# Efficient Algorithms for Hard Problems in Nondeterministic Tree Automata

*Ricardo Manuel de Oliveira Almeida*

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2017

# Abstract

We present PTIME language-preserving techniques for the reduction of nondeterministic tree automata, both for the case of finite trees and for infinite trees.

Our techniques are based on new transition removing and state merging results, which rely on binary relations that compare the downward and upward behaviours of states in the automaton. We use downward/upward simulation preorders and the more general but EXPTIME-complete trace inclusion relations, for which we introduce good under-approximations computable in polynomial time. We provide a complete picture of combinations of downward and upward simulation/trace inclusions which can be used in our reduction techniques.

We define an algorithm that puts together all the reduction results found for finite trees, and implemented it under the name `minotaut`, a tool built on top of the well-known tree automata library `libvata`. We tested `minotaut` on large collections of automata from program verification provenience, as well as on different classes of randomly generated automata. Our algorithm yields substantially smaller and sparser automata than all previously known reduction techniques, and it is still fast enough to handle large instances.

Taking reduction of automata on finite trees one step further, we then introduce saturation, a technique that consists of adding new transitions to an automaton while preserving its language. We implemented this technique on `minotaut` and we show how it can make subsequent state-merge and transition-removal operations more effective. Thus we obtain a PTIME algorithm that reduces the number of states of tree automata even more than before.

Additionally, we explore how `minotaut` alone can play an important role when performing hard operations like complementation, allowing to obtain smaller complement automata and at lower computation times overall. We then show how saturation can extend this contribution even further. An overview of the tool, highlighting some of its implementation features, is presented as well.

# Acknowledgements

First of all, I would like to thank my supervisor Richard Mayr for giving me the chance to work on such a fascinating topic. I am deeply grateful for the immense support and dedication provided throughout my PhD. Thank you for your patience and trust and for being a co-author of part of the work in this thesis.

I would also like to thank my examiners Tomáš Vojnar and Colin Stirling for such a careful read of my thesis. The issues raised and the feedback given were truly valuable.

I would like to thank EPSRC for the scholarship which allowed me to pursue this degree. I am also very grateful to LFCS for the financial support I received to attend and present my work at conferences.

I will always be grateful to my parents for all they have given me from day one. For educating me, for planting in me the taste for maths, for teaching me the discipline and the perseverance essential to achieve great accomplishments in life. For accepting so well my relocation to Scotland, for never making it obvious how difficult the distance between us can be. Thank you for believing in me and for supporting me always no matter what. This thesis is dedicated to you.

My life in Scotland would have never been the same without Natalia, who I met halfway through my PhD. Thank you for coming into my life and bringing so much more than I would ever have imagined. Thank you for always believing in me and for supporting me like no one else can. Thank you for, somehow, always managing to lift me up on the toughest moments. I will always be grateful to you.

# Declaration

I declare that this thesis has been composed by myself and that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included, and that this is explicitly stated below. Moreover, I declare that the work here presented has not been submitted for any other degree or professional qualification.

The work presented in Sections 2.1-2.7 was previously published in the proceedings of TACAS'16 as *Reduction of Nondeterministic Tree Automata* by Ricardo Almeida (author), Lukáš Holík and Richard Mayr (supervisor), to which the author made a substantial contribution.

26 May 2017

(Ricardo Manuel de Oliveira Almeida)

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Background

Tree automata are a generalization of word automata that accept trees instead of words. Different researchers have contributed to their formalisation, namely A. Church in the 50's and also B. Trakhtenbrot, J.R. Büchi, M.O. Rabin, Doner, Thatcher in the 60's, among others. Presently, the main reference for tree automata is *Tree Automata Techniques and Applications* (TATA) [CDG$^+$08], the result of contributions of various authors.

Tree automata can easily model notions such as the behaviour of computer programs manipulating complex structures (e.g., heaps and other tree-like structures), algebraic term representations of data, syntactical trees and XML documents and many others. Hence tree automata appear in many areas of computer science, such as formal verification, natural language processing, structured documents processing, and decision procedures of various logics. In the field of program verification, they have applications in model checking [ALdR06, ALdR05, BHRV06], term rewriting [Dur15] and shape analysis [AHJ$^+$13, HLR$^+$13, HHR$^+$11]. Thus several software packages for manipulating tree automata have been developed, e.g., MONA [BKM15], Timbuk [ea15], Autowrite [Dur15] and also `libvata` [LSV15], on which other verification tools are based (e.g., Forester [LSV$^+$17b] for shape analysis and the decision procedures SLIDE [IRV15, IRV14, IRS13] and SPEN [ELS$^+$17, ELSV14] for separation logic).

Infinite trees are a more recent concept but have been a subject of study for several decades now. The first formulation of automata on infinite trees was made by Rabin in the field of mathematical logic, namely for deciding the monadic second-order (MSO) theory of two successor functions (the infinite binary tree) [Rab69]. Rabin's famous theorem states that a set of infinite labelled trees can be defined in MSO logic if and only if it can be recognized by a finite automaton on infinite trees. Since MSO formulas can be interpreted as automata, deciding logical problems then translates into solving automata-theoretical questions. Besides Rabin tree automata, there are several other classes of automata on infinite trees, such as Büchi, Müller, Street and parity automata. Here we consider nondeterministic Büchi tree automata, an extension of Büchi automata on words [Büc90].

Automata on infinite trees have also been applied to solving program verification tasks. For instance, Vardi provided an automata-theoretic framework for the specification and verification of concurrent and nondeterministic programs [Var91]. In the case of words (i.e., linear trees), Büchi language inclusion has many applications in software verification, such as checking the termination of programs [HHP14].

## 1.2 Language Inclusion and Complementation

Many verification problems can thus be formulated as instances of the language inclusion problem. For example, it is possible to describe the behaviour of an implementation as an automaton $A$ and all the behaviours allowed by the specification as a different automaton $B$. Then, the problem of verifying whether the implementation meets the specification is equivalent to checking $L(A) \subseteq L(B)$. The language inclusion problem can be solved via the computation of the complement of $B$ and it is `PSPACE`-complete for word automata [KV98] and `EXPTIME`-complete for tree automata [CDG⁺08].

For NFAs, complementation is achieved by first determinizing the automaton, modifying it to make it complete and then complementing the set of final states. The classical approach to determinizing an NFA is the Rabin-Scott powerset construction (also known as the subset construction), which suffers from a $\mathcal{O}(2^n)$ worst-case complexity on the initial number of states [RS59].

For Büchi automata, various methods have been proposed to handle the complementation step. The first method was developed by Büchi himself [Büc90]. It is known as the Ramsey method, due to it relying on the infinite Ramsey theorem for large complete graphs [Ram87]. However, this construction has efficiency problems, as the automata experience a $2^{2^{O(n)}}$ blow-up in the state size. This construction was later improved by Sistla, Vardi and Wolfer, yielding a single-exponential complexity of $2^{O(n^2)}$ [SVW87]. A more efficient approach has been proposed by Safra [Saf88]. It is based on determinization and has a complexity of $2^{O(n \log n)}$. Later Piterman improved this method by using parity automata as the intermediate deterministic automata. This approach is known as the Safra-Piterman construction and has an upper bound of $n^{2n}$ [Pit06]. The rank-based approach, proposed by Kupferman and Vardi, uses rank functions to measure the progress made by a node in the run tree towards fair termination [KV01]. It has a complexity of $2^{O(n \log n)}$. A slice-based construction has been presented by Kähler and Wilke and observes a $(3n)^n$ blow-up in the number of states [KW08, VW08].

Efficient implementations for all these complementation constructions are provided in the tool `GOAL` [TTLH15, TCT+08]. The experimental assessment carried out in [TFVT14] compared their performance and concluded that, on average, the Safra-Piterman construction performs better than the other three methods in terms of time and state size. The authors also concluded that the Ramsey-based construction is not competitive for complementation though it is competitive for universality and inclusion testing. In [FKWV13] the authors unify the rank-based and the slice-based approaches, obtaining an improved algorithm with an upper bound of $(0.76n)^n$.

For automata on finite trees, the approach depends on whether we are considering bottom-up tree automata (which read trees from the leaves to the root) or top-down tree automata (which read trees in the inverse direction). While the two representations are equivalent, it has been proved that deterministic top-down tree automata are strictly less expressive than nondeterministic ones [CDG+08]. Thus, in general, determinization is not possible for top-down tree automata. In this case one can achieve the complement automaton by computing the difference algorithm [Hos10]. For the case of bottom-up tree automata, the powerset approach can be used as well [CDG+08].

Yet, whether one is performing an explicit complementation or not, it is common to start by making the automata smaller in a pre-processing step. As we will see, this effort often pays off as it makes hard operations and tests significantly easier to compute in general.

## 1.3 Automata Reduction

In this section we make a brief summary of some of the recent advancements on efficient reduction of nondeterministic automata on words and on trees. The main applications of reduction are the following:

1. Help in solving hard problems and operations like language inclusion/universality, complementation, etc.
2. If automata undergo a long chain of manipulations by operations like union, intersection, projection, etc., then intermediate results can be reduced several times on the way to keep the automata within a manageable size.
3. There are fixed-parameter tractable problems (e.g., in model checking where

an automaton encodes a logic formula) where the size of one automaton very strongly influences the overall complexity, and must be kept as small as possible.

### 1.3.1 Word Automata

For Büchi automata we have the graphical interactive tool `GOAL` [TTLH15, TCT+08], which manipulates automata and temporal logic. It features simulation-based reduction, followed by an on-the-fly variant of Piterman's construction [Pit06, TCT+08]. It supports several other tests and operations: on-the-fly model checking of a system automaton against a property automaton, emptiness, complement, determinization, among many others. Although the tool focuses primarily on Büchi automata it also features tests and operations on NFA, as well as a wide variety of operations for Logic Formulae, Games, among others.

However, significantly better results on reduction of Büchi automata have been achieved when considering generalized simulations on the state space that approximate language inclusion [CM13]. These relations are combined to define suitable criteria for *transition pruning*, i.e., deleting transitions in the automaton if other ones remain which are *better* w.r.t. the simulations being considered, as well as the more classical operation of *state quotienting*, in which states with the same *behaviour* (i.e., they are equivalent w.r.t. a suitable preorder) can be collapsed into just one state. Since language inclusion is PSPACE-complete for words, the authors define various types of forward and backward (when the transitions in the automaton are taken backwards) simulation which are computable in polynomial time and which under-approximate language inclusion.

These reduction techniques not only efficiently reduce Büchi automata but they also solve many instances of language inclusion during the computation of simulations already, since simulation preorders often witness language inclusion. The algorithm has been made available in the tool `RABIT` (RAmsey-based Büchi automata Inclusion Testing) [Lan17] and significantly outperforms previous methods, such as the `GOAL` tool. `RABIT` uses the simulation-based reduction for checking language inclusion between Büchi automata, and the Ramsey method to find counterexamples in cases where inclusion does not hold. The tool has been extended to handle NFA as well.

### 1.3.2 Tree Automata

The traditional approach for testing language inclusion on word automata can be applied to bottom-up tree automata as well. An alternative approach has been proposed where the determinization step is avoided by replacing the subset construction by the antichains of sets of states [BHH+08]. This approach is inspired by previous work on word automata, where antichains were used in dual forward and backward algorithms for inclusion and universality testing [WDHfR06]. In [BHH+08] the authors adapt that approach by showing how the forward algorithm can be extended to finite tree automata (using algorithms computing upwards).

While on word automata we have forward and backward simulations, on tree automata we speak of downward and upward simulations. However, as we will see, upward simulations do not correspond directly to backwards on words as they are substantially more complex. The notions of downward and upward simulations first appeared in [ALdR05], where they are used for proving the soundness of some acceleration techniques used in regular tree model checking. Computing a simulation for tree automata was first addressed in [ABH+08], where the authors devise state quotienting techniques based on downward and upward simulations. Both quotienting with downward simulation and with upward simulation are reduction techniques featured in `libvata` [LSV15], a highly-optimised library for the manipulation of finite tree automata. The library features several other operations on automata, such as complementation, union and intersection, and supports both explicit and semi-symbolic tree automata. A more intricate and coarser preorder, called combined preorder, combines downward with upward simulations and is suitable for quotienting as well [AHKV09].

Reduction via quotienting based on simulation preorders has also been addressed for the case of infinite trees, namely by using delayed downward simulation (a coarser notion than ordinary direct downward simulation) for alternating Büchi tree automata [vB08], of which Büchi tree automata are a particular case.

Downward and upward simulations on finite tree automata have been used for solving language inclusion directly as well. The downward approach to checking language inclusion in tree automata was first presented by Hosoya [HVP05] in the context of subtyping of XML types. This algorithm does not derive from the classical approach to solving the inclusion problem, and it has a complex structure with weak points of its own. A different approach combined simulations with antichains, where upward simulations are used to prune out unnecessary search paths of the antichains-based method [ACH+10]. However, this algorithm has the disadvantage of being compatible with upward simulations only. Since downward simulations are often larger and cheaper to compute than upward simulations, in [HLSV11] the authors improve Hosoya's algorithm to obtain a new algorithm that combines antichains with downward simulations. This yields an algorithm that significantly outperforms the upwards approach in most of the test cases performed. The algorithm is used in `libvata` for solving instances of the language inclusion problem.

## 1.4 Our Contribution

Our goal is to make automata more computationally tractable for solving hard automata problems and operations. We achieve this by reducing their size while retaining the language. Thus there is an algorithmic tradeoff between the effort for reduction and the complexity of the problem later considered for the automata.

In Chapter 2 we present two efficient algorithms for the reduction of nondeterministic automata on finite trees, in the sense of obtaining a smaller automaton with the same language, though not necessarily with the absolute minimal possible number of states. In fact, generally there is no unique nondeterministic automaton with the minimal possible number of states for a given language. The first algorithm, *Heavy*, is based on a combination of new transition pruning techniques and quotienting of the state space w.r.t. suitable equivalences. The pruning techniques are related to those presented for word automata in [CM13], but significantly more complex due to the fundamental asymmetry between the upward and downward directions in trees. The algorithm *Heavy* has been implemented as the tool `minotaut` [Alm16a] and, together with all the related pruning and quotienting results, it has been presented in [AHM16].

As mentioned above, transition pruning in word automata is based on the observation that certain transitions can be removed without changing the language, because other *better* transitions remain. One defines some strict partial order (p.o.) between transitions and removes all the transitions that are not maximal w.r.t. this order. A strict p.o. between transitions is called *good for pruning* (GFP) iff pruning w.r.t. it preserves the language of the automaton. Note that pruning reduces not only the number of transitions, but also, indirectly, the number of states. By removing transitions, some states may become 'useless', in the sense that they are unreachable from any initial state, or that it is impossible to reach any accepting state from them. Such useless states can then be removed from the automaton without changing its language. One can obtain computable strict p.o. between transitions by comparing the possible backward and forward behaviours of their source and target states, respectively. In order to do this, one uses computable relations like backward/forward simulation preorder and approximations of backward/forward trace inclusion via lookahead simulations. Such combinations of backward/forward trace/simulation orders on states may or may not induce strict p.o. between transitions that are GFP [CM13]. However, there is always a symmetry between backward and forward, since finite words can equally well be read in either direction.

This symmetry does not hold for tree automata, because a tree branches as one goes downward, while it might 'join in' side branches as one goes upward. Therefore, while downward simulation preorder (respectively downward language inclusion) between states in a tree automaton is a direct generalization of forward simulation preorder (respectively forward language inclusion) on words, the corresponding upward notions do not correspond to backward on words. Comparing upward behavior of states in tree automata depends also on the branches that 'join in' from the sides as one goes upward in the tree. Thus upward simulation/language inclusion is only defined *relative* to a given other relation that compares the downward behavior of states 'joining in' from the sides [ABH$^+$08]. So one speaks of "upward simulation *of* the identity relation" or "upward simulation *of* downward simulation". When one studies strict p.o. between transitions in tree automata in order to check whether they are GFP or not, one has combinations of three relations: the source states are compared by an upward relation $X(Y)$ of some downward relation $Y$, while the target states are compared w.r.t. some downward relation $Z$ (where $Z$ can be, and often must be, different from $Y$). This yields a richer landscape, and many counter-intuitive effects.

In Chapter 2 we provide a complete picture of combinations of upward and downward simulation/trace inclusions which are GFP for finite tree automata. Since trace/language inclusion is `EXPTIME`-complete for trees [CDG$^+$08], we describe methods to compute good approximations in polynomial time, by generalizing lookahead simulations from words [CM13] to trees. We also generalize results on quotienting of tree automata [Hol11] to larger relations, such as approximations of trace inclusion.

The second algorithm, *Sat*, combines *Heavy* with the dual notion of transition pruning, in which transitions are added to the automaton if *better* ones exist already. This technique is known as transition saturation and it has previously been defined for words [CM16]. One defines some reflexive binary relation between (existing or non-existing) transitions and adds all transitions that are not maximal w.r.t. the relation. A relation is called *good for saturation* (GFS) iff adding transitions w.r.t. it preserves the language of the automaton. We provide a comprehensive table of results showing which combinations of upward and downward simulation/trace inclusions are GFS for finite tree automata.

The motivation behind saturation is that it may allow for new merging of states and transition removal which were not possible by using *Heavy* alone. Therefore saturating an automaton which has been reduced with *Heavy* and then reducing it again might yield an automaton which is even smaller. Thus our algorithm *Sat* alternates between reducing an automaton with *Heavy* and saturating it for as long as new reductions are achieved. We present five different versions of *Sat*, each of them performing the alternation in a different way. *Sat* has been implemented in `minotaut` and, together with the saturation results used, the algorithm was described in [Alm16b].

We dedicate Chapter 3 to the experimental evaluation of the algorithms *Heavy* and

*Sat* and to some of the relevant aspects of their implementation. Both algorithms are available in the tool `minotaut` [Alm16a], which was built on top of the `libvata` [LSV15] library. We present the results of testing the performance of the algorithms on a given collection of tree automata from various applications of `libvata` in program verification, as well as on many classes of randomly generated tree automata. We show that *Heavy* yields substantially smaller automata than all previously known reduction techniques, which are mainly based on quotienting. Moreover, the thus obtained automata are also much sparser (i.e., they use fewer transitions per state and less nondeterministic branching) than the originals, which yields additional performance advantages in subsequent computations. Our *Sat* algorithm generally obtains automata with fewer states, but in some cases with more transitions, than the ones obtained with *Heavy* alone.

As mentioned before, one wishes to reduce automata in order to make them more efficient to handle in subsequent computations. Thus in Chapter 3 we present a second experimental evaluation showing that when computing complement automata, one obtains much smaller complements and at much lower times overall when the automata are first reduced with *Heavy*. Preceding the complementation of automata by reduction with *Sat* yields complements with even less states, but sometimes more transitions.

Chapter 4 is dedicated to reduction techniques for Büchi tree automata. As in the case of infinite words, downward relations (simulations and trace inclusions) on infinite trees offer a richer scenario then in the finite case, as one can speak not only of direct downward relations but also of delayed and fair relations. Since trace/language inclusion is `EXPTIME`-complete for trees, we describe methods to compute good under-approximations in polynomial time, by generalizing lookahead simulations [CM13] to infinite trees. Moreover, we explore yet another approximation of downward trace inclusion, called downward fixed-tree simulation, which has previously been defined for the case of words and shown to be PSPACE-complete [Cle11]. One can also define downward fixed-tree simulation in terms of direct/delayed/fair relations. We thus explore which possible downward and upward relations are suitable for quotienting. In particular, we show that delayed fixed-tree simulation is good for quotienting. Finally, we provide a complete picture of combinations of upward and direct/delayed/fair downward relations which are suitable for pruning transitions in Büchi tree automata.

# Chapter 2

# Finite Tree Automata

Tree automata are a generalization of word automata that accept trees instead of words [CDG$^+$08]. They have many applications in model checking [ALdR06, ALdR05, BHRV06], shape analysis [AHJ$^+$13, HLR$^+$13, HHR$^+$11] and related areas of formal software verification.

Our goal is to make tree automata more computationally tractable in practice. We present two efficient algorithms for the reduction of nondeterministic tree automata, in the sense of obtaining a smaller automaton with the same language, though not necessarily with the absolute minimal possible number of states. The reason to perform reduction is that the smaller reduced automaton is more efficient to handle in a subsequent computation, such as testing language inclusion, which is `EXPTIME`-complete for tree automata [CDG$^+$08]. Thus there is an algorithmic tradeoff between the effort for reduction and the complexity of the problem later considered for this automaton.

**Chapter outline.** We start this chapter by presenting the preliminary definitions for finite trees and tree automata in Section 2.1. In Section 2.2 the notions of downward/upward simulation and trace inclusion preorders on the states of an automaton are formalized, both for word automata and for tree automata. Since trace/language inclusion is `EXPTIME`-complete for trees [CDG$^+$08], in Section 2.3 we describe methods to compute good approximations of them in polynomial time, by generalizing lookahead simulations [CM13] to trees. We give both an intuition by simulation games between two players as well as a formal coinductive definition.

Section 2.4 introduces the first automata reduction technique, transition pruning, showing how the relations defined in the previous sections can be used to find transitions in an automaton which may be deleted because *better* ones remain. We provide a complete picture of which combinations of upward/downward simulation/trace inclusions are suitable for pruning on tree automata.

Section 2.5 is dedicated to the technique of state quotienting, according to which some states in the automaton may have the same *behaviour* and therefore can be merged into a single state. We generalize previous quotienting results based on simulations [Hol11] to larger relations, such as approximations of trace inclusion. We show which relations are or are not suitable for quotienting states in an automaton.

Section 2.6 puts together transition pruning and state quotienting to define the automata reduction algorithm *Heavy*. In Section 2.7 we show that one of the best known tree automata reduction techniques cannot achieve any further reduction on automata which have previously been reduced with *Heavy*.

Section 2.8 introduces the dual technique to transition pruning, in which transitions are added to an automaton if *better* ones exist already. This technique is known as transition saturation and it has previously been defined for word automata [CM16]. One defines some reflexive binary relation between (existing or non-existing) transitions in the automaton and adds all transitions that are not maximal w.r.t. this relation. A relation is called *good for saturation* (GFS) iff adding transitions w.r.t. it preserves the language of the automaton. We provide a comprehensive table of results showing which combinations of upward/downward simulation/trace inclusions are GFS on tree automata.

The motivation behind saturation is that it may allow for new merging of states and transition removal which were not possible by using *Heavy* alone. Therefore saturating an automaton which has been reduced with *Heavy* and then reducing it again might result in an even smaller automaton. Thus in Section 2.9 we introduce the algorithm *Sat*, which alternates between reducing an automaton with *Heavy* and saturating it for as long as new reductions are achieved in the automaton. We present five different versions of *Sat*, each of them performing the alternation in a different way.

## 2.1 Trees and Tree Automata

**Trees.** A *ranked alphabet* $\Sigma$ is a set of symbols together with a function $\# : \Sigma \to \mathbb{N}_0$. For $a \in \Sigma$, $\#(a)$ is called the *rank* of $a$. For $n \geq 0$, we denote by $\Sigma_n$ the set of all symbols of $\Sigma$ which have rank $n$.

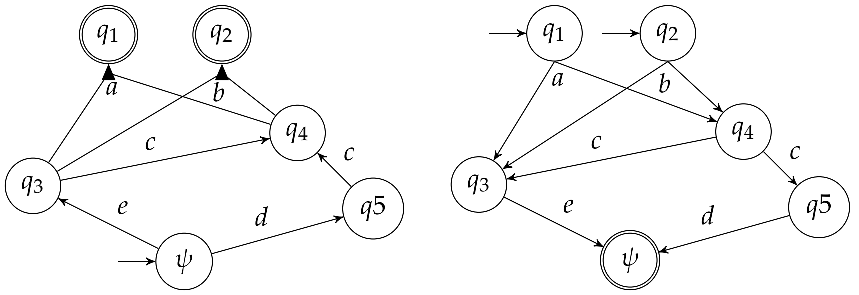
We define a *node* as a sequence of elements of $\mathbb{N}$, where $\varepsilon$ is the empty sequence. For a node $v \in \mathbb{N}^*$, any node $v'$ s.t. $v = v'v''$, for some node $v''$, is said to be a *prefix* of $v$, and if $v'' \neq \varepsilon$ then $v'$ is a *strict prefix* of $v$. For a node $v \in \mathbb{N}^*$, we define the $i$-th child of $v$ to be the node $vi$, for some $i \in \mathbb{N}$. Given a ranked alphabet $\Sigma$, a *tree* over $\Sigma$ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$ such that for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $vi \in dom(t)$ then **(1)** $v \in dom(t)$, and **(2)** $\#(t(v)) \geq i$. Note that the number of children of a node $v$ may be smaller than $\#(t(v))$. In this case we say that the node is *open*. Nodes which have exactly $\#(t(v))$ children are called *closed*. Nodes which do not have any children are called *leaves*. A tree is closed if all its nodes are closed, otherwise it is open. By $\mathbb{C}(\Sigma)$ we denote the set of all closed trees over $\Sigma$ and by $\mathbb{T}(\Sigma)$ the set of all trees over $\Sigma$. A tree $t$ is *linear* iff every node in $dom(t)$ has at most one child.

The *subtree* of a tree $t$ at $v$ is defined as the tree $t_v$ such that $dom(t_v) = \{v' \mid vv' \in dom(t)\}$ and $t_v(v') = t(vv')$ for all $v' \in dom(t_v)$. A tree $t'$ is a prefix of $t$ iff $dom(t') \subseteq dom(t)$ and for all $v \in dom(t')$, $t'(v) = t(v)$. For $t \in \mathbb{C}(\Sigma)$, the *height of a node $v$* of $t$ is given by the function $h$: if $v$ is a leaf then $h(v) = 1$, otherwise $h(v) = 1 + max(h(v1)), \ldots, h(v\#(t(v))))$. We define the height of a tree $t \in \mathbb{C}(\Sigma)$ as $h(\epsilon)$, i.e., as the number of levels of $t$, and often write it as simply $h(t)$.

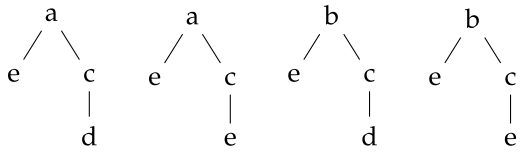
**Tree automata, bottom-up.** A (finite, nondeterministic) *bottom-up tree automaton* (BUTA) reads a tree from the leaves to the root. Formally, it is defined by a quadruple $A = (\Sigma, Q, \delta, F)$ where $Q$ is a finite set of states, $F \subseteq Q$ is a set of final states, $\Sigma$ is a ranked alphabet, and $\delta \subseteq Q^+ \times \Sigma \times Q$ is set of transition rules. A BUTA has an unique initial state, which we represent by $\psi$. The bottom-up representation is the most common tree automata representation found in the literature, although the initial state is often not made explicit: $\psi$ is omitted and any *initial transition* has a target state and a leaf symbol but no source state [CDG$^+$08, BHH$^+$08, HLSV11].

**Tree automata, top-down.** In analogy to automata on words, which read a word from the beginning to the end (i.e., left-to-right), one may also define tree automata reading a tree from its root to the leaves (i.e., top-down). A (finite, nondeterministic) *top-down tree automaton* (TDTA) is thus defined as a quadruple $(\Sigma, Q, \delta, I)$ where $Q$ is the finite set of states, $I \subseteq Q$ is the set of initial states, $\Sigma$ is a ranked alphabet, $\delta \subseteq Q \times \Sigma \times Q^+$ is the set of transition rules and $\psi \in Q$ denotes the (unique) final state. The transition rules satisfy that if $\langle q, a, \psi \rangle \in \delta$ then $\#(a) = 0$, and if $\langle q, a, q_1 \ldots q_n \rangle \in \delta$ (with $n > 0$) then $\#(a) = n$. A TDTA can be obtained from a BUTA by swapping the roles between the initial states and the final states, and by reversing the direction of the transition rules. See Figure 2.1 for an example of a tree automaton presented in both BUTA and TDTA form. In this thesis we adopt TDTA as the standard representation of a tree automaton in order to make the connection to word automata more explicit.

Let $A$ be a TDTA. A *run* of $A$ over a tree $t \in \mathbb{T}(\Sigma)$ (or a $t$-run in $A$) is a partial mapping $\pi : \mathbb{N}^* \rightarrow Q$ such that $v \in dom(\pi)$ iff either $v \in dom(t)$ or $v = v'i$ where $v' \in dom(t)$ and $i \leq \#(t(v'))$. Further, for every $v \in dom(t)$, there exists either **a)** a rule $\langle q, a, \psi \rangle$ such that $q = \pi(v)$ and $a = t(v)$, or **b)** a rule $\langle q, a, q_1 \ldots q_n \rangle$ such that $q = \pi(v)$, $a = t(v)$, and $q_i = \pi(vi)$



(a) $A_{BU} = (\Sigma, Q, \delta_{BU}, F = \{q_1, q_2\})$. (b) $A_{TD} = (\Sigma, Q, \delta_{TD}, I = \{q_1, q_2\})$.



(c) The trees accepted by $A_{BU}$ and $A_{TD}$.

Figure 2.1: Let $\Sigma$ be a ranked alphabet such that $\Sigma_0 = \{d, e\}$, $\Sigma_1 = \{c\}$ and $\Sigma_2 = \{a, b\}$. Consider the BUTA $A_{BU}$ and the TDTA $A_{TD}$, where $Q = \{q_1, \ldots, q_5\}$ and $\delta_{BU} = \{\langle \psi, e, q_3 \rangle, \langle \psi, d, q_5 \rangle, \langle q_5, c, q_4 \rangle, \langle q_3, c, q_4 \rangle, \langle q_3 q_4, a, q_1 \rangle, \langle q_3 q_4, b, q_2 \rangle\}$. $A_{TD}$ is obtained from $A_{BU}$ by reversing the transition rules in $\delta_{BU}$ and by swapping the roles of the accepting and the final states. The language accepted by the automata is $L = \{a(e, c(d)), a(e, c(e)), b(e, c(d)), b(e, c(e))\}$, as represented in c).

for each $i : 1 \leq i \leq \#(a)$. A *leaf of a run* $\pi$ over $t$ is a node $v \in dom(\pi)$ such that $vi \in dom(\pi)$ for no $i \in \mathbb{N}$. We call it *dangling* if $v \notin dom(t)$. Intuitively, the dangling nodes of a run over $t$ are all the nodes which are in $\pi$ but are missing in $t$ due to it being incomplete. Notice that dangling leaves of $\pi$ are children of open nodes of $t$. The prefix of depth $k$ of a run $\pi$ is denoted $\pi_k$. Runs are always finite since the trees we are considering too are finite.

We write $t \overset{\pi}{\Longrightarrow} q$ to denote that $\pi$ is a $t$-run of $A$ such that $\pi(\epsilon) = q$. We use $t \Longrightarrow q$ to denote that such run $\pi$ exists. A run $\pi$ is accepting if $t \overset{\pi}{\Longrightarrow} q \in I$. The *downward language of a state* $q$ in $A$ is defined by $D_A(q) = \{t \in \mathbb{C}(\Sigma) \mid t \Longrightarrow q\}$, while the *language* of $A$ is defined by $L(A) = \bigcup_{q \in I} D_A(q)$. The *upward language* of a state $q$ in $A$, denoted $U_A(q)$, is then defined as the set of open trees $t$, such that there exists an accepting $t$-run $\pi$ with exactly one dangling leaf $v$ s.t. $\pi(v) = q$. We omit the $A$ subscript notation when it is implicit which automaton we are considering.

## 2.2 Simulations and Trace Inclusion Relations

We consider different types of relations on states of a TDTA which under-approximate language inclusion. Note that words are but a special case of trees where every node has only one child, i.e., words are linear trees. *Downward* simulation/trace inclusion on TDTA corresponds to *direct forward* simulation/trace inclusion in the special case of word automata, and *upward* corresponds to *backward* [CM13].

**Forward simulation on word automata.** Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA. A *direct forward simulation D* is a binary relation on $Q$ such that if $q \, D \, r$, then

1. $q \in F \Longrightarrow r \in F$, and
2. for any $\langle q, a, q' \rangle \in \delta$, there exists $\langle r, a, r' \rangle \in \delta$ such that $q' \, D \, r'$.

The set of direct forward simulations on $A$ contains *id* and is closed under union and transitive closure. Thus there is a unique maximal direct forward simulation on $A$, which is a preorder. We call it *the direct forward simulation preorder on A* and write $\sqsubseteq^{\mathsf{di}}$.

**Forward trace inclusion on word automata.** Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA and $w = \sigma_1 \sigma_2 \ldots \sigma_n \in \Sigma^*$ a word of length $n$. A trace of $A$ on $w$ (or a $w$-trace) starting at $q$ is a sequence of transitions $\pi = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$ such that $q_0 = q$. The *direct forward trace inclusion* preorder $\subseteq^{\mathsf{di}}$ is a binary relation on $Q$ such that $q \subseteq^{\mathsf{di}} r$ iff

1. $(q \in F \Longrightarrow r \in F)$, and
2. for every word $w = \sigma_1 \sigma_2 \ldots \sigma_n \in \Sigma^*$ and for every $w$-trace (starting at $q$) $\pi_q = q \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$, there exists a $w$-trace (starting at $r$) $\pi_r = r \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} r_n$ such that $(q_i \in F \Longrightarrow r_i \in F)$ for each $i : 1 \leq i \leq n$.

Since $\pi_r$ is required to preserve the acceptance of the states in $\pi_q$, trace inclusion is a strictly stronger notion than language inclusion (see Figure 2.2 for an example on the particular case of NFA).

**Downward simulation on tree automata.** Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. A *downward simulation D* is a binary relation on $Q$ such that if $q \, D \, r$, then

1. $(q = \psi \Longrightarrow r = \psi)$, and
2. for any $\langle q, a, q_1 \ldots q_n \rangle \in \delta$, there exists $\langle r, a, r_1 \ldots r_n \rangle \in \delta$ s.t. $q_i D r_i$ for $i : 1 \leq i \leq n$.

Since the set of all downward simulations on $A$ is closed under union and under reflexive and transitive closure (cf. Lemma 4.1 in [Hol11]), it follows that there is one unique maximal downward simulation on $A$, and that relation is a preorder. We call it *the downward simulation preorder on* $A$ and write $\sqsubseteq^{\mathsf{dw}}$.

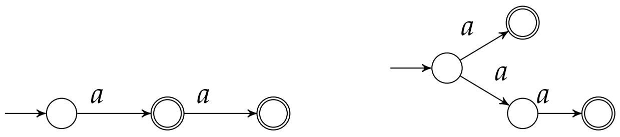
**Downward trace inclusion on tree automata.** Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. The *downward trace inclusion* preorder $\subseteq^{\mathsf{dw}}$ is a binary relation on $Q$ s.t. $q \subseteq^{\mathsf{dw}} r$ iff for every tree $t \in \mathbb{C}(\Sigma)$ and for every $t$-run $\pi_q$ with $\pi_q(\epsilon) = q$ there exists another $t$-run $\pi_r$ s.t. $\pi_r(\epsilon) = r$.

Generally, one way of making downward language inclusion on the states of an automaton coincide with downward trace inclusion is by modifying the automaton to guarantee that **1)** there is one unique final state which has no outgoing transitions, **2)** from any other state reachable from the initial state, there is a path ending in that final state. Note that in a TDTA these two conditions are automatically satisfied: **1)** since the final state is reached after reading a leaf of the tree, and **2)** because only complete trees are in the language of the automaton. Thus, in a TDTA, downward language inclusion and downward trace inclusion coincide.

**Backward simulation on word automata.** Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA. A *backward simulation* $B$ is a binary relation on $Q$ s.t. if $q\ B\ r$, then

1. $(q \in F \implies r \in F)$ and $(q \in I \implies r \in I)$, and
2. for any $\langle q', a, q\rangle \in \delta$, there exists $\langle r', a, r\rangle \in \delta$ s.t. $q'\ B\ r'$.

Like for forward simulation, there is a unique maximal backward simulation on $A$, which is a preorder. We call it *the backward simulation preorder on* $A$ and write $\sqsubseteq^{\mathsf{bw}}$.



(a) Automaton $A_1$.
(b) Automaton $A_2$.

Figure 2.2: An example of two NFAs for which language inclusion holds but trace inclusion does not: the trace for $aa$ does not preserve acceptance in the second state in $A_2$.

**Backward trace inclusion on word automata.** Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA and $w = \sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$ a word of length $n$. A $w$-trace of $A$ ending at $q$ is a sequence of transitions $\pi = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$ such that $q_n = q$. The *backward trace inclusion* preorder $\subseteq^{\mathsf{bw}}$ is a binary relation on $Q$ such that $q \subseteq^{\mathsf{bw}} r$ iff

1. $(q \in F \Longrightarrow r \in F)$ and $(q \in I \Longrightarrow r \in I)$, and
2. for every word $w = \sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$ and for every $w$-trace (ending at $q$) $\pi_q = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q$, there exists a $w$-trace (ending at $r$) $\pi_r = r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} r$ such that $(q_i \in F \Longrightarrow r_i \in F \wedge q_i \in I \Longrightarrow r_i \in I)$ for each $i : 1 \leq i \leq n$.

**Upward simulation on tree automata.** Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. Given a binary relation $R$ on $Q$, an *upward simulation* $U(R)$ induced by $R$ is a binary relation on $Q$ such that if $q\ U(R)\ r$, then

1. $(q = \psi \Longrightarrow r = \psi)$ and $(q \in I \Longrightarrow r \in I)$, and
2. for any $\langle q', a, q_1 \ldots q_n\rangle \in \delta$ with $q_i = q$ (for some $i : 1 \leq i \leq n$), there exists $\langle r', a, r_1 \ldots r_n\rangle \in \delta$ such that $r_i = r$, $q'\ U(R)\ r'$ and $q_j\ R\ r_j$ for each $j : 1 \leq j \neq i \leq n$.

Similarly to the case of downward simulation, for any given relation $R$, there is a unique maximal upward simulation induced by $R$ which is a preorder (cf. Lemma 4.2 in [Hol11]). We call it *the upward simulation preorder on A induced by R* and write $\sqsubseteq^{\mathsf{up}}(R)$.

**Upward trace inclusion on tree automata.** Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. Given a binary relation $R$ on $Q$, the *upward trace inclusion preorder* $\subseteq^{\mathsf{up}}(R)$ *induced by* $R$ is a binary relation on $Q$ such that $q \subseteq^{\mathsf{up}}(R)\, r$ iff the following holds: for every tree $t \in T(\Sigma)$ and for every $t$-run $\pi_q$ with $\pi_q(v) = q$ for some dangling leaf $v$ of $\pi_q$, there exists a $t$-run $\pi_r$ s.t.

1. $\pi_r(v) = r$,
2. for all prefixes $v'$ of $v$, $(\pi_q(v') \in I \Longrightarrow \pi_r(v') \in I)$, and
3. if $v'x \in dom(\pi_q)$, for some strict prefix $v'$ of $v$ and some $x \in \mathbb{N}$ s.t. $v'x$ is not a prefix of $v$, then $\pi_q(v'x)\ R\ \pi_r(v'x)$.

Downward simulation preorder is computable in polynomial time [ABH+08]. The complexity on the upwards case depends on the relation $R$, i.e., upward simulation is polynomial-time computable when $R$ itself can be computed in polynomial time (e.g., $R = \sqsubseteq^{\mathsf{dw}}$ or in more trivial cases such as $R = Q \times Q$ or $R = id$ [ABH+08]), otherwise it has the same complexity as $R$. Although efficient to compute in general, downward and upward simulation preorders typically yield only small under-approximations of the corresponding trace inclusions, which are significantly harder to compute. Downward trace inclusion is `EXPTIME`-complete for TDTA [CDG⁺08], while forward trace inclusion is `PSPACE`-complete for word automata. The complexity of upward trace inclusion depends on $R$ (e.g., it is `PSPACE`-complete for $R = id$ or $R = Q \times Q$ but `EXPTIME`-complete for $R = \subseteq^{\mathsf{dw}}$).

## 2.3 Lookahead Simulations

Simulation preorders are generally not very good under-approximations of trace inclusion, since they are much smaller on many automata. Thus we consider better approximations that are still efficiently computable.

For word automata, more general *lookahead simulations* were introduced in [CM13]. These provide a practically useful tradeoff between the computational effort and the size of the obtained relations. Lookahead simulations can also be seen as a particular restriction of the more general (but less practically useful) *multipebble simulations* [Ete02]. We generalize lookahead simulations to tree automata in order to compute good under-approximations of trace inclusions. In Section 2.2 we defined ordinary simulation and trace inclusion relations coinductively. Let us first see how these relations can be interpreted in terms of games and then generalize them to the lookahead case. Formal coinductive definitions for lookahead downward and upward simulations follow.

**Intuition by Simulation Games.** Simulation preorders on labelled transition systems can be characterized by a game between two players, Spoiler and Duplicator. In the ordinary simulation case, given a pair of states $(q_0, r_0)$, Spoiler wants to show that $(q_0, r_0)$ is not contained in the simulation preorder relation, while Duplicator has the opposite goal. Starting in the initial configuration $(q_0, r_0)$, Spoiler chooses a transition $q_0 \xrightarrow{\sigma} q_1$ and Duplicator must imitate it *stepwise* by choosing a transition by the same symbol $r_0 \xrightarrow{\sigma} r_1$. This yields a new configuration $(q_1, r_1)$ from which the game continues. If a player cannot move the other wins. Duplicator wins every infinite game. Simulation holds iff Duplicator wins.

In ordinary simulation, Duplicator only knows Spoiler's very next step (as previously defined in Section 2.2), while in *k-lookahead simulation* Duplicator knows Spoiler's next $k$ steps in advance (unless Spoiler's move ends in a deadlocked state - i.e., a state with no transitions). As the parameter $k$ increases, the $k$-lookahead simulation relation becomes larger and thus approximates the trace inclusion relation better and better. Trace inclusion can also be characterized by a game. In the trace inclusion game, Duplicator knows *all* steps of Spoiler in the entire game in advance.

**Lookahead downward simulation on TDTA.** We say that a tree $t$ is $k$-bounded iff for all leaves $v$ of $t$, either **a)** $|v| = k$, or **b)** $|v| < k$ and $v$ is closed.

Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. A *k-lookahead downward simulation* $L^{k-\mathsf{dw}}$ is a binary relation on $Q$ such that if $q\ L^{k-\mathsf{dw}}\ r$, then the following holds: Let $\pi_k$ be a run on a $k$-bounded tree $t_k$ with $\pi(\epsilon) = q$ s.t. every leaf node of $\pi_k$ is either at depth $k$ or downward-deadlocked (i.e., no more downward transitions exist). Then there must exist a run $\pi'_k$ over a nonempty prefix $t'_k$ of $t_k$ s.t. (1) $\pi'_k(\epsilon) = r$, and (2) for every leaf $v$ of $\pi'_k$, $\pi_k(v)\ L^{k-\mathsf{dw}}\ \pi'_k(v)$. Since, for given $A$ and $k \geq 1$, lookahead downward simulations are closed under union, there exists a unique maximal one, and we call it *the k-lookahead downward simulation on A*, denoted by $\sqsubseteq^{k\text{-}\mathsf{dw}}$.

While $\sqsubseteq^{k\text{-}\mathsf{dw}}$ is trivially reflexive, it is not transitive in general (cf. [CM13], Appendix B). Since we only use it as a means to under-approximate the transitive trace inclusion relation $\subseteq^{\mathsf{dw}}$ (and require a preorder to induce an equivalence relation), we work with its transitive closure $\preceq^{k\text{-}\mathsf{dw}} := (\sqsubseteq^{k\text{-}\mathsf{dw}})^+$. In particular, $\preceq^{k\text{-}\mathsf{dw}} \subseteq \subseteq^{\mathsf{dw}}$. The asymmetric restriction is given by $\prec^{k\text{-}\mathsf{dw}} = \preceq^{k\text{-}\mathsf{dw}} \setminus (\preceq^{k\text{-}\mathsf{dw}})^{-1}$.

**Lookahead upward simulation on TDTA.** Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. We have that a *k-lookahead upward simulation* on $A$ induced by a relation $R$ is a binary relation $L^{k-\mathsf{up}}(R)$ on $Q$ s.t. if $q\ L^{k-\mathsf{up}}(R)\ r$, then the following holds: Let $\pi$ be a run over a tree $t \in \mathbb{T}(\Sigma)$ with $\pi(v) = q$ for some bottom leaf $v$ s.t. either $|v| = k$ or $0 < |v| < k$ and $\pi(\epsilon)$ is upward-deadlocked (i.e., no more upward transitions exist).

Then there must exist $v', v''$ such that $v = v'v''$ and $|v''| \geq 1$ and a run $\pi'$ over $t_{v'}$ s.t. the following holds: (1) $\pi'(v'') = r$; (2) $\pi(v')\ L^{k-\mathsf{up}}(R)\ \pi'(\epsilon)$; (3) $\pi(v'x) \in I \implies \pi'(x) \in I$ for all prefixes $x$ of $v''$; (4) if $v'xy \in dom(\pi)$ for some strict prefix $x$ of $v''$ and some $y \in \mathbb{N}$ where $xy$ is not a prefix of $v''$ then $\pi(v'xy)\ R\ \pi'(xy)$.

Since, for given $A$, $k \geq 1$ and $R$, lookahead upward simulations are closed under union, there exists a unique maximal one and we call it *the k-lookahead upward simulation induced by R on A*, denoted by $\sqsubseteq^{k\text{-}\mathsf{up}}(R)$. Since both $R$ and $\sqsubseteq^{k\text{-}\mathsf{up}}(R)$ are not necessarily transitive, we first compute the transitive closure $R^+$, and we then compute $\preceq^{k\text{-}\mathsf{up}}(R) := (\sqsubseteq^{k\text{-}\mathsf{up}}(R^+))^+$, which under-approximates the upward trace inclusion $\subseteq^{\mathsf{up}}(R^+)$.

For every fixed $k$, $k$-lookahead simulation is computable in polynomial time, both in the case of downward simulation and of upward simulation (when $R$ is a polynomial-time computable relation, e.g., $\sqsubseteq^{\mathsf{dw}}$, $Q \times Q$ or $id$). However, the complexity of computing lookahead simulation rises quickly in $k$: it is doubly exponential for downward lookahead simulation and single exponential in the upwards case (when taking $\sqsubseteq^{\mathsf{dw}}$, $Q \times Q$ or $id$ as the relation $R$), due to the downward branching of trees. A crucial trick makes it possible to practically compute it for nontrivial $k$: Spoiler's moves are built incrementally, and Duplicator need not respond to all of Spoiler's announced $k$ next steps, but only to a prefix of them, after which he may request fresh information [CM13]. Thus Duplicator just uses the minimal lookahead necessary to win the current step.

## 2.4 Transition Pruning

We define pruning relations on a TDTA $A = (\Sigma, Q, \delta, I)$. The intuition is that certain transitions may be deleted without changing the language, because *better* transitions remain. We perform this pruning (i.e., deletion) of transitions by comparing their endpoints over the same symbol $\sigma \in \Sigma$. Given two binary relations $R_{\mathsf{u}}$ and $R_{\mathsf{d}}$ on $Q$, we define the following relation to compare transitions:

$$P(R_{\mathsf{u}}, R_{\mathsf{d}}) = \{(\langle p, \sigma, r_1 \cdots r_n \rangle, \langle p', \sigma, r'_1 \cdots r'_n \rangle) \mid p \, R_{\mathsf{u}} \, p' \text{ and } \langle r_1 \cdots r_n \rangle \, \hat{R}_{\mathsf{d}} \, \langle r'_1 \cdots r'_n \rangle\},$$

where $\hat{R}_{\mathsf{d}}$ results from lifting $R_{\mathsf{d}} \subseteq Q \times Q$ to $\hat{R}_{\mathsf{d}} \subseteq Q^n \times Q^n$, as defined below. The function $P$ is monotone in the two arguments. If $t P t'$ then $t$ may be pruned because $t'$ is *better* than $t$.

We want $P(R_{\mathsf{u}}, R_{\mathsf{d}})$ to be a strict partial order (p.o.), i.e., irreflexive and transitive (and thus acyclic). There are two cases in which $P(R_{\mathsf{u}}, R_{\mathsf{d}})$ is guaranteed to be a strict p.o.:

1. $R_{\mathsf{u}}$ is some strict p.o. $<_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the standard lifting $\hat{\leq}_{\mathsf{d}}$ of some p.o. $\leq_{\mathsf{d}}$ to tuples. I.e., $\langle r_1 \cdots r_n \rangle \hat{\leq}_{\mathsf{d}} \langle r'_1 \cdots r'_n \rangle$ iff $\forall_{1 \leq i \leq n} . r_i \leq_{\mathsf{d}} r'_i$. The transitions in each pair of $P(<_{\mathsf{u}}, \leq_{\mathsf{d}})$ depart from different states and therefore are necessarily different.

2. $R_{\mathsf{u}}$ is some p.o. $\leq_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the lifting $\hat{<}_{\mathsf{d}}$ of some strict p.o. $<_{\mathsf{d}}$ to tuples. In this case the transitions in each pair of $P(\leq_{\mathsf{u}}, <_{\mathsf{d}})$ may have the same origin but must go to different tuples of states. Since for two tuples $\langle r_1 \cdots r_n \rangle$ and $\langle r'_1 \cdots r'_n \rangle$ to be different it suffices that $r_i \neq r'_i$ for some $1 \leq i \leq n$, we define $\hat{<}_{\mathsf{d}}$ as a binary relation such that $\langle r_1 \cdots r_n \rangle \hat{<}_{\mathsf{d}} \langle r'_1 \cdots r'_n \rangle$ iff $\forall_{1 \leq i \leq n} . r_i \leq_{\mathsf{d}} r'_i$, and $\exists_{1 \leq i \leq n} . r_i <_{\mathsf{d}} r'_i$.

Let $A = (\Sigma, Q, \delta, I)$ be a TDTA and let $P \subseteq \delta \times \delta$ be a strict partial order. The pruned automaton is defined as $\mathit{Prune}(A, P) = (\Sigma, Q, \delta', I)$ where $\delta' = \{(p, \sigma, r) \in \delta \mid \nexists (p', \sigma, r') \in \delta . (p, \sigma, r) P (p', \sigma, r')\}$. Note that the pruned automaton $\mathit{Prune}(A, P)$ is unique. The transitions are removed without requiring the re-computation of the relation $P$, which could be expensive. Since removing transitions cannot introduce new trees in the language, $L(\mathit{Prune}(A, P)) \subseteq L(A)$. If the reverse inclusion holds too (so that the language is preserved), we say that $P$ is *good for pruning* (GFP), i.e., $P$ is GFP iff $L(\mathit{Prune}(A, P)) = L(A)$.

We now provide a complete picture of which combinations of simulation and trace inclusion relations are GFP. Recall that simulations are denoted by square symbols $\sqsubseteq$

while trace inclusions are denoted by round symbols $\subseteq$. For every partial order $R$, the corresponding strict p.o. is defined as $R \backslash R^{-1}$.

Not all notions of relations that compare the behavior of states (simulation, trace inclusion, etc) are GFP. For instance, $P(\subset^{\mathsf{bw}}, \subset^{\mathsf{di}})$ is not GFP even for word automata (see Figure 2(a) in [CM13] for a counterexample). As mentioned before, words correspond to linear trees. Thus $P(\subset^{\mathsf{up}}(R), \subset^{\mathsf{dw}})$ is not GFP for tree automata, regardless of the relation $R$ (which can be any relation, since in upward simulations the inducing relation is only used when there is a node branching into more than one child).

Figure 2.3 presents several more counterexamples. For word automata, $P(\subset^{\mathsf{bw}}, \sqsubseteq^{\mathsf{di}})$ and $P(\sqsubseteq^{\mathsf{bw}}, \subset^{\mathsf{di}})$ are not GFP (Figure 2.3b and 2.3c), even though $P(\subseteq^{\mathsf{bw}}, \sqsubset^{\mathsf{di}})$ and $P(\sqsubset^{\mathsf{bw}}, \subseteq^{\mathsf{di}})$ are (cf. [CM13]). Thus $P(\subset^{\mathsf{up}}(R), \sqsubseteq^{\mathsf{dw}})$ and $P(\sqsubseteq^{\mathsf{up}}(R), \subset^{\mathsf{dw}})$ are not GFP for tree automata, regardless of $R$. For tree automata, $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}), id)$ and $P(\sqsubset^{\mathsf{up}}(\subset^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$ are not GFP (Figure 2.3a and 2.3d).

Moreover, a complex counterexample (see Figure 2.4) is needed to show that $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}), \subset^{\mathsf{dw}})$ is not GFP.

The following theorems and corollaries provide several relations which are GFP. Note that the GFP property is downward closed, i.e., if $R \subseteq R'$ and $R'$ is GFP then $R$ too is GFP (or if $R$ is not GFP then $R'$ too is not).

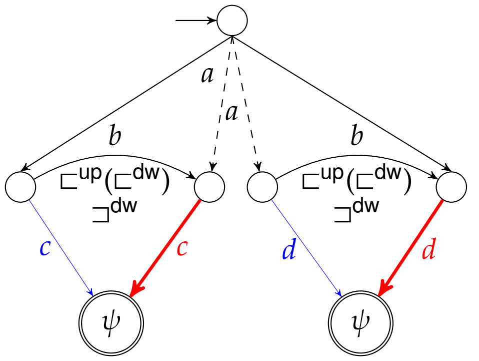
**Theorem 2.4.1.** *For every strict partial order $R \subset \subseteq^{\mathsf{dw}}$, it holds that $P(id, R)$ is GFP.*

*Proof.* Let $A' = Prune(A, P(id, R))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an accepting $t$-run $\pi$ in $A$. We show that there exists an accepting $t$-run $\pi'$ in $A'$.

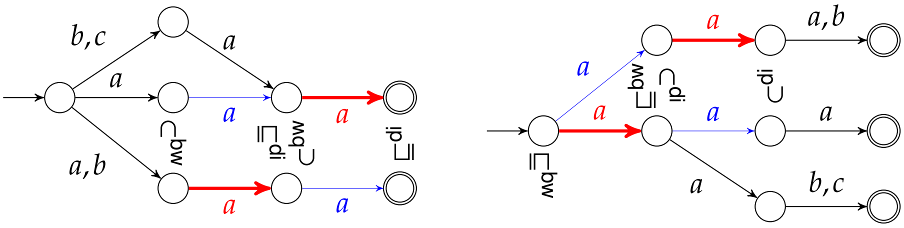
We will call an accepting $t$-run $\tilde{\pi}$ in $A$ *i-good* if its first $i$ levels use only transitions of $A'$. Formally, for every node $v \in dom(t)$ with $|v| < i$, $\langle \tilde{\pi}(v), t(v), \tilde{\pi}(v1) \ldots \tilde{\pi}(v\#(t(v))) \rangle$ is a transition of $A'$. By induction on $i$, we will show that there exists an $i$-good accepting run on $t$ for every $i \leq h(t)$. In the base case $i = 0$, the claim is trivially true since every accepting $t$-run of $A$, and particularly $\pi$, is 0-good.

For the induction step, let us assume that the claim holds for some $i$. Since $A$ is finite, for every transition *trans* there are only finitely many $A$-transitions *trans′* such that *trans* $P(id, R)$ *trans′*. And since $P(id, R)$ is transitive and irreflexive, for each transition *trans* in $A$ we have that either **1)** *trans* is maximal w.r.t. $P(id, R)$, or **2)** there exists a $P(id, R)$-larger transition *trans′* which is maximal w.r.t. $P(id, R)$. Thus for every state $p$ and every symbol $\sigma$, there exists a transition by $\sigma$ departing from $p$ which is still in $A'$.

Therefore, for every $i$-good accepting run $\pi^i$ on $t$, one easily obtains an accepting run $\pi^{i+1}$ which is $(i+1)$-good. In the first $i$ levels of $t$, $\pi^{i+1}$ is identical to $\pi^i$. In the $(i+1)$-th level of $t$, we have that for any transition $trans = \langle \pi^i(v), t(v), \pi^i(v1) \ldots \pi^i(v\#(t(v))) \rangle$,

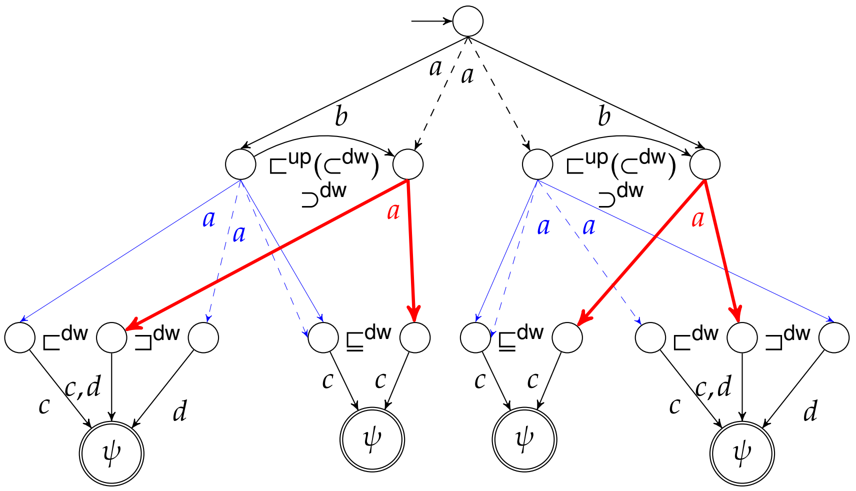

a
a
b
b
$\sqsubset^{\text{up}}(\sqsubset^{\text{dw}})$
$\sqsupset^{\text{dw}}$
$\sqsubset^{\text{up}}(\sqsubset^{\text{dw}})$
$\sqsupset^{\text{dw}}$
c
c
d
d
ψ
ψ

(a) $P(\sqsubset^{\text{up}}(\sqsubset^{\text{dw}}), id)$ is not GFP: if we remove the blue transitions, the automaton no longer accepts the tree $a(c,d)$. We are considering $\Sigma_0 = \{c,d\}$, $\Sigma_1 = \{b\}$ and $\Sigma_2 = \{a\}$.



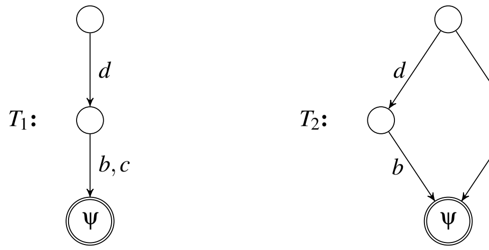
(b) $P(\subset^{\text{bw}}, \sqsubseteq^{\text{di}})$ is not GFP for words: if we remove the blue transitions, the automaton no longer accepts the word *aaa*.

(c) $P(\sqsubseteq^{\text{bw}}, \subset^{\text{di}})$ is not GFP for words: if we remove the blue transitions, the automaton no longer accepts the word *aaa*.

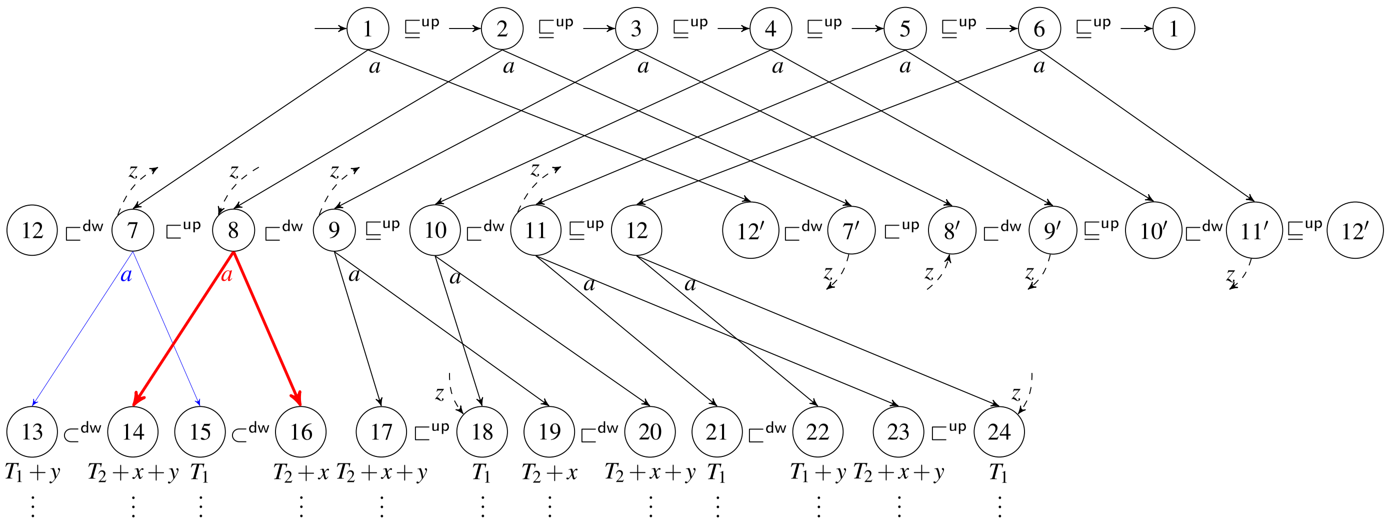


(d) $P(\sqsubset^{\text{up}}(\subset^{\text{dw}}), \sqsubset^{\text{dw}})$ is not GFP: if we remove the blue transitions, the tree $a(a(c,c),a(c,c))$ is no longer accepted. We are considering $\Sigma_0 = \{c,d\}$, $\Sigma_1 = \{b\}$ and $\Sigma_2 = \{a\}$.

Figure 2.3: GFP counterexamples. A transition is drawn in dashed when a different transition by the same symbol departing from the same state already exists. We draw a transition in thick red when it is better than another transition (drawn in thin blue).



(a) Consider the macros $T_1$ and $T_2$, which are used in $b$) to introduce new transitions and states in the third level of the automaton. Note that each of the intermediate states in $T_2$ is smaller w.r.t $\subset^{\mathsf{dw}}$ than the intermediate state in $T_1$. Thus, when the first state in $T_2$ is smaller w.r.t $\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}})$ than the first state in $T_1$, each of the $d$-transitions in $T_2$ is a blue one and the $d$-transition in $T_1$ is red. Let us also consider the symbols $x$ and $y$ of rank 1. We can extend a macro $T$ by adding transitions from the first state to some new state by $x$, by $y$ or by both and we denote these by $T_x$, $T_y$ or $T_{x+y}$, resp.


1 $\sqsubset^{\mathsf{up}}$ 2 $\sqsubset^{\mathsf{up}}$ 3 $\sqsubset^{\mathsf{up}}$ 4 $\sqsubset^{\mathsf{up}}$ 5 $\sqsubset^{\mathsf{up}}$ 6 $\sqsubset^{\mathsf{up}}$ 1
$a$
$z$
12 $\sqsubset^{\mathsf{dw}}$ 7 $\sqsubset^{\mathsf{up}}$ 8 $\sqsubset^{\mathsf{dw}}$ 9 $\sqsubseteq^{\mathsf{up}}$ 10 $\sqsubset^{\mathsf{dw}}$ 11 $\sqsubseteq^{\mathsf{up}}$ 12
12′ $\sqsubset^{\mathsf{dw}}$ 7′ $\sqsubset^{\mathsf{up}}$ 8′ $\sqsubset^{\mathsf{dw}}$ 9′ $\sqsubseteq^{\mathsf{up}}$ 10′ $\sqsubset^{\mathsf{dw}}$ 11′ $\sqsubseteq^{\mathsf{up}}$ 12′
13 $\subset^{\mathsf{dw}}$ 14
15 $\subset^{\mathsf{dw}}$ 16
17 $\sqsubset^{\mathsf{up}}$ 18
19 $\sqsubset^{\mathsf{dw}}$ 20
21 $\sqsubset^{\mathsf{dw}}$ 22
23 $\sqsubset^{\mathsf{up}}$ 24
$T_1+y$
$T_2+x+y$
$T_1$
$T_2+x$
$T_2+x+y$
$T_1$
$T_2+x$
$T_2+x+y$
$T_1$
$T_1+y$
$T_2+x+y$
$T_1$

(b) We consider $\Sigma_0 = \{b,c\}$, $\Sigma_1 = \{d,x,y,z\}$ and $\Sigma_2 = \{a\}$. The dashed arrows represent transitions by $z$ from/to some new state. Each of the six initial states has an $a$-transition to one of the states from 7 to 12 on the left and one of the states from 7′ to 12′ on the right. Any state $n'$ on the right side of the automaton does exactly the same downwardly as the state $n$ on the left side, and thus needs not be expanded in the figure. We abbreviate $\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}})$ to simply $\sqsubset^{\mathsf{up}}$ and $\sqsubseteq^{\mathsf{up}}(\sqsubset^{\mathsf{dw}})$ to $\sqsubseteq^{\mathsf{up}}$.

Figure 2.4: $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}),\subset^{\mathsf{dw}})$ is not GFP since the automaton presented in $b$) cannot read the tree $a(a(d(b),d(b)),a(d(c),d(c)))$ (or any tree with just $a$'s in the first two levels) without using a blue transition: a run starting in state $1$ encounters a blue transition from $7$, as illustrated in the figure; and since $7'$ and $8'$ do the same downwardly as $7$ and $8$, respectively, and since $7' \sqsubset^{\mathsf{up}} 8'$, we have that there is a blue transition from $7'$ as well, and so $2$ cannot be used either; since $17 \sqsubset^{\mathsf{up}} 18$ we have that, as explained in $a$), there is a blue transition departing from $17$, thus a run starting at $3$ too cannot be used; and since $9$ is downwardly imitated by $9'$, a run using this state finds a blue transition as well, and so $4$ is not safe; since $23 \sqsubset^{\mathsf{up}} 24$, a blue transition from $23$ exists and so $5$ cannot be used; finally, since $11$ is imitated by $11'$, we have that a run using this state encounters a blue transition as well, and so $6$ too is not safe.

for $|v| = i$, either *trans* is $P(id,R)$-maximal, and so we take $\pi^{i+1}(vj) := \pi^{i}(vj)$ for all $1 \leq j \leq \#(t(v))$, or there exists a $P(id,R)$-larger transition $trans' = \langle \pi^{i}(v), t(v), q_1 \ldots q_{\#(t(v))} \rangle$ that is $P(id,R)$-maximal. By the definition of $P(id,R)$, we have that $\langle \pi^{i}(v1) \ldots \pi^{i}(v\#(t(v))) \rangle \, \hat{R} \, \langle q_1 \ldots q_{\#(t(v))} \rangle$, and we take $\pi^{i+1}(vj) := q_j$ for all $1 \leq j \leq \#(t(v)))$. Since $R \subset \subseteq^{\mathsf{dw}}$, we have that for every $1 \leq j \leq \#t(v)$, there is a run $\pi_j$ of $A$ such that $t_{v_j} \xRightarrow{\pi_j} q_j$. The run $\pi^{i+1}$ on $t$ can hence be completed from every $q_j$ by the run $\pi_j$, which concludes the proof of the induction step.

Since a $h(t)$-good run is a run in $A'$, the theorem is proven. □

**Corollary 2.4.1.** *It follows from Theorem 2.4.1 and the fact that GFP is downward closed that* $P(id, \subset^{\mathsf{dw}})$ *and* $P(id, \sqsubseteq^{k\text{-}\mathsf{dw}})$*, for any* $k \geq 1$*, are GFP.*

**Theorem 2.4.2.** *For every strict partial order* $R \subset \subseteq^{\mathsf{up}}(id)$*, it holds that* $P(R, id)$ *is GFP.*

*Proof.* Let $A' = Prune(A, P(R, id))$. We will show that for every accepting run $\pi$ of $A$ on a tree $t$, there exists an accepting run $\hat{\pi}$ of $A'$ on $t$.

Let us first define some auxiliary notation. For an accepting run $\pi$ of $A$ on a tree $t$, $bad(\pi)$ is the smallest subtree of $t$ which contains *all nodes* $v$ of $t$ where $\pi$ uses a transition of $A - A'$, i.e., a transition which is not $P(R,id)$-maximal (where by $\pi$ using a transition at node $v$ we mean that the symbol of the transition is $t(v)$, $\pi(v)$ is the left-hand side of the transition, and the vector of $\pi$-values of children of $v$ is its right-hand side). We will use the following auxiliary claim.

(C) For every accepting run $\pi$ of $A$ on a tree $t$ with $|bad(\pi)| > 1$, there is an accepting run $\pi'$ of $A$ on $t$ where $bad(\pi')$ is a proper subtree of $bad(\pi)$.

To prove (C), assume that $v$ is a leaf of $bad(\pi)$ labeled by a transition $\langle p, \sigma, r_1 \ldots r_n \rangle$. By the definition of $P(R,id)$ and by the minimality of $bad(\pi)$, there exists a $P(R,id)$-maximal transition $\tau = \langle p', \sigma, r_1 \ldots r_n \rangle$ where $p \subset^{\mathsf{up}}(id)\, p'$. Since $p \subset^{\mathsf{up}}(id)\, p'$, it follows from the definition of $\subset^{\mathsf{up}}(id)$ that there exists a run $\pi'$ of $A$ on $t$ that differs from $\pi$ only in labels of prefixes of $v$ (including $v$ itself) with $\pi'(v) = p'$. In other words, $bad(\pi')$ differs from $bad(\pi)$ only in that it does not contain a certain subtree rooted by some prefix node of $v$. This subtree contains at least $v$ itself, since $\pi'$ uses the $P(R,id)$-maximal transition $\tau$ to label $v$. The tree $bad(\pi')$ is hence a proper subtree of $bad(\pi)$, which concludes the proof of (C).

With (C) in hand, we are ready to prove the theorem. By finitely many applications of (C), starting from $\pi$, we obtain an accepting run $\hat{\pi}$ on $t$ where $bad(\hat{\pi})$ is empty (we only need finitely many applications since $bad(\pi)$ is a finite tree, and every application of (C) yields a run with a strictly smaller *bad* subtree). Thus $\hat{\pi}$ is using only $P(R,id)$-maximal transitions. Since $R$ and hence also $P(R,id)$ are strict p.o., $A' = Prune(A, P(R,id))$ contains all $P(R, id)$-maximal transitions of $A$, which means that $\hat{\pi}$ is an accepting run of $A'$ on $t$. ☐

**Corollary 2.4.2.** *By Theorem 2.4.2 and the fact that GFP is downward closed,* $P(\subset^{\mathsf{up}}(id), id)$ *and* $P(\sqsubset^{k\text{-}\mathsf{up}}(id), id)$*, for any* $k \geq 1$*, are GFP.*

**Definition 2.4.1.** *Given a tree automaton $A$, a binary relation $W$ on its states is called a* downup-relation *iff the following condition holds: If $p\ W\ q$ then for every tree $t \in \mathbb{T}(\Sigma)$ and accepting $t$-run $\pi$ from $p$ there exists an accepting $t$-run $\pi'$ from $q$ such that* $\forall_{v \in \mathbb{N}^*}\ \pi(v) \sqsubseteq^{\mathsf{up}}(W)\ \pi'(v)$.

**Lemma 2.4.1.** *Any relation $V$ satisfying* **1)** *$V$ is a downward simulation, and* **2)** $id \subseteq V \subseteq\ \sqsubseteq^{\mathsf{up}}(V)$ *is a downup-relation. In particular, id is a downup-relation, but* $\sqsubseteq^{\mathsf{dw}}$ *and* $\sqsubseteq^{\mathsf{up}}(id)$ *are not.*

**Theorem 2.4.3.** *For every downup-relation $W$, it holds that* $P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}})$ *is GFP.*

*Proof.* Let $A' = Prune(A, P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}}))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an accepting $t$-run $\hat{\pi}$ in $A$. We show that there is an accepting $t$-run $\hat{\pi}'$ in $A'$.

For each accepting $t$-run $\pi$ in $A$, let $level_i(\pi)$ be the tuple of states that $\pi$ visits at depth $i$ in the tree, read from left to right. Formally, let $\langle x_1, \ldots, x_k \rangle$ with $x_j \in \mathbb{N}^i$ be the set of all tree positions of depth $i$ s.t. $x_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = \langle \pi(x_1), \ldots, \pi(x_k) \rangle \in Q^k$. By lifting partial orders on $Q$ to partial orders on tuples, we can compare such tuples w.r.t. $\sqsubseteq^{\mathsf{up}}(W)$. We say that an accepting $t$-run $\pi$ is $i$-good iff it does not contain any transition from $A - A'$ from any position $v \in \mathbb{N}^*$ with $|v| < i$. I.e., no pruned transition is used in the first $i$ levels of the tree.

We now define a strict partial order $<_i$ on the set of accepting $t$-runs in $A$. Let $\pi <_i \pi'$ iff $\exists k \leq i.\ level_k(\pi) \sqsubset^{\mathsf{up}}(W)\ level_k(\pi')$ and $\forall l < k. level_l(\pi) \sqsubseteq^{\mathsf{up}}(W)\ level_l(\pi')$. Note that $<_i$ only depends on the first $i$ levels of the run. Given $A$, $t$ and $i$, there are only finitely many different such $i$-prefixes of accepting $t$-runs. By our assumption that $\hat{\pi}$ is an accepting $t$-run in $A$, the set of accepting $t$-runs in $A$ is non-empty. Thus, for any $i$, there must exist some accepting $t$-run $\pi$ in $A$ that is maximal w.r.t. $<_i$.

We now show that this $\pi$ is also $i$-good, by assuming the contrary and deriving a contradiction. Suppose that $\pi$ is not $i$-good. Then it must contain a transition $\langle p, \sigma, r_1 \cdots r_n \rangle$ from $A - A'$ used at the root of some subtree $t'$ of $t$ at some level $j < i$. Since $A' = Prune(A, P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}}))$, there must exist another transition $\langle p', \sigma, r_1' \cdots r_n' \rangle$ in $A'$ s.t. (1) $\langle r_1, \ldots, r_n \rangle \subseteq^{\mathsf{dw}} \langle r_1', \ldots, r_n' \rangle$ and (2) $p \sqsubset^{\mathsf{up}}(W)\ p'$.

First consider the implications of (2). Upward simulation propagates upward stepwise (though only in non-strict form after the first step). So $p'$ can imitate the upward path of $p$ to the root of $t$, maintaining $\sqsubseteq^{\mathsf{up}}(W)$ between the corresponding states. The states on side branches joining in along the upward path from $p$ can be matched by $W$-larger states in joining side branches along the upward path from $p'$. From Definition 2.4.1 we obtain that these $W$-larger states in $p'$s joining side branches can accept their subtrees of $t$ via computations that are everywhere $\sqsubseteq^{\mathsf{up}}(W)$ larger than corresponding states in computations from $p'$s joining side branches. So there must be an accepting run $\pi'$ on $t$ s.t. (3) $\pi'$ is at state $p'$ at the root of $t'$ and uses transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$ from $p'$, and (4) for all $v \in \mathbb{N}^*$ where $t(v) \notin t'$ we have $\pi(v) \sqsubseteq^{\mathsf{up}}(W)\ \pi'(v)$. Moreover, by conditions (1) and (3), $\pi'$ can be extended from $r'_1, \ldots, r'_n$ to accept also the subtree $t'$. Thus $\pi'$ is an accepting $t$-run in $A$. By conditions (2) and (4) we obtain that $\forall l \leq j.\ level_l(\pi) \sqsubseteq^{\mathsf{up}}(W)\ level_l(\pi')$. By (2) we get even $level_j(\pi) \sqsubset^{\mathsf{up}}(W)\ level_j(\pi')$ and thus $\pi <_j \pi'$. Since $j < i$ we also have $\pi <_i \pi'$ and thus $\pi$ was not maximal w.r.t. $<_i$. Contradiction. So we have shown that for every $t \in L(A)$ there exists an $i$-good accepting run for every finite $i$.

If $t \in L(A)$ then there exists an accepting $t$-run $\hat{\pi}$ in $A$. Then there exists an accepting $t$-run $\hat{\pi}'$ that is $i$-good, where $i$ is the height of $t$. Thus $\hat{\pi}'$ is a run in $A'$ and $t \in L(A')$. □

**Corollary 2.4.3.** *It follows from the fact that GFP is downward closed that, for any $V$ as defined in Lemma 2.4.1, $P(\sqsubset^{\mathsf{up}}(V), \subseteq^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \subset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \sqsubseteq^{k\text{-}\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \sqsubset^{k\text{-}\mathsf{dw}})$, for any $k \geq 1$, and $P(\sqsubset^{\mathsf{up}}(V), id)$ are GFP. In particular, $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(id), \subset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(id), \sqsubseteq^{k\text{-}\mathsf{dw}})$ and $P(\sqsubset^{\mathsf{up}}(id), \sqsubset^{k\text{-}\mathsf{dw}})$, for any $k \geq 1$, are GFP.*

**Theorem 2.4.4.** *$P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$ is GFP.*

*Proof.* Let $A' = Prune(A, P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an accepting $t$-run $\hat{\pi}$ in $A$. We show that there is an accepting $t$-run $\hat{\pi}'$ in $A'$.

For each accepting $t$-run $\pi$ in $A$, let $level_i(\pi)$ be the tuple of states that $\pi$ visits at depth $i$ in the tree, read from left to right. Formally, let $\langle x_1, \ldots, x_k \rangle$ with $x_j \in \mathbb{N}^i$ be the set of all tree positions of depth $i$ s.t. $x_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = \langle \pi(x_1), \ldots, \pi(x_k) \rangle \in Q^k$. By lifting partial orders on $Q$ to partial orders on tuples we can compare such tuples w.r.t. $\sqsubseteq^{\mathsf{dw}}$. We say that an accepting $t$-run $\pi$ is $i$-good if it does not contain any transition from $A - A'$ from any position $v \in \mathbb{N}^*$ with $|v| < i$. I.e., no pruned transitions are used in the first $i$ levels of the tree.

We now show, by induction on $i$, the following property (C): For every $i$ and every accepting $t$-run $\pi$ in $A$ there exists an $i$-good accepting $t$-run $\pi'$ in $A$ s.t. $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$.

The base case is $i = 0$. Every accepting $t$-run $\pi$ in $A$ is trivially 0-good itself and thus satisfies (C).

For the induction step, let $S$ be the set of all $(i-1)$-good accepting $t$-runs $\pi'$ in $A$ s.t. $level_{i-1}(\pi) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi')$. Since $\pi$ is an accepting $t$-run, by induction hypothesis,

$S$ is non-empty. Let $S' \subseteq S$ be the subset of $S$ containing exactly those runs $\pi' \in S$ that additionally satisfy $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$. From $level_{i-1}(\pi) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi')$ and the fact that $\sqsubseteq^{\mathsf{dw}}$ is preserved downward-stepwise, we obtain that $S'$ is non-empty. Now we can select some $\pi' \in S'$ s.t. $level_i(\pi')$ is maximal, w.r.t. $\sqsubseteq^{\mathsf{dw}}$, relative to the other runs in $S'$. We claim that $\pi'$ is $i$-good and $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$. The second part of this claim holds because $\pi' \in S'$.

We show that $\pi'$ is $i$-good by contraposition. Suppose that $\pi'$ is not $i$-good. Then it must contain a transition $\langle p, \sigma, r_1 \cdots r_n \rangle$ from $A - A'$. Since $\pi'$ is $(i-1)$-good, this transition must start at depth $(i-1)$ in the tree. Since $A' = Prune(A, P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$, there must exist another transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$ in $A'$ s.t. $p \subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})\, p'$ and $\langle r_1, \ldots, r_n \rangle \sqsubset^{\mathsf{dw}} \langle r'_1, \ldots, r'_n \rangle$. From the definition of $\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$ we obtain that there exists another accepting $t$-run $\pi_1$ in $A$ (that uses the transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$) s.t. $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_1)$. The run $\pi_1$ is not necessarily $i$-good or $(i-1)$-good. However, by induction hypothesis, there exists some accepting $t$-run $\pi_2$ in $A$ that is $(i-1)$-good and satisfies $level_{i-1}(\pi_1) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi_2)$. Since $\sqsubseteq^{\mathsf{dw}}$ is preserved stepwise, there also exists an accepting $t$-run $\pi_3$ in $A$ (that coincides with $\pi_2$ up-to depth $(i-1)$), which is $(i-1)$-good and satisfies $level_i(\pi_1) \sqsubseteq^{\mathsf{dw}} level_i(\pi_3)$. In particular, $\pi_3 \in S'$.

From $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_1)$ and $level_i(\pi_1) \sqsubseteq^{\mathsf{dw}} level_i(\pi_3)$ we obtain $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_3)$. This contradicts our condition above that $\pi'$ must be $level_i$ maximal w.r.t. $\sqsubseteq^{\mathsf{dw}}$ in $S'$. This concludes the induction step and the proof of property (C).

If $t \in L(A)$ then there exists an accepting $t$-run $\hat{\pi}$ in $A$. By property (C), there exists an accepting $t$-run $\hat{\pi}'$ that is $i$-good, where $i$ is the height of $t$. Therefore $\hat{\pi}'$ does not use any transition from $A - A'$ and is thus also a run in $A'$. So we obtain $t \in L(A')$. $\square$

**Corollary 2.4.4.** *It follows from Theorem 2.4.4 and the fact that GFP is downward closed that* $P(\subset^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\sqsubseteq^{k\text{-}\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\sqsubset^{k\text{-}\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\subseteq^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$, $P(\subset^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$, $P(\sqsubseteq^{k\text{-}\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$, *for any* $k \geq 1$, *and* $P(id, \sqsubset^{\mathsf{dw}})$ *are GFP.*

The table in Figure 2.5 summarizes all our results, providing a complete picture of which combinations of upward and downward relations are or are not good for pruning. Note that negative results propagate to larger relations and positive results propagate to smaller relations (i.e., GFP is downward closed).

| | | $R_{\mathsf{d}}$ | | | | |
|---|---|---|---|---|---|---|
| $R_{\mathsf{u}} \backslash R_i$ | | $id$ | $\sqsubset^{\mathsf{dw}}$ | $\sqsubseteq^{\mathsf{dw}}$ | $\subset^{\mathsf{dw}}$ | $\subseteq^{\mathsf{dw}}$ |
| $id$ | $id$ | – | ✓ | – | ✓ | – |
| $\sqsubset^{\mathsf{up}}$ | $id$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | $\sqsubset^{\mathsf{dw}}$ | × | ✓ | × | × | × |
| | $\sqsubseteq^{\mathsf{dw}}$ | × | ✓ | × | × | × |
| | downup-rel. | ✓ | ✓ | ✓ | ✓ | ✓ |
| | $\subset^{\mathsf{dw}}$ | × | × | × | × | × |
| | $\subseteq^{\mathsf{dw}}$ | × | × | × | × | × |
| $\sqsubseteq^{\mathsf{up}}$ | $id$ | – | ✓ | – | × | – |
| | $\sqsubset^{\mathsf{dw}}$ | – | ✓ | – | × | – |
| | $\sqsubseteq^{\mathsf{dw}}$ | – | ✓ | – | × | – |
| | $\subset^{\mathsf{dw}}$ | – | × | – | × | – |
| | $\subseteq^{\mathsf{dw}}$ | – | × | – | × | – |
| $\subset^{\mathsf{up}}$ | $id$ | ✓ | ✓ | × | × | × |
| | $\sqsubset^{\mathsf{dw}}$ | × | ✓ | × | × | × |
| | $\sqsubseteq^{\mathsf{dw}}$ | × | ✓ | × | × | × |
| | $\subset^{\mathsf{dw}}$ | × | × | × | × | × |
| | $\subseteq^{\mathsf{dw}}$ | × | × | × | × | × |
| $\subseteq^{\mathsf{up}}$ | $id$ | – | ✓ | – | × | – |
| | $\sqsubset^{\mathsf{dw}}$ | – | ✓ | – | × | – |
| | $\sqsubseteq^{\mathsf{dw}}$ | – | ✓ | – | × | – |
| | $\subset^{\mathsf{dw}}$ | – | × | – | × | – |
| | $\subseteq^{\mathsf{dw}}$ | – | × | – | × | – |

Figure 2.5: GFP relations $P(R_{\mathsf{u}}(R_i), R_{\mathsf{d}})$ for tree automata. We use ✓ to mark the coarsest GFP relations found, and ✓ to mark the (finer) relations which under-approximate them. Relations which are not GFP are marked with × and – is used to mark relations where the test does not apply due to them being reflexive (and therefore not asymmetric).

## 2.5 State Quotienting

A classic method for reducing the size of automata is state quotienting. Given a suitable equivalence relation on the set of states, each equivalence class is collapsed into just one state. From a preorder $\sqsubseteq$ one obtains an equivalence relation $\equiv := \sqsubseteq \cap \sqsupseteq$. We now define quotienting w.r.t. $\equiv$. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA and let $\sqsubseteq$ be a preorder on $Q$. Given $q \in Q$, we denote by $[q]$ its equivalence class w.r.t $\equiv$. For $P \subseteq Q$, $[P]$ denotes the set of equivalence classes $[P] = \{[p] \mid p \in P\}$. We define the quotient automaton w.r.t. $\equiv$ as $A/\equiv := (\Sigma, [Q], \delta_{A/\equiv}, [I])$, where $\delta_{A/\equiv} = \{\langle [q], \sigma, [q_1] \ldots [q_n] \rangle \mid \langle q, \sigma, q_1 \ldots q_n \rangle \in \delta_A\}$. It is trivial that $L(A) \subseteq L(A/\equiv)$ for any $\equiv$. If the reverse inclusion also holds, i.e., if $L(A) = L(A/\equiv)$, we say that $\equiv$ is *good for quotienting* (GFQ).

It was shown that $\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}$ and $\sqsubseteq^{\mathsf{up}}(id) \cap \sqsupseteq^{\mathsf{up}}(id)$ are GFQ [Hol11]. Here we generalize this result from simulation to trace equivalence. Let $\equiv^{\mathsf{dw}} := \subseteq^{\mathsf{dw}} \cap \supseteq^{\mathsf{dw}}$ and $\equiv^{\mathsf{up}}(R) := \subseteq^{\mathsf{up}}(R) \cap \supseteq^{\mathsf{up}}(R)$. The theorems below show that $\equiv^{\mathsf{dw}}$ and $\equiv^{\mathsf{up}}(id)$ are GFQ, and the corollaries propagate the results to their corresponding lookahead simulations. Note that the GFQ property is downward closed, i.e., if $R \subseteq R'$ and $R'$ is GFQ then $R$ too is GFQ (or if $R$ is not GFQ then $R'$ too is not).

**Theorem 2.5.1.** *$\equiv^{\mathsf{dw}}$ is GFQ.*

*Proof.* Let $A' := A/\equiv^{\mathsf{dw}}$. It is trivial that $L(A) \subseteq L(A')$. For the reverse inclusion, we will show by induction on the height $i$ of $t$, that for any tree $t$, if $t \in D_{A'}([q])$ for some $[q] \in [Q]$, then $t \in D_A(q)$. This guarantees $L(A') \subseteq L(A)$ since if $[q] \in [I]$ then there is some $q' \in I$ such that $q' \equiv^{\mathsf{dw}} q$ and thus, by the definition of $\equiv^{\mathsf{dw}}$, $D_A(q') = D_A(q)$.

In the base case $i = 1$, $t$ is a leaf-node $\sigma$, for some $\sigma \in \Sigma$. By hypothesis, $t \in L(A')$. So there exists $[q] \in [I]$ such that $t \Longrightarrow_{A'} [q]$. So $\langle [q], \sigma, [\psi] \rangle \in \delta_{A'}$. Since $[\psi] = \{\psi\}$, there exists $q' \in [q]$ such that $\langle q', \sigma, \psi \rangle \in \delta_A$. Since $[q] \in [I]$ there is some $q'' \in I$ with $q'' \equiv^{\mathsf{dw}} q \equiv^{\mathsf{dw}} q'$. We have $t \in D_A(q') = D_A(q'') \subseteq L(A)$.

Let us now consider $i > 1$. Let $\sigma$ be the root of the tree $t$, and let $t_1, t_2, \ldots, t_n$, where $n = \#(\sigma)$, denote each of the immediate subtrees of $t$. As we assume $t \in L(A')$, there exists $[q] \in [I]$ such that $\langle [q], \sigma, [q_1][q_2] \ldots [q_n] \rangle \in \delta_{A'}$, for some $[q_1], [q_2], \ldots, [q_n] \in [Q]$, such that $t_i \in D_{A'}([q_i])$ for every $i$. By the definition of $\delta_{A'}$, there are $q'_1 \in [q_1]$, $q'_2 \in [q_2]$, …, $q'_n \in [q_n]$ and $q' \in [q]$, such that $\langle q', \sigma, q'_1 q'_2 \ldots q'_n \rangle \in \delta_A$. By induction hypothesis, we obtain $t_i \in D_A(q_i)$ for every $i$. Since $q_i \equiv^{\mathsf{dw}} q'_i$, it follows that $t_i \in D_A(q'_i)$ for every $i$ and thus $t \in D_A(q')$. By $q \equiv^{\mathsf{dw}} q'$, we conclude that $t \in D_A(q)$. □

**Corollary 2.5.1.** *It follows from Theorem 2.5.1 and the fact that GFQ is downward closed that $\sqsubseteq^{k\text{-}\mathsf{dw}} \cap \sqsupseteq^{k\text{-}\mathsf{dw}}$ is GFQ for any $k \geq 1$.*

**Theorem 2.5.2.** *$\equiv^{\mathsf{up}}(id)$ is GFQ.*

*Proof.* Let $\equiv := \equiv^{\mathsf{up}}(id)$ and $A' := A/\equiv$. It is trivial that $L(A) \subseteq L(A')$. For the reverse inclusion, we will show, by induction on the height $h$ of $t$, that for any tree $t$, if $t \in D_{A'}([q])$ for some $[q] \in [Q]$, then $t \in D_A(q')$ for some $q' \in [q]$. This guarantees $L(A') \subseteq L(A)$ since if $[q] \in [I]$ then, given that $\equiv$ preserves the initial states, $q' \in I$.

In the base case $h = 1$, the tree is a leaf-node $\sigma$, for some $\sigma \in \Sigma$. By hypothesis, $t \in L(A')$. So there exists a $[q] \in [I]$ such that $t \Longrightarrow_{A'} [q]$, and so $\langle [q], \sigma, [\psi] \rangle \in \delta_{A'}$. By the definition of $\delta_{A'}$ and since $[\psi] = \{\psi\}$ ($\equiv$ preserves acceptance), we have that there exists $q' \in [q]$ such that $\langle q', \sigma, \psi \rangle \in \delta_A$, and hence $t \Longrightarrow_A q'$.

Let us now consider $h > 1$. As we assume $t \in D_{A'}([q])$, there must exist a transition $\langle [q], \sigma, [q_1] \ldots [q_n] \rangle \in \delta_{A'}$, for $n = \#\sigma$ and some $[q_1], \ldots, [q_n] \in [Q]$ such that $t_i \in D_{A'}([q_i])$ for every $i : 0 \leq i \leq n$, where the $t_i$s are the subtrees of $t$. We define the following auxiliary notion: a transition $\langle r, \sigma, r_1 \ldots r_n \rangle$ of $A$ satisfying $r \in [q]$ and $\forall_{1 \leq k \leq n} . r_k \in [q_k]$ is said to be $j$-good iff $\forall_{1 \leq k \leq j} . t_k \in D_A(r_k)$. We will use induction on $j$ to show that there is a $j$-good transition for any $j$, which implies that there is some state $\hat{r} \in [q]$ such that $t \in D_A(\hat{r})$.

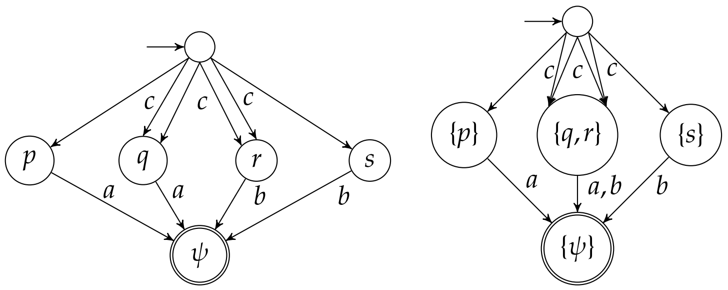
The base case is $j = 0$. By the definition of $\delta_{A'}$ and the fact that $\langle [q], \sigma, [q_1] \ldots [q_n] \rangle \in \delta_{A'}$, there exist $q'_1 \in [q_1], \ldots, q'_n \in [q_n]$ and $q' \in [q]$ such that $\langle q', \sigma, q'_1 \ldots q'_n \rangle \in \delta_A$. This transition is trivially 0-good.

To show the induction step, assume a transition $trans = \langle r, \sigma, r_1 \ldots r_n \rangle$ that is $j$-good for $j \geq 0$, i.e., each $r_i$ is in $[q_i]$, $r \in [q]$, and $\forall_{1 \leq i \leq j} . t_i \in D_A(r_i)$. By the hypothesis of the outer induction on $h$, there is $r'_{j+1} \in [r_{j+1}]$ such that $t_{j+1} \in D_A(r'_{j+1})$. Notice that $r_{j+1} \equiv r'_{j+1}$. Since *trans* is a transition of $A$, there is a run $\pi'$ of $A$ on a tree $t'$ of the height 1 with the root symbol $\sigma$, and where $\pi'(1) = r_1, \ldots, \pi'(n) = r_n$, and $\pi'(\epsilon) = r$. Since $r_{j+1} \equiv r'_{j+1}$, then, by the definition of $\equiv$, there is another $t'$-run $\pi''$ such that $r' = \pi''(\epsilon) \in [q]$, $\pi''(j+1) = r'_{j+1}$, and $\forall_{i \neq j+1} . \pi''(i) = \pi'(i) = r_i$. This run uses the transition $trans' = \langle r', \sigma, r_1 \ldots r_j r'_{j+1} r_{j+2} \ldots r_n \rangle$ in $A$. Since *trans* is $j$-good and $t_{j+1} \in D_A(r'_{j+1})$, we have that $trans'$ is $(j+1)$-good. This concludes the inner induction on $j$, showing that there exists an $n$-good transition. Hence $t \in D_A(\hat{r})$ for some $\hat{r} \in [q]$, which proves the outer induction on the height $h$ of the tree, concluding the whole proof. □

**Corollary 2.5.2.** *By Theorem 2.5.2 we have that* $\sqsubseteq^{k\text{-}\mathsf{up}}(id) \cap \sqsupseteq^{k\text{-}\mathsf{up}}(id)$ *is GFQ for any* $k \geq 1$.

Figure 2.6 presents a counterexample showing that $\equiv := \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) \cap \sqsupseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$ is not GFQ. This is an adaptation from the Example 5 in [Hol11], where the inducing relation is referred to as the *downward bisimulation equivalence* and the automata are seen bottom-up.

The table in Figure 2.7 summarizes our results on relations which are or are not good for quotienting. Note that, since GFQ is downward closed, negative results propagate to larger relations.



(a) Automaton $A$. (b) Automaton $A/\equiv$.

Figure 2.6: $\equiv := \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) \cap \sqsupseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$ is not GFQ. We are considering $\Sigma_0 = \{a, b\}$ and $\Sigma_2 = \{c\}$. Computing all the necessary relations to quotient $A$ w.r.t. $\equiv$, we obtain $\sqsubseteq^{\mathsf{dw}} = \{(p, q), (r, s)\} = \sqsupseteq^{\mathsf{dw}}$ and $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) = \{(q, r), (r, q)\}$. Thus $\equiv = \{(q, r), (r, q)\}$. Computing $A/\equiv$, we verify that $c(b, a)$ is now accepted by the automaton $A/\equiv$, while it was not in the language of $A$.

| $R$ | | |
|---|---|---|
| $\sqsubseteq^{\mathsf{dw}}$ | | ✓ |
| $\subseteq^{\mathsf{dw}}$ | | ✓ |
| $\sqsubseteq^{\mathsf{up}}$ | $id$ | ✓ |
| | $\sqsubset^{\mathsf{dw}}$ | – |
| | $\sqsubseteq^{\mathsf{dw}}$ | × |
| | $\subset^{\mathsf{dw}}$ | – |
| | $\subseteq^{\mathsf{dw}}$ | × |
| $\subseteq^{\mathsf{up}}$ | $id$ | ✓ |
| | $\sqsubset^{\mathsf{dw}}$ | – |
| | $\sqsubseteq^{\mathsf{dw}}$ | × |
| | $\subset^{\mathsf{dw}}$ | – |
| | $\subseteq^{\mathsf{dw}}$ | × |

Figure 2.7: GFQ relations $R$ for tree automata. The largest GFQ relations found are marked with ✓, and ✓ marks their corresponding under-approximations. Relations which are not GFQ are marked with ×. The relations marked with – are not even reflexive in general, unless all transitions are linear (in which case we have a word automaton and the relations $\sqsubseteq^{\mathsf{up}}(\sqsubset^{\mathsf{dw}})$ and $\sqsubseteq^{\mathsf{up}}(\subset^{\mathsf{dw}})$ coincide with $\sqsubseteq^{\mathsf{up}}(id)$ and $\subseteq^{\mathsf{up}}(\sqsubset^{\mathsf{dw}})$ and $\subseteq^{\mathsf{up}}(\subset^{\mathsf{dw}})$ coincide with $\subseteq^{\mathsf{up}}(id)$).

## 2.6 Reducing Automata: an algorithm

Our tree automata reduction algorithm (tool available [Alm16a]) combines the transition pruning techniques we saw in Section 2.4 with the state quotienting techniques described in Section 2.5. Trace inclusions are under-approximated by lookahead simulations, as seen in Section 2.3, where higher lookaheads are harder to compute but yield better (i.e., larger) approximations. Let us use the parameters $x, y \geq 1$ to describe the lookahead used for computing downward and upward lookahead simulations, respectively. Downward lookahead simulation is harder to compute than upward lookahead simulation, since the number of possible moves is doubly exponential in $x$ (due to the downward branching of the tree) while for upward-simulation it is only single exponential in $y$.

Besides pruning and quotienting, we also use the operation *RU* that removes useless states, i.e., states that either cannot be reached from any initial state or from which no tree can be accepted. Let $Op(x, y)$ be the following sequence of operations on tree automata:

1) *RU*,
2) quotienting with $\leq^{x\text{-dw}}$,
3) pruning with $P(id, <^{x\text{-dw}})$,
4) *RU*,
5) quotienting with $\leq^{y\text{-up}}(id)$,
6) pruning with $P(<^{y\text{-up}}(id), id)$,
7) pruning with $P(\sqsubset^{\text{up}}(id), \leq^{x\text{-dw}})$,
8) *RU*,
9) quotienting with $\leq^{y\text{-up}}(id)$,
10) pruning with $P(\leq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$,
11) *RU*.

It is language preserving by the Theorems of Sections 2.4 and 2.5. The order of the operations is chosen according to some considerations of efficiency. For instance, some steps are performed one after the other because they make use of the same preorder, which thus need not be recomputed in-between steps: the $\leq^{x\text{-dw}}$ preorder is used in both 2) and 3) without requiring recomputation, and the same happening with $\leq^{y\text{-up}}(id)$ used in steps 5) and 6). Yet which of the two, the quotienting or the pruning, ought to be performed first has a different explanation. As we saw in Section 2.4, transition pruning operates by performing comparisons between the states in the endpoints of every two transitions in the automaton. And we know that, in general, state quotienting reduces the number of states in an automaton. Thus, quotienting is performed before pruning in order to decrease the number of comparisons during the pruning. Still, no order of steps is ideal for all automata instances, and we also experimented with an alternative $Op'(x,y)$ defined as follows:

1) $RU$,
2) quotienting with $\preceq^{y\text{-up}}(id)$,
3) pruning with $P(\preceq^{y\text{-up}}(\sqsubseteq^{\mathsf{dw}}),\sqsubset^{\mathsf{dw}})$,
4) $RU$,
5) quotienting with $\preceq^{x\text{-dw}}$,
6) pruning with $P(id,\prec^{x\text{-dw}})$,
7) $RU$,
8) quotienting with $\preceq^{y\text{-up}}(id)$,
9) pruning with $P(\sqsubset^{\mathsf{up}}(id),\preceq^{x\text{-dw}})$,
10) $RU$.

However, on the tests performed we found that there was no significant difference in terms of automata reduction between the two options, and we thus adopted $Op(x,y)$ as the default one.

Let us now define our $Heavy(x,y)$ algorithm, whose behavior depends on the parameterizing lookaheads $x$ and $y$. $Heavy(1,1)$ just iterates $Op(1,1)$ until a fixpoint is reached, while the general algorithm $Heavy(x,y)$ does not iterate $Op(x,y)$, but uses a double loop instead: it iterates the sequence $Heavy(1,1)Op(x,y)$ until a fixpoint is reached. The reason behind the double loop is to keep the runtime down, as ordinary simulations are significantly easier to compute than lookahead simulations, which results in $Heavy(1,1)$ being much faster than $Heavy(x,y)$ with higher lookaheads. There is thus a tradeoff between computation time and reducibility.

While in $Heavy(x,y)$ the lookaheads "jump" directly from (1,1) to $(x,y)$ when the first fixpoint is reached, one could instead make this increment progressive, trying all possible downward lookaheads between 1 and $x$. Consider thus the algorithm $HeavyProg(x)$ which applies the sequence of reductions $Op(1,1), Op(2,3), \ldots, Op(x, 2^x - 1)$ to an input automaton, but restarts the sequence whenever an actual reduction happens. Thus, $Op$ only attempts a certain downward lookahead when all smaller lookaheads cannot yield any further reductions. For each downward lookahead $x'$ between 1 and $x$, the upward lookahead used is calculated as $2^{x'} - 1$ (due to the downward branching and upward linear structure of a tree).

*HeavyProg* will have advantages and disadvantages compared to *Heavy*, depending on the automata instance being reduced. *HeavyProg* can perform better than *Heavy* for automata which do become smaller and smaller using reductions with the intermediary lookaheads. For automata where increasing the lookaheads does not pave the way for further reductions, *Heavy* will outperform *HeavyProg* since it reaches a fixpoint sooner. Note that for $x < 3$, $HeavyProg(x)$ and $Heavy(x, 2^x - 1)$ behave exactly the same.

It is easy to see that what the *Heavy* and *HeavyProg* algorithms output is not the absolute minimum TDTA for that language (such an automaton in general is not even unique for nondeterministic automata), but a local minimum w.r.t. $Op(x,y)$, i.e., the smallest TDTA that our reduction steps can obtain.

## 2.7 Combined Preorder

One of the best methods previously known for reducing TA performs state quotienting based on a combination of downward and upward simulation [AHKV09]. However, as we will see below, this method cannot achieve any further reduction on an automaton which has been previously reduced with the techniques we described above.

Let $\oplus$ be an operator defined as follows: given two preorders $H$ and $S$ over a set $Q$, for $x, y \in Q$, $x(H \oplus S)y$ iff (*i*) $x(H \circ S)y$ and (*ii*) $\forall z \in Q : yHz \implies x(H \circ S)z$. Let $D$ be a downward simulation preorder and $U$ an upward simulation preorder induced by $D$. A combined preorder $W$ is defined as $W = D \oplus U^{-1}$ and satisfies that $D \subseteq W \subseteq D \circ U^{-1}$. Thus, for any states $x, y$ such that $x(D \oplus U^{-1})y$, there exists a state $z$, called a mediator, such that $xDz$ and $yUz$. A preorder $W$ can be computed following its definition in time $O(\mathsf{min}(|D| \cdot |Q|, |U| \cdot |Q|))$ [AHKV09]. Since, in general and for a fixed number of states, larger preorders $D$ yield larger preorders $U$ induced by $D$, the larger $D$ is the longer it takes to compute $W$. When $D$ and $U$ are the maximal simulation preorders $\sqsubseteq^{\mathsf{dw}}$ and $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$, respectively, we speak of *the combined preorder*. In [AHKV09] the authors show the impossibility of relaxing the need of downward simulations. In other words, if one replaces $\sqsubseteq^{\mathsf{dw}}$ by a larger under-approximation of $\subseteq^{\mathsf{dw}}$, the language is not guaranteed to be preserved during the quotienting.

In the following, Lemmas 2.7.1 and 2.7.2 are used by Theorem 2.7.1 to show that any quotienting with the equivalence relation induced by a combined preorder is subsumed by $Heavy(1,1)$. Note that any automaton $A$ which has been reduced with $Heavy(1,1)$ satisfies (1) $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) = A/(\sqsubseteq^{\mathsf{up}}(id) \cap \sqsupseteq^{\mathsf{up}}(id))$ due to the repeated quotienting, and (2) $A = Prune(A, P(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$ due to the repeated pruning.

**Lemma 2.7.1.** Let $A$ be an automaton and $p$ and $q$ two states. If 1) $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$ and 2) $A = Prune(A, P(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$, then $(p \sqsubseteq^{\mathsf{dw}} q \wedge p(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))q) \implies p = q$.

*Proof.* From 1) it follows that $\sqsubseteq^{\mathsf{dw}}$ is antisymmetric, so if $p \sqsubseteq^{\mathsf{dw}} q$ then $p \sqsubset^{\mathsf{dw}} q \vee p = q$.

From $p(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))q$, it follows that for any transition $\langle p',\sigma,p_1\dots p_i\dots p_{\#(\sigma)}\rangle$ with $p_i = p$ there exists a transition $\langle q',\sigma,q_1\dots q_i\dots q_{\#(\sigma)}\rangle$ with $q_i = q$ such that $p'(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))q'$ and $p_j \sqsubseteq^{\mathsf{dw}} q_j$ for all $j : 1 \leq j \neq i \leq \#(\sigma)$. From $p = p_i \sqsubseteq^{\mathsf{dw}} q_i = q$, we have that $(p_1\dots p\dots p_{\#(\sigma)}) \hat{\sqsubseteq}^{\mathsf{dw}} (q_1\dots q\dots q_{\#(\sigma)})$. From 2) it follows that there is no $k : 1 \leq k \leq \#(\sigma)$ such that $p_k \sqsubset^{\mathsf{dw}} q_k$. In particular, $\neg(p \sqsubset^{\mathsf{dw}} q)$. Thus we conclude that $p = q$. □

**Lemma 2.7.2.** Let $A$ be an automaton and $p$ and $q$ two states. If $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$, then $(p \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})q) \wedge (q \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})p) \implies (p \sqsubseteq^{\mathsf{up}}(id)q) \wedge (q \sqsubseteq^{\mathsf{up}}(id)p)$.

*Proof.* Since $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$, for any two states $x$ and $y$ we have that $(x \sqsubseteq^{\mathsf{dw}} y) \implies (x \sqsubset^{\mathsf{dw}} y \vee x = y)$.

Let $p$ and $q$ be states s.t. $p \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})q$ and $q \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})p$. By the definition of $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$ it follows that for any transition $\langle p',\sigma,p_1\dots p_i\dots p_{\#(\sigma)}\rangle$ with $p_i = p$ there exists a transition $\langle q',\sigma,q_1\dots q_i\dots q_{\#(\sigma)}\rangle$ with $p' \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})q'$ and $q_i = q$ such that for any $j : 1 \leq j \neq i \leq \#(\sigma) \cdot p_j \sqsubseteq^{\mathsf{dw}} q_j$, and vice-versa. We can thus construct an infinite sequence of matching transitions where, for every index $j \neq i$, the sequence of states at component $j$ is $\sqsubseteq^{\mathsf{dw}}$-increasing. However, since $A$ only has a finite number of states (and transitions), all these sequences must converge to some equivalence class w.r.t. $\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}$. Thus, for any transition $\langle p',\sigma,p_1\dots p_i\dots p_{\#(\sigma)}\rangle$ with $p_i = p$ there exists a transition $\langle q',\sigma,q_1\dots q_i\dots q_{\#(\sigma)}\rangle$ with $p' \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})q'$ and $q_i = q$ such that for any $j : 1 \leq j \neq i \leq \#(\sigma) \cdot p_j \sqsubseteq^{\mathsf{dw}} q_j \wedge q_j \sqsubseteq^{\mathsf{dw}} p_j$, and vice-versa. However, since $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$, we obtain that $p_j = q_j$ for $j : 1 \leq j \neq i \leq \#(\sigma)$. By repeating the same argument for the new pair of states $p'$ and $q'$, we get that $(p \sqsubseteq^{\mathsf{up}}(id)q) \wedge (q \sqsubseteq^{\mathsf{up}}(id)p)$ as required. Hence $(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}) \cap (\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))^{-1}) \subseteq (\sqsubseteq^{\mathsf{up}}(id) \cap (\sqsubseteq^{\mathsf{up}}(id))^{-1})$. □

**Theorem 2.7.1.** Let $A$ be an automaton such that:
(1) $A = A/(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) = A/(\sqsubseteq^{\mathsf{up}}(id) \cap \sqsupseteq^{\mathsf{up}}(id))$, and
(2) $A = Prune(A, P(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$.
Then $A = A/(W \cap W^{-1})$, where $W = \sqsubseteq^{\mathsf{dw}} \oplus (\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))^{-1}$.

*Proof.* We show that $(p\,W q) \wedge (q\,W p) \implies p = q$, which implies $A = A/(W \cap W^{-1})$. Let $p\,W q$ and $q\,W p$, then by the definition of $W$, there exist mediators $r$ such that $p \sqsubseteq^{\mathsf{dw}} r$ and $q \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})r$ and $s$ such that $q \sqsubseteq^{\mathsf{dw}} s$ and $p \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})s$. By the definition of $W$, we have that $p\,(\sqsubseteq^{\mathsf{dw}} \circ (\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))^{-1})\,s$ and $q\,(\sqsubseteq^{\mathsf{dw}} \circ (\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}))^{-1})\,r$. Thus, there exist mediators $t$ such that $p \sqsubseteq^{\mathsf{dw}} t$ and $s \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})t$ and $u$ such that $q \sqsubseteq^{\mathsf{dw}} u$ and $r \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})u$. By the transitivity of $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$ we obtain that $p \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})t$ and $q \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})u$. From 1), 2) and Lemma 2.7.1 we obtain that $p = t$ and $q = u$. So we have $s \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})p$ and $r \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})q$. By Lemma 2.7.2 we obtain that $s \sqsubseteq^{\mathsf{up}}(id)p$ and $p \sqsubseteq^{\mathsf{up}}(id)s$, and $r \sqsubseteq^{\mathsf{up}}(id)q$ and $q \sqsubseteq^{\mathsf{up}}(id)r$.

Thus by (1) we obtain that $p = s$ and $q = r$. Since $p \sqsubseteq^{\mathsf{dw}} r$ and $q \sqsubseteq^{\mathsf{dw}} s$, we conclude that $p = q$. □

## 2.8 Transition Saturation

In Section 2.4, we described the transition pruning technique, which removes a transition if a *better* one remains. In this section, we introduce its dual notion, saturation, which adds a transition if a *better* one exists already. The motivation behind saturation is to pave the way for further reductions when the *Heavy* algorithm has reached a fixpoint on the automaton, as we will make clear further below. Saturation has been defined for the words case before [CM16], here we apply it to tree automata.

Let $A = (\Sigma, Q, \delta, I)$ be a TDTA, $\Delta = Q \times \Sigma \times Q^+$ and $S \subseteq \Delta \times \Delta$ a reflexive binary relation on $\Delta$. The *S-saturated automaton* is defined as $Sat(A,S) := (\Sigma, Q, \delta_S, I)$, where

$$\delta_S = \{\langle p', a, q'_1 \ldots q'_{\#(a)}\rangle \in \Delta \mid \exists \langle p, a, q_1 \ldots q_{\#(a)}\rangle \in \delta \cdot \langle p', a, q'_1 \ldots q'_{\#(a)}\rangle \, S \, \langle p, a, q_1 \ldots q_{\#(a)}\rangle\}.$$

Since $S$ is reflexive, any transition in the initial automaton is preserved and so $A \subseteq Sat(A,S)$. When the converse inclusion also holds, we say that $S$ is *good for saturation* (GFS). Note that the GFS property is downward closed in the space of reflexive relations, i.e., if $R$ is GFS and $id \subseteq R' \subseteq R$, then $R'$ too is GFS (or if $R'$ is not GFS, then $R$ too is not GFS).

Given two binary relations $R_s$ and $R_t$ on $Q$, we define $S(R_s, R_t) = \{(\langle p, \sigma, r_1 \cdots r_n\rangle, \langle p', \sigma, r'_1 \cdots r'_n\rangle) \mid pR_sp' \text{ and } \langle r_1 \cdots r_n\rangle \hat{R}_t \langle r'_1 \cdots r'_n\rangle\}$, where $\hat{R}_t$ is the standard lifting of $R_t \subseteq Q \times Q$ to $\hat{R}_t \subseteq Q^n \times Q^n$. Informally, a transition $t'$ is added to the automaton if there exists already a transition $t$ s.t. its source state is $R_s$-larger than the source state of $t'$, and its target states are $\hat{R}_t$-larger than the target states of $t'$. Note that, in general, $S(R_s, R_t)$ is not transitive. However, all $S(R_s, R_t)$ relations which we will prove to be good for saturation use transitive relations for both $R_s$ and $R_t$, thus making $S(R_s, R_t)$ itself transitive.

The following theorems and corollaries provide several relations which are GFS. Note that the GFS property is downward closed in the space of reflexive relations, i.e., if $R$ is GFS and $id \subseteq R' \subseteq R$, then $R'$ too is GFS (or if $R'$ is not GFS, then $R$ too is not GFS).

**Theorem 2.8.1.** *$S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$ is GFS.*

*Proof.* Let $A$ be a TDTA and $A_S = Sat(A, S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}}))$. We will use induction on $n \geq 1$ to show that for every tree $t$ of height $n$ and every run $\pi_S$ of $A_S$ s.t. $t \overset{\pi_S}{\Longrightarrow} p$, for some state $p$, there exists a run $\pi$ of $A$ s.t. $t \overset{\pi}{\Longrightarrow} p$. This shows, in particular, that $A_S \subseteq A$.

In the base case $n = 1$, $t$ is a leaf-node $\sigma$, for some $\sigma \in \Sigma$. Thus for every run $\pi_S$ of $A_S$ such that $t \overset{\pi_S}{\Longrightarrow} p$, for some state $p$, there exists $\langle p, \sigma, \psi\rangle \in \delta_S$. By the definition of $\delta_S$, there exists $\langle q,\sigma,\psi\rangle \in \delta$ s.t. $q \subseteq^{\mathsf{dw}} p$. Consequently, there exists a run $\pi$ in $A$ s.t. $t \overset{\pi}{\Longrightarrow} q$. By $q \subseteq^{\mathsf{dw}} p$, there also exists a run $\pi'$ of $A$ s.t. $t \overset{\pi'}{\Longrightarrow} p$.

For the induction step, let $t$ be a tree of height $n > 1$ and $a$ its root symbol. Thus for every run $\pi_S$ of $A_S$ s.t. $t \overset{\pi_S}{\Longrightarrow} p$, for some state $p$, there exist $\langle p,a,q_1 \ldots q_{\#(a)}\rangle \in \delta_S$ and, for each $i : (1 \le i \le \#(a))$, a run $\pi_{S_i}$ of $A_S$ s.t. $t_i \overset{\pi_{S_i}}{\Longrightarrow} q_i$. By the definition of $\delta_S$, there exists $\langle p',a,q'_1 \ldots q'_{\#(a)}\rangle \in \delta$ s.t. $p' \subseteq^{\mathsf{dw}} p$ and, for every $i : (1 \le i \le \#(a))$, $q'_i \supseteq^{\mathsf{dw}} q_i$. Applying the induction hypothesis to each of the subtrees $t_i$, we know that for every $t_i$-run $\pi_{S_i}$ of $A_S$ ending in $q_i$ there is also a $t_i$-run $\pi_i$ of $A$ ending in $q_i$. And since $q'_i \supseteq^{\mathsf{dw}} q_i$ for every $i : (1 \le i \le \#(a))$, for each $t_i$ there exists a run $\pi'_i$ of $A$ s.t. $t_i \overset{\pi'_i}{\Longrightarrow} q'_i$. Since there exists $\langle p',a,q'_1 \ldots q'_{\#(a)}\rangle \in \delta$, we obtain that there is a run $\pi''$ of $A$ s.t. $t \overset{\pi''}{\Longrightarrow} p'$. From $p' \subseteq^{\mathsf{dw}} p$, it follows that there is also a run $\pi'''$ of $A$ s.t. $t \overset{\pi'''}{\Longrightarrow} p$. □

**Corollary 2.8.1.** *It follows from Theorem 2.8.1 and the fact that GFS is downward closed that* $S(\supseteq^{\mathsf{dw}},\sqsubseteq^{k\text{-}\mathsf{dw}})$, $S(\supseteq^{\mathsf{dw}},id)$, $S(\sqsupseteq^{l\text{-}\mathsf{dw}},\subseteq^{\mathsf{dw}})$, $S(\sqsupseteq^{l\text{-}\mathsf{dw}},\sqsubseteq^{k\text{-}\mathsf{dw}})$, $S(\sqsupseteq^{l\text{-}\mathsf{dw}},id)$, $S(id,\subseteq^{\mathsf{dw}})$ *and* $S(id,\sqsubseteq^{k\text{-}\mathsf{dw}})$ *are GFS, for any* $k,l \ge 1$.

For Theorem 2.8.2 we use the auxiliary Lemma 2.8.1 and the following definitions. For every tree $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, let $level_i(\pi)$ be the tuple of states that $\pi$ visits at depth $i$ in the tree, read from left to right. Formally, let $\langle v_1,\ldots,v_n\rangle$, with each $v_j \in \mathbb{N}^i$, be the set of all tree positions of depth $i$ s.t. each $v_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = \langle \pi(v_1),\ldots,\pi(v_n)\rangle \in Q^n$. We say that $st \in Q^*$ is a subtuple of $level_i(\pi)$, and write $st \le level_i(\pi)$, if all states in $st$ also appear in $level_i(\pi)$ and in the same order. By lifting preorders on $Q$ to preorders on $Q^n$, we can compare tuples of states w.r.t. $\subseteq^{\mathsf{up}}(id)$.

**Lemma 2.8.1.** *Let* $A$ *be a TDTA and* $\langle p_1,\ldots,p_n\rangle$ *and* $\langle q_1,\ldots,q_n\rangle$ *two tuples of states of* $A$ *such that* $\langle p_1,\ldots,p_n\rangle \subseteq^{\mathsf{up}}(id)\langle q_1,\ldots,q_n\rangle$. *Then, for every* $t \in \mathbb{T}(\Sigma)$, *every accepting t-run* $\pi$ *and every tuple* $\langle v_1,\ldots,v_n\rangle$ *of some leaves of* $\pi$ *of the same depth* $i$ *(i.e.,* $\langle v_1,\ldots,v_n\rangle \le level_i(\pi)$*) s.t.* $\langle \pi(v_1),\ldots,\pi(v_n)\rangle = \langle p_1,\ldots,p_n\rangle$, *there exists an accepting t-run* $\pi'$ *of* $A$ *such that* $\langle \pi'(v_1),\ldots,\pi'(v_n)\rangle = \langle q_1,\ldots,q_n\rangle$ *and* $\pi'(v) = \pi(v)$ *for every leaf* $v$ *of* $\pi'$ *other than* $v_1,\ldots,v_n$.

*Proof.* Let $\pi$ be an accepting $t$-run of $A$ s.t. $\langle \pi(v_1),\ldots,\pi(v_n)\rangle = \langle p_1,\ldots,p_n\rangle$. We say that an accepting $t$-run $\pi''$ is $i$-good iff i) for every node $v_j$ of $\pi''$, with $j \le i$, $\pi''(v_j) = q_j$, and ii) for every $v_j$, with $i < j \le n$, $\pi''(v_j) = p_j$. We will show, by induction on $i$, that for every $i$ there exists an accepting $t$-run $\pi'''$ which is $i$-good and s.t. $\pi'''(v) = \pi(v)$ for every leaf $v$ of $\pi'''$ other than $v_1,\ldots,v_n$. For the particular case of $i = n$ this proves the lemma.

The base case $i = 0$ is trivial, since the accepting $t$-run $\pi$ is 0-good itself.

For the induction step, let $\pi_1$ be an accepting $(i-1)$-good $t$-run of $A$. If $i > n$, the lemma holds trivially. Otherwise, we have $\pi_1(v_i) = p_i \subseteq^{\mathsf{up}}(id)\ q_i$ and thus there exists an accepting $t$-run $\pi_2$ of $A$ s.t. $\pi_2(v_i) = q_i$. And since the upward trace inclusion is parameterized by $id$, it follows, in particular, that for every leaf $v$ other than $v_i$, $\pi_2(v) = \pi_1(v)$. Thus, $\pi_2$ is an accepting $i$-good $t$-run of $A$. Moreover, we have that, on leaves other than $v_1,\ldots,v_n$, the run $\pi_2$ coincides with $\pi_1$ and consequently, by the induction hypothesis, with $\pi$. □

**Theorem 2.8.2.** *$S(\subseteq^{\mathsf{up}}(id),\supseteq^{\mathsf{up}}(id))$ is GFS.*

*Proof.* Let $A$ be a TDTA and $A_S = Sat(A, S(\subseteq^{\mathsf{up}}(id),\supseteq^{\mathsf{up}}(id)))$. If $\hat{t} \in A_S$, then there exists an accepting $\hat{t}$-run $\hat{\pi}$ of $A_S$. We will show that there exists an accepting $\hat{t}$-run of $A$, which proves $A_S \subseteq A$.

Let us first define an auxiliary notion. For every $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, we say that $\pi$ is $i$-good iff it does not contain any transition of $\delta_S - \delta$ from any position $v \in \mathbb{N}^*$ s.t. $|v| < i$, i.e., all transitions used in the first $i$ levels of the tree are of $A$.

Next, we will show, by induction on $i$, that for every $i$ there exists an accepting $i$-good $\hat{t}$-run $\hat{\pi}'$ of $A_S$ s.t. $level_i(\hat{\pi}') = level_i(\hat{\pi})$. For $i$ equal to the height of $\hat{t}$, this implies that there exists an accepting $\hat{t}$-run of $A$.

The base case $i = 0$ is trivial, since $\hat{\pi}$ is 0-good itself.

For the induction step, let us first define some auxiliary notions. For every $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, we say that $level_{i'}(\pi)$ is $j$-good iff $\pi$ does not contain a transition of $\delta_S - \delta$ from a state $\pi(v_k)$, s.t. $k \le j$ and $\pi(v_k)$ is the $k$-th state of $level_{i'}(\pi)$. We now say that an accepting $\hat{t}$-run $\hat{\pi}''$ of $A_S$ is $(i-1, j)$-good iff i) it is $(i-1)$-good, ii) $level_{i-1}(\hat{\pi}'')$ is $j$-good, and iii) $level_i(\hat{\pi}'') = level_i(\hat{\pi})$.

We will now show, by induction on $j$, that for every $j$ there exists an accepting $(i-1, j)$-good $\hat{t}$-run of $A_S$. Since trees are finitely-branching, we have that for a sufficiently large $j$ there is an accepting $\hat{t}$-run $\hat{\pi}'''$ of $A_S$ which is $i$-good. And since, in particular, $level_i(\hat{\pi}''') = level_i(\hat{\pi})$, this will conclude the outer induction.

For the base case $(i-1, 0)$, we know by the hypothesis of the outer induction that there exists an accepting $(i-1)$-good $\hat{t}$-run $\pi_1$ s.t. $level_{i-1}(\pi_1) = level_{i-1}(\hat{\pi})$. Then the $\hat{t}$-run $\pi_2$ which, on the levels below $i$, coincides with $\pi_1$ and, on the levels from $i$ up, coincides with $\hat{\pi}$ too is accepting and $(i-1)$-good. Thus $\pi_2$ is $(i-1, 0)$-good.

For the induction step, let $\pi_1$ be an accepting $(i-1, j-1)$-good $\hat{t}$-run of $A_S$, and let $\pi_1'$ be the prefix of $\pi_1$ which only uses transitions of $A$. $\pi_1'$ is thus an accepting run of $A$ over some prefix tree $\hat{t}'$ of $\hat{t}$. Let $v_j$ be the node of $\hat{t}$ s.t. $\pi_1'(v_j)$ is the $j$-th state of $level_{i-1}(\pi_1')$ and $\sigma = \hat{t}(v_j)$ a symbol of rank $r$.

If $r = 0$, then $v_j$ is a leaf of $\hat{t}$ and so there exists a transition $\langle \pi_1'(v_j), \sigma, \psi \rangle$ in $A_S$. By the definition of $\delta_S$, there exists a transition $\langle p, \sigma, \psi \rangle$ in $A$ s.t. $\pi_1'(v_j) \subseteq^{\mathsf{up}}(id)\, p$. Thus there exists an accepting $\hat{t}'$-run $\pi_2$ of $A$ s.t. $\pi_2(v_j) = p$ and for any leaf $v$ of $\pi_2$ other than $v_j$,

$\pi_2(v) = \pi'_1(v)$. We now obtain a run over $\hat{t}$ again by extending $\pi_2$ downwards according to $\pi_1$, i.e., $\pi_2(vv') := \pi_1(vv')$, for every leaf $v$ of $\pi_2$ other than $v_j$ and for every $v' \in \mathbb{N}^*$. It follows that $level_i(\pi_2) = level_i(\pi_1) = level_i(\hat{\pi})$. $\pi_2$ is clearly a $(i-1)$-good $\hat{t}$-run of $A_S$ and $level_{i-1}(\pi_2)$ is $j$-good. Thus $\pi_2$ is an accepting $(i-1, j)$-good $\hat{t}$-run of $A_S$.

If $r > 0$, then $v_j$ is not a leaf and so there exists a transition $\langle \pi'_1(v_j), \sigma, \pi_1(v_j1) \ldots \pi_1(v_jr) \rangle$ in $A_S$. By the definition of $\delta_S$, there exists a transition *trans*: $\langle p, \sigma, q_1 \ldots q_r \rangle$ in $A$ s.t. $\pi'_1(v_j) \subseteq^{\mathsf{up}}(id)p$ and 1) $\langle q_1 \ldots q_r \rangle \subseteq^{\mathsf{up}}(id) \langle \pi_1(v_j1) \ldots \pi_1(v_jr) \rangle$. From $\pi'_1(v_j) \subseteq^{\mathsf{up}}(id)\, p$ we have that there exists an accepting $\hat{t}'$-run $\pi_2$ of $A$ s.t. $\pi_2(v_j) = p$ and $\pi_2(v) = \pi'_1(v)$, for every leaf $v$ of $\pi_2$ other than $v_j$. Extending $\pi_2$ with *trans* we obtain an accepting run of $A$ s.t. $\pi_2(v_jk) := q_k$ for each child $v_jk$ of $v_j$. Applying Lemma 2.8.1 to 1), we obtain that there exists an accepting run $\pi_3$ of $A$ over the same prefix tree of $\hat{t}$ as $\pi_2$ s.t. 2) $\pi_3(v_jk) = \pi_1(v_jk)$ for each child $v_jk$ of $v_j$, and $\pi_3(v) = \pi_2(v) = \pi_1(v)$ for every leaf $v$ of $\pi_3$ other than $v_j1, \ldots, v_jr$. We now obtain a run over $\hat{t}$ again by extending $\pi_3$ downwards according to $\pi_1$, i.e., 3) $\pi_3(vv') := \pi_1(vv')$, for every leaf $v$ of $\pi_3$ other than $v_j1, \cdots, v_jr$ and for every $v' \in \mathbb{N}^*$. $\pi_3$ is clearly a $(i-1)$-good $\hat{t}$-run of $A_S$ and $level_{i-1}(\pi_3)$ is $j$-good. From 2) and 3), we obtain that $level_i(\pi_3) = level_i(\pi_1) = level_i(\hat{\pi})$. Thus $\pi_3$ is an accepting $(i-1, j)$-good $\hat{t}$-run of $A_S$. □

**Corollary 2.8.2.** *From Theorem 2.8.2 it follows that* $S(\subseteq^{\mathsf{up}}(id), \sqsupseteq^{k\text{-}\mathsf{up}}(id))$, $S(\subseteq^{\mathsf{up}}(id), id)$, $S(\sqsubseteq^{l\text{-}\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$, $S(\sqsubseteq^{l\text{-}\mathsf{up}}(id), \sqsupseteq^{k\text{-}\mathsf{up}}(id))$, $S(\sqsubseteq^{l\text{-}\mathsf{up}}(id), id)$, $S(id, \supseteq^{\mathsf{up}}(id))$ *and* $S(id, \sqsupseteq^{k\text{-}\mathsf{up}}(id))$ *are GFS, for any* $k, l \geq 1$.

The figures below provide several counterexamples of GFS relations:

- Figure 2.8 shows that $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$.
- Figure 2.9 shows that $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS.
- Figure 2.10 shows that $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS.
- Figure 2.11 is inspired by an example for a similar result for linear trees (i.e., words) [CM16]. It shows that $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$.

The table in Figure 2.12 summarizes all our results on relations which are or are not good for saturation. Note that, since GFS is downward closed, negative results propagate to larger relations.

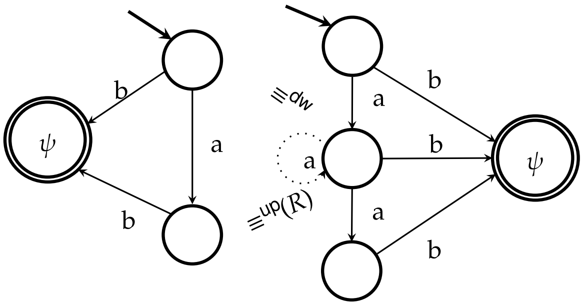


Figure 2.8: $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$: if we add the dotted transition, the linear tree $aaab$ is now accepted. The symbol $b$ has rank $0$ and $a$ rank $1$.

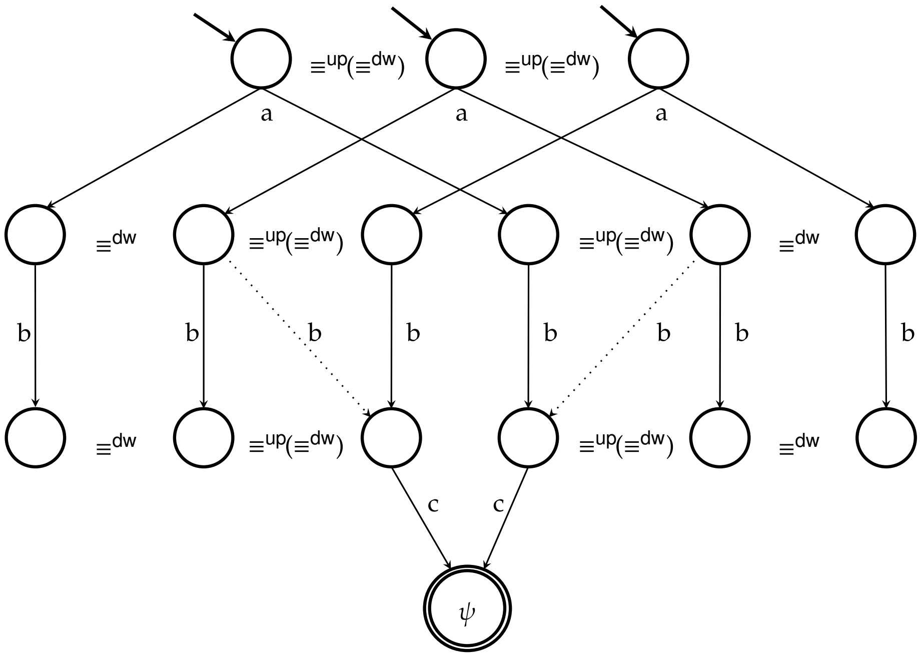


Figure 2.9: $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS: if we add the dotted transitions, the tree $a(b(c), b(c))$ is now accepted. The symbols $c$, $b$ and $a$ have ranks $0$, $1$ and $2$, respectively.

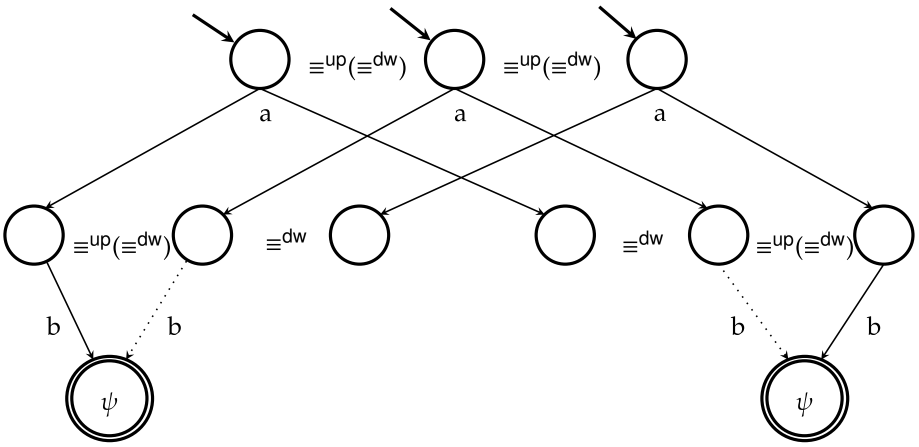

$\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}})$
$\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}})$
a
a
a
$\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}})$
$\equiv^{\mathsf{dw}}$
$\equiv^{\mathsf{dw}}$
$\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}})$
b
b
b
b
$\psi$
$\psi$

Figure 2.10: $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS: if we add the dotted transitions, the tree $a(b,b)$ is now accepted. The symbols $b$ and $a$ have ranks $0$ and $1$, respectively.

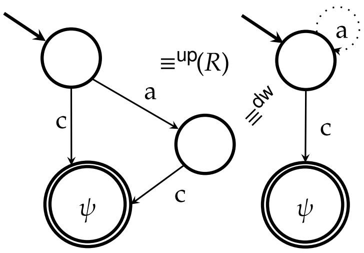


Figure 2.11: $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$ (example adapted from [CM16]): if we add the dotted transition, the linear tree $aac$ is now accepted. The symbol $c$ has rank $0$ and $a$ has rank $1$.

| $S$ | $id$ | $\sqsubseteq^{\mathsf{dw}}$ | $\subseteq^{\mathsf{dw}}$ | $\sqsupseteq^{\mathsf{up}}(id)$ | $\supseteq^{\mathsf{up}}(id)$ | $\sqsupseteq^{\mathsf{up}}(\sqsupseteq^{\mathsf{dw}})$ | $\supseteq^{\mathsf{up}}(\supseteq^{\mathsf{dw}})$ |
|---|---|---|---|---|---|---|---|
| $id$ | ✓ | ✓ | ✓ | ✓ | ✓ | × | × |
| $\sqsupseteq^{\mathsf{dw}}$ | ✓ | ✓ | ✓ | × | × | × | × |
| $\supseteq^{\mathsf{dw}}$ | ✓ | ✓ | ✓ | × | × | × | × |
| $\sqsubseteq^{\mathsf{up}}(id)$ | ✓ | × | × | ✓ | ✓ | × | × |
| $\subseteq^{\mathsf{up}}(id)$ | ✓ | × | × | ✓ | ✓ | × | × |
| $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$ | × | × | × | × | × | × | × |
| $\subseteq^{\mathsf{up}}(\subseteq^{\mathsf{dw}})$ | × | × | × | × | × | × | × |

Figure 2.12: GFS relations for tree automata. The largest GFS relations found are marked with ✓, and ✓ marks their corresponding under-approximations. Relations which are not GFS are marked with ×.

## 2.9 Reducing Automata by Adding Transitions: a new algorithm

We saw in Section 2.6 that, given an automaton $A_1$, the reduced automaton $A_2 := Heavy(x,y)(A_1)$ is a local minimum w.r.t. $Heavy(x,y)$, i.e., no smaller automaton can be obtained by applying the steps in $Op(x,y)$ again. The motivation behind saturation is to change this scenario. By adding transitions to $A_2$ using a $S(R_s,R_t)$ relation which is GFS, we obtain an automaton $A_3 := S(R_s,R_t)(A_2)$ which is language equivalent to $A_2$ but (possibly) denser. Thus, some states in $A_3$ might now be collapsible using a GFQ preorder, since saturation helps making them more 'identical' to each other. In its turn, state quotienting may then lead to new transitions becoming superfluous in the automaton as well. Thus, applying $Heavy(x,y)$ to $A_3$ might reach a new local minimum which is now smaller than $A_2$. In particular, $A_4 := Heavy(x,y)(A_3)$ might have

1. fewer states,
2. as many states but fewer transitions, or
3. as many states and at least as many transitions.

We say that in scenarios 1. and 2. the automaton $A_4$ is *better* than $A_2$.

Just as in the case of the *Heavy* algorithm, there is no ideal order to apply the saturation and the reduction techniques, so different possible algorithms have been defined and tested. We present five different versions of $Sat(x,y)$, where $x,y \geq 1$ are the lookaheads used for computing downward and upward simulations, respectively (see Figure 2.13). In all versions of *Sat*, we chose an order for the operations that ensures that the effect of the saturations is not necessarily cancelled by the reductions immediately after.

Intuitively, *Sat1* starts by applying both saturations together, in an attempt to obtain a highly dense automaton where more states may be quotiented. Since the upwards saturation may allow for new transitions to be added in a downward saturation, *Sat2* places the two saturation steps inside an inner loop, which is executed until no more transitions can be added. *Sat3* differs from *Sat2* in that it swaps the direction of both the saturation and the reduction steps. *Sat4* takes a different approach. By interleaving each downward saturation with the upward reductions it may allow, *Sat4* prevents the automaton from becoming too dense. And since each upward reduction not only may allow for new downward saturations to be performed but may also have its effect cancelled if the upward saturation is performed immediately after, in *Sat5* (called *Sat2* in [Alm16b]) downward saturation and upward reductions are iterated in an inner loop before performing any upward saturation. All versions return the 'best' automaton encountered.

## 2.10 Conclusion

In this chapter we have seen polynomial-time algorithms for reducing finite tree automata, based on the language-preserving techniques of state quotienting, transition pruning and transition saturation. We provided a complete picture of downward and upward equivalences which are suitable for quotienting, and combinations of upward and downward relations (ordinary simulations or trace inclusions) which can be used for pruning transitions in an automaton. Similarly, we presented our results on relations which are good for saturation. Many of these results are nontrivial and counterintuitive, as it is shown by the irregular layout of the table of results in Section 2.4, as well as some rather elaborate pruning counter-examples.

We defined the first reduction algorithm as *Heavy*, which applies all our transition pruning and state quotienting results, alternating between them until a fixpoint is reached (i.e., no further reduction is achieved). The second reduction algorithm, *Sat*, combines *Heavy* with our transition-saturation methods, alternating between them until a fixpoint is reached.

We showed that quotienting with the combined preorder - one of the best reductions methods known for tree automata - cannot achieve any further reduction on automata which have already been reduced with *Heavy*. In the following chapter we will see how, in practice, this method is outperformed by *Heavy*. We will also analyse how well *Heavy* and *Sat* behave in practice by testing their performance in a variety of automata samples of different provenience. Apart from that, we will look into some of they key aspects of their computation as well as some implementation details that help the algorithms become more efficient.

```
Sat1(x,y)
Loop:
  Sat. with $S(\geq^{x\text{-dw}}, \leq^{x\text{-dw}})$
  Sat. with $S(\leq^{y\text{-up}}(id), \geq^{y\text{-up}}(id))$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\sqsubset^{\text{up}}(id), \leq^{x\text{-dw}})$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\leq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Run $Heavy(x, y)$
```

```
Sat2(x,y)
Loop:
  Loop:
    Sat. with $S(\geq^{x\text{-dw}}, \leq^{x\text{-dw}})$
    Sat. with $S(\leq^{y\text{-up}}(id), \geq^{y\text{-up}}(id))$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\sqsubset^{\text{up}}(id), \leq^{x\text{-dw}})$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\leq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Run $Heavy(x, y)$
```

```
Sat3(x,y)
Loop:
  Loop:
    Sat. with $S(\leq^{y\text{-up}}(id), \geq^{y\text{-up}}(id))$
    Sat. with $S(\geq^{x\text{-dw}}, \leq^{x\text{-dw}})$
  Run $Heavy(x, y)$
```

```
Sat4(x,y)
Loop:
  Sat. with $S(\geq^{x\text{-dw}}, \leq^{x\text{-dw}})$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\sqsubset^{\text{up}}(id), \leq^{x\text{-dw}})$
  Quot. with $\leq^{y\text{-up}}(id)$
  Prune with $P(\leq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Sat. with $S(\leq^{y\text{-up}}(id), \geq^{y\text{-up}}(id))$
  Run $Heavy(x, y)$
```

```
Sat5(x,y)
Loop:
  Loop:
    Sat. with $S(\geq^{x\text{-dw}}, \leq^{x\text{-dw}})$
    Quot. with $\leq^{y\text{-up}}(id)$
    Prune with $P(<^{y\text{-up}}(id), id)$
    Prune with $P(\sqsubset^{\text{up}}(id), \leq^{x\text{-dw}})$
    Quot. with $\leq^{y\text{-up}}(id)$
    Prune with $P(\leq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Sat. with $S(\leq^{y\text{-up}}(id), \geq^{y\text{-up}}(id))$
  Run $Heavy(x, y)$
```

Figure 2.13: Five different saturation-based reduction algorithms. Loops are performed until the automaton no longer becomes *better*. All versions return the 'best' automaton encountered.

# Chapter 3

# Experiments and Implementation

This chapter gives a practical look into the algorithms defined in Chapter 2. In Section 3.1 we detail various experiments conducted with both algorithms and the results obtained. Subsection 3.1.1 tests the reduction performance of $Heavy(x,y)$ for different values of lookaheads $x$ and $y$. In Subsection 3.1.2 we give evidence of how adding transitions with $Sat(x,y)$ can help obtain smaller automata, not just in the number of states but also in the number of transitions. Finally, Subsection 3.1.3 explores how reducing with our algorithms aids in obtaining smaller complement automata and faster.

In Section 3.2 we explain the matrix-refinement approach to the computation of downward lookahead simulation, based on which some optimizations were built to make the overall computation more efficient: the pre-refinement step, which under-approximates non-simulation, and caching of attacks in the simulation game, which avoids unnecessary recomputations.

The implementation of downward/upward simulation and of our reduction algorithms *Heavy* and *Sat* are provided in the tool `minotaut` [Alm16a], to which Section 3.3 is dedicated. Subsection 3.3.1 describes the tool architecture and how the bridge with the `libvata` library is established. In Subsection 3.3.2 we present the tool interface, providing examples of how the user can test the algorithms presented in this thesis, as well as a diverse collection of other automata operations and analysis methods which we have also implemented. Lastly, Subsection 3.3.3 details a few of the relevant aspects of the implementation of the algorithms, namely how moves and the caching of attacks in the simulation game are stored and used.

We conclude the chapter with Section 3.4, where we summarise the experimental results obtained.

## 3.1 Experimental Results

We carried out our experiments using three sets of automata from the `libvata` distribution, as well as randomly generated tree automata. The first set are 27 moderate-sized automata (87 states and 816 transitions on average) derived from abstract regular tree model checking applications. We designate this sample as `moderate-artmc`. The second set has the same provenience but consists of 62 automata which are significantly larger (586 states and 8865 transitions on average). We thus designate this sample as `nonmoderate-artmc`. The third set are 14,498 automata (57 states and 266 transitions on average) from the shape analysis tool Forester [LSV+17b], and we designate this sample simply as `forester`. The random tree automata were generated according to a generalization of the Tabakov-Vardi model of random word automata [TV07]. Given parameters $n$,$s$, $td$ (transition density) and $ad$ (acceptance density), it generates tree automata with $n$ states, $s$ symbols (each of rank 2), $n * td$ randomly assigned transitions for each symbol, and $n * ad$ randomly assigned leaf rules.

All experiments were run on an Intel® Core™ i5 @ 3.20GHz x 4 machine with 8GB of RAM using a 64-bit version of Ubuntu 16.04.

### 3.1.1 Reducing Finite Tree Automata with $Heavy(x, y)$

In this section we compare the reduction performance of the $Heavy(x, y)$ algorithm against some simpler methods, and also against quotienting with the combined preorder (the method with the greatest reduction ratio so far, to the best of our knowledge; see Section 2.7). Let us recall that a state in the automaton is *useless* if it cannot be reached from any initial state or if no tree can be accepted from it. Consider the following methods:

**RU:** Removing useless states. (Previously present in `libvata`.)

**RUQ:** **RU** plus quotienting with $\sqsubseteq^{\mathsf{dw}}$. (Previously present in `libvata`.)

**RUQP:** **RUQ** plus pruning with $P(id, \sqsubset^{\mathsf{dw}})$. (Not in `libvata`, but simple.)

**RUC:** **RU** plus quotienting with the combined preorder.

**Heavy:** $Heavy(1, 1)$, $Heavy(2, 4)$ and $HeavyProg(3)$. (New.)

As described above, we tested these algorithms on the samples `moderate-artmc`, `nonmoderate-artmc` and `forester` and on randomly generated automata. In the following we show how the different methods performed for each sample, in terms of reduction achieved and computation time. The average reduction values are presented in terms of percent of states and transitions left after the reduction relative to before. Thus, smaller values are better, as they correspond to greater reductions.

**`moderate-artmc`.** *Heavy*(1,1), on average, reduced the number of states and transitions of the 27 automata in this sample *to* 27% and 14% of the original sizes, respectively. **RUC** performed slightly worse, reducing, on average, the number of states and transitions to 29% and 23%, respectively. In contrast, **RU** did not perform any reduction in any case. **RUQ**, on average, reduced the number of states and transitions only to 81% and 80% of the original sizes, and **RUQP** reduced the number of states and transitions to 81% and 32%. Most of the methods performed very fast, although **RUC** was significantly slower than any of the other ones. The average computation times of *Heavy*(1,1), **RUC**, **RUQP**, **RUQ** and **RU** were, respectively, 0.05s, 1.43s, 0.03s, 0.006s and 0.001s.

The charts in Figure 3.1 provide the reduction achieved in terms of percentage intervals. Table 3.1 contains individual results for the reduction applied by *Heavy*(1,1) to each automaton in the sample. The columns give the following information: name of each automaton; $\#Q_i$: original number of states; $\#Delta_i$: original number of transitions; $\#Q_f$: number of states after reduction; $\#Delta_f$: number of transitions after reduction; Q reduction: the reduction ratio for states in percent $100 * \#Q_f / \#Q_i$ (smaller is better); Delta reduction: the reduction ratio for transitions in percent $100 * \#Delta_f / \#Delta_i$; and Time(s): the computation time in seconds. Note that the reduction ratios for transitions are smaller than the ones for states, i.e., the automata get sparser.

**`nonmoderate-artmc`.** *Heavy*(1,1), on average, reduced the number of states and transitions of the 62 automata in this sample *to* 4.2% and 0.7% of the original sizes, respectively. **RUC** reduced the number of states and transitions to 6.6% and 3.4%. In contrast, **RU** did not perform any reduction in any case. **RUQ**, on average, reduced the number of states and transitions to 75.2% and 74.8% of the original sizes and **RUQP** reduced the number of states and transitions to 75.2% and 15.8%. The average computation times of *Heavy*(1,1), **RUC**, **RUQP**, **RUQ** and **RU** were, respectively, 2.7s, 30.7s, 2.1s, 0.2s and 0.02s. Note how the **RUC** method reduced only slightly less than *Heavy*(1,1), but was much slower.

The charts in Figure 3.2 provide the reduction achieved in terms of percentage intervals. Tables 3.2 and 3.3 contain individual results for the reduction applied by *Heavy*(1,1) to each automaton in the sample. The column names should be read as in the table for the `moderate-artmc` sample.

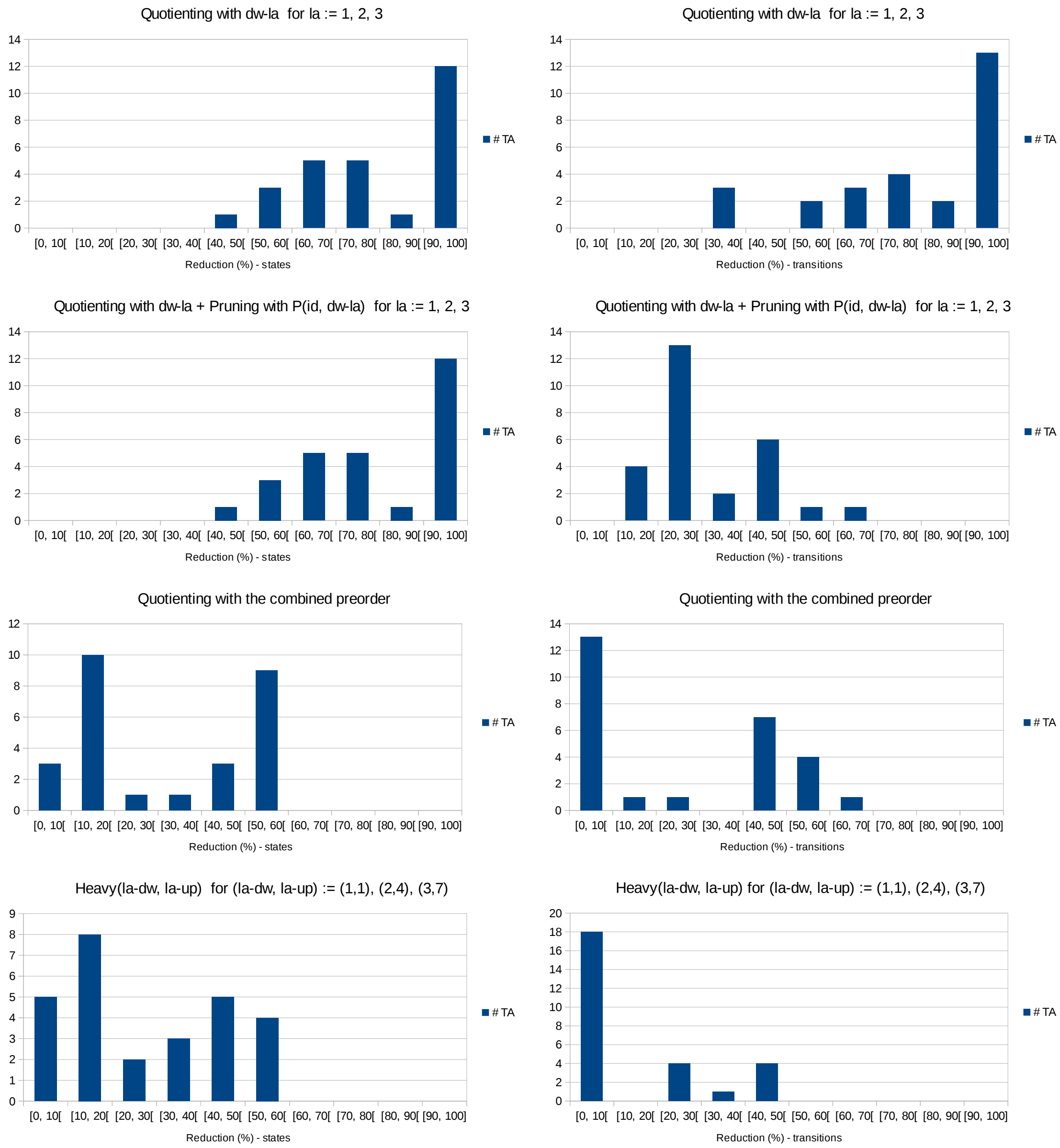


Figure 3.1: Reduction of 27 moderate-sized tree automata by methods **RUQ** (top row), **RUQP** (second row), **RUC** (third row) and Heavy (bottom row). A bar of height $h$ at an interval $[x, x+10[$ means that $h$ of the 27 automata were reduced to a size between $x\%$ and $(x+10)\%$ of their original size. The reductions in the numbers of states/transitions are shown on the left/right, respectively. Each method reduced more than the ones above, which is shown by a greater concentration of bars on the left side of the charts. On this set of automata, the methods $Heavy(2,4)$ and $Heavy(3,7)$ gave exactly the same results as $Heavy(1,1)$.

| TA name | $\#Q_i$ | $\#Delta_i$ | $\#Q_f$ | $\#Delta_f$ | Q reduction | Delta reduction | Time(s) |
|---|---|---|---|---|---|---|---|
| A0053 | 54 | 159 | 27 | 66 | 50.00 | 41.51 | 0.015 |
| A0054 | 55 | 241 | 28 | 93 | 50.91 | 38.59 | 0.024 |
| A0055 | 56 | 182 | 27 | 73 | 48.21 | 40.11 | 0.017 |
| A0056 | 57 | 230 | 24 | 55 | 42.11 | 23.91 | 0.017 |
| A0057 | 58 | 245 | 24 | 58 | 41.38 | 23.67 | 0.020 |
| A0058 | 59 | 257 | 25 | 65 | 42.37 | 25.29 | 0.019 |
| A0059 | 60 | 263 | 24 | 59 | 40.00 | 22.43 | 0.022 |
| A0060 | 61 | 244 | 32 | 111 | 52.46 | 45.49 | 0.034 |
| A0062 | 63 | 276 | 32 | 112 | 50.79 | 40.59 | 0.029 |
| A0063 | 64 | 571 | 11 | 23 | 17.19 | 4.03 | 0.027 |
| A0064 | 65 | 574 | 11 | 23 | 16.92 | 4.01 | 0.024 |
| A0065 | 66 | 562 | 11 | 23 | 16.67 | 4.09 | 0.026 |
| A0070 | 71 | 622 | 11 | 23 | 15.49 | 3.70 | 0.016 |
| A0080 | 81 | 672 | 26 | 58 | 32.10 | 8.63 | 0.043 |
| A0082 | 83 | 713 | 26 | 65 | 31.33 | 9.12 | 0.047 |
| A0083 | 84 | 713 | 26 | 65 | 30.95 | 9.12 | 0.048 |
| A0086 | 87 | 1402 | 26 | 112 | 29.89 | 7.99 | 0.103 |
| A0087 | 88 | 1015 | 12 | 23 | 13.64 | 2.27 | 0.060 |
| A0088 | 89 | 1027 | 12 | 23 | 13.48 | 2.24 | 0.063 |
| A0089 | 90 | 1006 | 12 | 21 | 13.33 | 2.09 | 0.064 |
| A0111 | 112 | 1790 | 11 | 42 | 9.82 | 2.35 | 0.139 |
| A0117 | 118 | 2088 | 25 | 106 | 21.19 | 5.08 | 0.177 |
| A0120 | 121 | 1367 | 12 | 21 | 9.92 | 1.54 | 0.068 |
| A0126 | 127 | 1196 | 11 | 23 | 8.66 | 1.92 | 0.083 |
| A0130 | 131 | 1504 | 11 | 23 | 8.40 | 1.53 | 0.044 |
| A0172 | 173 | 1333 | 11 | 23 | 6.36 | 1.73 | 0.098 |
| A0177 | 178 | 1781 | 26 | 58 | 14.61 | 3.26 | 0.085 |
| Average | 87.07 | 816.04 | 19.78 | 53.59 | 26.97 | 13.94 | 0.052 |

Table 3.1: Results of reducing 27 moderate-sized tree automata from model checking with our $Heavy(1,1)$ algorithm. The columns give the name of each automaton, $\#Q_i$: its original number of states, $\#Delta_i$: its original number of transitions, $\#Q_f$: the number of states after reduction, $\#Delta_f$: the number of transitions after reduction, the reduction ratio for states in percent $100 * \#Q_f / \#Q_i$ (smaller is better), the reduction ratio for transitions in percent $100 * \#Delta_f / \#Delta_i$ (smaller is better), and the computation time in seconds. Note that the reduction ratios for transitions are smaller than the ones for states, i.e., the automata get not only smaller but also sparser.

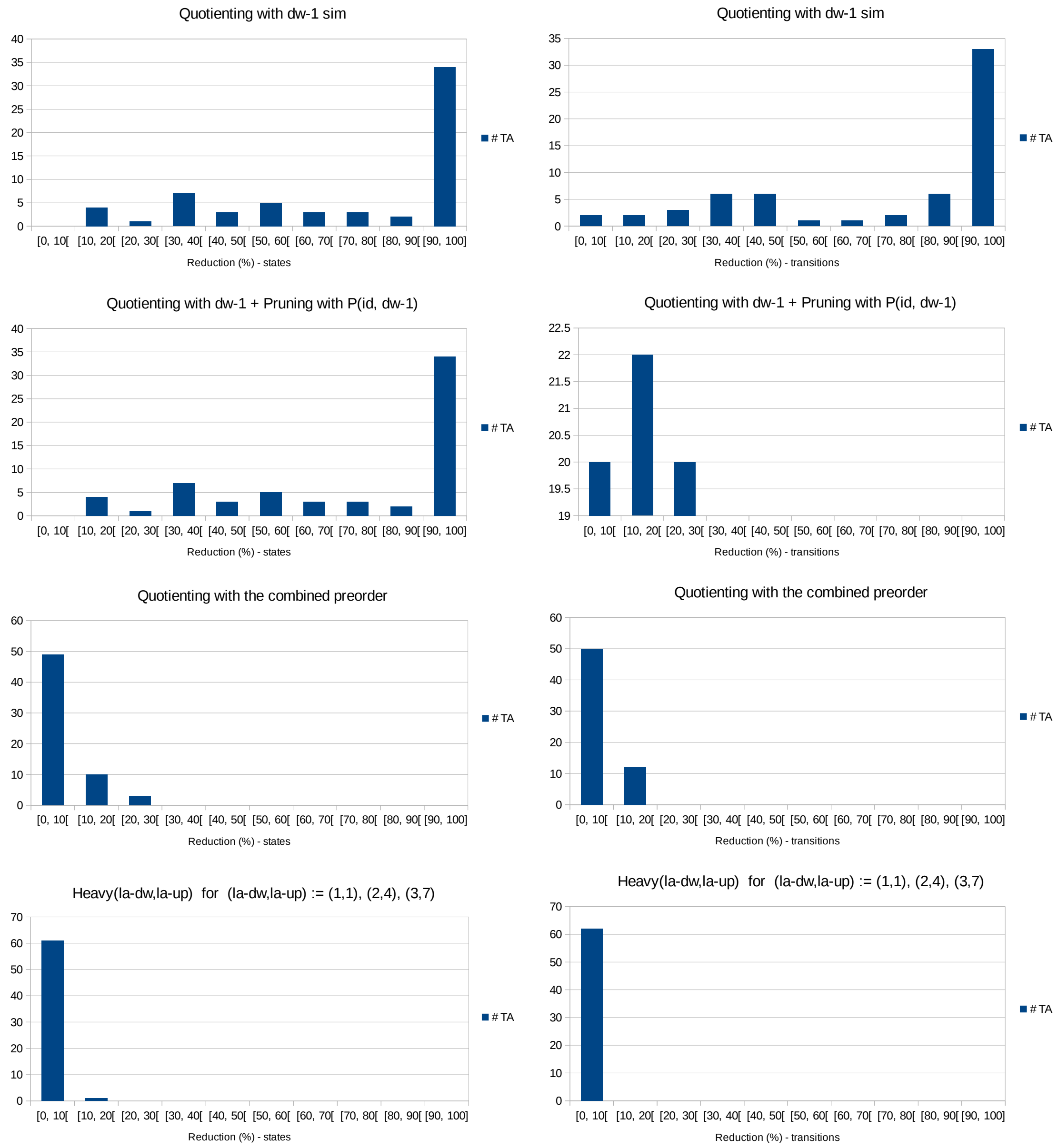


Figure 3.2: Reduction of 62 large tree automata by methods **RUQ** (top row), **RUQP** (second row), **RUC** (third row) and Heavy (bottom row). A bar of height $h$ at an interval $[x, x+10[$ means that $h$ of the 62 automata were reduced to a size between $x\%$ and $(x+10)\%$ of their original size. The reductions in the numbers of states/transitions are shown on the left/right, respectively. Each method reduced more than the ones above, which is shown by a greater concentration of bars on the left side of the charts. On this set of automata, the methods $Heavy(2,4)$ and $Heavy(3,7)$ gave exactly the same results as $Heavy(1,1)$.

| TA name | $\#Q_i$ | $\#Delta_i$ | $\#Q_f$ | $\#Delta_f$ | Q reduction | Delta reduction | Time(s) |
|---|---|---|---|---|---|---|---|
| A246 | 247 | 2944 | 11 | 42 | 4.45 | 1.43 | 0.40 |
| A301 | 302 | 4468 | 12 | 21 | 3.97 | 0.47 | 0.29 |
| A310 | 311 | 3343 | 24 | 52 | 7.72 | 1.56 | 0.59 |
| A312 | 313 | 3367 | 11 | 23 | 3.51 | 0.68 | 0.21 |
| A315 | 316 | 3387 | 24 | 52 | 7.59 | 1.54 | 0.58 |
| A320 | 321 | 3623 | 26 | 65 | 8.10 | 1.79 | 0.56 |
| A321 | 322 | 3407 | 24 | 52 | 7.45 | 1.53 | 0.62 |
| A322 | 323 | 3651 | 35 | 100 | 10.84 | 2.74 | 0.67 |
| A323 | 324 | 6199 | 26 | 112 | 8.02 | 1.81 | 1.48 |
| A328 | 329 | 3517 | 26 | 58 | 7.90 | 1.65 | 0.50 |
| A329 | 330 | 5961 | 24 | 100 | 7.27 | 1.68 | 1.36 |
| A334 | 335 | 3936 | 11 | 23 | 3.28 | 0.58 | 0.72 |
| A335 | 336 | 3738 | 26 | 58 | 7.74 | 1.55 | 0.56 |
| A339 | 340 | 5596 | 12 | 21 | 3.53 | 0.38 | 0.49 |
| A348 | 349 | 3681 | 11 | 23 | 3.15 | 0.62 | 0.27 |
| A354 | 355 | 3522 | 24 | 52 | 6.76 | 1.48 | 0.70 |
| A355 | 356 | 3895 | 25 | 55 | 7.02 | 1.41 | 0.45 |
| A369 | 370 | 4134 | 24 | 52 | 6.49 | 1.26 | 0.31 |
| A387 | 388 | 4117 | 24 | 52 | 6.19 | 1.26 | 0.51 |
| A390 | 391 | 5390 | 11 | 23 | 2.81 | 0.43 | 1.15 |
| A400 | 401 | 5461 | 11 | 23 | 2.74 | 0.42 | 1.36 |
| A447 | 448 | 7924 | 12 | 23 | 2.68 | 0.29 | 2.55 |
| A483 | 484 | 5592 | 25 | 55 | 5.17 | 0.98 | 0.51 |
| A487 | 488 | 4891 | 16 | 28 | 3.28 | 0.57 | 0.33 |
| A488 | 489 | 8493 | 12 | 21 | 2.45 | 0.25 | 2.86 |
| A489 | 490 | 8516 | 12 | 21 | 2.45 | 0.25 | 2.93 |
| A491 | 492 | 8708 | 12 | 21 | 2.44 | 0.24 | 3.03 |
| A493 | 494 | 7523 | 12 | 21 | 2.43 | 0.28 | 0.69 |
| A494 | 495 | 8533 | 12 | 21 | 2.42 | 0.25 | 2.97 |
| A496 | 497 | 8618 | 12 | 21 | 2.41 | 0.24 | 2.81 |
| A498 | 499 | 8612 | 12 | 21 | 2.40 | 0.24 | 3.10 |

Table 3.2: Results on reducing 62 large automata from model checking with our $Heavy(1,1)$ algorithm (continues in Table 3.3).

| TA name | $\#Q_i$ | $\#Delta_i$ | $\#Q_f$ | $\#Delta_f$ | Q reduction | Delta reduction | Time(s) |
|---|---|---|---|---|---|---|---|
| A501 | 502 | 8632 | 12 | 21 | 2.39 | 0.24 | 2.95 |
| A532 | 533 | 8867 | 12 | 23 | 2.25 | 0.26 | 3.20 |
| A569 | 570 | 8351 | 26 | 58 | 4.56 | 0.69 | 0.98 |
| A589 | 590 | 9606 | 12 | 21 | 2.03 | 0.22 | 3.20 |
| A620 | 621 | 9218 | 12 | 21 | 1.93 | 0.23 | 1.45 |
| A646 | 647 | 6054 | 19 | 34 | 2.94 | 0.56 | 0.65 |
| A667 | 668 | 8131 | 26 | 58 | 3.89 | 0.71 | 1.12 |
| A670 | 671 | 11021 | 34 | 76 | 5.07 | 0.69 | 5.80 |
| A673 | 674 | 11157 | 25 | 55 | 3.71 | 0.49 | 5.38 |
| A676 | 677 | 11043 | 34 | 76 | 5.02 | 0.69 | 5.85 |
| A678 | 679 | 11172 | 26 | 56 | 3.83 | 0.50 | 5.32 |
| A679 | 680 | 11032 | 34 | 76 | 5.00 | 0.69 | 5.88 |
| A689 | 690 | 11207 | 31 | 71 | 4.49 | 0.63 | 5.59 |
| A691 | 692 | 11047 | 34 | 76 | 4.91 | 0.69 | 5.61 |
| A692 | 693 | 11066 | 34 | 76 | 4.91 | 0.69 | 6.10 |
| A693 | 694 | 11188 | 34 | 76 | 4.90 | 0.68 | 6.05 |
| A694 | 695 | 11191 | 34 | 76 | 4.89 | 0.68 | 6.09 |
| A695 | 696 | 11070 | 34 | 76 | 4.89 | 0.69 | 5.80 |
| A700 | 701 | 11245 | 36 | 81 | 5.14 | 0.72 | 6.13 |
| A701 | 702 | 11244 | 36 | 83 | 5.13 | 0.74 | 6.00 |
| A703 | 704 | 11255 | 34 | 76 | 4.83 | 0.68 | 6.09 |
| A723 | 724 | 9376 | 26 | 58 | 3.59 | 0.62 | 1.28 |
| A728 | 729 | 11903 | 12 | 21 | 1.65 | 0.18 | 2.97 |
| A756 | 757 | 8884 | 26 | 58 | 3.43 | 0.65 | 1.34 |
| A837 | 838 | 13038 | 11 | 23 | 1.31 | 0.18 | 5.34 |
| A881 | 882 | 15575 | 12 | 21 | 1.36 | 0.13 | 3.36 |
| A980 | 981 | 21109 | 12 | 21 | 1.22 | 0.10 | 4.64 |
| A1003 | 1004 | 21302 | 12 | 21 | 1.20 | 0.10 | 3.99 |
| A1306 | 1307 | 19699 | 25 | 55 | 1.91 | 0.28 | 2.88 |
| A1404 | 1405 | 18839 | 24 | 52 | 1.71 | 0.28 | 3.09 |
| A2003 | 2004 | 30414 | 24 | 52 | 1.20 | 0.17 | 6.98 |

Table 3.3: Results on reducing 62 large automata from model checking with our $Heavy(1,1)$ algorithm (continued from Table 3.2).

**forester.** *Heavy*(1,1), on average, reduced the number of states and transitions of the 14,498 automata in this sample *to* 76.4% and 67.9% of the original, respectively. **RUC** performed slightly worse, reducing the number of states and transitions to 76.5% and 68.3%, resp. **RUQP** reduced the states and transitions only to 94.1% and 87.6%, resp., and **RUQ** to 94.1% and 88.5%. The charts in Figure 3.3 provide these results in terms of percentage intervals. The average computation times of *Heavy*(1,1), **RUC**, **RUQP**, **RUQ** and **RU** were, respectively, 0.21s, 0.39s, 0.014s, 0.004s, and 0.0006s.

Due to the particular structure of the automata in these 3 sample sets, *Heavy*(2,4) and *HeavyProg*(3) had hardly any advantage over *Heavy*(1,1). However, in general they can perform significantly better.

**random automata.** We also tested the algorithms on randomly generated tree automata with 100 states, 2 symbols, acceptance density of 0.8 and varying transition densities (*td*). Figure 3.4 shows the results in terms of reduction in states and transitions for each value of *td*. Note that *Heavy*(2,4) reduces much better than *Heavy*(1,1) for $td \geq 3.5$. For higher lookaheads, we tried both *Heavy*(3,7) and *HeavyProg*(3), of which the latter behaved better. This difference of performance is explained by the fact that, in general, using downward lookahead 2 reduced much better than 1, thus benefiting *HeavyProg*(3), which performs reductions using a lookahead that is progressively incremented from 1 to 3. Still, the computation of the 3-lookahead downward simulation itself is sufficiently hard to make *HeavyProg*(3) not always terminate in due time. For this reason we had to impose a timeout of 1,800 seconds (30 minutes), after which *HeavyProg*(3) simply returns the smallest automaton achieved. This reason may explain why, in average values, the advantage of using lookahead 3 instead of 2 was not so impressive. The average computation times taken by *Heavy* and *HeavyProg* using different lookaheads and by the simpler methods can be seen in Figure 3.5.

Due to the complexity of computing the 3-lookahead downward simulation preorder, *HeavyProg*(3) is clearly slower by several orders of magnitude than any other methods. *Heavy*(1,1) is significantly faster than *Heavy*(2,4), which has its time peak at $td = 4.3$.

In general, **RUC** is a simpler method compared to *Heavy*(1,1), which may explain the lower average running times for all automata samples with $td < 5.3$. However, as noted above, not only the computation time of the combined preorder grows with the downward simulation but also its size. Thus computing this relation and then quotienting with it tends to take longer for particularly dense automata, which yield larger downward simulation preorders in general.

**RUQP** is faster than **RUC** and at $td = 4.5$ it has its highest average value (0.13s).

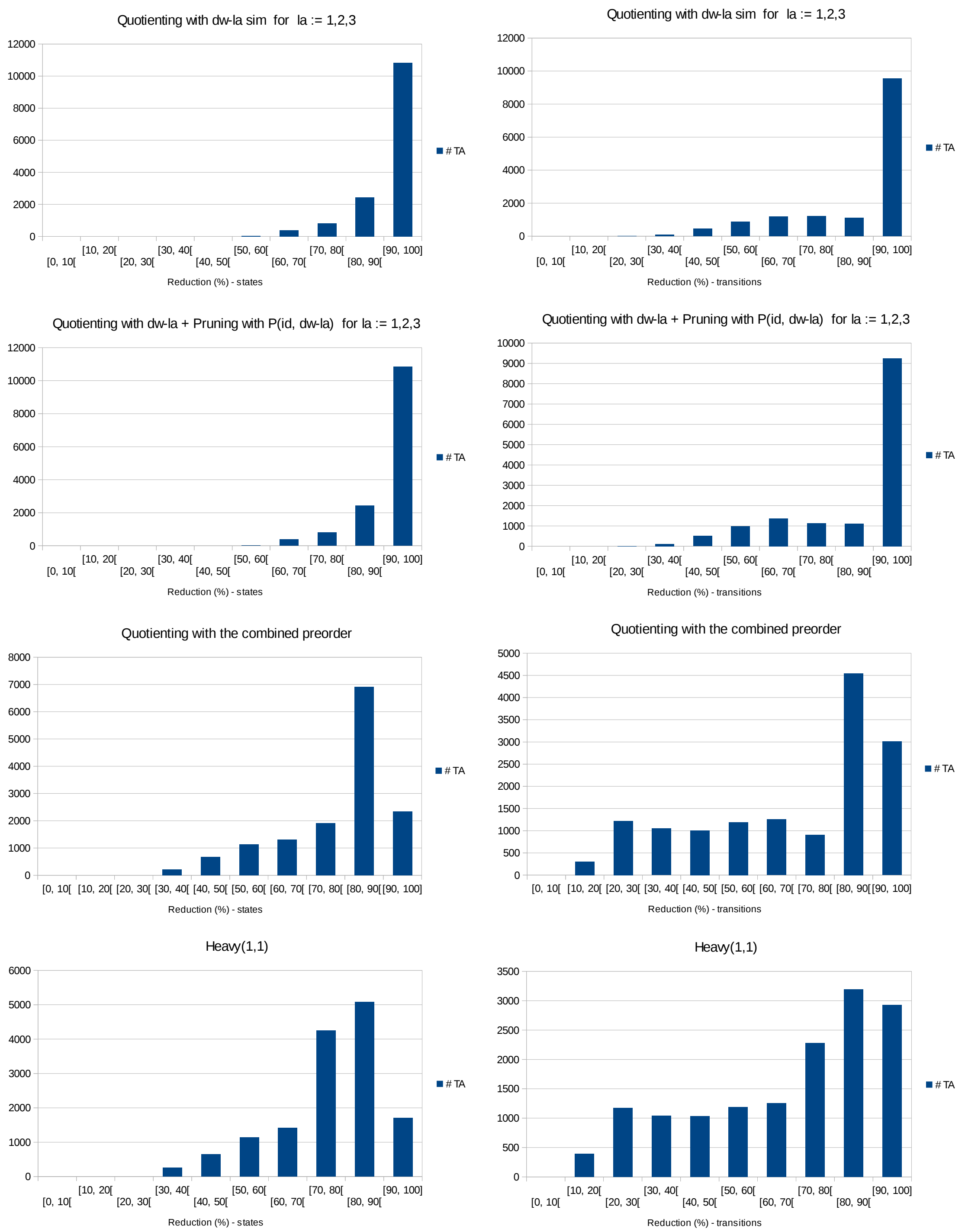


Figure 3.3: Reduction of 14,498 tree automata from the Forester tool [LSV+17b], by methods **RUQ** (top row), **RUQP** (second row), **RUC** (third row) and Heavy (bottom row). A bar of height $h$ at an interval $[x, x+10[$ means that $h$ of the 14,498 automata were reduced to a size between $x\%$ and $(x+10)\%$ of their original size. The reductions in the numbers of states/transitions are shown on the left/right, respectively. Both **RUC** and $Heavy(1,1)$ performed significantly better than **RUQ** and **RUQP**, and $Heavy(1,1)$ performed slightly better than **RUC** . Using Heavy with lookaheads higher than 1 made very little difference in this sample set.

The operations **RUQ** and **RU** are simpler and thus faster than **RUQP**.

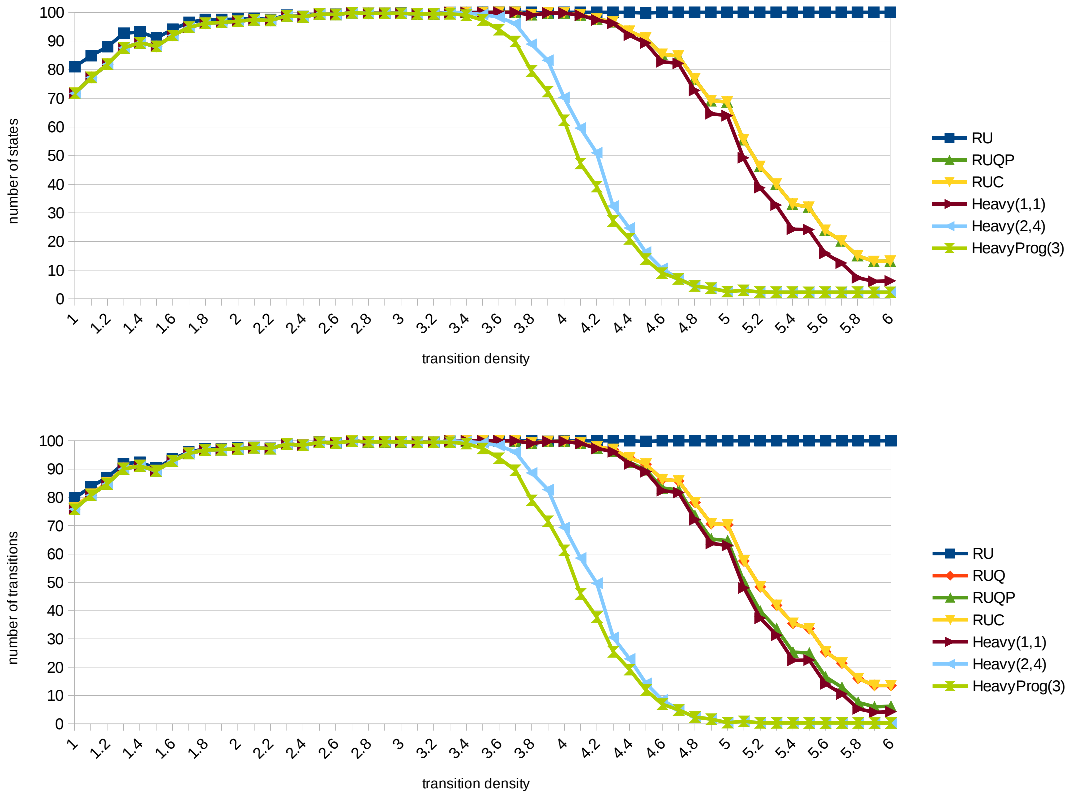


Figure 3.4: Reduction of Tabakov-Vardi random tree automata with $n = 100$, $s = 2$ and $ad = 0.8$ and varying $ad$. The top chart shows the average reduction (in percent) in terms of number of states and the bottom chart shows the reduction in the number of transitions (smaller is better) obtained with the various methods. Each data point in the charts is the average of 400 random automata. For $td \geq 4.3$, RUQ and RUC had a similar performance, reducing much more than RU; RUQP and $\mathit{Heavy}(1,1)$ reduced even better. For $td \geq 3.5$, $\mathit{Heavy}(2,4)$ reduced much better than $\mathit{Heavy}(1,1)$, and $\mathit{HeavyProg}(3)$ reduced even more.

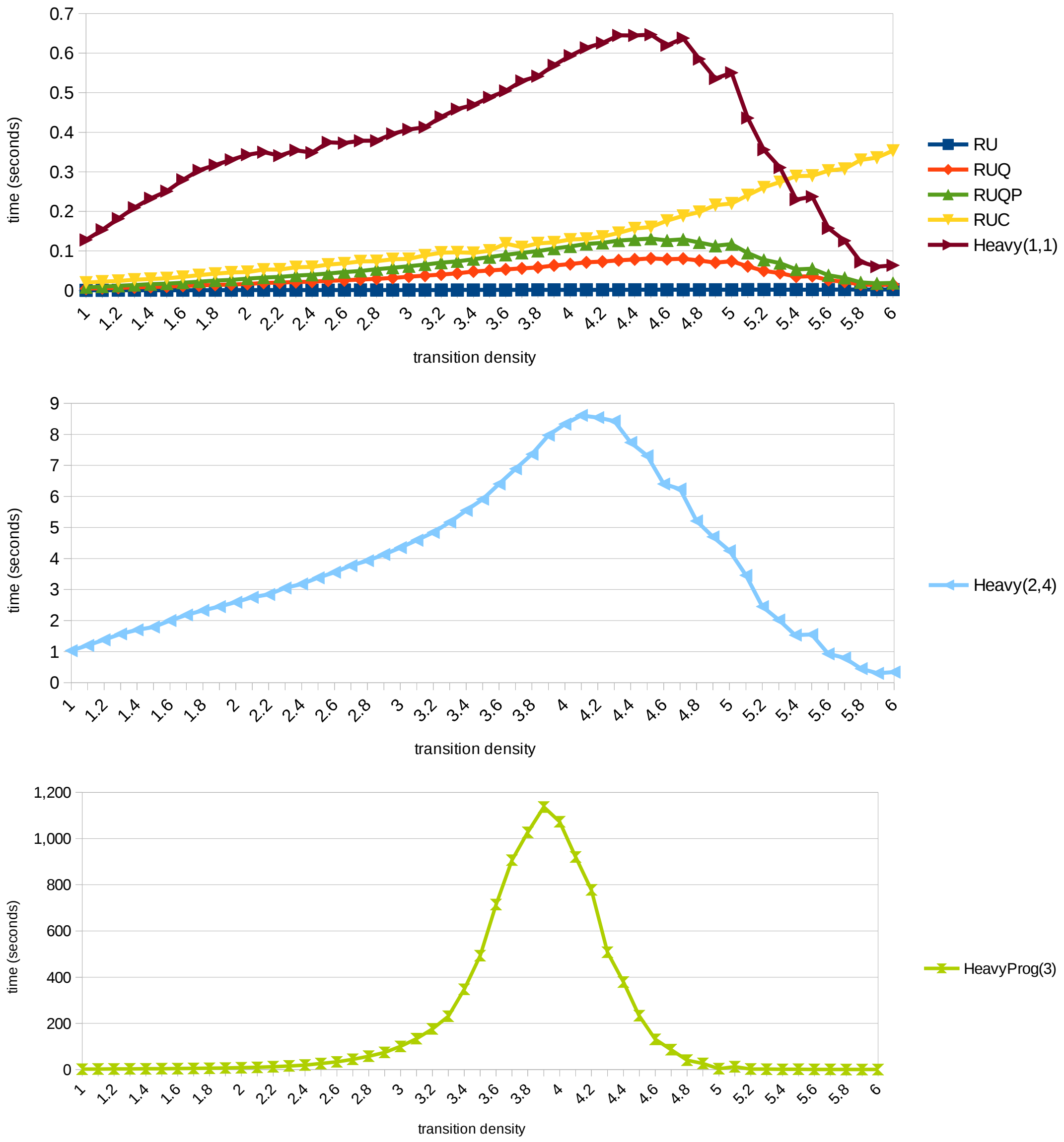


Figure 3.5: Reduction of Tabakov-Vardi random tree automata with $n = 100$, $s = 2$ and $ad = 0.8$. The charts show how the average time (in seconds) taken by each method varied with the transition density of the sample. We compared the methods **RU**, **RUQ**, **RUQP**, **RUC**, $Heavy(1,1)$ (top chart), $Heavy(2,4)$ (middle chart) and $HeavyProg(3)$ (bottom chart). Each data point in the charts is the average of 400 random automata.

The experiments show that our *Heavy*$(x, y)$ algorithm can significantly reduce the size of many classes of nondeterministic tree automata, and that it is sufficiently fast to handle instances with hundreds of states and thousands of transitions.

### 3.1.2 Reducing Finite Tree Automata with *Sat*$(x, y)$

The next results focus on the evaluation of the saturation-based reduction algorithms defined in Section 2.9. We can see in Figure 3.6 that, on average, the five different versions of *Sat* produced automata containing *both* fewer states and, especially, fewer transitions than *Heavy* alone. However, this came at the expense of longer running times. The fastest version of *Sat* was *Sat*1, but it achieved significantly less reduction in the number of transitions than *Sat*3, *Sat*4 or *Sat*5. The experiments presented in the next subsection provide more insight on the reduction performance of *Sat*5.

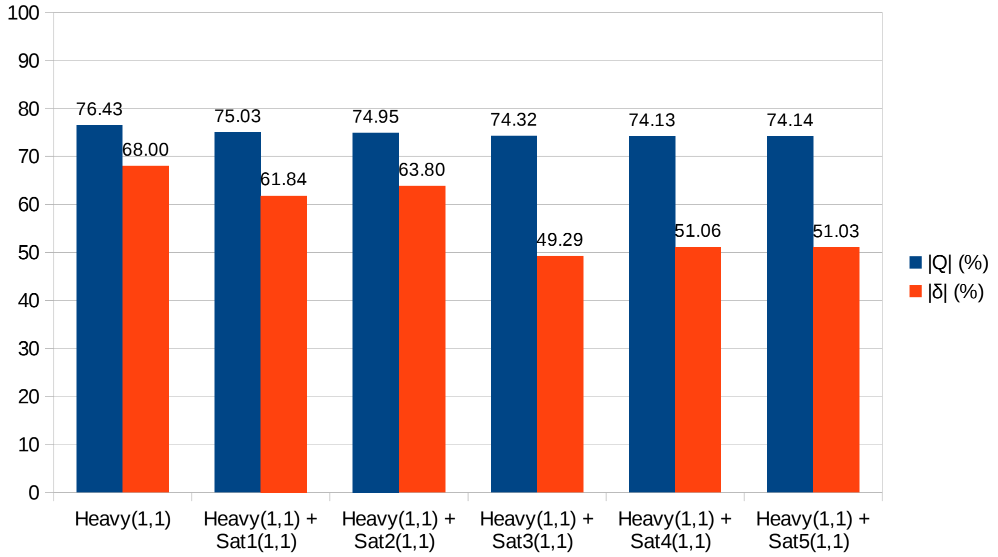
### 3.1.3 Reducing and Complementing Finite Tree Automata

The results that follow focus on the advantage of reducing automata when computing their complement (which is implemented in `libvata` as the difference algorithm [Hos10]). We started by testing on a subset of the `forester` sample (Figure 3.7), and we compared direct complementation with reducing automata (with *Heavy*$(1,1)$ optionally followed by *Sat*5$(1,1)$) prior to the complementation and with a final reduction using *Heavy*$(1,1)$. Due to memory reasons, direct complementation was not feasible for large automata. Thus the sample used is the subset of `forester` containing all automata with at most 14 states, in a total of 760 automata.

Note that one can obtain further reductions with *Heavy* after the complementation since the difference algorithm does not determinize the automata. In fact, in general determinization of top-down tree automata is not possible [CDG$^{+}$08].

As we can see, all reduction methods yielded significantly smaller complement automata, both in terms of states and in terms of transitions, than direct complementation, on average, while running either with similar times or substantially faster. This difference was particularly notorious when the automata were first reduced with both *Heavy*$(1,1)$ and *Sat*5$(1,1)$, which, compared to direct complementation, resulted in automata with fewer states (18 vs 27, on average) and fewer transitions (649 vs 1750) and at much lower times (0.02s vs 4.86s). Applying *Heavy*$(1,1)$ in the end reduced the automata even more, with a very low time cost.

The next experiments were performed on sets of randomly generated tree automata. Figures 3.8 and 3.9 show the results of complementing automata with 4 states, 2 symbols, acceptance density of 0.8 and varying transition densities (*td*). While the automata tested are very small, for some values of *td* their complements are quite



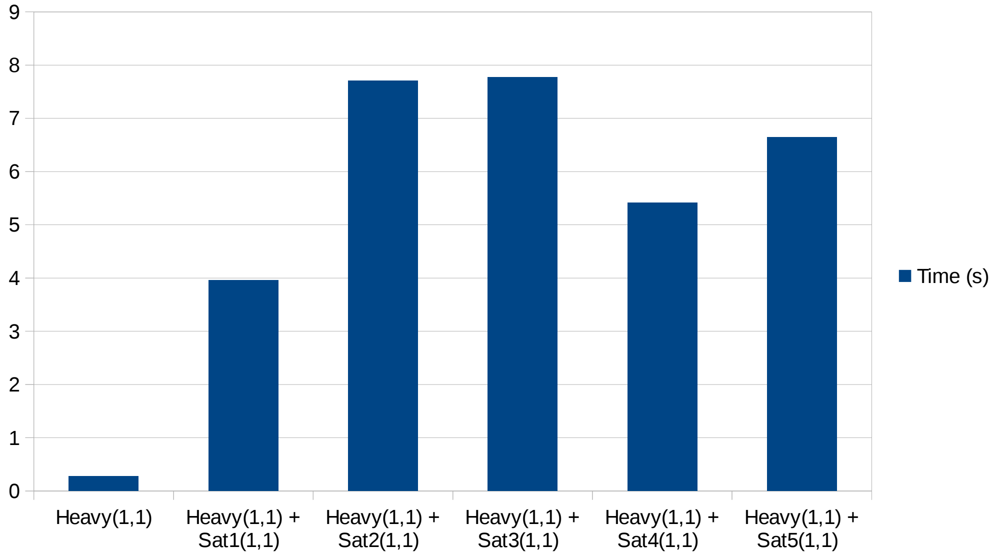


Figure 3.6: Reduction of Forester automata using saturation methods. The top chart gives the average number of states and transitions that remained (in percent) after application of each method; the bottom chart compares their running times. All five versions reduced more than $Heavy$ alone, but were significantly slower. $Sat1$ was the fastest version, but achieved less reduction in the number of transitions than $Sat3$, $Sat4$ or $Sat5$.

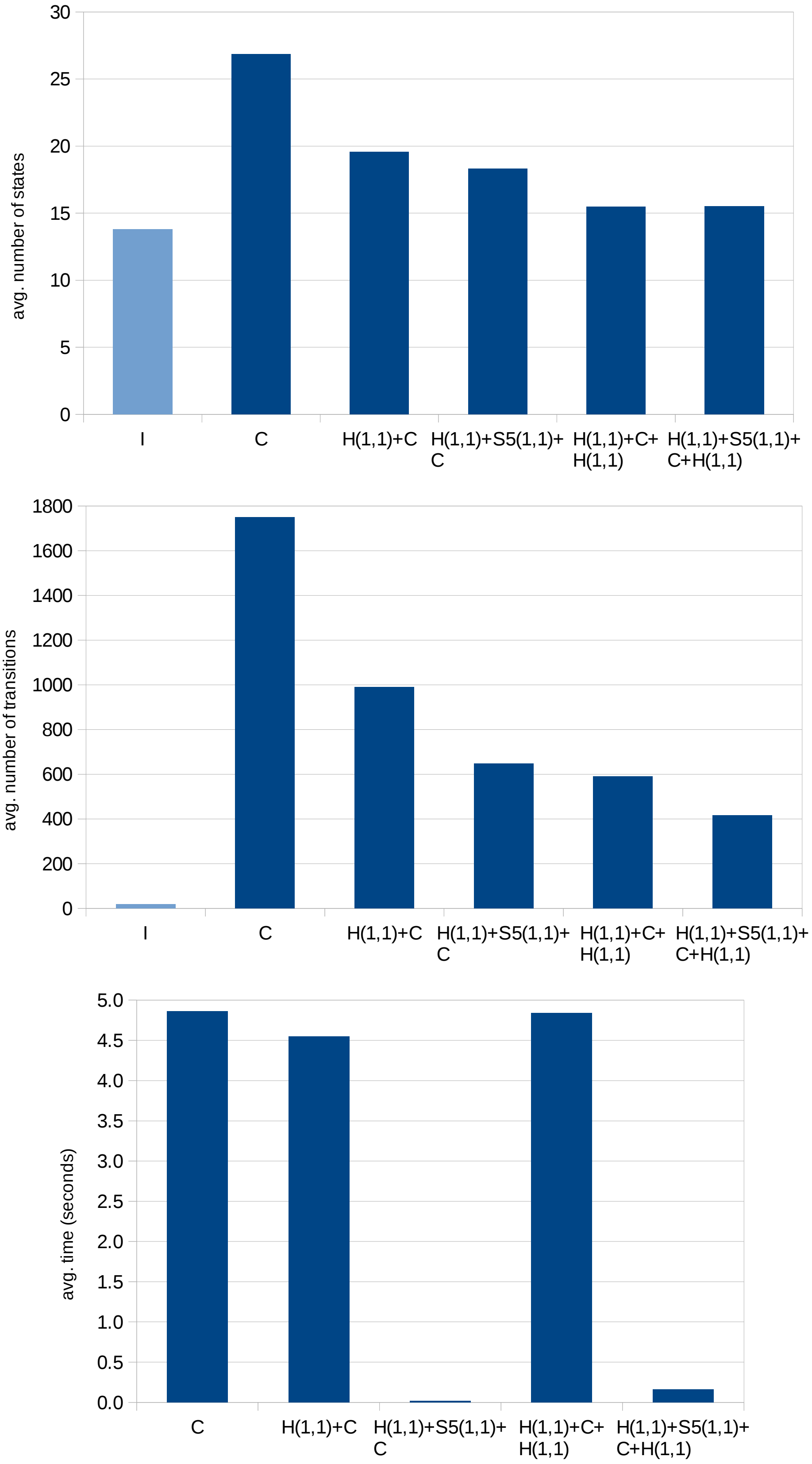


Figure 3.7: Reducing and complementing Forester automata with at most 14 states. The complement automata have fewer transitions and are faster to compute if the complementation is preceded by applying $Heavy(1,1)$ and $Sat5(1,1)$ - H(1,1)+S5(1,1)+C - or just $Heavy(1,1)$ - H(1,1)+C. Applying $Heavy(1,1)$ in the end reduces even more. We include the initial number (I) of states and transitions for comparison purposes.

complex (more than 400 transitions on average). As we can see, applying *Heavy* not only before but also after the complementation on average yielded significantly smaller automata than direct complementation, especially in terms of transitions (less than one third for nearly all values of *td*), while running on similar times (all average times were below 0.1s). Moreover, the nondeterminism introduced in the complement automata by the saturation method did pave the way for further reductions in the state space which were not possible with *Heavy* alone. This came at the cost of higher running times and also of returning automata with more transitions - but with still far less transitions then those obtained with direct complementation. Note that for very dense automata ($td \geq 4.0$), the average size of the complement became particularly small. This is because more than half of the automata generated with such *td* were universal, and thus their complements were empty.

We also tested our algorithms on random automata with 7 states (Figure 3.10), whose complement automata can have, on average, up to 100 states and more than 30,000 transitions. As above, reducing automata with *Heavy* both before and after the complementation returned automata with significantly fewer transitions than direct complementation (3,000 vs 35,000 in some cases), but the former was clearly slower (avg. times up to 90s) than the latter (avg. times up to 2.5s) on the automata region where the difference between the two methods was most drastic. Still, for highly dense automata ($td \geq 4$), direct complementation was responsible for the highest times recorded (avg. times between 135s and 2170s). Due to the size of the complement automata, the saturation methods revealed to be too slow to be viable in this case.

Based on our experimental results, we can conclude that *Sat*5, in general, achieves a greater reduction than *Heavy* alone, but at the expense of yielding automata with possibly more transitions. When performing a computationally hard operation such as complementation, both algorithms play a key role in obtaining smaller complements (in both states and transitions), by achieving reductions both before and after the complementation step.

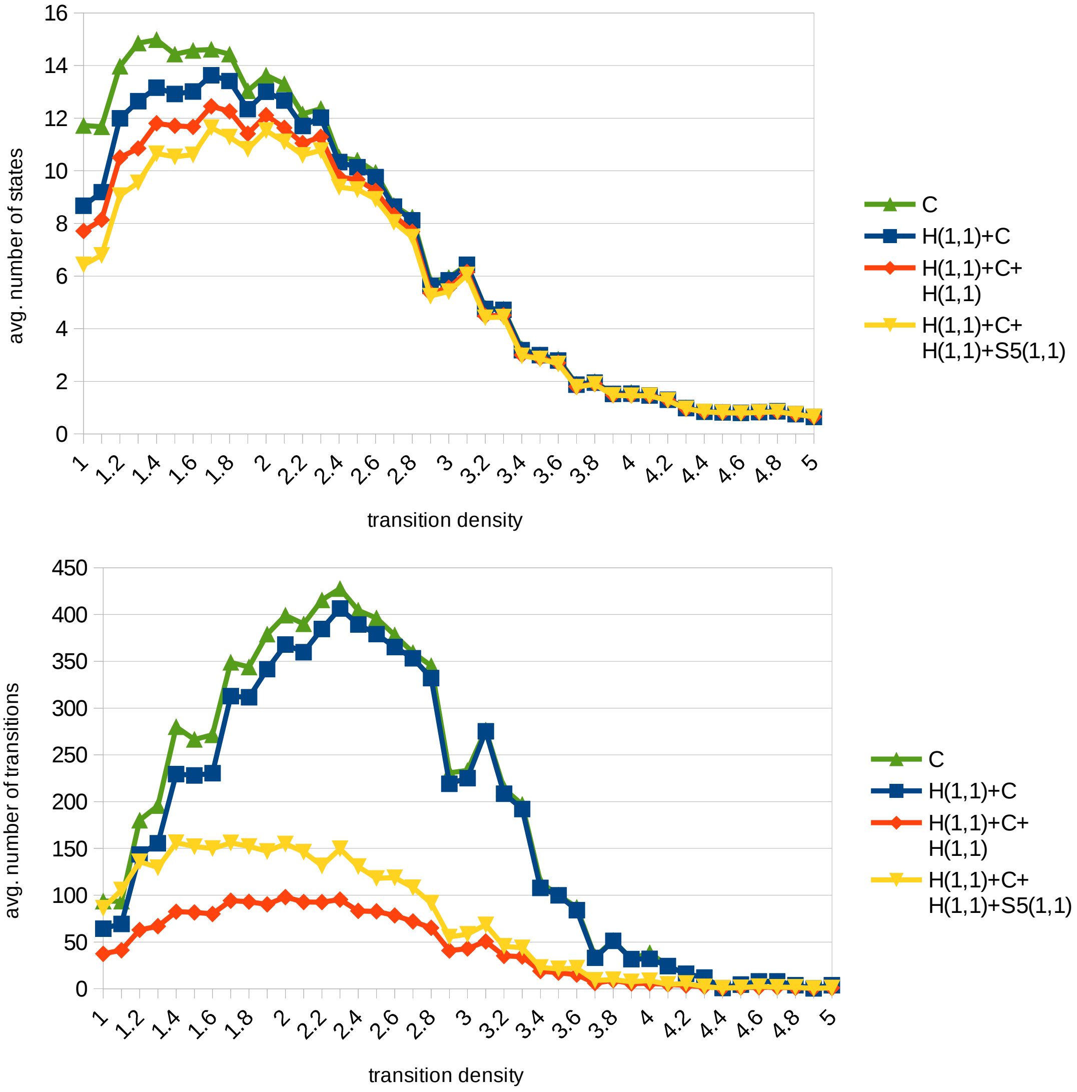


Figure 3.8: Reducing and complementing Tabakov-Vardi random automata with 4 states. Each data point is the average of 300 automata. Applying $Heavy(1,1)$ before the complementation (H(1,1)+C) yielded automata with fewer states and transitions, on avg., than direct complementation (C). When $Heavy(1,1)$ is also used after the complementation - H(1,1)+C+H(1,1), the difference is even more significant, and even more when $Sat5(1,1)$ is used - H(1,1)+C+H(1,1)+S5(1,1).

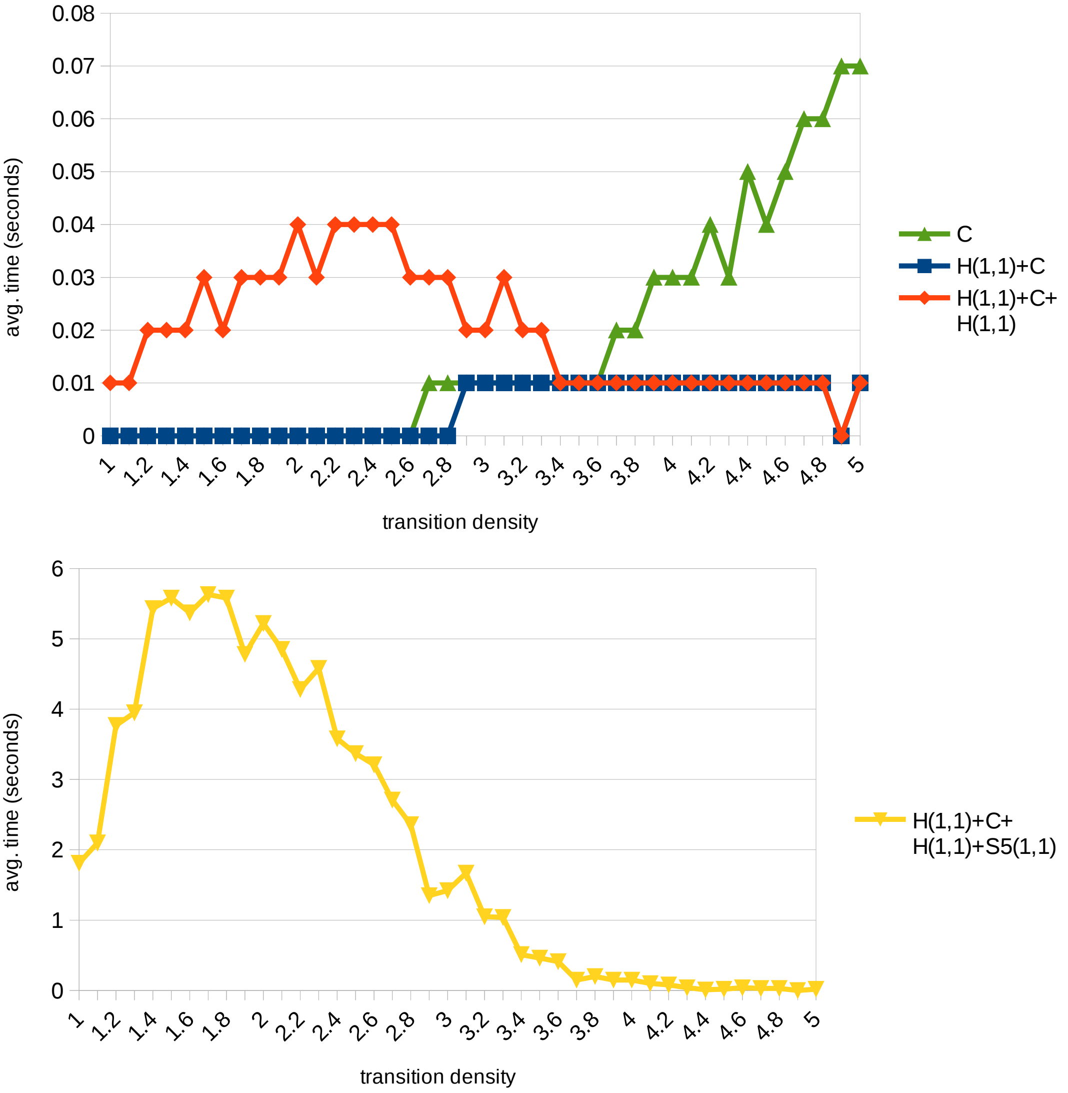


Figure 3.9: The time for reducing and complementing Tabakov-Vardi random automata with 4 states. Each data point is the average of 300 automata. The sample automata were small enough to make the complementation step very fast, whether $Heavy(1,1)$ was used before/after or not (top chart). Applying $Sat5(1,1)$ after the complementation, however, made the overall computation significantly slower except for high values of $td$ (bottom chart), even though this method yielded automata with fewer states, in general.

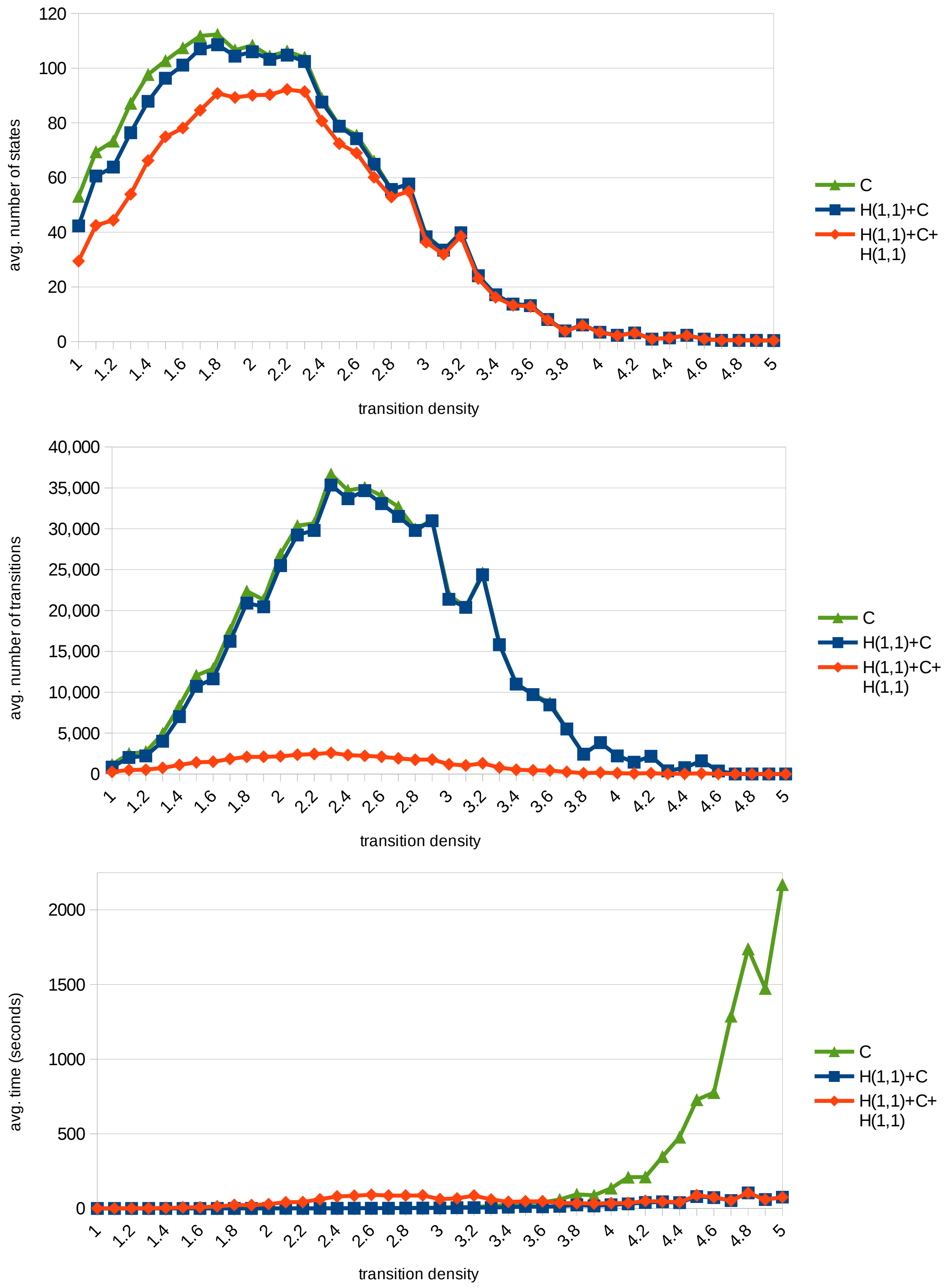


Figure 3.10: Reducing and complementing Tabakov-Vardi random automata with 7 states. Each data point is the average of 300 automata. Applying $Heavy(1,1)$ before the complementation (H(1,1)+C) yields smaller automata than direct complementation (C), on avg. When Heavy is used both before and after the complementation (H(1,1)+C+H(1,1)), the gap is even more significant, but the greater reductions took longer to compute. C still took the longest times recorded, for highly dense automata.

## 3.2 Efficient Computation of Lookahead Simulations

We performed some optimizations on the computation of the maximal downward lookahead simulation used in *Heavy*$(x,y)$. In the following we describe the key aspects of the computation in terms of a downward simulation game between Spoiler and Duplicator. Upward simulation is similar but simpler, since the tree branches downward.

**Fixpoint iteration with incremental moves.** We represent binary relations over $Q$ as boolean matrices of dimension $|Q| \times |Q|$. Starting with a matrix $W$ in which all entries are set to TRUE, the algorithm consists of a downward refinement loop of $W$ that converges to the maximal downward $k$-lookahead simulation. In each iteration of the refinement loop, for each pair $p,q$ where $W[p][q]$ is still TRUE:

- Spoiler tries an attack *atk* consisting of a possible move from $p$ of some depth $d \leq k$. Each such attack is built incrementally, for $d = 1,2,\ldots,k$, in order to give Duplicator a chance to respond already to *atk* or even to a prefix of it.
- Duplicator then attempts to defend against a (possibly non-strict) prefix *atk′* of *atk*, by finding a move *def* from $q$ by the same symbols as in *atk′* s.t. every leaf-state in *def* is in relation $W$ with the corresponding leaf-state in *atk′*. (Duplicator's search is done in depth-first mode.) If successful, Duplicator declares victory against this particular attack *atk* and Spoiler tries a new one, since extending *atk* to a higher depth is pointless. If unsuccessful and $d < k$, Spoiler builds an attack of the next depth level $d+1$, by extending *atk* with one new transition from each of its leaf-states. The extra information might enable Duplicator to find a successful defence then.
- Duplicator fails if he could not defend against an attack *atk* of the maximal depth, either where *atk* has depth $d = k$ or $d < k$ but *atk* cannot be extended any more due to all its leaf-states having no outgoing transitions.
- If Duplicator could defend against every attack (or some prefix of it) by Spoiler then $W[p][q]$ stays true, for now.
- In the worst case, for each Spoiler's attack of depth $d$, Duplicator must search through all defences of depth up-to $d$, but often Duplicator wins sooner.
- Similarly, in the worst case, Spoiler needs to try all possible attacks of depth $k$, but often Duplicator already wins against prefixes of some depth $d < k$.

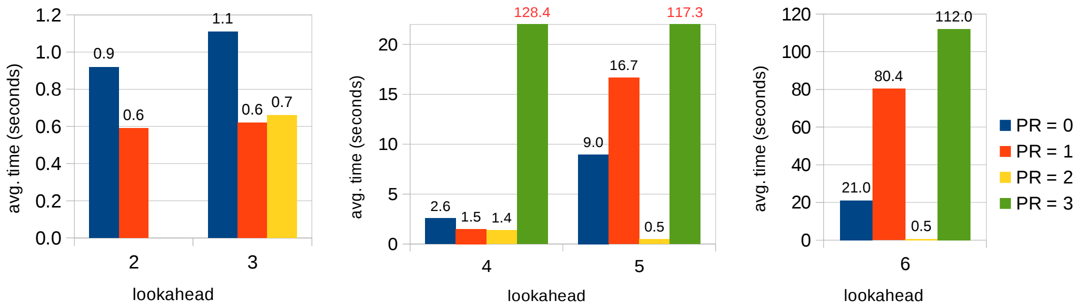
Since the outcome of a local game depends on the values of $W$, the refinement loop might converge only after several iterations. The reached fixpoint represents $\sqsubseteq^{k\text{-dw}}$, that is generally not transitive (for $k > 1$). Since we only use it as a means to under-approximate the transitive trace inclusion relation $\subseteq^{\mathsf{dw}}$ (and require a preorder to induce an equivalence relation), we work with its transitive closure $\preceq^{k\text{-dw}}$.

**An Optimization Based on Pre-Refinement.** Following an approach implemented in `RABIT` [Lan17] for NFA, we under-approximate non-simulation as follows. If there exists a tree of bounded depth $d$ that can be read from state $p$ but not from state $q$, then the pair $(p,q)$ cannot be in $k$-lookahead simulation for any $k$. The pre-refinement step iterates through all pairs $(p,q)$ and sets $W[p][q]$ to `false` if such a tree is found witnessing non-simulation.

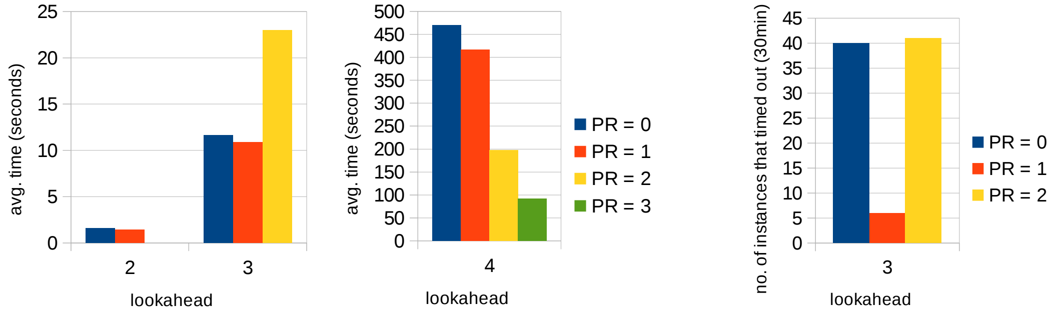
Our experiments show (see Figure 3.11) that, for most automata samples, running a pre-refinement with some modest depth $d$ suffices to speed up the $k$-lookahead downward simulation computation. Moreover, in several samples the computation time dropped to half or less. Our tests included samples described in Section 3.1, namely a subset of the `forester` sample, the `moderate-artmc` and the `nonmoderate-artmc` samples and Tabakov-Vardi randomly generated tree automata with different values of transition density. We can see that it is possible to generalize a recommended value of pre-refinement for each lookahead used, i.e., a depth value $d$ for each lookahead such that pre-refining with $d$ is overall as fast as without it and in some cases significantly faster. We thus use depth 1 for lookahead values of 2 or 3, and depth 2 for higher lookaheads. All the experiments described in Section 3.1 or anywhere else in this thesis were run with this heuristic in practice.

We now present an optimization that allows to compute lookahead simulation faster. The idea is that attacks which are *good* (i.e., successful) or *bad* (i.e., unsuccessful) may be remembered to skip unnecessary re-computations. We defined and tested three different versions that vary in the scope in which an attack is seen as *good* or *bad*. In the description that follows, we use the automata in Figure 3.12 as an example, for a state in the computation of the 3-lookahead downward simulation in which $W = \{(p_5,q_7),(p_5,q_{11})\}$.

**Local caching of Spoiler's attacks.** In this version of the caching, whenever Spoiler's attack uses a transition branching into $n$ sub-attacks, Duplicator tries a transition by the corresponding symbol and memorizes which of the next $n$ states can/cannot defend against the corresponding sub-attack from Spoiler. For example, in a round of the simulation game from $(p_2,q_3)$ in the automata in Figure 3.12, Spoiler is attempt-

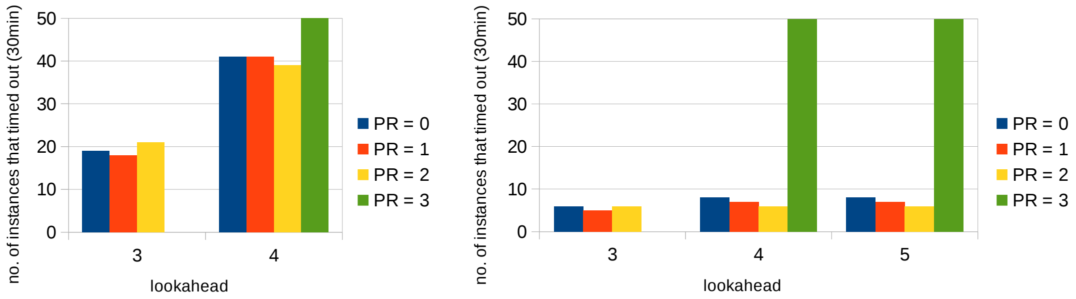


(a) Sample: 100 Forester automata.



(b) Sample: moderate artmc (a total of 27 automata).

(c) Sample: non-moderate artmc (a total of 62 automata).



(d) Sample: 50 random automata with transition density of 4.0.

(e) Sample: 50 random automata with transition density of 5.0.

Figure 3.11: The pre-refinement step speeds up the computation of the downward lookahead simulation preorder. We performed tests with various lookahead values and using both automata from program verification provenience and Tabakov-Vardi random tree automata with 100 states, 2 symbols and acceptance density of 0.8. Note that in several cases the computation time dropped to half or less. For larger/denser automata, we measured the number of automata for which computing the lookahead simulation relation finished in less than 30 minutes.

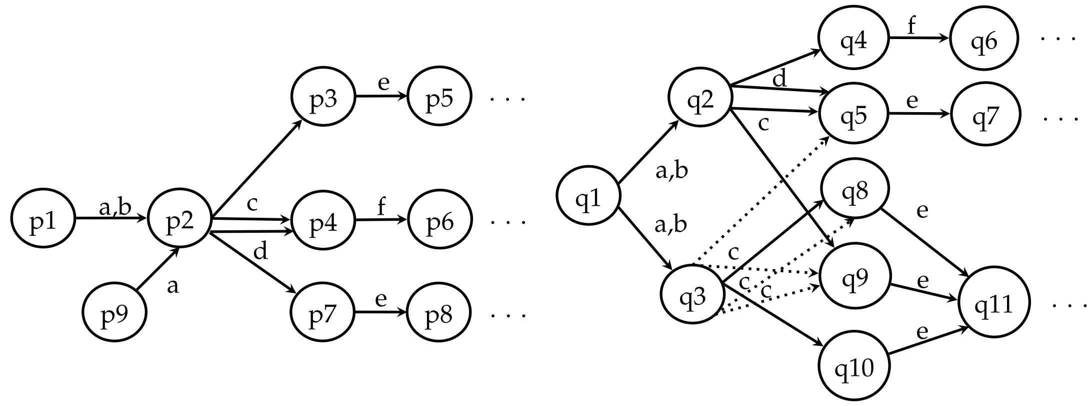


Figure 3.12: For $W = \{(p_5,q_7),(p_5,q_{11})\}$, all versions of the optimization allow some attacks to be skipped when computing the 3-lookahead downward simulation game. We draw dotted transitions from $q_3$ to make the example more legible.

ing the attack $c(e,f)$ leading to $p_5,p_6$. Say that Duplicator tries responding with the $c$-transition to $q_8,q_9$. Since there is a $e$-transition from $q_8$ to $q_{11}$ and $p_5\,W\,q_{11}$, he caches the information that, against $q_8$, the first sub-attack from Spoiler is a *bad* one. Since he succeeded to defend against the first sub-attack, Duplicator now proceeds to try to find a defence from $q_9$ against the second sub-attack. However, as $q_9$ cannot read $f$, Duplicator caches the second sub-attack as a *good* one against $q_9$. But Duplicator does not declare defeat just yet, and he now tries a different $c$-transition from $q_3$, this time leading to $q_8,q_{10}$. He can skip checking if $q_8$ can defend against the first sub-attack as it has been recorded as a *bad* one against $q_8$ already. But since $q_{10}$ cannot defend against the second sub-attack (as it can only read $e$), he caches this attack as a *good* one against $q_{10}$, and proceeds to attempt the last $c$-transition from $q_3$, leading to $q_5,q_9$. From state $q_5$ a successful defence against the first sub-attack is possible, but the second sub-attack was previously recorded as being *good* against $q_9$, thus Duplicator declares defeat immediately without any further exploration.

**Semi-global caching of Spoiler's attacks.** Now for an entire attack from Spoiler we keep record of which sub-attacks are successful/unsuccessful against the states explored so far by Duplicator. Consider the game configuration $(p_1,q_1)$ and that Spoiler is attempting the attack $ac(e,f)$, leading to states $p_5,p_6$. Say that Duplicator first tries to defeat this attack by using the $a$-transition to $q_2$ and then the $c$-transition leading to $q_5,q_9$. Since $q_5$ reads $e$ onto $q_7$ and $p_5\,W\,q_7$, Duplicator records that the sub-attack from $p_3$ is a *bad* one against $q_5$. He thus proceeds to try to find a defence from $q_9$ against the sub-attack from $p_4$, but since $q_9$ can only read $e$, he caches this sub-attack as a *good* one against $q_9$. Since he was not successful, Duplicator now tries to find a defence against the attack by taking the $a$-transition from $q_1$ to $q_3$ followed by the transition to $q_8,q_9$. The state $q_8$ is able to defend against the sub-attack from $p_3$, and he caches this attack accordingly, since both states read just $e$ and $p_5 W q_{11}$. However, he need not attempt a defence from $q_9$ against $p_4$, since such attack has previously been recorded as being *good* against $q_9$ already.

**Global caching of Spoiler's attacks.** Here we expand the scope to the entire $W$-refinement. For example, the *good* attacks from $p_2$ against $q_2$ (or against $q_3$) can be recalled even when playing a game from a different configuration, say, $(p_9,q_1)$. However, the information about the *bad* attacks cannot be used outside of the game in which it was saved, since Duplicator's successful defences can be based on the state of $W$ at each time. Note the asymmetry between *good* and *bad* attacks: *good* attacks remain *good* for the rest of the entire lookahead simulation computation, but *bad* attacks may become *good* after $W$ changes.

The three versions present a trade-off between expressiveness and space required to encode attacks (see Subsection 3.3.3.2 for details of implementation). Our tests indicate that the semi-global version indeed speeds up the computation on automata with high transition overlaps (i.e., where many states are shared by different transitions), such as the `moderate-artmc` sample, and for relatively high lookaheads (3, 4 and 5). Thus, one could first measure the transition overlaps in a given automaton (see Subsection 3.3.2.6 for instructions on how to do this with `minotaut`) and then choose to use the semi-global caching or not depending on the results obtained. However, note that this might not compensate, as the overlaps measure can itself be very time-consuming.

## 3.3 Tool

This section takes a brief look at the tool `minotaut` (**Min**imization **O**f **T**ree **Aut**omata), where the algorithms *Heavy* and *Sat* introduced in Sections 2.6 and 2.9, respectively, are available. The tool was written in C++ and built as an extension of `libvata` [LSV12, LSV17a], a highly optimized library for the manipulation of tree automata, and the source code has been made publicly available [Alm16a]. In Subsection 3.3.1 we describe the tool architecture and the interaction with `libvata`. Subsection 3.3.2 exemplifies how the user can test the algorithms, as well as a diverse collection of other automata operations and analysis methods which have been implemented. Lastly, Subsection 3.3.3 details a few of the relevant aspects of the implementation, namely how moves in the simulation game are stored and extended.

### 3.3.1 Architecture and Overview

The architecture of `minotaut` is illustrated in Figure 3.13. The tool relies on `libvata` for the internal encoding of explicit tree automata and for reading automata from the input, as well as for the following operations on automata: 1) Useless states removal (used in both $Heavy(x,y)$ and $Sat(x,y)$), which is implemented in `libvata` as a state reachability problem, 2) State quotienting (used in both $Heavy(x,y)$ and $Sat(x,y)$), which is based on `libvata`'s method for collapsing states in an automaton given an equivalence relation, and 3) Complementation (in which our reduction algorithms play a key role in obtaining small complement automata and faster, as concluded by our experimental assessment in Subsection 3.1.3), which is provided in `libvata` as an implementation of the difference algorithm [Hos10]. The remaining operations and simulation computations that we use were implemented as part of `minotaut`, except as noted below.

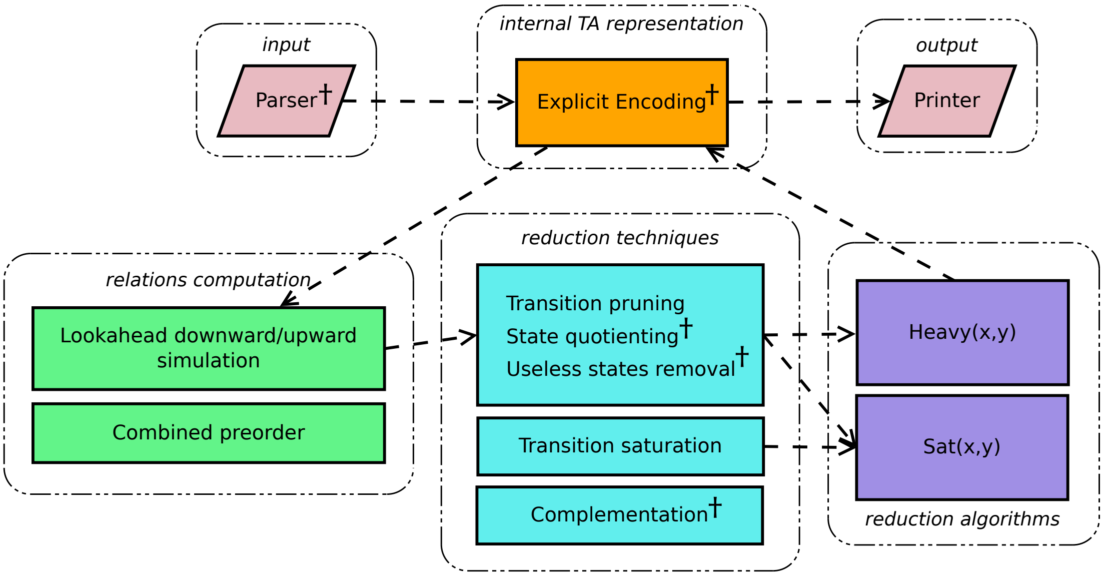


Figure 3.13: The architecture of our tool `minotaut`. Features marked with † are borrowed from `libvata`.

Note that in this thesis tree automata are represented top-down and with a final state made explicit (see Section 2.1). The `libvata` library uses the bottom-up representation and the initial state is left implicit, i.e., transition rules by leaf symbols (also known as *leaf-rules*) have a target state but no source state [ABH$^+$08, LSV12, CDG$^+$08]. Thus, after parsing an input automaton, `minotaut` takes care of artificially adding an explicit initial state from which every leaf rule will now depart. This way our simulation algorithms still adapt naturally to `libvata`'s representation, only inverting the direction considered.

For the computation of the lookahead downward/upward simulations we considered adapting `libvata`'s algorithms for the ordinary simulations (i.e., with lookahead 1). However, due to the particularities of their implementation, which significantly favours efficiency (for instance, a TA is converted to a LTS prior to the computation of the simulation relation), it seemed more feasible to instead follow the approach taken in `RABIT` [Lan17, CM13], which introduces algorithms for computing lookahead simulations for finite (and infinite) words (this approach is detailed in Section 3.2). An adaptation to the bottom-up direction used in `libvata` had to be done as well. Still, for efficiency reasons, the steps in the implementation of $Heavy(x,y)$ that require the computation of ordinary downward simulation are resorted to invoking `libvata`'s method and then making the appropriate translation to our simulation representation.

The combined preorder (see Section 2.7) is a comparatively coarse preorder combining ordinary downward and upward simulations and which is GFQ (i.e., it is language-preserving for quotienting) [AHKV09]. Up until now, it was, to the best of our knowledge, the technique with the greatest reduction ratio, in average, and thus we implemented it for comparison purposes (see our experimental results in Subsection 3.1.1).

The transition pruning technique detailed in Section 2.4 was implemented as a double loop on the transitions of an automaton: each transition is compared to every other transition until a *better* one is found, in which case the first one is pruned; if no *better* transition is found then the transition is not removed. When pruning with a strict partial order $P(R_s, \hat{R}_t)$, a transition is found to be *better* than a different one by performing a membership test in $R_s$ between the two source states and a membership test in $R_t$ between each two target states in the same position.

Although transition saturation (introduced in Section 2.8) is the dual notion to pruning, its implementation was significantly different, given that transitions in the automaton are compared against other transitions which do not exist (yet). Saturating with $S(R_s, R_t)$ is implemented as follows: for each transition $p \xrightarrow{a} p_1 \ldots p_n$, for each $p'$ s.t. $(p, p') \in R_s$ and for each $(p'_1 \ldots p'_n)$ s.t. $(p_1 \ldots p_n)\hat{R}_t(p'_1 \ldots p'_n)$ (i.e., for each combination of states s.t. each $p'_i$ satisfies $(p_i, p'_i) \in R_t$)), the transition $p' \xrightarrow{a} p'_1 \ldots p'_n$ is added to the automaton.

Finally, during the code testing phase, we used `libvata`'s language inclusion test to check that the implementations of our reduction algorithms $Heavy(x,y)$ and $Sat(x,y)$ preserved the language of the automata tested.

### 3.3.2 Interface and Automata Operations Provided

This section describes the usage of the most relevant operations on tree automata that have been implemented in `minotaut`. They can be tested from the command line as exemplified below.

#### 3.3.2.1 Reducing with *Heavy*

The minimize command applies the *Heavy*$(x, y)$ algorithm to each automaton received. The input parameters must be the downward and upward lookaheads ($x$ and $y$) and then a sequence (of 1 or more) automata files (or directories containing automata files) respecting the Timbuk syntax [ea15].
Example:

```
$ ./minotaut minimize −output=my_small_ta_folder −timeout=60 2 4
    moderate−artmc/A0053 forester/B33465936_40

     Automaton 1 − moderate−artmc/A0053:      Q_i: 54  Trans_i: 159
    TransDens_i: 0.20    Q_f/Q_i: 50.00%  Trans_f/Trans_i: 41.51%  TD_f/TD_i:
    83.02%    Time: 0.06s

     Automaton 2 − forester/B33465936_40:     Q_i: 64  Trans_i: 122
    TransDens_i: 0.10    Q_f/Q_i: 90.62%  Trans_f/Trans_i: 93.44%  TD_f/TD_i:
    103.11%   Time: 0.56s

Average results:     Q_i: 59.00   Trans_i: 140.50  TransDens_i: 0.15
    Q_f/Q_i: 70.31%  Trans_f/Trans_i: 67.48%  TD_f/TD_i: 93.06%    Time: 0.31s

The End
```

For each input automaton, minimize starts by printing its initial number of states (Q_i), number of transitions (Trans_i) and transition density (TransDens_i). Then it reduces the automaton with *Heavy*$(x, y)$, outputting the reduction obtained in terms of states (Q_f/Q_i), transitions (Trans_f/Trans_i) and also in terms of transition density (TD_f/TD_i), as well as the time in seconds that *Heavy*$(x, y)$ took to complete. After all input automata have been processed, the average of each measure is printed. The following are a few notes on the options accepted by minimize:

- If instead of giving a list of automata files as input we wished to give a list of directories, each of them containing (only) automata files, we would use the option −input_format=dir. Should we then only be interested in processing $n$ automata files from each directory, the option −max_numb_tests=n could be used.

- All the initial and final measures mentioned above are printed following the corresponding label, which happens by default but could also be made explicit by using the option –output_stat_format=human. Other possible values for this option are: machine, that makes the whole output more 'machine-readable' by omitting the labels in every row; and log, that, when used in combination with –input_format=dir, outputs the individual results for every automaton in a given directory to a dedicated log file, and outputs the average results per directory to an additional log file.
- The command minimize allows us not only to measure how much *Heavy*$(x, y)$ reduces each automaton and how long it takes, but also to save the reduced automaton in a separate file. By using the option –output=my_small_ta_folder in the example above, at the end of the execution two text files containing the reduced automata, A0053_minimized_with_Heavy(2,4) and B33465936_40_minimized_with_Heavy(2,4), are saved in my_small_ta_folder. When used together with the –input_format=dir option, for each input directory a subfolder is created in my_small_ta_folder where all the output automata relative to that directory are saved.
- By default, *Heavy*$(x, y)$ tries to reduce a given automaton for at most 1,800 seconds (30 minutes) and outputs the smallest automaton achieved. We can change the timeout value to any number of seconds by using the –timeout option.
- More advanced options are available as well, regarding measures of the optimizations to the computation of lookahead downward simulation. The depth d used in the pre-refinement step can be fixed using –BranchPR=d. By default, it is given by the heuristic defined in Section 3.2. In order to choose the version of the caching of *good*/*bad* attacks to use, we have the options –HGA_strategy/–HBA_strategy. The accepted values are global, semi–global, local and none, where the latter is the default value. However, as mentioned in Section 3.2, it is worth using the value semi–global when handling automata with high transition overlap (see the command measure_trans_overlap on Subsection 3.3.2.6 below on how to measure the transition overlap in an automaton).

The I/O routine that handles minimize and the other commands described below also makes record of other measures, such as automata reduction in terms of percentage intervals, keeping track of the automaton that suffered the greatest reduction, how many times timeout was thrown and the size of the smallest input automaton that triggered it, etc. However, these measures were left out of the interface for a matter of simplicity.

#### 3.3.2.2 Reducing with *Heavy* and *Sat*

The minimize_with_saturation command applies *Heavy*$(x, y)$ followed by *Sat*$(x', y')$ to each automaton received. The input parameters must be the downward and upward lookaheads used in *Heavy* ($x$ and $y$), the downward and upward lookaheads used in *Sat* ($x'$ and $y'$) and then a sequence (of 1 or more) automata files (or directories containing automata files) written in the Timbuk syntax. The version of *Sat* to be used by default is *Sat5*, but this can be changed by using the option –sat_version=1/2/3/4/5. All of the optional parameters described in the Subsection 3.3.2.1 above for the minimize command can be used with minimize_with_saturation as well.
Example:

```
$ ./minotaut minimize_with_saturation 2 4 1 1 moderate-artmc/A0053
    forester/B33465936_40

    Automaton 1 - moderate-artmc/A0053: Q_f/Q_i: 50.00% Trans_f/Trans_i:
    41.51% TD_f/TD_i: 83.02%    Time: 0.22s

    Automaton 2 - forester/B33465936_40:     Q_f/Q_i: 89.06%
    Trans_f/Trans_i: 95.90% TD_f/TD_i: 107.68%   Time: 2.17s

Average results:     Q_i: 59.00   Trans_i: 140.50 TransDens_i: 0.15
    Q_f/Q_i: 69.53% Trans_f/Trans_i: 68.71% TD_f/TD_i: 95.35%    Time: 1.20s

The End
```

The example above takes as input the same automata as on the previous example for the minimize command. Note how running *Heavy*$(2,4)$ followed by *Sat5*$(1,1)$ allowed to achieve a greater state reduction in the second automaton in comparison with *Heavy*$(2,4)$ alone.

#### 3.3.2.3 Reducing and Complementing

The minimize_and_complement command allows us to measure how much smaller the complement of an input automaton is if it is first reduced with *Heavy*$(x, y)$. As we saw in Section 3.1, for most automata the reduction step significantly speeds up the complementation operation. The two options –applyHeavyInTheEnd=true and –applyHeavyAndSatInTheEnd=true allow us to additionally reduce the complement automaton respectively using *Heavy*$(x, y)$ again and using *Heavy*$(x, y)$+*Sat5*$(x, y)$. The input parameters must be the downward and upward lookaheads ($x$ and $y$) used by *Heavy* (and by *Sat5*, if enabled) and then a sequence (of 1 or more) automata files (or directories containing automata files) respecting the Timbuk syntax.

Example:

```
$ ./minotaut minimize_and_complement −applyHeavyInTheEnd=true
    −applyHeavyAndSatInTheEnd=true −output=minimized_complement_TA 1 1
    small_TA/27.timbuk small_TA/207.timbuk small_TA/64.timbuk

     Automaton 1 − small_TA/27.timbuk: Q_i: 5.00 Trans_i: 13.00  Q_h/Q_i:
    100.00%     Trans_h/Trans_i: 100.00%     Time_h: 0.00s    Q_hc: 17.00
    Trans_hc: 252.00     Time_hc: 0.00s  Q_hch: 15.00     Trans_hch: 93.00
    Time_hch: 0.03s Q_hchs: 14.00     Trans_hchs: 275.00  Time_hchs: 2.88s
    Q_c: 17.00   Trans_c: 252.00 Time_c: 0.00s

     Automaton 2 − small_TA/207.timbuk: Q_i: 5.00     Trans_i: 13.00
    Q_h/Q_i: 0.00%   Trans_h/Trans_i: 0.00%   Time_h: 0.00s    Q_hc: 2.00
    Trans_hc: 3.00   Time_hc: 0.00s  Q_hch: 2.00 Trans_hch: 3.00 Time_hch:
    0.00s Q_hchs: 2.00     Trans_hchs: 3.00     Time_hchs: 0.00s     Q_c:
    17.00  Trans_c: 235.00 Time_c: 0.00s

     Automaton 3 − small_TA/64.timbuk: Q_i: 5.00 Trans_i: 13.00  Q_h/Q_i:
    40.00% Trans_h/Trans_i: 15.38% Time_h: 0.00s    Q_hc: 3.00   Trans_hc:
    6.00   Time_hc: 0.00s  Q_hch: 3.00 Trans_hch: 6.00 Time_hch: 0.00s
    Q_hchs: 3.00     Trans_hchs: 6.00     Time_hchs: 0.01s     Q_c: 15.00
    Trans_c: 163.00 Time_c: 0.00s

Average results:     Q_i: 5.00    Trans_i: 13.00  Q_h/Q_i: 46.67%
    Trans_h/Trans_i: 38.46% Q_hc: 7.33   Trans_hc: 87.00 Time_hc: 0.00
    Q_hch: 6.67 Trans_hch: 34.00     Time_hch: 0.01  Q_hchs: 6.33
    Trans_hchs: 94.67    Time_hchs: 0.96 Q_c: 16.33   Trans_c: 216.67 Time_c:
    0.00

The End
```

For each automaton, the output results can be read as follows: the measures on the initial automaton (e.g., `Q_i`); the measures of the reduction (in percentage) observed after running *Heavy* (e.g., `Q_h/Q_i`) together with how long it took (`Time_h`); the absolute number of states and transitions of the complemented reduced automaton (e.g., `Q_hc`), followed by how long *Heavy* and the complementation altogether took to compute (`Time_hc`); the measures of the complement automaton reduced with $Heavy(x,y)$ (e.g., `Q_hch`); the measures of the complement automaton reduced with $Heavy(x,y)+Sat5(x,y)$ (e.g., `Q_hchs`); and finally, the measures of the automaton obtained by directly complementing the input automaton (e.g., `Q_c`).

By using `−output=minimized_complement_TA`, we keep, for each input automaton, the last reduced version computed of its complement (in this case, the result of

$Heavy(x,y)+Sat5(x,y)$) in the minimized_complement_TA folder. All the other optional parameters described above for the previous commands can be used here as well.

#### 3.3.2.4 Measuring the Size

The measure_size command receives an input automaton and simply outputs its number of states (Q), number of transitions (Trans) and transition density (TransDens).

#### 3.3.2.5 Measuring the Nondeterminism

The measure_non_det command receives an input automaton and calculates two different measures of non-determinism (ND):

- ND_states - the number of states (in percentage) from which there are nondeterministic transitions (top-down), i.e., at least two transitions with the same symbol.
- ND_trans - the number of transitions (in percentage) for which there is another transition with the same source state (top-down) and with the same symbol.

This command also accepts the optional parameter –output_stat_format=human/machine which was described already in Subsection 3.3.2.1.

#### 3.3.2.6 Measuring the Transition Overlap

Two transitions are said to overlap if they have the same source state, symbol and at least one target state in common and appearing in the same position. The measure_trans_overlap command receives an input automaton and calculates three different measures of transition overlap (TOL):

- TOL_1 - the number of transitions (in percentage) that overlap with some other.
- TOL_2 - an average of, for each transition that overlaps, how many of its target states (in percentage) are also target states of a different transition with the same source state and symbol and preserving their respective positions.
- TOL_3 - an average of, for each transition that overlaps, how many times each of its target states appears at the same position in some other transition with the same source state and symbol.

Example:

```
$ ./minotaut measure_trans_overlap forester/B33465936_40
    TOL_1: 60.00%    TOL_2: 75.97%    TOL_3: 5.22
```

This command also accepts the optional parameter `–output_stat_format=human/machine` which was described already in Subsection 3.3.2.1.

## 3.3.3 Implementation Details

### 3.3.3.1 Attacks and Defences in the Simulation Game

As we have seen in Section 2.3, the computation of simulation relations can be characterized by a game between players Spoiler and Duplicator. In each configuration of the game, Spoiler and Duplicator alternately make moves in the automaton, which we refer to as attacks and defences, respectively. In the case of lookahead simulation, these moves may be incremented throughout the gameplay.

The moves played by Spoiler and Duplicator can be seen as tree structures where each node contains not only a symbol but also the state from which the symbol is read by the respective player. In the downward simulation game, when a move is extended downwards, for every `Step node` which is currently a leaf of the move, `node.children` is assigned a vector of newly-created `Step` nodes corresponding to states reachable from `node.p` by some transition.

The fields `parent` and `index` are only used in the upward simulation game and aid in the extension of a move upwards. In each `Step node` in the move, `node.parent` is used to point to the node of which `node` is a child (and so `node.parent` is set to `NULL` only when `node` is the root of the move). And for any `node` which is not the root, `node.index` contains the position at which `node` appears in the `children` vector of its parent node. This significantly speeds up Duplicator's exploration of which transitions can be used to extend his defence upwards: if `Step node_S` and `Step node_D` are nodes at the same depth corresponding, respectively, to moves of Spoiler and Duplicator, then in order to respond to the transition taken upwards from `node_S.p` (if `node_S` is not the root of Spoiler's attack), Duplicator need only consider transitions (by `node_S.s`) in which `node_D.p` appears as the target state at position `node_S.index`.

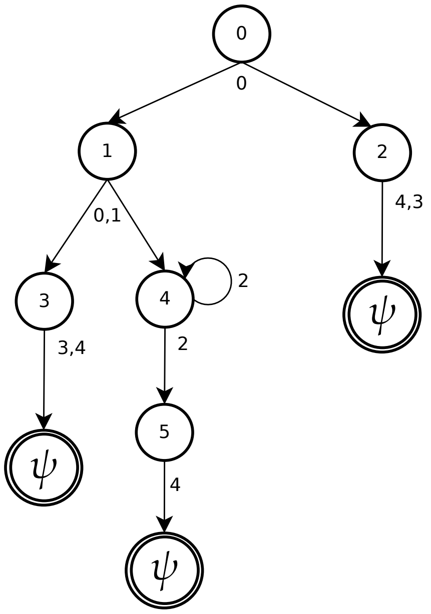
Finally, as we will see in Subsection 3.3.3.2, the fields `code` and `node` are used as keys for caching attacks as *good* or *bad*.

```
class Step
{
    int p;
    int s;
    vector<Step> children;
    Step* parent;
    int index;
    code c;
    node v;
}
```

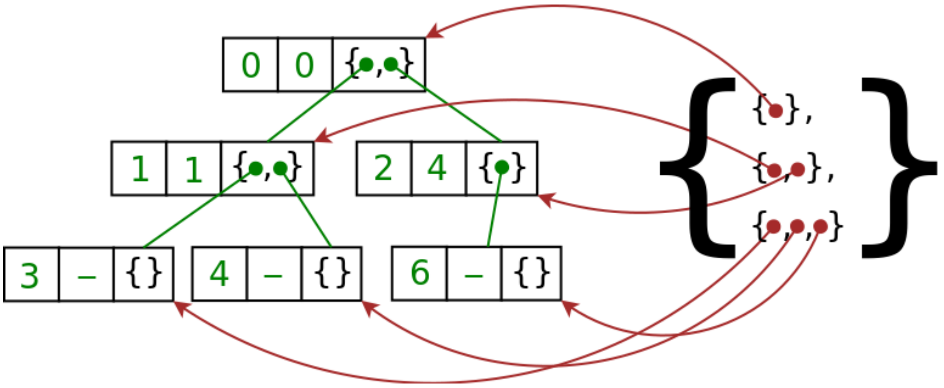
Listing 3.1: A simplified version of the class `Step`.

Let us consider an example, where Spoiler and Duplicator play the 3-lookahead downward simulation game in the automaton in Figure 3.14(a). Both states and symbols are represented by numbers to ease the parallelism between game definition and its implementation, and the special final state $\psi$ is always given the greatest index in the state space. Let us say that Spoiler plays from state 1 and, at a certain point in the computation, he tries the attack based on the open tree $0(1,4)$. The `Step` object `atk` in Figure 3.14(b,left) represents the structure of this attack.

Consider now that Duplicator is able to defend against the attack `atk` launched by Spoiler. The rules of the simulation game dictate that Spoiler must then extend his attack to depth 3. This means continuing each leaf node in `atk` with some possible new transition and launch the attack again (note that, as is the case in this scenario, some leaf nodes may be at $\psi$ already and therefore cannot be extended with a transition). Instead of traversing down `atk` from its root to the leaves every time it needs to be extended, we use an auxiliary structure that contains shortcuts to each and every `Step` in the attack. It consists of a matrix `mtx` of pointers where each row $i$ contains pointers to the nodes of depth $i$, from left to right, of `atk`. The instantiation of `mtx` for `atk` can be seen in Figure 3.14(b,right). The computation now proceeds by generating on-the-fly every possible combination of transitions from each leaf `Step` and extending `atk` accordingly (updating `mtx` with a new row of pointers) until one such extended attack reveals to be successful.



(a) An automaton where Spoiler and Duplicator play the 3-lookahead downward simulation game. Spoiler launches attacks from 0.



(b) Spoiler's attack over symbols 0(1,4) is represented by the `Step` object `atk` (left). Some field values of the class `Step` have been omitted since they are not needed here. A matrix `mtx` (right) stores pointers that allow immediate access to nodes in every level of `atk`.

Figure 3.14: The class `Step` implements a tree-structure which is used to store attacks and defences in the simulation game, and a matrix of type `vector<vector<Step*>>` aids significantly in the step of incrementing an attack. In this example, (b) presents a possible instantiation of the two structures during a 3-lookahead simulation game on the automaton in (a).

#### 3.3.3.2 Caching Good and Bad Attacks

In Section 3.2 we introduced an optimization to the computation of the downward lookahead simulation game in which Duplicator can skip computing a defence against an attack from Spoiler if it has previously been cached as a *good* or *bad* attack. The way attacks are encoded and stored varies significantly depending on which version of the caching is being used: global, semi-global or local.

**Global caching of attacks.** This version of the caching stores the attacks from Spoiler in a matrix where each entry is a set containing encodings of attacks. We use two different structures, `historyGA` and `historyBA` (for storing the *good* attacks and the *bad* ones, respectively) of type `vector<vector<set<code>>>`, where `code` is the type used to encode an attack, as detailed below. The intuition is that, at a given point in the computation, the set in each `historyGA[p][q]` (or `historyBA[p][q]`) contains the attacks from `p` which are *good* (or *bad*) against `q`. Thus, for each attack launched by Spoiler, Duplicator first checks if it is already present in one of these two structures. If it is not, he then tries to build a defence against it. If Duplicator is successful, he inserts the attack in the set at the appropriate entry of `historyBA`. If he is not, he adds it to `historyGA`. We experimented both using the ordered `std::set` implementation for sets and using the `std::unordered_set` implementation, but the former had a faster performance than the latter. This is explained by the fact that, for ordered sets, both insert and look-up operations are of logarithmic complexity in the size of the structure, while in the unordered case they are linear in the worst case [C:s, C:ub].

As explained in Section 3.2, in the global version of the caching, *good* attacks remain *good* for the rest of the lookahead simulation computation (i.e., this information can be reused even when computing the game from a different configuration $(p,q)$), while *bad* attacks may become *good* afterwards. This happens since, when entries in the current over-approximation of lookahead simulation ($W$) change from `true` to `false`, defences previously built by Duplicator may no longer be valid. Thus, in the current version of this optimization, at the end of each game, the structure `historyBA` is replaced by a fresh one with no information cached. A possible improvement to this optimization could be to clean up `historyBA` completely or just partially after some relevant entries in $W$ change. For example, whenever an entry $W[p'][q']$ changes from `true` to `false`, one could remove from each set `historyBA[p][q]` those attacks that visit $p'$. This, of course, would be expensive to compute and only by implementing and testing it could we conclude if it compensates in terms of overall running time or not.

In the current implementation, `code` is used as an alias for `string`. The encoding of an attack is a sorted representation of the states and symbols that it contains.

Consider an attack from Spoiler, represented as the object `atk` of the class `Step` (see Subsection 3.3.3.1). For every node `n` in `atk`, the encoding of the (sub-)attack with root at `n` is generated by the method `genCode()` as follows:

- If `n` is a leaf-node in `atk` (i.e., if `n.children` = {}), then `n.genCode() = n.p + ''.'' + n.s` (the `''.''` character is used as a separator between states and symbols), where `n.p` and `n.s` are converted to `string`.
- If `n` is not a leaf node, then `n.genCode() = n.p + ''.'' + n.s + ''.'' + n.children[0].genCode() + ''.'' + n.children[1].genCode() + ''.'' + ... + ''.'' + n.children[rank(s)].genCode()`, i.e., the code of each child node of `n` is included in the code of `n`.

The `string` concatenation operation (represented above as +) in general has linear complexity on the length of the string [C:c]. Using an integer type for `code`, even in its longer representations, would easily result in overflow errors when caching attacks over large automata. The code is generated once for each attack that Duplicator receives from Spoiler. Thus, due to it being invoked very often, the method `genCode()` described above is defined as an inline function to decrease the execution time of the program [C:i]. A natural alternative to generating the code of an attack (or sub-attack) every time it is required is to generate the code of each node `n` once and store it in the field `n.c` of type `code`. However, when an attack is extended by Spoiler, the codes of the already existing nodes are no longer valid. Since there is no trivial way to update the code of a node in an attack after it has been extended, this field had to be reset in every node after each extension of the attack. Thus, the experimental evaluation we performed revealed that this store-and-reset alternative brought no benefit in terms of running time.

We attempted yet another alternative to the caching of attacks, consisting of recording not the encoding of an attack but the attack itself. The structures `historyGA` and `historyBA` were thus of type `vector<vector<set<Step>>>`. Since `Step` is a user-defined class, this required us to overload the less-than operator, which is the comparison object used to sort the elements of a `set` internally in C++ [C:l]. For a matter of consistency and good programming practice, we overloaded the remaining binary infix comparison operators as well, despite them not being necessary in the current implementation. Still, in our experimental evaluation we concluded that this method is slower than the alternatives based on encoding the attacks.

**Semi-global caching of attacks.** The semi-global version stores both *good* and *bad* attacks in a single structure `historyA` of type `vector<map<node,bool>>`, i.e., a vector of maps from `node` (the type used to represent an attack, as described below) to `bool`. The intuition is that, at any point of the computation, each `historyA[p]` contains those attacks which are known to be *good* against `p` (in which case the mapped value will be `true`) and those which are known to be *bad* (`false`). Comparing to the global version of the caching described above, this version has the advantage that we only need to make one single look-up on the structure to tell if computing a defence against the given attack is necessary or not. As in the global version, Duplicator then caches an attack as either *good* or *bad* after he finds that he is not able or is able to build a defence against it from `p`. Unlike in the global version, however, all information is reset when Spoiler attempts a new attack.

Similarly to the case with sets, using the ordered C++ `std::map` implementation, as opposed to `std::unordered_map`, was more efficient in terms of running time [C:m, C:ua].

The type `node` is used to represent cached attacks and, as with the type `code` in the global caching, it is used as an alias for `string` in the current implementation. As explained in Section 3.2, for a given attack from Spoiler, the semi-global caching records which sub-attacks in Spoiler's move have already been seen as being *good* or *bad* against the states explored so far by Duplicator. Since attacks have a tree-structure, sub-attacks can be seen as subtrees, and so we cache them referring to their root node in the bigger tree of the attack. This node representation is stored in the field `node v` of a `Step` object, and its construction follows the definition of a node $v$ in the domain of a tree $t$ as a sequence of integer numbers (cf. Section 2.1), where the empty sequence $\varepsilon$ is represented by the empty string `‘‘ ’’`. In any attack, the root node `r` is constructed with `r.v = ‘‘ ’’` by default. For every other node `n`, its code `n.v` need only be constructed once and when `n` is being added as a new leaf node of the attack during an extension: if `n’` is the parent node of `n`, then `n.v = n’.v + i`, where `i` is the index of `n’.children` such that `n’.children[i] = n`.

The fact that the `v` field is trivial to obtain and is computed only once for each node, and that inspecting if the attack has been cached as good/bad requires a single look-up, certainly help to explain why this version of the caching outperformed the global one in terms of running time in our experiments.

**Local caching of attacks.** As we saw in Section 3.2, in this version of the caching, whenever Spoiler's attack uses a transition branching into $n$ sub-attacks, Duplicator tries a transition by the corresponding symbol and memorizes which of the next $n$ states can/cannot defend against the corresponding sub-attack from Spoiler. Our implementation uses two separate matrices, `historyGA` and `historyBA`, of type `vector<vector<bool>>` where all entries are initialized to `false`, to cache *good* and *bad* attacks, respectively. The intuition is that, at any point of the computation, and for a given attack where `Step n` is the root node, if `historyGA[q][i]` (or `historyBA[q][i]`) is set to `true` then we have previously concluded that, from the state `q`, it is not possible (or it is possible) to defend against the sub-attack starting at `n.children[i]`. If both `historyGA[q][i]` and `historyBA[q][i]` are set to `false`, then Duplicator tries to compute a defence and sets either `historyBA[q][i]` or `historyGA[q][i]` to `true`, depending on whether he was successful or not. Due to the locality of this version of the caching, all information is reset when Spoiler proceeds to try a different transition in his attack.

## 3.4 Conclusion

In this chapter we detailed an exhaustive experimental evaluation of the reduction algorithms *Heavy* and *Sat* for nondeterministic finite tree automata, introduced in Chapter 2. As we could see in Section 3.1, *Heavy* performs very well in practice, yielding substantial reductions on automata from model checking and shape analysis, as well as on different classes of randomly generated automata. In the previous chapter we saw that quotienting with the combined preorder cannot achieve any further reduction on automata which have already been reduced with *Heavy*. In the experimental evaluation described in this chapter, this quotienting method was also significantly outperformed by *Heavy* in most automata samples, either in terms of reduction achieved or of running time.

The second reduction algorithm, *Sat*, alternates between *Heavy* and our transition-saturation techniques until a fixpoint is reached. Based on our experiments, it is easy to see that the nondeterminism introduced by the saturation techniques in the reduced automata does indeed, on average, pave the way for further reductions which are not possible with *Heavy* alone. This translates into not only more states being quotiented but also, in several cases, more transitions being pruned afterwards. Moreover, we saw how both *Heavy* and *Sat* can make hard computations like complementation much more tractable, allowing both to obtain smaller complements and at much lower computation times.

# Chapter 4

# Infinite Tree Automata

Infinite trees have been a subject of study for several decades now. The first formulation of automata on infinite trees was made by Rabin in the field of mathematical logic, namely for deciding the monadic second-order (MSO) theory of two successor functions (the infinite binary tree) [Rab69].

Automata on infinite trees have also been applied to solve program verification tasks. For instance, Vardi has provided an automata-theoretic framework for the specification and verification of concurrent and nondeterministic programs [Var91]. In the case of words (i.e., linear trees), Büchi language inclusion has applications in procedures for checking program termination [HHP14]. Language inclusion is thus a theoretical problem that verification tasks are often resorted to, as one can encode both a program and its specification as automata and then test language inclusion between the two to check the desired property.

However, language inclusion is a hard problem in automata theory: it is PSPACE-complete for words and EXPTIME-complete for trees. A common strategy to work around this consists of first making the automata smaller in size so that they become more tractable to test for inclusion. In this chapter we introduce new and efficient reduction techniques for Büchi tree automata.

In the following we present the outline of this chapter. Note that some of the main ideas presented adapt naturally from the case of finite trees, thus many definitions and proofs of theorems follow closely the ones presented in Chapter 2.

**Chapter outline.** We start by presenting the preliminary definitions for infinite trees and infinite tree automata in Section 4.1. In Section 4.2 we formalise the notions of downward/upward simulation and trace inclusion preorders on the states of an automaton on infinite trees. As in the case of infinite words, downward relations (simulations and trace inclusions) on infinite trees offer a richer scenario than in the finite case, as one can speak not only of direct downward relations but also of delayed and fair simulations/trace inclusions. Since trace/language inclusion is `EXPTIME`-complete for trees, we describe methods to compute good approximations in polynomial time, by generalizing lookahead simulations [CM13] to infinite trees. Moreover, we explore yet another under-approximation of downward trace inclusion, called downward fixed-tree simulation, which has previously been defined for words [Cle11]. Downward fixed-tree simulations too can be defined as a direct, delayed or fair simulation.

Section 4.3 is dedicated to the technique of state quotienting, according to which some states in the automaton may have the same *behaviour* and therefore can be merged into a single state. We show which upward and direct/delayed/fair relations are or are not suitable for quotienting states in an automaton. In particular, we show that delayed downward fixed-tree simulation is suitable for quotienting.

Finally, in Section 4.4 we describe the technique of transition pruning for the case of infinite tree automata. We show how the relations defined in this chapter can be used to find transitions in an automaton which may be deleted because *better* ones remain. We provide a complete picture of which combinations of upward and direct/delayed/fair downward relations are suitable for transition pruning.

## 4.1 Infinite Trees and Büchi Tree Automata

**Infinite trees.** A *ranked alphabet* $\Sigma$ is a finite set of symbols together with a function $\# : \Sigma \to \mathbb{N}$. For $a \in \Sigma$, $\#(a)$ is called the *rank* of $a$. For $n > 0$, we denote by $\Sigma_n$ the set of all symbols of $\Sigma$ which have rank $n$.

We define a *node* as a sequence of elements of $\mathbb{N}$, where $\varepsilon$ is the empty sequence. For nodes $v, v', v'' \in \mathbb{N}^*$, if $v = v'v''$ we say that $v'$ is a *prefix* of $v$ and we write $v' \sqsubseteq v$, and if $v'' \neq \varepsilon$ then $v'$ is a *strict prefix* of $v$ and we write $v' \sqsubset v$. The length or depth of a node $v$ is denoted by $|v|$ and we define that $|\varepsilon| = 0$ and $|v \cdot v'| = 1 + |v'|$, where $v \in \mathbb{N}$ and $v' \in \mathbb{N}^*$. For a node $v \in \mathbb{N}^*$, we define the $i$-th child of $v$ to be the node $vi$, for some $i \in \mathbb{N}$. Given a ranked alphabet $\Sigma$, a *tree* over $\Sigma$ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$ such that for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $vi \in dom(t)$ then **(1)** $v \in dom(t)$ and **(2)** $\#(t(v)) \geq i$.

Note that the number of children of a node $v$ may be smaller than $\#(t(v))$. In this case we say that the node is *open*. Nodes which have exactly $\#(t(v))$ children are called *closed*. A tree is closed if all its nodes are closed, otherwise it is open. By $\mathbb{C}(\Sigma)$ we denote the set of all closed trees over $\Sigma$ and by $\mathbb{T}(\Sigma)$ the set of all trees over $\Sigma$. A tree $t$ is *linear* iff every node in $dom(t)$ has at most one child.

The *subtree* of a tree $t$ at $v$ is defined as the tree $t_v$ such that $dom(t_v) = \{v' \mid vv' \in dom(t)\}$ and $t_v(v') = t(vv')$ for all $v' \in dom(t_v)$. A tree $t'$ is a prefix of $t$ iff $dom(t') \subseteq dom(t)$ and for all $v \in dom(t')$, $t'(v) = t(v)$.

We note that while closed trees are defined as in the finite case (see Chapter 2), here we have that every node of a closed tree must have infinitely many successors, since all symbol ranks are larger than 0. Thus a closed tree is necessarily infinite (i.e., it has an infinite number of nodes), although an infinite tree is not necessarily closed.

Given a tree $t \in \mathbb{T}(\Sigma)$, we say that $\rho \subseteq dom(t)$ is a *path* of $t$ iff 1) $\varepsilon \in \rho$, 2) if $v \in \rho$, then $vi \in \rho$ for exactly one $i : 1 \leq i \leq \#(t(v))$, and 3) if $vi \in \rho$ then $v \in \rho$. It follows that, for any pair of distinct nodes $v$ and $v'$ in $\rho$, either $v$ is a prefix of $v'$ or vice versa. Intuitively, a path $\rho$ of a tree $t$ is a set containing the nodes of $t$ visited during a possible traversal of the tree, starting from the root node and choosing at each node *one* of its children nodes to visit next. Thus in general a path $\rho$ of a tree $t$ is not unique, unless $t$ is linear. We denote by $t|_\rho$ the sequence of symbols of $t$ visited by the path $\rho$. It is easy to see that if $t$ is closed, then $t|_\rho$ is necessarily infinite.

**Büchi tree automata.** We consider nondeterministic Büchi tree automata (BTA) which read closed trees downwards starting from their roots. A BTA is a quintuple $A = (\Sigma, Q, \delta, I, F)$ where $Q$ is a finite set of states, $I \subseteq Q$ is a set of initial states, $\Sigma$ is a finite ranked alphabet, $\delta \subseteq Q \times \Sigma \times Q^+$ is a finite set of transition rules and $F \subseteq Q$ is a set of accepting states. The transition rules satisfy that if $\langle q, a, q_1 \ldots q_n \rangle \in \delta$ then $\#(a) = n$. To make further definitions and proofs simpler, we consider only tree automata which are complete (i.e., from every state there exists a transition by each symbol).

A *run* of $A$ over a tree $t \in \mathbb{T}(\Sigma)$ (or a $t$-run in $A$) is a partial mapping $\pi : \mathbb{N}^* \to Q$ such that $v \in dom(\pi)$ iff either $v \in dom(t)$ or $v = v'i$ where $v' \in dom(t)$ and $i \leq \#(t(v'))$. Further, for every $v \in dom(t)$, there exists a rule $\langle q, a, q_1 \ldots q_n \rangle$ such that $q = \pi(v)$, $a = t(v)$, and $q_i = \pi(vi)$ for each $i : 1 \leq i \leq \#(a)$. A *leaf of a run* $\pi$ over $t$ is a node $v \in dom(\pi)$ such that $vi \in dom(\pi)$ for no $i \in \mathbb{N}$. We call it *dangling* if $v \notin dom(t)$. Intuitively, the dangling nodes of a run over $t$ are all the nodes which are in $\pi$ but are missing in $t$ due to it being open. Notice that dangling leaves of $\pi$ are children of open nodes of $t$. The prefix of depth $k$ of a run $\pi$ is denoted $\pi_k$. Runs over closed trees are necessarily infinite.

For every tree $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, let $level_i(\pi)$ be the tuple of states that $\pi$ visits at depth $i$ in the tree, read from left to right. Formally, let $\langle v_1, \ldots, v_n \rangle$, with each $v_j \in \mathbb{N}^i$, be the set of all tree positions of depth $i$ s.t. each $v_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = \langle \pi(v_1), \ldots, \pi(v_n) \rangle \in Q^n$. We say that $st \in Q^*$ is a subtuple of $level_i(\pi)$, and write $st \leq level_i(\pi)$, if all states in $st$ also appear in $level_i(\pi)$ and in the same order.

We write $t \overset{\pi}{\Longrightarrow} q$ to denote that $\pi$ is a $t$-run of $A$ such that $\pi(\epsilon) = q$. We use $t \Longrightarrow q$ to denote that such run $\pi$ exists. A run $\pi$ is initial if $t \overset{\pi}{\Longrightarrow} q \in I$. In the related literature, runs are sometimes treated as being necessarily initial, but here we make this distinction for clarity of further definitions and proofs. Given a run $\pi$ over $t$ and $\rho$ a path of $t$, we denote by $\pi|_\rho$ the restriction of $\pi$ to $\rho$. Since $\pi|_\rho$ has a linear structure, we can represent it by $\pi|_\rho = \pi(v_0)t(v_0)\pi(v_1)t(v_1)\pi(v_2)\ldots$, where $v_i \in \rho$ and $|v_i| = i$. We can talk of a prefix $\chi$ of $\pi|_\rho$, where the length is given by the length of the sequence $t(v_0)t(v_1)\ldots$ of symbols being visited by $\chi$: $|\pi(v_0)| = 0$ and $|\pi(v_0)\ldots t(v_{n-1})\pi(v_n)| = n$.

We say that the run $\pi$ of a tree $t$ is *fair* iff, for any path $\rho$ of $t$,

$$\{q' \in Q \mid \pi(v) = q' \text{ for infinitely many } v \in \rho\} \cap F \neq \emptyset.$$

Informally, $\pi$ is fair iff every path of the tree has infinitely many nodes labelled with an accepting state. The *downward language of a state* $q$ in $A$ is the set of all closed trees that can be read from $q$ by fair runs. Formally, it is defined by

$$D_A(q) \;=\; \{t \in \mathbb{C}(\Sigma) \mid \text{there exists a fair } t\text{-run } \pi \text{ s.t. } t \overset{\pi}{\Longrightarrow} q\}.$$

In other words, a closed tree $t$ can be read from $q$ iff every path of $t$ has infinitely many nodes labelled with an accepting state by some $t$-run. The *language* of $A$ is thus defined by $L(A) = \bigcup_{q \in I} D_A(q)$. The *upward language* of a state $q$ in $A$, denoted $U_A(q)$, is defined as the set of open trees $t$, such that there exists an initial and fair $t$-run $\pi$ with exactly one dangling leaf $v$ s.t. $\pi(v) = q$. We omit the $A$ subscript notation when it is implicit which automaton we are considering.

## 4.2 Simulations and Trace Inclusion Relations

We consider different types of relations on states of a BTA which under-approximate language inclusion. We generalize several notions of simulation and trace inclusion from BA to BTA, since infinite words are a particular case of infinite trees where every node has exactly one child. *Direct/delayed/fair downward* simulation/trace inclusion on BTA corresponds to direct/delayed/fair *forward* simulation/trace inclusion on BA, and *upward* corresponds to backward [CM13].

As described in Section 2.2, simulation preorders on labelled transition systems can be characterized by a game between players Spoiler and Duplicator. Given a pair of states $(p,q)$, Spoiler wants to show that $(p,q)$ is not contained in the simulation preorder while Duplicator has the opposite goal. In the downwards case, the game starts at the configuration $(p,q)$ and Spoiler makes a move using a transition $p \xrightarrow{a} \langle p_1,\ldots,p_n\rangle$, where $n = \#(a)$, and Duplicator must respond by making a move using a transition by the same symbol $q \xrightarrow{a} \langle q_1,\ldots,q_n\rangle$. Then Spoiler chooses some $i : 1 \geq i \geq n$ and the game continues from configuration $(p_i,q_i)$. The game in the words case is a simplified version as the rank $n$ is 1 for every symbol [CM13]. Since the automata are assumed to be complete, the game goes on forever and Spoiler and Duplicator build two infinite runs, $\pi$ and $\pi'$ respectively, over a closed tree s.t. $\pi(\varepsilon) = p$ and $\pi'(\varepsilon) = q$. The winning condition depends on the type of downward simulation being considered: *direct* [DHW91], *delayed* [EWS05] or *fair* simulation [HKR02]. For $x \in \{\mathsf{di}, \mathsf{de}, \mathsf{f}\}$, Duplicator wins the game iff $C^x(\pi, \pi')$ holds, where

$$C^{\mathsf{di}}(\pi,\pi') \iff \forall_{v \in dom(t)} \cdot (\pi(v) \in F \Rightarrow \pi'(v) \in F)$$

$$C^{\mathsf{de}}(\pi,\pi') \iff \forall_{\text{path } \rho \text{ of } t} \forall_{v \in \rho} \cdot (\pi(v) \in F \Rightarrow \exists_{v' \in \rho} \cdot (v \sqsubseteq v' \land \pi'(v') \in F))$$

$$C^{\mathsf{f}}(\pi,\pi') \iff \pi \text{ is fair} \Rightarrow \pi' \text{ is fair.}$$

We define the downward $x$-simulation relation $\sqsubseteq^{\mathsf{dw}\text{-}x} \subseteq Q \times Q$ (or simply $\sqsubseteq^x$) as the set of all pairs $(p,q)$ for which Duplicator has a winning strategy in the $x$-simulation game. Trivially, we have that $\sqsubseteq^{\mathsf{di}} \subseteq \sqsubseteq^{\mathsf{de}} \subseteq \sqsubseteq^{\mathsf{f}}$. It is also easy to see that, for any $x \in \{\mathsf{di}, \mathsf{de}, \mathsf{f}\}$, $\sqsubseteq^x$ is preserved downward-stepwise: if $p \sqsubseteq^x q$ and there is a tree $t \in \mathbb{T}(\Sigma)$ and a run $\pi$ s.t. $t \overset{\pi}{\Longrightarrow} p$, then there exists a run $\pi'$ s.t. $t \overset{\pi'}{\Longrightarrow} q$ and, for every $v \in dom(t)$, $\pi(v) \sqsubseteq^x \pi'(v)$.

In the upwards case, the game is modified so that the transitions are taken upwards, and each configuration continues onto the configuration containing the source states of the transitions taken by Spoiler and Duplicator respectively. The two players build two runs, $\pi$ and $\pi'$, of an open tree, starting from its unique dangling leaf $v_d$ and ending in the root node. For an initial configuration $(p,q)$, $\pi$ and $\pi'$ must satisfy $\pi(v_d) = p$ and $\pi'(v_d) = q$. Given a relation $R \subseteq Q \times Q$, we say that Duplicator wins the game iff $C^{\mathsf{up}(R)}(\pi, \pi')$ holds, where

$$C^{\mathsf{up}(R)}(\pi,\pi') \iff \begin{cases} \forall_{v \in dom(t)} \cdot (\pi(v) \in F \Rightarrow \pi'(v) \in F), \text{ and} \\ \forall_{v \in dom(t)} \cdot (\pi(v) \in I \Rightarrow \pi'(v) \in I), \text{ and} \\ \forall_{v \sqsubset v_d} \forall_{v' \in \mathbb{N}^*} \cdot (vv' \not\sqsubseteq v_d \Rightarrow \pi(vv') R \pi'(vv')) \end{cases}.$$

From a games perspective, the parameterizing relation $R$ means that Duplicator can only respond to a given transition $t$ from Spoiler with a transition $t'$ if each state that joins in from the sides in $t'$ is larger w.r.t. R than the state in the same position in $t$.

We define *the upward simulation preorder of* $R$, denoted $\sqsubseteq^{\mathsf{up}}(R) \subseteq Q \times Q$, as the set containing all pairs $(p,q)$ for which Duplicator has a winning strategy in the $\mathsf{up}(R)$-simulation game. Dually to the downwards case, upward simulation is preserved upward-stepwise.

All these simulation relations are preorders on the state space of the automaton and can be computed efficiently. As in the case of automata on finite trees (see Section 2.2), downward simulation is computed in polynomial time and the complexity of computing upward simulation depends on the parameterizing relation $R$ (it is polynomial-time computable when $R$ can be computed in polynomial time, i.e., when $R = \sqsubseteq^{\mathsf{dw}}$, $R = id$, $R = Q \times Q$, etc).

However, while attractive from the practical point of view, simulations are generally very small in size. In order to obtain coarser under-approximations of language inclusion, the simulation game is modified to give Duplicator some *lookahead* into Spoiler's moves [CM13]. While on the ordinary downward simulation game Spoiler and Duplicator announce a move by one transition at each turn, in the *k-lookahead downward simulation* game, at each configuration $(p,q)$ Spoiler announces a move consisting of a run $\pi$ of depth $k$ and with root at $p$ over some (open) tree $t$. Duplicator must respond by choosing a move consisting of a run $\pi'$ with root at $q$ over some (possibly non-strict) prefix of $t$. Let $v_1,\ldots,v_n$ be the leaf nodes of $\pi'$. Then Spoiler chooses some $i : 1 \leq i \leq n$ and the game continues from configuration $(\pi(v_i),\pi'(v_i))$. The game goes on forever and Spoiler and Duplicator build two infinite runs $\pi_S$ and $\pi_D$, respectively, over some tree $t$.We say that Duplicator wins the $k$-lookahead downward $x$-simulation, where $x \in \{\mathsf{di},\mathsf{de},\mathsf{f}\}$, iff $C^x(\pi_S,\pi_D)$ holds. We define *the k-lookahead downward x-simulation* as the set containing all $(p,q)$ for which Duplicator has a winning strategy and we denote it by $\sqsubseteq^{k\text{-}\mathsf{dw}\text{-}x}$ (or simply $\sqsubseteq^{k\text{-}x}$).

In the *k-lookahead upward simulation* game, at each configuration $(p,q)$ Spoiler announces a move consisting of a run $\pi$ over an open tree $t$ s.t. $v_d$ is the only dangling leaf of $\pi$ and $\pi(v_d) = p$. The run $\pi$ must have depth $k$, unless the root node of $\pi$ has no incoming transitions. Duplicator then responds by choosing a node $v'$ of $t$ s.t. $v_d = v'v''$ for some $v'' \neq \varepsilon$ (i.e., $v'$ is a strict prefix of $v_d$) and a move consisting of a run $\pi_{v'}$ over $t_{v'}$ s.t. $\pi_{v'}(v'') = q$. The game then continues from configuration $(\pi(v'),\pi_{v'}(\varepsilon))$. Let $\pi_S$ and $\pi_D$ be the runs over some open tree $t$ built by Spoiler and Duplicator during the entire game. Given a relation $R \subseteq Q \times Q$, we say that Duplicator wins the game iff $C^{\mathsf{up}(R)}(\pi_S,\pi_D)$ holds. We define *the k-lookahead upward simulation of* $R$ as the set containing all $(p,q)$ for which Duplicator has a winning strategy and we denote it by $\sqsubseteq^{k\text{-}\mathsf{up}}(R)$.

Like on finite trees, in the case of $k = \infty$, i.e., when Spoiler announces his full run $\pi$ at once, we talk of *trace inclusion* and write $\subseteq^{\mathsf{dw}\text{-}x}$ (or simply $\subseteq^x$), where $x \in \{\mathsf{di},\mathsf{de},\mathsf{f}\}$, and $\subseteq^{\mathsf{up}}(R)$. While interesting from the theoretical point of view, trace inclusions are impractical to compute, as direct downward trace inclusion is EXPTIME-complete [CDG$^+$08] and upward trace inclusion is PSPACE-complete for $R = \sqsubseteq^{\mathsf{dw}}$, $R = Q \times Q$ or $R = id$ (but EXPTIME-complete for $R = \subseteq^{\mathsf{dw}}$). On the other hand, $k$-lookahead simulations are polynomial-time computable for a fixed $k$ both in the downward direction and in the upward direction (when taking $\sqsubseteq^{\mathsf{dw}}$, $Q \times Q$ or $id$ as the parameterizing relation $R$). However, the complexity of computing lookahead simulations rises quickly with $k$, as it is doubly exponential on the downwards case and single exponential on the upwards case (when $R$ is $\sqsubseteq^{\mathsf{dw}}$, $Q\times Q$ or $id$). Lookahead simulations thus offer a practical tradeoff between complexity and the size of the computed relations: as the parameter $k$ increases, the $k$-lookahead simulation becomes larger and larger, approximating the corresponding trace inclusion relation better and better.

Yet another notion of simulation that under-approximates downward trace inclusion is *downward fixed-tree simulation*. It was first introduced for word automata as fixed-word simulation [Cle11, Cle12], when searching for the largest relation contained in trace inclusion and which is still suitable for state quotienting. The authors also showed that computing fixed-word simulation is PSPACE-complete [Cle11]. Similarly to the words case, we can talk of *direct*, *delayed* or *fair* downward fixed-tree simulation. For $x\in\{\mathsf{di},\mathsf{de},\mathsf{f}\}$ and a configuration $(p,q)$, Spoiler starts by announcing a complete tree $t$ that can be read from $p$, and then a variation of the ordinary downward simulation game, denoted $(t,x)$-game, is initiated. Spoiler makes a move by taking a transition $p\xrightarrow{t(\varepsilon)}\langle p_1,\ldots,p_n\rangle$, where $n=\#(t(\varepsilon))$, and Duplicator must respond by making a move by some transition $q\xrightarrow{t(\varepsilon)}\langle q_1,\ldots,q_n\rangle$. This yields $n$ $(p_i,q_i)$ configurations from which the corresponding $(t_i,x)$-game is played. As in the ordinary case, the game goes on forever and Spoiler and Duplicator build two infinite runs, $\pi$ and $\pi'$ respectively, over $t$ s.t. $\pi(\varepsilon)=p$ and $\pi'(\varepsilon)=q$. The winning condition is the same as in the corresponding ordinary simulation game. For any $t\in\mathbb{C}(\Sigma)$ and $x\in\{\mathsf{di},\mathsf{de},\mathsf{f}\}$, if Duplicator wins the $(t,x)$-game from a configuration $(p,q)$, we write $p\sqsubseteq_t^x q$. If $p\sqsubseteq_t^x q$ holds for every tree $t$, then we say that $p$ is smaller than $q$ w.r.t. fixed-tree simulation and we write $p\sqsubseteq_{\mathsf{fx}}^{\mathsf{x}} q$. Clearly, fixed-tree simulation is a preorder between simulation and trace inclusion. Even though $\sqsubseteq_{\mathsf{fx}}^{\mathsf{x}}$ is not preserved downward-stepwise in general (see Figure 4.1 for an example), it is easy to see that, for a fixed tree $t$, $\sqsubseteq_t^{\mathsf{x}}$ does propagate downwards along the nodes of $t$. Formally, if $p\sqsubseteq_t^{\mathsf{x}} q$ and $\pi$ is a run s.t. $t\overset{\pi}{\Longrightarrow}p$, then there exists a run $\pi'$ s.t. $t\overset{\pi'}{\Longrightarrow}q$ and, for every $v\in dom(t)$, $\pi(v)\sqsubseteq_{t_v}^{\mathsf{x}}\pi'(v)$.

## 4.3 State Quotienting Techniques

A classic method for reducing the size of automata is state quotienting. Given a suitable equivalence relation on the state space, each equivalence class is collapsed into just one state. From a preorder $\sqsubseteq$, one obtains an equivalence relation $\equiv:=\sqsubseteq\cap\sqsupseteq$. We now define quotienting w.r.t. $\equiv$. Let $A=(\Sigma,Q,\delta,I,F)$ be a BTA and let $\sqsubseteq$ be a preorder on $Q$. Given $q\in Q$, we denote by $[q]$ its equivalence class w.r.t. $\equiv$. For $P\subseteq Q$, $[P]$ denotes the set of equivalence classes $[P]=\{[p]\mid p\in P\}$. We define the quotient automaton w.r.t. $\equiv$ as $A/{\equiv}:=(\Sigma,[Q],\delta_{A/\equiv},[I],[F])$, where $\delta_{A/\equiv}=\{\langle[q],a,[q_1]\ldots[q_n]\rangle\mid\langle q,a,q_1\ldots q_n\rangle\in\delta_A\}$.

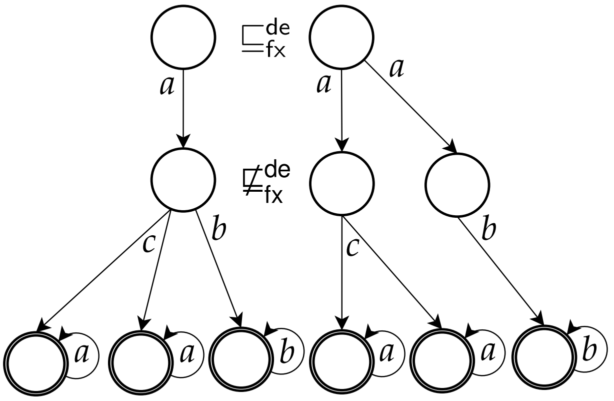

$\sqsubseteq^{\mathsf{de}}_{\mathsf{fx}}$
a
a
a
$\not\sqsubseteq^{\mathsf{de}}_{\mathsf{fx}}$
c
b
c
b
a
a
b
a
a
b

Figure 4.1: $\sqsubseteq^{\mathsf{de}}_{\mathsf{fx}}$ is not preserved downward-stepwise.

Is is trivial that $L(A) \subseteq L(A/{\equiv})$ for any $\equiv$. If the reverse inclusion also holds, i.e., if $L(A/{\equiv}) \subseteq L(A)$, we say that $\equiv$ is *good for quotienting* (GFQ).

Let $\equiv^{\mathsf{di}} := \subseteq^{\mathsf{di}} \cap \supseteq^{\mathsf{di}}$. It has been shown that $\equiv^{\mathsf{di}}$ is GFQ for infinite words (this follows from [Ete02]). In Theorem 4.3.1 we adapt the proof for finite trees from Chapter 2 to the infinite case.

**Theorem 4.3.1.** *$\equiv^{\mathsf{di}}$ is GFQ.*

*Proof.* Let $A' := A/{\equiv^{\mathsf{di}}}$. It is trivial that $L(A) \subseteq L(A')$. For the reverse inclusion, we will show that, for any tree $t$, if $t \in D_{A'}([q])$ for some $[q] \in [Q]$, then $t \in D_A(q)$. This guarantees $L(A') \subseteq L(A)$, since if $[q] \in [I]$ then there is some $q_i \in I$ such that $[q_i] = [q]$ and thus $t \in D_A(q_i)$.

By hypothesis, $t \in D_{A'}([q])$, and so there exists a fair run $\hat{\pi}$ of $A'$ such that $t \overset{\hat{\pi}}{\Longrightarrow} [q]$. We will show that there exists a run $\hat{\pi}'$ of $A$ such that $t \overset{\hat{\pi}'}{\Longrightarrow} q$ and which preserves the acceptance in $\hat{\pi}$. Since $\hat{\pi}'$ too is fair, we will obtain that $t \in D_A(q)$.

Let $\sigma$ be the root node of the tree $t$ and let $t_1,\ldots,t_n$, where $n = \#(\sigma)$, denote each of the immediate subtrees of $t$. From $t \overset{\hat{\pi}}{\Longrightarrow} [q]$ we have that there exists a transition $\langle [q], \sigma, [q_1]\ldots[q_n] \rangle$ in $A'$, for some $[q_1],\ldots,[q_n] \in [Q]$, such that $t_i \in D_{A'}([q_i])$ for every $i$. By the definition of $\delta_{A'}$, there exists a transition 1) $\langle q', \sigma, q'_1 \ldots q'_n \rangle$ in $A$, for states $q'_1 \in [q_1],\ldots,q'_n \in [q_n]$ and also a state $q' \in [q]$ satisfying 2) $[q] \in [F] \Rightarrow q' \in F$. From $t_i \in D_{A'}([q_i]) = D_{A'}([q'_i])$ we obtain, for every $i$, that there is a $\hat{\pi}_i$ such that $t_i \overset{\hat{\pi}_i}{\Longrightarrow} [q'_i]$. Applying the same reasoning downwards, we obtain, for each $t_i$, a run $\hat{\pi}'_i$ such that $t_i \overset{\hat{\pi}'_i}{\Longrightarrow} q'_i$ and which preserves the acceptance in $\hat{\pi}_i$. Thus, taking the transition 1) and the runs $\hat{\pi}_i$, one obtains a run $\hat{\pi}''$ of $A$ such that $t \overset{\hat{\pi}''}{\Longrightarrow} q'$. From 2) we have that $\hat{\pi}''$ preserves the acceptance in $\hat{\pi}$ and thus too is fair. Finally, from $q' \equiv^{\mathsf{di}} q$ we conclude that there is a fair run $\hat{\pi}'$ of $A$ such that $t \overset{\hat{\pi}'}{\Longrightarrow} q$. $\square$

Since the GFQ property is downward closed, we obtain that the direct downward ordinary simulation, $\sqsubseteq^{\mathsf{di}}$, too is GFQ. However, there are coarser notions of downward ordinary simulations which are GFQ. Let $\equiv^{\mathsf{de}} := \sqsubseteq^{\mathsf{de}} \cap \sqsupseteq^{\mathsf{de}}$ and $\equiv^{\mathsf{de}}_{\mathsf{fx}} := \sqsubseteq^{\mathsf{de}}_{\mathsf{fx}} \cap \sqsupseteq^{\mathsf{de}}_{\mathsf{fx}}$ be the equivalences induced by the delayed ordinary simulation and the delayed fixed-word simulation, respectively. It has been shown that $\equiv^{\mathsf{de}}$ [EWS05] and $\equiv^{\mathsf{de}}_{\mathsf{fx}}$ [Cle11] are GFQ for infinite word automata. Below we show that these results carry over to the trees case. Theorem 4.3.2 uses the auxiliary Lemma 4.3.1 to prove that the delayed fixed-tree simulation is suitable for quotienting as well. This result then extends to the particular case of the delayed ordinary simulation. As for trace inclusion, it has been shown that $\subseteq^{\mathsf{de}} \cap \supseteq^{\mathsf{de}}$ is not GFQ for infinite words [Cle11], therefore it also does not hold for the more general case of infinite trees.

**Lemma 4.3.1.** *Let $A$ be a BTA, $t \in \mathbb{T}(\Sigma)$ a tree, $\rho = \{v_1, v_2, \ldots\}$ a path of $t$ and $\pi$ a $t$-run of $A/\equiv^{\mathsf{de}}_{\mathsf{fx}}$ s.t. $\pi|_\rho = [p_1]a_1[p_2]a_2\ldots$. If $p_1 \sqsubseteq^{\mathsf{de}}_{\mathsf{fx}} p'_1$, then there exists a $t$-run $\pi'$ of $A$ s.t. $\pi'|_\rho = p'_1 a_1 p'_2 a_2 \ldots$ and $p_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} p'_i$, for every $i \geq 1$.*

*Proof.* We will use induction on $n \geq 0$ to build, for every $n$, the prefix $\chi_n$ of length $n$ of $\pi'|_\rho$. Together with König's Lemma, this will show that $\pi'|_\rho$ indeed exists.

The base case $n = 0$ is trivial, simply take $\chi_0 = p'_1$.

For the induction step, assume the prefix $\chi_n = p'_1 \ldots p'_n$ has already been built. From $\pi|_\rho$ and by the definition of $A/\equiv^{\mathsf{de}}_{\mathsf{fx}}$, we obtain that there exist $q_n$ and $q_{n+1}$ s.t. 1) $q_n \equiv^{\mathsf{de}}_{\mathsf{fx}} p_n$, 2) $q_{n+1} \equiv^{\mathsf{de}}_{\mathsf{fx}} p_{n+1}$ and 3) $\langle q_n, a_n, r_1 \ldots r_j \ldots r_{\#(a_n)} \rangle$ in $A$ s.t. $r_j = q_{n+1}$. From 1) we have, in particular, that $q_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} p_n$. Since $p_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} p'_n$ we have by the transitivity of $\sqsubseteq^{\mathsf{de}}_{t_{v_n}}$ that $q_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} p'_n$. Thus, since $\sqsubseteq^{\mathsf{de}}_{t_{v_n}}$ propagates downwards, it follows from 3) that there exists a transition $\langle p'_n, a_n, r'_1 \ldots r'_j \ldots r'_{\#(a_n)} \rangle$ in $A$ s.t. $q_{n+1} \sqsubseteq^{\mathsf{de}}_{t_{v_{n+1}}} r'_j$. From 2) we obtain that $p_{n+1} \sqsubseteq^{\mathsf{de}}_{t_{v_{n+1}}} r'_j$. Taking $p'_{n+1} := r'_j$, we have that $\chi_{n+1} := \chi_n a_n p'_{n+1}$ is a prefix of length $n+1$ of $\pi'|_\rho$, for $\pi'$ some $t$-run of $A$, satisfying $p_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} p'_i$, for every $i \in \{1, \ldots, n+1\}$. □

**Theorem 4.3.2.** *$\equiv^{\mathsf{de}}_{\mathsf{fx}}$ is GFQ.*

*Proof.* It is trivial that $L(A) \subseteq L(A/\equiv^{\mathsf{de}}_{\mathsf{fx}})$. For the reverse inclusion, we will show that, for any $t \in \mathbb{C}(\Sigma)$, if $t \in D_{A/\equiv^{\mathsf{de}}_{\mathsf{fx}}}([q_1])$ for some $[q_1] \in [Q]$, then $t \in D_A(q_1)$. In particular, if $[q_1] \in [I]$, then there is some $q'_1 \in I$ s.t. $[q'_1] = [q_1]$ and so $t \in D_A(q'_1)$. Thus, $t \in L(A)$.

By hypothesis, $t \in D_{A/\equiv^{\mathsf{de}}_{\mathsf{fx}}}([q_1])$, and so there exists a fair run $\hat{\pi}$ of $A/\equiv^{\mathsf{de}}_{\mathsf{fx}}$ s.t. $t \xRightarrow{\hat{\pi}} [q_1]$. We will show that there exists a fair run $\hat{\pi}'$ of $A$ s.t. $t \xRightarrow{\hat{\pi}'} q_1$. Let $\rho$ be a path of $t$, $t|_\rho = a_1 a_2 \ldots$ the sequence of symbols of $t$ along $\rho$ and $\hat{\pi}|_\rho = [q_1]a_1[q_2]a_2\ldots$ the restriction of $\hat{\pi}$ to $\rho$. We will show that the restriction $\hat{\pi}'|_\rho$ of the desired run $\hat{\pi}'$ of $A$ to $\rho$ exists, by successively building prefixes of $\hat{\pi}'|_\rho$. Applying this to every path $\rho$ of $t$, we show that $\hat{\pi}'$ indeed exists.

Let $\chi_e$ be a prefix of $\hat{\pi}'|_\rho$ s.t. $\chi_e$ contains at least $e$ elements from $F$. We will show that there is an infinite sequence $\chi_0, \chi_1, \chi_2, \ldots$. So the limit of the sequence is a prefix that visits elements from $F$ an infinite number of times, i.e., it is $\hat{\pi}'|_\rho$ itself. Let us represent the nodes of $\rho$ by $v_1, v_2, \ldots$, where $v_1$ is the root node of $t$ and each $v_i$ is the parent node of $v_{i+1}$, for every $i \geq 1$. We then use $\rho_i$ to denote the path of $t_{v_i}$ contained in $\rho$, i.e., $\rho_i = \{v \mid v_i \cdot v \in \rho\}$. We will also make use of the following auxiliary definition: a prefix $\chi_e = q'_1 a_1 q'_2 \ldots q'_n$ of $\hat{\pi}'|_\rho$ is *good* iff $q_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} q'_n$. We will use induction on $e \geq 0$ to show that, for every $e$, there exists a prefix $\chi_e$ of $\hat{\pi}'|_\rho$ which is good. This will prove that the sequence of $\chi'_i s$ is infinite.

The base case $e = 0$ is trivial, just take $\chi_0 = q_1$.

For the induction step, assume that the prefix $\chi_e = q_1 a_1 q'_2 a_2 \ldots q'_n$ exists and is good, i.e., $q_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} q'_n$. Since $\hat{\pi}$ is fair, $\hat{\pi}|_\rho$ visits elements from $[F]$ an infinite number of times, and so there is an $o > n$ s.t. $\hat{\pi}|_\rho = [q_1]a_1 \ldots [q_n]a_n \ldots [q_o]a_o \ldots$ and $[q_o] \in [F]$. Let $\overset{*}{\pi}_1$ be the $t_{v_n}$-run of $A/\equiv^{\mathsf{de}}_{\mathsf{fx}}$ s.t. $\overset{*}{\pi}_1|_{\rho_n} = [q_n]a_n \ldots [q_o]a_o \ldots$. Applying Lemma 4.3.1 to the tree $t_{v_n}$, the path $\rho_n$, the run $\overset{*}{\pi}_1$ and the fact that $q_n \sqsubseteq^{\mathsf{de}}_{t_{v_n}} q'_n$, we obtain that there exists a $t_{v_n}$-run $\overset{*}{\pi}_2$ of $A$ s.t. $\overset{*}{\pi}_2|_{\rho_n} = q'_n a_n \ldots q'_o a_o \ldots$ and $q_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} q'_i$ for every $i \geq n$. Now let $\overset{*}{\pi}_3$ be the $t_{v_o}$-run of $A/\equiv^{\mathsf{de}}_{\mathsf{fx}}$ s.t. $\overset{*}{\pi}_3|_{\rho_o} = [q_o]a_o \ldots$. From $[q_o] \in [F]$ it follows that there is a $q''_o \in F$ s.t. $q''_o \equiv^{\mathsf{de}}_{\mathsf{fx}} q_o$. Thus applying Lemma 4.3.1 to $t_{v_o}$, $\rho_o$, $\overset{*}{\pi}_3$ and $q_o \sqsubseteq^{\mathsf{de}}_{t_{v_o}} q''_o$, we obtain that there is a $t_{v_o}$-run $\overset{*}{\pi}_4$ of $A$ s.t. $\overset{*}{\pi}_4|_{\rho_o} = q''_o a_o \ldots$ and $q_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} q''_i$ for every $i \geq o$. And since $q''_o \sqsubseteq^{\mathsf{de}}_{t_{v_o}} q_o$ and $q''_o \in F$, it follows from the definition of $\sqsubseteq^{\mathsf{de}}_{t_{v_o}}$ that there exists a $t_{v_o}$-run $\overset{*}{\pi}_5$ of $A$ s.t. $\overset{*}{\pi}_5|_{\rho_o} = q'''_o a_o q'''_{o+1} \ldots$, where $q'''_o = q_o$, and $q'''_p \in F$ for some $p \geq o$. Moreover, since $\sqsubseteq^{\mathsf{de}}_{t_{v_o}}$ propagates downwards along the nodes of $t_{v_o}$, we have that $q''_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} q'''_i$ for every $i \geq o$. By the transitivity of $\sqsubseteq^{\mathsf{de}}_{t_{v_i}}$ we have that $q_i \sqsubseteq^{\mathsf{de}}_{t_{v_i}} q'''_i$ and, in particular, $q_p \sqsubseteq^{\mathsf{de}}_{t_{v_p}} q'''_p$. Thus $\chi_{e+1} := \chi_e a_n q'_{n+1} \ldots q'_{o-1} a_{o-1} q'''_o \ldots q'''_p$ is a good prefix of $\hat{\pi}'|_\rho$ visiting at least $e+1$ elements from F. □

Fair simulation is not GFQ for infinite words [HKR02], thus it also does not hold for the trees case. Consequently, $\subseteq^{\mathsf{f}}$ too is not GFQ.

Figure 4.2 presents a counterexample showing that $\equiv := \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}} \cap \sqsupseteq^{\mathsf{di}}) \cap \sqsupseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}} \cap \sqsupseteq^{\mathsf{di}})$ is not GFQ. This is an adaptation of the Example 5 for finite trees in [Hol11], where the inducing relation is referred to as the *downward bisimulation equivalence* and the automata are seen bottom-up.

Theorem 4.3.3 shows that upward trace inclusion induced by the identity is suitable for quotienting. Let $\equiv^{\mathsf{up}} := \subseteq^{\mathsf{up}}(id) \cap \supseteq^{\mathsf{up}}(id)$. For this proof we use the auxiliary Lemma 4.3.3 and the lifting of $\subseteq^{\mathsf{up}}(id)$ to tuples of states.

**Lemma 4.3.2.** *Let $A$ be a BTA and $\langle p_1, \ldots, p_n \rangle$ and $\langle q_1, \ldots, q_n \rangle$ two tuples of states of $A$ s.t. $\langle p_1, \ldots, p_n \rangle \subseteq^{\mathsf{up}}(id) \langle q_1, \ldots, q_n \rangle$. Then for every $t \in \mathbb{T}(\Sigma)$ and every initial t-run $\pi$ of depth i, for some $i \geq 1$, s.t. $level_i(\pi) = \langle p_1, \ldots, p_n \rangle$, there exists an initial t-run $\hat{\pi}$ of the same depth s.t. $level_i(\hat{\pi}) = \langle q_1, \ldots, q_n \rangle$ and $\hat{\pi}$ preserves the acceptance of states in $\pi$.*

*Proof.* Let $A$ be an initial $t$-run of depth $i$ s.t. $level_i(\pi) = \langle p_1,\dots,p_n\rangle$. For $\pi'$ an arbitrary $t$-run of the same depth, we say that $level_i(\pi')$ is $j$-good iff 1) for every node $v_k$ in $level_i(\pi')$ s.t. $k \le j$, $\pi'(v_k) = q_k$, and 2) for every node $v_k$ in $level_i(\pi')$ s.t. $j < k \le n$, $\pi'(v_k) = p_k$. We will show, using induction on $j \ge 0$, that for every $j$ there exists an initial $t$-run $\pi^*$ s.t. $level_i(\pi^*)$ is $j$-good and $\pi^*$ preserves the acceptance in $\pi$. For the particular case of $j = n$, this proves the lemma.

The base case $j = 0$ is trivial, as the initial $t$-run $\pi$ is 0-good itself.

For the induction step, let $\pi'$ be an initial $t$-run s.t. $level_i(\pi')$ is $(j-1)$-good and $\pi'$ preserves the acceptance in $\pi$. If $j > n$, the lemma trivially holds. Otherwise, by $p_j \subseteq^{\mathsf{up}}(id)\, q_j$ there exists a $t$-run $\pi''$ s.t. $level_i(\pi'') = \langle q_1,\dots,q_j,p_{j+1},\dots,p_n\rangle$. Since $\subseteq^{\mathsf{up}}(id)$ preserves both initiality and acceptance, $\pi''$ is an initial $t$-run s.t. $level_i(\pi'')$ is $j$-good and which preserves the acceptance in $\pi'$, which we assume by induction to preserve the acceptance in $\pi$. □

**Theorem 4.3.3.** $\equiv^{\mathsf{up}}$ *is GFQ.*

*Proof.* Let $A' := A/\equiv^{\mathsf{up}}$. It is trivial that $L(A) \subseteq L(A')$. For the reverse inclusion, we will show that for any $t \in \mathbb{C}(\Sigma)$, if $t \in L(A')$ then $t \in L(A)$. By hypothesis, there exists an initial and fair $t$-run $\pi'$ of $A$ by successively building larger and larger finite prefixes of $\pi'$. Let $\pi_i$ be the finite prefix of $\pi$ of depth $i$, and $t_i$ the prefix tree of $t$ visited by $\pi_i$, and let $[q_1],[q_2],\dots,[q_n] \in [Q]$ be such that $level_i(\pi_i) = \langle [q_1],[q_2],\dots,[q_n]\rangle$. We say that a $t_i$-run $\pi'_i$ of $A$ of depth $i$ is *good* if 1) $level_i(\pi'_i) = \langle q_1,q_2,\dots,q_n\rangle$, and 2) if $\pi_i(v) \in [F]$ then $\pi'_i(v) \in F$, for any $v \in dom(\pi_i)$ - i.e., $\pi'_i$ preserves the acceptance in $\pi_i$.

In the following, we will use induction on $i \ge 0$ to build, for every $i$, an initial and *good* run $\pi^*_i$ of $A$ of depth $i$ over a prefix $t_i$ of $t$.

For $i = 0$, take $\pi'_0(\varepsilon) = q$ s.t. $\pi_0(\varepsilon) = [q]$. From the fact that $\subseteq^{\mathsf{up}}(id)$ preserves the initiality and the acceptance of states, we have, respectively, $q \in I$, since $[q] \in [I]$, and if $[q] \in [F]$ then $q \in F$. Therefore, $\pi'_0$ is an initial and *good* run of $A$ of depth 0.

For $i > 0$, let $\pi_i$ be the initial run of $A'$ of depth $i$, and $t_i$ the prefix of $t$ visited by $\pi_i$, s.t. $level_i(\pi_i) = \langle [q_1],[q_2],\dots,[q_n]\rangle$, for some $[q_1],[q_2],\dots,[q_n] \in [Q]$. We will use the following auxiliary definition: for $\hat{\pi}$ some $t_i$-run of $A$, we say that $level_i(\hat{\pi}) = \langle q'_1,q'_2,\dots,q'_n\rangle$ is $j$-good iff, for every $k : 1 \le k \le j$, $q'_k = q_k$. Let $\pi_{i-1}$ be the prefix of $\pi_i$ of depth $i-1$, and $t_{i-1}$ the prefix of $t$ visited by $\pi_{i-1}$, and $[r_1],\dots,[r_m] \in [Q]$ some states s.t. $level_{i-1}(\pi_{i-1}) = \langle [r_1],\dots,[r_m]\rangle$. We assume, by induction hypothesis, that an initial and *good* $t_{i-1}$-run $\pi'_{i-1}$ of $A$ of depth $i-1$ has already been built. Since $\pi'_{i-1}$ is *good*, we have, in particular, that $level_{i-1}(\pi'_{i-1}) = \langle r_1,\dots,r_m\rangle$. From the definition of $\delta_{A'}$, we have that, for each of the $m$ transitions $\langle [r_k],\sigma_k,[q_{k1}]\dots[q_{k\#(\sigma_k)}]\rangle \in \delta_{A'}$ s.t. $[r_k] \in level_{i-1}(\pi_i)$ and $[q_{k1}]\dots[q_{k\#(\sigma_k)}] \in level_i(\pi_i)$, there exist $r'_k \equiv^{\mathsf{up}} r_k$, $q'_{k1} \equiv^{\mathsf{up}} q_{k1},\dots,q'_{k\#(\sigma_k)} \equiv^{\mathsf{up}} q_{k\#(\sigma_k)}$ s.t.

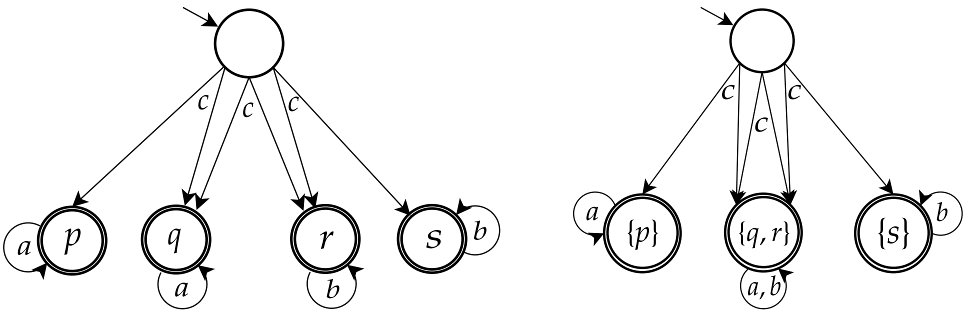
$\langle r'_k, \sigma_k, q'_{k1} \ldots q'_{k\#(\sigma_k)} \rangle \in \delta_A$. Applying Lemma 4.3.2 to $\langle r'_1, \ldots, r'_m \rangle \sqsupseteq^{\mathsf{up}} (id) \langle r_1, \ldots, r_m \rangle$, we obtain that there exists an initial $t_{i-1}$-run $\pi''_{i-1}$ of $A$ s.t. $level_{i-1}(\pi''_{i-1}) = \langle r'_1, \ldots, r'_m \rangle$ and which preserves the acceptance in $\pi'_{i-1}$. Thus, extending $\pi''_{i-1}$ with each of the $m$ $A$-transitions $\langle r'_k, \sigma_k, q'_{k1} \ldots q'_{k\#(\sigma_k)} \rangle$ we obtain an initial run $\pi'_i$ of $A$ over $t_i$ and s.t. $level_i(\pi'_i) = \langle q'_{11}, \ldots, q'_{1\#(\sigma_1)}, \ldots, q'_{m1}, \ldots, q'_{m\#(\sigma_m)} \rangle$. We will use induction on $j \geq 1$ to build an initial $t_i$-run $\pi^*_i$ of $A$ which preserves the acceptance in $\pi'_i$ and s.t. $level_i(\pi^*_i) \sqsubseteq^{\mathsf{up}} (id) \langle q_1, \ldots, q_n \rangle$ and $level_i(\pi^*_i)$ is $j$-good for every $j$. Thus $\pi^*_i$ is an initial and *good* $t_i$-run of $A$ as required.

For $j = 1$, we can assume without loss of generality that $q'_{11} = q_{11}$, since $[q'_{11}] = [q_{11}]$. Thus $\pi'_i$ itself is an initial $t_i$-run of $A$ s.t. $level_i(\pi'_i) \sqsubseteq^{\mathsf{up}} (id) \langle q_1, \ldots, q_n \rangle$ and $level_i(\pi'_i)$ is 1-good.

For $j > 1$, assume an initial $t_i$-run $\pi''_i$ of $A$ preserving the acceptance in $\pi'_i$ and s.t. $level_i(\pi''_i) \sqsubseteq^{\mathsf{up}} (id) \langle q_1, \ldots, q_n \rangle$ and $level_i(\pi''_i)$ is $(j-1)$-good has already been built. Let $q'_j$ be the $j$-th state in $level_i(\pi'')$. By $q'_j \sqsubseteq^{\mathsf{up}} (id) q_j$, there exists a $t_i$-run $\pi'''_i$ of $A$ s.t. $level_i(\pi'''_i)$ differs from $level_i(\pi''_i)$ only in its $j$-th state, which is $q_j$. Since $\sqsubseteq^{\mathsf{up}} (id)$ preserves both initiality and acceptance, $\pi'''_i$ is initial and preserves the acceptance in $\pi''_i$, which we assume by hypothesis to preserve the acceptance in $\pi'_i$. Thus, $\pi'''_i$ is an initial run of $A$ over $t_i$ preserving the acceptance in $\pi'_i$ and s.t. $level_i(\pi'''_i) \sqsubseteq^{\mathsf{up}} (id) \langle q_1, \ldots, q_n \rangle$ and $level_i(\pi'''_i)$ is $j$-good.

Thus we have shown that there exist initial runs $\pi^*_0, \pi^*_1, \pi^*_2, \ldots$ of $A$ over larger and larger prefixes of $t$ which preserve the acceptance in the $t$-run $\pi$ of $A'$. Since $A$ is finitely branching, by König's Lemma there exists an initial and infinite run $\pi^*_\omega$ of $A$ over $t$ preserving the acceptance in $\pi$. Since $\pi$ is fair, $\pi^*_\omega$ too is fair, and therefore $t \in L(A)$. □

The table in Figure 4.3 summarizes all our results on relations which are or are not good for quotienting for infinite tree automata. Note that negative results propagate to larger relations and positive results propagate to smaller relations (i.e., GFQ is downward closed).



(a) Automaton $A$. (b) Automaton $A/\equiv$.

Figure 4.2: $\equiv := \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}} \cap \sqsupseteq^{\mathsf{di}}) \cap \sqsupseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}} \cap \sqsupseteq^{\mathsf{di}})$ is not GFQ. We are considering $\Sigma_1 = \{a,b\}$ and $\Sigma_2 = \{c\}$. Computing all the necessary relations to quotient $A$ w.r.t. $\equiv$, we obtain $\sqsubseteq^{\mathsf{di}} = \{(p,q),(r,s)\} = \sqsupseteq^{\mathsf{di}}$ and $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}} \cap \sqsupseteq^{\mathsf{di}}) = \{(q,r),(r,q)\}$. Thus $\equiv = \{(q,r),(r,q)\}$. Computing $A/\equiv$, we verify that $c(bb\ldots,aa\ldots)$ is now accepted by the automaton $A/\equiv$, while it was not in the language of $A$.

| $R$ | | |
|---|---|---|
| $\sqsubseteq^{\mathsf{di}}$ | | ✓ |
| $\sqsubseteq^{\mathsf{di}}_{\mathsf{fx}}$ | | ✓ |
| $\sqsubseteq^{\mathsf{de}}$ | | ✓ |
| $\sqsubseteq^{\mathsf{de}}_{\mathsf{fx}}$ | | ✓ |
| $\sqsubseteq^{\mathsf{f}}$ | | × |
| $\sqsubseteq^{\mathsf{f}}_{\mathsf{fx}}$ | | × |
| $\subseteq^{\mathsf{di}}$ | | ✓ |
| $\subseteq^{\mathsf{de}}$ | | × |
| $\subseteq^{\mathsf{f}}$ | | × |
| $\sqsubseteq^{\mathsf{up}}$ | $id$ | ✓ |
| | $\sqsubseteq^{\mathsf{di/de/f}}$ | × |
| | $\subseteq^{\mathsf{di/de/f}}$ | × |
| $\subseteq^{\mathsf{up}}$ | $id$ | ✓ |
| | $\sqsubseteq^{\mathsf{di/de/f}}$ | × |
| | $\subseteq^{\mathsf{di/de/f}}$ | × |

Figure 4.3: GFQ relations $R$ for infinite tree automata. The largest GFQ relations found are marked with ✓, and ✓ marks their corresponding under-approximations. Relations which are not GFQ are marked with ×.

## 4.4 Transition Pruning

In this section we define pruning relations on a BTA, following very closely the intuition and definitions presented in Section 2.4 for finite trees. The motivation behind this technique is that some transitions may be deleted without changing the language, because *better* ones remain. We proceed by comparing the endpoints of transitions over the same symbol. Given two binary relations $R_{\mathsf{u}}$ and $R_{\mathsf{d}}$ on $Q$, we compare two transitions using the following relation:

$$P(R_{\mathsf{u}},R_{\mathsf{d}}) = \{(\langle p,\sigma,r_1\cdots r_n\rangle,\langle p',\sigma,r'_1\cdots r'_n\rangle) \mid p\, R_{\mathsf{u}}\, p' \wedge \langle r_1\cdots r_n\rangle\, \hat{R}_{\mathsf{d}}\, \langle r'_1\cdots r'_n\rangle\},$$

where $\hat{R}_{\mathsf{d}}$ results from lifting $R_{\mathsf{d}} \subseteq Q\times Q$ to $\hat{R}_{\mathsf{d}} \subseteq Q^n \times Q^n$, as defined below. If $t\,P\,t'$ then we say that $t'$ is *better* than $t$ and $t$ may be removed. We want $P(R_{\mathsf{u}},R_{\mathsf{d}})$ to be a strict partial order (s.p.o.), i.e., irreflexive and transitive (and thus acyclic). There are two cases in which $P(R_{\mathsf{u}},R_{\mathsf{d}})$ is guaranteed to be a s.p.o.:

1. $R_{\mathsf{u}}$ is some s.p.o. $<_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the standard lifting $\hat{\leq}_{\mathsf{d}}$ of some p.o. $\leq_{\mathsf{d}}$ to tuples. I.e., $\langle r_1\cdots r_n\rangle\, \hat{\leq}_{\mathsf{d}}\, \langle r'_1\cdots r'_n\rangle$ iff $\forall_{1\leq i\leq n}\,.\, r_i \leq_{\mathsf{d}} r'_i$. The transitions in each pair of $P(<_{\mathsf{u}},\leq_{\mathsf{d}})$ depart from different states and therefore are necessarily different.

2. $R_{\mathsf{u}}$ is some p.o. $\leq_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the lifting $\hat{<}_{\mathsf{d}}$ of some s.p.o. $<_{\mathsf{d}}$ to tuples. In this case the transitions in each pair of $P(\leq_{\mathsf{u}},<_{\mathsf{d}})$ may have the same origin but must go to different tuples of states. Since for two tuples $\langle r_1\cdots r_n\rangle$ and $\langle r'_1\cdots r'_n\rangle$ to be different it suffices that $r_i \neq r'_i$ for some $1 \leq i \leq n$, we define $\hat{<}_{\mathsf{d}}$ as a binary relation such that $\langle r_1\cdots r_n\rangle\, \hat{<}_{\mathsf{d}}\, \langle r'_1\cdots r'_n\rangle$ iff $\forall_{1\leq i\leq n}.\, r_i \leq_{\mathsf{d}} r'_i$, and $\exists_{1\leq i\leq n}.\, r_i <_{\mathsf{d}} r'_i$.

For $A = (\Sigma,Q,\delta,I,F)$ a BTA and $P \subseteq \delta\times\delta$ a s.p.o., the pruned automaton is defined as $Prune(A,P) = (\Sigma,Q,\delta',I,F)$ where $\delta' = \{(p,\sigma,r) \in \delta \mid \nexists (p',\sigma,r') \in \delta. (p,\sigma,r)\, P\, (p',\sigma,r')\}$. Note that the pruned automaton $Prune(A,P)$ is unique. Since deleting transitions cannot add new trees to the language, it is trivial that $L(Prune(A,P)) \subseteq L(A)$. If we also have $L(A) \subseteq L(Prune(A,P))$, the language is preserved and we say that $P$ is *good for pruning* (GFP).

In the following we provide a complete picture of which combinations of simulation and trace inclusion relations are GFP. Most of the counterexamples are obtained via slight modifications to those in Section 2.4 for finite trees. The counterexample for $P(\sqsubset^{\mathsf{up}}(\equiv^{\mathsf{de}}),\sqsubset^{\mathsf{di}})$ in Figure 4.6, however, is completely new. As in the case of finite trees, the GFP property is downward closed, i.e., if $R \subseteq R'$ and $R'$ is GFP then $R$ too is GFP (or if $R$ is not GFP then $R'$ too is not). For every p.o. $R$, the corresponding s.p.o. is given by $R\backslash R^{-1}$.

Theorem 4.4.1 proves that $P(id,R)$ is GFP for every strict partial order $R \subset \subseteq^{\mathsf{di}}$. Thus, in particular, $P(id,\subset^{\mathsf{di}})$ and $P(id,\sqsubset^{\mathsf{di}})$ are GFP. It has been shown that $P(id,\sqsubset^{\mathsf{de}})$ is not

GFP [CM13] for infinite words, and so the result does not hold for the more general case of infinite trees either. Consequently, $P(id, \subset^{\mathsf{de}})$, $P(id, \sqsubset^{\mathsf{f}})$ and $P(id, \subset^{\mathsf{f}})$ too are not GFP.

**Theorem 4.4.1.** *For every strict partial order* $R \subset \subseteq^{\mathsf{di}}$, $P(id,R)$ *is GFP.*

*Proof.* Let $A' = Prune(A, P(id,R))$. We show $L(A) \subseteq L(A')$. By hypothesis, $t \in L(A)$ and so there exists an initial and fair $t$-run $\pi$ of $A$. We will show that there exists an initial and fair $t$-run $\pi'$ of $A'$.

We say that a $t$-run $\hat{\pi}$ of $A$ is *i-good* if it does not use any transition of $A - A'$ in its first $i$ levels. Formally, for every node $v \in dom(t)$ with $|v| < i$, $\langle \hat{\pi}(v), \sigma, \hat{\pi}(v1) \ldots \hat{\pi}(v\#(\sigma))\rangle$, where $\sigma = t(v)$, is a transition of $A'$. We will use induction on $i \geq 0$ to show that, for every $i$, there exists an $i$-good initial and fair $t$-run $\hat{\pi}'$ of $A$.

The base case $i = 0$ is trivial, as the $t$-run $\pi$ itself is 0-good.

For the induction step, let us assume that an $(i-1)$-good initial and fair run $\pi^{i-1}$ of $A$ over $t$ exists. Since $A$ is finite, for every transition *trans* there are only finitely many $A$-transitions *trans*$'$ such that *trans* $P(id,R)$ *trans*$'$. And since $P(id,R)$ is a strict partial order, for each transition *trans* in $A$ we have that either **1)** *trans* is maximal w.r.t. $P(id,R)$, or **2)** there exists a $P(id,R)$-larger transition *trans*$'$ which is maximal w.r.t. $P(id,R)$. Thus, for every state $p$ and every symbol $\sigma$, there exists a transition by $\sigma$ departing from $p$ which is still in $A'$.

Therefore, from $\pi^{i-1}$ one easily obtains an $i$-good initial $t$-run $\pi^i$ preserving the acceptance in $\pi$ in its first $i$ levels. In the first $(i-1)$ levels of $t$, $\pi^i$ is identical to $\pi^{i-1}$. In the $i$-th level of $t$, we have that for each transition $trans = \langle \pi^{i-1}(v), \sigma, \pi^{i-1}(v1) \ldots \pi^{i-1}(v\#(\sigma))\rangle$, where $|v| = i-1$ and $\sigma = t(v)$, either 1) *trans* is $P(id,R)$-maximal and so we take $\pi^i(vj) := \pi^{i-1}(vj)$, for every $j : 1 \leq j \leq \#(\sigma)$, or 2) there exists a $P(id,R)$-larger transition $trans' = \langle \pi^{i-1}(v), \sigma, q_1 \ldots q_{\#(\sigma)}\rangle$ which is $P(id,R)$-maximal. By the definition of $P(id,R)$, we have that $\langle \pi^{i-1}(v1) \ldots \pi^{i-1}(v\#(\sigma))\rangle \hat{R} \langle q_1 \ldots q_{\#(\sigma)}\rangle$ and thus we take $\pi^i(vj) := q_j$, for every $j : 1 \leq j \leq \#(\sigma)$. Since $R \subset \subseteq^{\mathsf{di}}$, we have that, for every $j : 1 \leq j \leq \#(\sigma)$, there exists a run $\pi_j$ of $A$ such that $t_{v_j} \overset{\pi_j}{\Longrightarrow} q_j$. The run $\pi^i$ can now be completed from each $q_j$ by using the corresponding run $\pi_j$. By the definition of $\subseteq^{\mathsf{di}}$, $\pi^i$ preserves the acceptance in $\pi^{i-1}$. Therefore, $\pi^i$ is a fair run of $A$, which concludes the proof of the induction step.

We have proved that it is possible to obtain initial and fair $t$-runs $\pi'$ of $A$ which are $i$-good for arbitrary large values of $i \geq 0$. Thus, since $A$ is finitely branching it follows from König's Lemma that there exists an initial and fair $t$-run $\pi''$ which is $i$-good for any $i$, i.e., $\pi''$ does not use any transition from $A - A'$ at all. Thus $\pi''$ is a run of $A'$, and so $t \in L(A')$. □

In Theorem 4.4.2 we prove that $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$ is GFP, from which it follows that

$P(\sqsubset^{\mathsf{up}}(id), \sqsubseteq^{\mathsf{di}})$ and $P(\sqsubset^{\mathsf{up}}(id), id)$ too are GFP. However, the counterexample in Figure 4.4 shows that $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}}), \subset^{\mathsf{di}})$ is not GFP, by adapting the complex counterexample for finite trees (Figure 2.4 in Section 2.4) to the infinite case. Therefore $P(\sqsubset^{\mathsf{up}}(R), R')$ and $P(\subset^{\mathsf{up}}(R), R')$ are not GFP for any $R \in \{\sqsubseteq^{\mathsf{di}}, \sqsubseteq^{\mathsf{de}}, \sqsubseteq^{\mathsf{f}}, \subseteq^{\mathsf{di}}, \subseteq^{\mathsf{de}}, \subseteq^{\mathsf{f}}\}$ and $R' \in \{\subset^{\mathsf{di}}, \subset^{\mathsf{de}}, \subset^{\mathsf{f}}\}$.

**Theorem 4.4.2.** *$P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$ is GFP.*

*Proof.* Let $A' = Prune(A, P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}}))$. We will show that for every initial and fair run of $A$ over some tree $t \in \mathbb{C}(\Sigma)$, there exists a $t$-run $\hat{\pi}$ of $A'$.

By hypothesis, there exists an initial and fair $t$-run $\pi$ of $A$. Let $\pi_i$ be a prefix of $\pi$ of some depth $i \geq 1$, and $t_i$ the prefix of $t$ visited by $\pi_i$. We will show that a $t_i$-run $\pi'_i$ of $A'$ starting from an initial state exists and that $\pi'_i$ preserves the acceptance of states in $\pi_i$. This will prove that it is possible to build larger and larger prefixes of an initial $t$-run $\hat{\pi}$ of $A'$ that preserves the acceptance in $\pi$. Since $A$ is finitely branching, by König's Lemma we obtain that the run $\hat{\pi}$ itself exists and is fair.

We will make use of some auxiliary notions. For $\pi_i$ a (finite) prefix of an (infinite) run $\pi$ of $A$ over a tree $t \in \mathbb{C}(\Sigma)$ (and $t_i \in \mathbb{T}(\Sigma)$ the prefix of $t$ visited by $\pi_i$), we say that $bad(\pi_i)$ is the smallest subtree of $t_i$ which contains all nodes $v$ of $t_i$ where $\pi_i$ uses a transition of $A - A'$ - i.e., a transition which is not $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$-maximal. We denote by $|bad(\pi_i)|$ the number of nodes of $bad(\pi_i)$. To prove the theorem we will make the following auxiliary claim:

(C) For every prefix $\pi_i$ of an initial run of $A$ over a tree $t$ with $|bad(\pi_i)| > 1$, there exists a prefix $\pi'_i$ of a different initial $t$-run of $A$, s.t. $\pi'_i$ preserves the acceptance in $\pi_i$ and $|bad(\pi'_i)| < |bad(\pi_i)|$.

To prove (C), assume that $v$ is a leaf of $bad(\pi_i)$ labelled by some transition $\langle p, \sigma, r_1 \ldots r_{\#(\sigma)} \rangle$, where $\sigma = t(v)$ and $\langle r_1 \ldots r_{\#(\sigma)} \rangle = \langle \pi_i(v1) \ldots \pi_i(v\#(\sigma)) \rangle$. By the definition of $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$ and by the minimality of $bad(\pi_i)$, there exists a $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$-maximal transition $\tau = \langle p', \sigma, r'_1 \ldots r'_{\#(\sigma)} \rangle$ where $p \sqsubset^{\mathsf{up}}(id) p'$ and $\langle r_1 \ldots r_{\#(\sigma)} \rangle \subseteq^{\mathsf{di}} \langle r'_1 \ldots r'_{\#(\sigma)} \rangle$. Thus, it follows that there exists a $t_i$-run $\pi'_i$ of $A$ s.t. $\langle \pi'_i(v1) \ldots \pi'_i(v\#(\sigma)) \rangle = \langle r'_1 \ldots r'_{\#(\sigma)} \rangle$, and which otherwise differs from $\pi_i$ only in labels of $v$ ($\pi'_i(v) = p'$) and any of its strict prefixes $v^*$ ($\pi'_i(v^*) \sqsupset^{\mathsf{up}} (id) \pi_i(v^*)$). In other words, $bad(\pi'_i)$ differs from $bad(\pi_i)$ in that it does not contain a subtree rooted at $v$ or at one of its strict prefixes. This subtree contains at least $v$ and its children nodes, $(v1 \ldots v\#(\sigma))$, which are labelled by $\pi'_i$ using the $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$-maximal transition $\tau$. Thus, we obtain $|bad(\pi'_i)| < |bad(\pi_i)|$. Since $\sqsubset^{\mathsf{up}}(id)$ preserves the initiality of states, we have that $\pi'_i$ starts from an initial state of $A$. Moreover, since both $\sqsubset^{\mathsf{up}}(id)$ and $\subseteq^{\mathsf{di}}$ preserve the acceptance of states, $\pi'_i$ preserves the acceptance in $\pi_i$ and we conclude the proof of the claim.

The theorem is now proved as follows. By finitely many applications of (C), starting from $\pi_i$, we obtain the prefix $\pi_i^*$ (of the same depth) of an initial $t$-run $\pi'$ s.t. $\pi_i^*$ preserves the acceptance in $\pi_i$ and $bad(\pi_i^*)$ is empty. Thus $\pi_i^*$ uses only $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$-maximal transitions. Since $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$ is a strict partial order, $A'$ contains all $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{di}})$-maximal transitions of $A$, which means that $\pi_i^*$ is a run of $A'$ too. □

Theorem 4.4.3 below shows that $P(R, id)$ is GFP for every strict partial order $R \subset \subseteq^{\mathsf{up}}(id)$. The proof of Theorem 4.4.3 is obtained from the proof of Theorem 4.4.2 with a small modification: since the relation comparing the target states of the transitions is *id*, in this case the $t_i$-run $\pi_i'$ actually coincides with $\pi_i$ in the labels of children nodes of $v$. The counterexample in Figure 4.5 shows that we cannot replace *id* by $\sqsubset^{\mathsf{di}}$ as the inducing relation, by adapting the counterexample for $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}), id)$ from Section 2.4 (Figure 2.3a) to the infinite case. Therefore $P(\sqsubset^{\mathsf{up}}(R), id)$ and $P(\subset^{\mathsf{up}}(R), id)$ are not GFP for any $R \in \{\sqsubseteq^{\mathsf{di}}, \sqsubseteq^{\mathsf{de}}, \sqsubseteq^{\mathsf{f}}, \subseteq^{\mathsf{di}}, \subseteq^{\mathsf{de}}, \subseteq^{\mathsf{f}}\}$.

**Theorem 4.4.3.** *For every strict partial order $R \subset \subseteq^{\mathsf{up}}(id)$, $P(R, id)$ is GFP.*

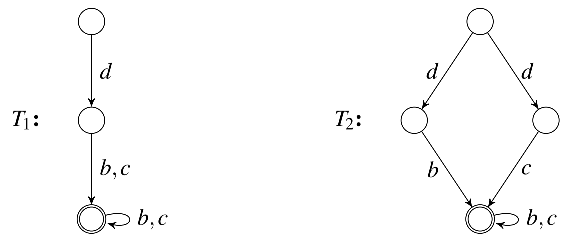
*Proof.* Let $A' = Prune(A, P(R, id))$. We will show that for every initial and fair run of $A$ over some tree $t \in \mathbb{C}(\Sigma)$, there exists a $t$-run $\hat{\pi}$ of $A'$.

By hypothesis, there exists an initial and fair $t$-run $\pi$ of $A$. Let $\pi_i$ be a prefix of $\pi$ of some depth $i \geq 1$, and $t_i$ the prefix of $t$ visited by $\pi_i$. We will show that a $t_i$-run $\pi_i'$ of $A'$ starting from an initial state exists and that $\pi_i'$ preserves the acceptance of states in $\pi_i$. This will prove that it is possible to build larger and larger prefixes of an initial $t$-run $\hat{\pi}$ of $A'$ that preserves the acceptance in $\pi$. Since $A$ is finitely branching, by König's Lemma we obtain that the run $\hat{\pi}$ itself exists and is fair.

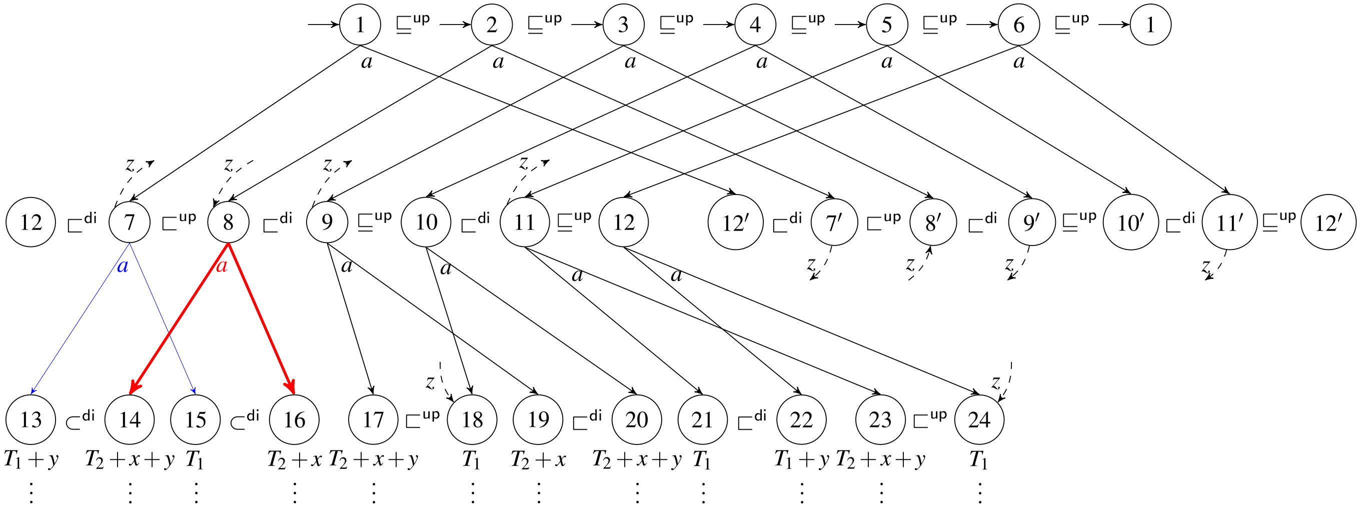
We will make use of some auxiliary notions. For $\pi_i$ a (finite) prefix of an (infinite) run $\pi$ of $A$ over a tree $t \in \mathbb{C}(\Sigma)$ (and $t_i \in \mathbb{T}(\Sigma)$ the prefix of $t$ visited by $\pi_i$), we say that $bad(\pi_i)$ is the smallest subtree of $t_i$ which contains all nodes $v$ of $t_i$ where $\pi_i$ uses a transition of $A - A'$ - i.e., a transition which is not $P(R, id)$-maximal. We denote by $|bad(\pi_i)|$ the number of nodes of $bad(\pi_i)$. To prove the theorem we will make the following auxiliary claim:

(C) For every prefix $\pi_i$ of an initial run of $A$ over a tree $t$ with $|bad(\pi_i)| > 1$, there exists a prefix $\pi_i'$ of a different initial $t$-run of $A$, s.t. $\pi_i'$ preserves the acceptance in $\pi_i$ and $|bad(\pi_i')| < |bad(\pi_i)|$.

To prove (C), assume that $v$ is a leaf of $bad(\pi_i)$ labelled by some transition $\langle p, \sigma, r_1 \ldots r_{\#(\sigma)} \rangle$, where $\sigma = t(v)$ and $\langle r_1 \ldots r_{\#(\sigma)} \rangle = \langle \pi_i(v1) \ldots \pi_i(v\#(\sigma)) \rangle$. By the definition of $P(R, id)$ and by the minimality of $bad(\pi_i)$, there exists a $P(R, id)$-maximal transition $\tau = \langle p', \sigma, r_1 \ldots r_{\#(\sigma)} \rangle$



(a) Consider the macros $T_1$ and $T_2$, which are used in $b$) to introduce new transitions and states in the third level of the automaton. Note that each of the intermediate states in $T_2$ is smaller w.r.t $\subset^{\mathsf{di}}$ than the intermediate state in $T_1$. Thus, when the first state in $T_2$ is smaller w.r.t $\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}})$ than the first state in $T_1$, each of the $d$-transitions in $T_2$ is a blue one and the $d$-transition in $T_1$ is red. Let us also consider the symbols $x$ and $y$ of rank 1. We can extend a macro $T$ by adding transitions from the first state to some new state by $x$, by $y$ or by both and we denote these by $T_x$, $T_y$ or $T_{x+y}$, resp.


1 ⊑up 2 ⊑up 3 ⊑up 4 ⊑up 5 ⊑up 6 ⊑up 1
a
12 ⊏di 7 ⊏up 8 ⊏di 9 ⊑up 10 ⊏di 11 ⊑up 12
12′ ⊏di 7′ ⊑up 8′ ⊏di 9′ ⊑up 10′ ⊏di 11′ ⊑up 12′
z
13 ⊂di 14 15 ⊂di 16 17 ⊏up 18 19 ⊏di 20 21 ⊏di 22 23 ⊏up 24
$T_1+y$
$T_2+x+y$
$T_1$
$T_2+x$
$T_2+x+y$
$T_1$
$T_2+x$
$T_2+x+y$
$T_1$
$T_1+y$
$T_2+x+y$
$T_1$

(b) We consider $\Sigma_1 = \{b,c,d,x,y,z\}$ and $\Sigma_2 = \{a\}$. The dashed arrows represent transitions by $z$ from/to some new state. Each of the six initial states has an $a$-transition to one of the states from 7 to 12 on the left and one of the states from 7′ to 12′ on the right. Any state $n'$ on the right side of the automaton does exactly the same downwardly as the state $n$ on the left side, and thus needs not be expanded in the figure. We abbreviate $\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}})$ to simply $\sqsubset^{\mathsf{up}}$ and $\sqsubseteq^{\mathsf{up}}(\sqsubset^{\mathsf{di}})$ to $\sqsubseteq^{\mathsf{up}}$.

Figure 4.4: (Adapted from Fig. 2.4 for finite tree automata.) $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}}),\subset^{\mathsf{di}})$ is not GFP since the automaton presented in $b$) cannot read the tree $a(a(d(bb\ldots),d(bb\ldots)),a(d(cc\ldots),d(cc\ldots)))$ (or any tree with just $a$'s in the first two levels) without using a blue transition: a run starting in state $1$ encounters a blue transition from $7$, as illustrated in the figure; and since $7'$ and $8'$ do the same downwardly as $7$ and $8$, respectively, and since $7' \sqsubset^{\mathsf{up}} 8'$, we have that there is a blue transition from $7'$ as well, and so $2$ cannot be used either; since $17 \sqsubset^{\mathsf{up}} 18$ we have that, as explained in $a$), there is a blue transition departing from $17$, thus a run starting at $3$ too cannot be used; and since $9$ is downwardly imitated by $9'$, a run using this state finds a blue transition as well, and so $4$ is not safe; since $23 \sqsubset^{\mathsf{up}} 24$, a blue transition from $23$ exists and so $5$ cannot be used; finally, since $11$ is imitated by $11'$, we have that a run using this state encounters a blue transition as well, and so $6$ too is not safe.

where $p \subset^{\mathsf{up}}(id)p'$. From the definition of $\subset^{\mathsf{up}}(id)$ we obtain that there exists a $t_i$-run $\pi'_i$ of $A$ which differs from $\pi_i$ only in labels of $v$ ($\pi'_i(v) = p'$) and any of its strict prefixes $v^*$ ($\pi'_i(v^*) \supseteq^{\mathsf{up}}(id)\pi_i(v^*)$). In other words, $bad(\pi'_i)$ differs from $bad(\pi_i)$ in that it does not contain a subtree rooted at $v$ or at one of its strict prefixes. Thus, we obtain $|bad(\pi'_i)| < |bad(\pi_i)|$, and since $\subseteq^{\mathsf{up}}(id)$ preserves the initiality and the acceptance of states, this concludes the proof of the claim.

The theorem is now proved as follows. By finitely many applications of (C), starting from $\pi_i$, we obtain the prefix $\pi^*_i$ (of the same depth) of an initial $t$-run $\pi'$ s.t. $\pi^*_i$ preserves the acceptance in $\pi_i$ and $bad(\pi^*_i)$ is empty. Thus $\pi^*_i$ uses only $P(R,id)$-maximal transitions. Since $P(R,id)$ is a strict partial order, $A'$ contains all $P(R,id)$-maximal transitions of $A$, which means that $\pi^*_i$ is a run of $A'$ too. □

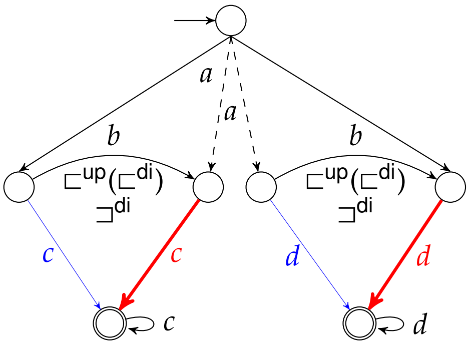

a
a
b
b
$\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}})$
$\sqsupset^{\mathsf{di}}$
$\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}})$
$\sqsupset^{\mathsf{di}}$
c
c
d
d
c
d

Figure 4.5: (Adapted from Fig. 2.3a for finite tree automata.) $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{di}}),id)$ is not GFP: if we remove the (thin) blue transitions, the automaton no longer accepts the tree $a(cc\ldots,dd\ldots)$. We are considering $\Sigma_1 = \{b,c,d\}$ and $\Sigma_2 = \{a\}$.

In Theorem 4.4.4 we prove that $P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}),\sqsubset^{\mathsf{di}})$ is GFP, from which it follows that $P(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}),\sqsubset^{\mathsf{di}})$ and $P(\subseteq^{\mathsf{up}}(id),\sqsubset^{\mathsf{di}})$ too are GFP. However, the inducing relation $\sqsubseteq^{\mathsf{di}}$ cannot be replaced by a coarser notion of simulation, since $P(\sqsubset^{\mathsf{up}}(\sqsubseteq^{\mathsf{de}}),\sqsubset^{\mathsf{di}})$ is not GFP (which follows from the counterexample in Figure 4.6). Consequently, $P(\sqsubset^{\mathsf{up}}(R),R')$ and $P(\subset^{\mathsf{up}}(R),R')$ too are not GFP for any relations $R \in \{\sqsubseteq^{\mathsf{de}},\subseteq^{\mathsf{de}},\sqsubseteq^{\mathsf{f}},\subseteq^{\mathsf{f}}\}$ and $R' \in \{\sqsubset^{\mathsf{di}},\sqsubset^{\mathsf{de}},\sqsubset^{\mathsf{f}}\}$. Moreover, using direct trace inclusion as the inducing relation too does not work, as it is shown by the counterexample in Figure 4.7, which adapts the counterexample for $P(\sqsubset^{\mathsf{up}}(\subset^{\mathsf{dw}}),\sqsubset^{\mathsf{dw}})$ for finite trees (Figure 2.3d in Section 2.4).

**Theorem 4.4.4.** *$P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}),\sqsubset^{\mathsf{di}})$ is GFP.*

*Proof.* Let $A' = Prune(A,P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}),\sqsubset^{\mathsf{di}}))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an initial and fair $t$-run $\hat{\pi}$ of $A$. We show that there is an initial and fair $t$-run $\hat{\pi}'$ of $A'$.

We say that a $t$-run $\pi$ is $i$-good if it does not contain any transition from $A - A'$ from any position $v \in \mathbb{N}^*$ s.t. $|v| < i$. I.e., no pruned transitions are used in the first $i$ levels of the tree. We will use induction on $i \geq 0$ to show that, for every $i$ and every initial and fair $t$-run $\pi$ of $A$ there exists an $i$-good initial and fair $t$-run $\pi'$ in $A$ s.t. $level_i(\pi) \sqsubseteq^{\mathsf{di}} level_i(\pi')$.

The base case $i = 0$ is trivial, as every $t$-run of $A$ is 0-good itself.

For the induction step, let $S$ be the set of all $(i-1)$-good initial and fair $t$-runs $\pi'$ of $A$ s.t. $level_{i-1}(\pi) \sqsubseteq^{\mathsf{di}} level_{i-1}(\pi')$. Since $\pi$ is an initial and fair $t$-run, by induction hypothesis, $S$ is non-empty. Let $S' \subseteq S$ be the subset of $S$ containing those runs $\pi' \in S$ that additionally satisfy $level_i(\pi) \sqsubseteq^{\mathsf{di}} level_i(\pi')$. From $level_{i-1}(\pi) \sqsubseteq^{\mathsf{di}} level_{i-1}(\pi')$ and the fact that $\sqsubseteq^{\mathsf{di}}$ is preserved downward-stepwise, we obtain that $S'$ is non-empty. Now we can select some $\pi' \in S'$ s.t. $level_i(\pi')$ is maximal w.r.t. $\sqsubseteq^{\mathsf{di}}$ relative to the other runs in $S'$. We claim that $\pi'$ is $i$-good and $level_i(\pi) \sqsubseteq^{\mathsf{di}} level_i(\pi')$. The second part of this claim holds because $\pi' \in S'$.

We show that $\pi'$ is $i$-good by contraposition. Suppose $\pi'$ is not $i$-good. Then it must contain a transition $\langle p,\sigma,q_1 \ldots q_n\rangle$ from $A - A'$, where $n = \#(\sigma)$. Since $\pi'$ is $(i-1)$-good, this transition must start at depth $(i-1)$ in the tree. Since $A' = Prune(A, P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}), \sqsubset^{\mathsf{di}}))$, there must exist another transition $\langle p',\sigma,q'_1 \ldots q'_n\rangle$ in $A'$ s.t. $p \subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}}) p'$ and $\langle q_1,\ldots,q_n\rangle \sqsubset^{\mathsf{di}} \langle q'_1,\ldots,q'_n\rangle$. From the definition of $\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{di}})$ we obtain that there exists another initial and fair $t$-run $\pi_1$ in $A$ (that uses the transition $\langle p',\sigma,q'_1 \ldots q'_n\rangle$) s.t. $level_i(\pi') \sqsubset^{\mathsf{di}} level_i(\pi_1)$. The run $\pi_1$ is not necessarily $i$-good or $(i-1)$-good. However, by induction hypothesis, we know there exists a $(i-1)$-good initial and fair $t$-run $\pi_2$ in $A$ s.t. $level_{i-1}(\pi_1) \sqsubseteq^{\mathsf{di}} level_{i-1}(\pi_2)$. Since $\sqsubseteq^{\mathsf{di}}$ is preserved stepwise, there also exists an initial and fair $t$-run $\pi_3$ in $A$ (that coincides with $\pi_2$ up-to depth $(i-1)$), which is $(i-1)$-good and satisfies $level_i(\pi_1) \sqsubseteq^{\mathsf{di}} level_i(\pi_3)$. In particular, $\pi_3 \in S'$. From $level_i(\pi') \sqsubset^{\mathsf{di}} level_i(\pi_1)$ and $level_i(\pi_1) \sqsubseteq^{\mathsf{di}} level_i(\pi_3)$ we obtain $level_i(\pi') \sqsubset^{\mathsf{di}} level_i(\pi_3)$. This contradicts our condition above that $\pi'$ must be $level_i$-maximal w.r.t. $\sqsubseteq^{\mathsf{di}}$ in $S'$. This concludes the induction step.

If $t \in L(A)$ then there exists an initial and fair $t$-run $\hat{\pi}$ of $A$. By the property we proved above, it is possible to obtain initial and fair $t$-runs $\hat{\pi}'$ of $A$ which are $i$-good for arbitrary values of $i \geq 0$. Thus, since $A$ is finitely branching it follows from König's Lemma that there exists an initial and fair $t$-run $\hat{\pi}^*$ which is $i$-good for any $i$, i.e., $\hat{\pi}^*$ does not use any transition from $A - A'$ at all. Therefore, $\hat{\pi}^*$ is a run of $A'$, and so $t \in L(A')$. □

Finally, it has been shown that $P(\sqsubset^{\mathsf{bw}}, \sqsubset^{\mathsf{de}})$ and $P(\subset^{\mathsf{bw}}, \subset^{\mathsf{di}})$ are not GFP for infinite words [CM13]. From the first result we have that $P(\sqsubset^{\mathsf{up}}(R), R')$ and $P(\subset^{\mathsf{up}}(R), R')$ are not GFP for trees regardless of $R$ and for any relation $R' \in \{\sqsubset^{\mathsf{de}}, \sqsubset^{\mathsf{f}}, \subset^{\mathsf{de}}, \subset^{\mathsf{f}}\}$. From the second result we obtain that $P(\subset^{\mathsf{up}}(R), \subset^{\mathsf{di}})$ too is not GFP for any $R$.

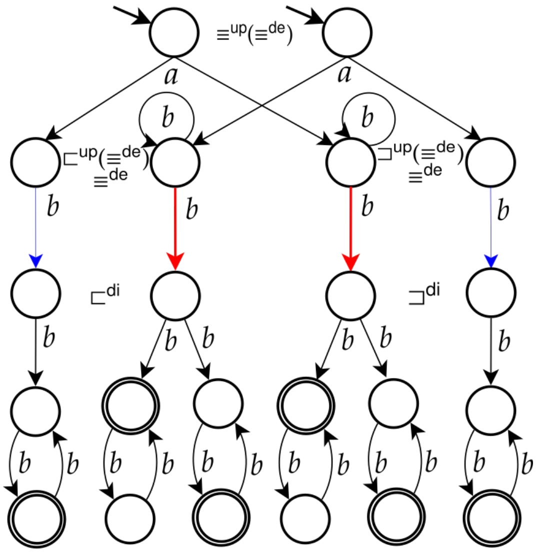


Figure 4.6: $P(\sqsubset^{\mathrm{up}}(\equiv^{\mathrm{de}}),\sqsubset^{\mathrm{di}})$ is not GFP: if we remove the blue (thin) transitions, the language of the automaton becomes empty.

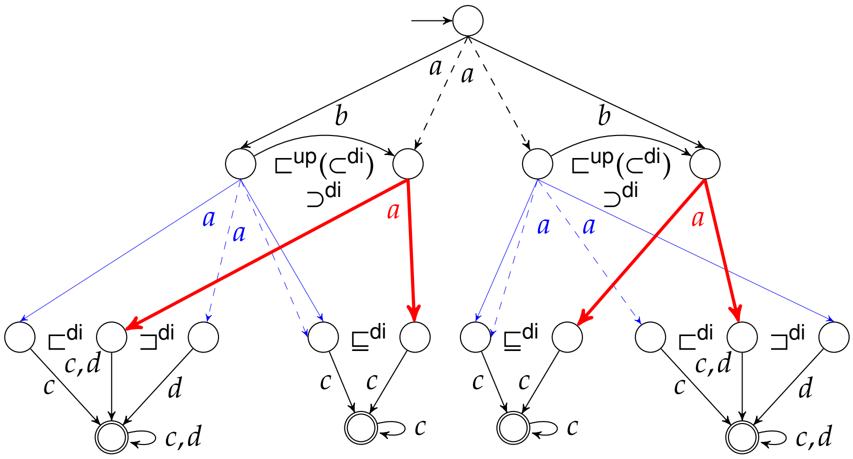


Figure 4.7: (Adapted from Fig. 2.3d for finite tree automata.) $P(\sqsubset^{\mathrm{up}}(\subset^{\mathrm{di}}),\sqsubset^{\mathrm{di}})$ is not GFP: if we remove the (thin) blue transitions, the tree $a(a(cc\ldots,cc\ldots),a(cc\ldots,cc\ldots))$ is no longer accepted. We are considering $\Sigma_1 = \{b,c,d\}$ and $\Sigma_2 = \{a\}$.

The table in Figure 4.8 summarizes our results, providing a complete picture of which combinations of strict upward and strict direct/delayed/fair downward relations are or are not good for pruning. Note that negative results propagate to larger relations and positive results propagate to smaller relations (i.e., GFP is downward closed).

| | | $R_{\mathsf{d}}$ | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $R_{\mathsf{u}} \backslash R_i$ | | $id$ | $\sqsubset^{\mathsf{di}}$ | $\sqsubset^{\mathsf{de}}$ | $\sqsubset^{\mathsf{f}}$ | $\subset^{\mathsf{di}}$ | $\subset^{\mathsf{de}}$ | $\subset^{\mathsf{f}}$ |
| $id$ | | × | ✓ | × | × | ✓ | × | × |
| $\sqsubset^{\mathsf{up}}$ | $id$ | ✓ | ✓ | × | × | ✓ | × | × |
| | $\sqsubseteq^{\mathsf{di}}$ | × | ✓ | × | × | × | × | × |
| | $\sqsubseteq^{\mathsf{de}}$ | × | × | × | × | × | × | × |
| | $\sqsubseteq^{\mathsf{f}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{di}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{de}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{f}}$ | × | × | × | × | × | × | × |
| $\subset^{\mathsf{up}}$ | $id$ | ✓ | ✓ | × | × | × | × | × |
| | $\sqsubseteq^{\mathsf{di}}$ | × | ✓ | × | × | × | × | × |
| | $\sqsubseteq^{\mathsf{de}}$ | × | × | × | × | × | × | × |
| | $\sqsubseteq^{\mathsf{f}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{di}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{de}}$ | × | × | × | × | × | × | × |
| | $\subseteq^{\mathsf{f}}$ | × | × | × | × | × | × | × |

Figure 4.8: GFP relations $P(R_{\mathsf{u}}(R_i), R_{\mathsf{d}})$ for infinite tree automata. The coarsest GFP relations in the table are marked with ✓, and the (finer) relations which under-approximate them are marked with ✓. Relations which are not GFP are marked with ×. Note that $P(id, id)$ is reflexive (and therefore not asymmetric) and so it is not GFP.

## 4.5 Conclusion

In this chapter we explored transition pruning and state quotienting techniques for Büchi tree automata. We proved that the pruning and quotienting results for finite trees carry over to the case of infinite trees, but also that delayed downward fixed-tree simulation (which subsumes PTIME-computable delayed downward simulation) is suitable for quotienting.

# Chapter 5

# Conclusion and Future Work

In this thesis we presented efficient techniques for reducing nondeterministic automata, both for the case of finite trees and for infinite trees. The techniques are based primarily on pruning transitions and quotienting states in the automata. We presented new combinations of upward and downward relations (simulations and trace inclusions) which preserve the language during pruning. Similarly, we showed which downward and upward preorders are suitable for quotienting. Many of these results are nontrivial and counterintuitive, as it is shown by the irregular layout of the table of pruning results in Chapter 2.

Based on the pruning and quotienting results for finite tree automata, in Chapter 2 we defined the reduction algorithm *Heavy*. We proved that quotienting with the combined preorder - one of the best reduction methods known for reducing tree automata - cannot achieve any further reduction on automata on finite trees which have already been reduced with *Heavy*. In Chapter 3, we gave experimental evidence that, in practice, this method is outperformed by *Heavy* on average, either in terms of reduction achieved or overall running times.

We introduced yet another reduction technique in Chapter 2 for finite trees, called transition saturation, which is the dual notion of pruning. It consists of adding new transitions to the automaton w.r.t. a combination of upward and downward relations, and we presented a comprehensive landscape of combinations of relations which are language-preserving for finite tree automata. Taking reduction one step further, in Chapter 2 we presented a second algorithm, *Sat*, that combines *Heavy* with the saturation techniques.

In Chapter 3 we performed an experimental evaluation of the reduction algorithms *Heavy* and *Sat* for finite tree automata. We concluded that *Heavy* performs very well in practice, yielding substantial reductions both on automata from program verification provenience and on different classes of randomly generated automata. It was also clear that *Sat*, on average, quotients more states and, in several cases, prunes more transitions than *Heavy*.

A second experimental phase detailed in Chapter 3 showed how both *Heavy* and *Sat* can make hard computations like complementation much more tractable, allowing both to obtain smaller complement automata and at much lower computation times.

In the future we would like to study how the transition saturation results for finite trees carry over to the case of infinite trees, also exploring possible good results using delayed or fair simulations and trace inclusions. We would also like to define algorithms in the fashion of *Heavy* and *Sat* but for the case of infinite trees, implement them either as part of `minotaut` or on top of an already-existing efficient tool supporting infinite tree automata, and conduct experiments to assess how the reduction techniques behave in practice for infinite tree automata.

Finally, it would be interesting to explore the possible contribution that our reduction techniques may have in finding more efficient decision algorithms for different logics. For instance, automata on finite trees have recently been used to decide the Weak monadic Second-order theory of 2 Successors (W2S2) logic [BKM15], as well as the fragment of separation logic [ELSV14, IRV14].

# Bibliography

[ABH+08] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati, and Tomás Vojnar. Computing simulations over tree automata. In *TACAS*, volume 4963 of *LNCS*, pages 93–108, 2008.

[ACH+10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, and Tomás Vojnar. When simulation meets antichains. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2010.

[AHJ+13] Parosh Aziz Abdulla, Lukás Holík, Bengt Jonsson, Ondrej Lengál, Cong Quy Trinh, and Tomás Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2013.

[AHKV09] Parosh Aziz Abdulla, Lukás Holík, Lisa Kaati, and Tomás Vojnar. A uniform (bi-)simulation-based framework for reducing tree automata. *Electr. Notes Theor. Comput. Sci.*, 251:27–48, 2009.

[AHM16] Ricardo Almeida, Lukás Holík, and Richard Mayr. Reduction of nondeterministic tree automata. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 717–735. Springer, 2016.

[ALdR05] Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Simulation-based iteration of tree transducers. In *Proc. TACAS '05-11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, 2005.

[ALdR06] Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Tree regular model checking: A simulation-based approach. *J. Log. Algebr. Program.*, 69(1-2):93–121, 2006.

[Alm16a] R. Almeida. MinOTAut. `https://github.com/ric-almeida/heavy-minotaut`, 2016.

[Alm16b] Ricardo Almeida. Reducing nondeterministic tree automata by adding transitions. In Jan Bouda, Lukás Holík, Jan Kofron, Jan Strejcek, and Adam Rambousek, editors, *Proceedings 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2016, Telč, Czech Republic, 21st-23rd October 2016.*, volume 233 of *EPTCS*, pages 33–51, 2016.

[BHH$^+$08] Ahmed Bouajjani, Peter Habermehl, Lukás Holík, Tayssir Touili, and Tomás Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In Oscar H. Ibarra and Bala Ravikumar, editors, *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2008.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking of complex dynamic data structures. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70. Springer, 2006.

[BKM15] D. Basin, N. Karlund, and A. Møller. Mona. `http://www.brics.dk/mona`, 2015.

[Büc90] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[C:c] C++ documentation (string concatenation). `http://www.cplusplus.com/reference/string/string/operator+/`. Accessed: 2017-01-14.

[CDG$^+$08] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2008. release November, 18th 2008.

[C:i] C++ documentation (set comparison object). `http://www.cplusplus.com/reference/set/set/key_comp/`. Accessed: 2017-01-15.

[C:l] C++ documentation (inline functions). `http://www.cplusplus.com/articles/2LywvCM9/`. Accessed: 2017-01-15.

[Cle11] Lorenzo Clemente. Büchi automata can have smaller quotients. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2011.

[Cle12] Lorenzo Clemente. *Generalized simulation relations with applications in automata theory*. PhD thesis, University of Edinburgh, UK, 2012.

[C:m] C++ documentation (map container). `http://www.cplusplus.com/reference/map/map/`. Accessed: 2017-01-16.

[CM13] Lorenzo Clemente and Richard Mayr. Advanced automata minimization. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 63–74. ACM, 2013.

[CM16] Lorenzo Clemente and Richard Mayr. Efficient reduction of nondeterministic automata with application to language inclusion testing. *Submitted to LMCS*, March 2016.

[C:s] C++ documentation (set container). `http://www.cplusplus.com/reference/set/set/`. Accessed: 2017-01-16.

[C:ua] C++ documentation (unordered map container). `http://www.cplusplus.com/reference/unordered_map/unordered_map/`. Accessed: 2017-01-16.

[C:ub] C++ documentation (unordered set container). `http://www.cplusplus.com/reference/unordered_set/unordered_set/`. Accessed: 2017-01-16.

[DHW91] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In Kim Guldstrand Larsen and Arne Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 255–265. Springer, 1991.

[Dur15] I. Durand. Autowrite. `http://dept-info.labri.fr/~idurand/autowrite/`, 2015.

[ea15] T. Genet et al. Timbuk. `http://www.irisa.fr/celtique/genet/timbuk/`, 2015.

[ELS+17] Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, Tomás Vojnar, and Zhilin Wu. SPEN: Separation logic entailment. `https://www.irif.fr/~sighirea/spen/`, 2017.

[ELSV14] Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, and Tomás Vojnar. Compositional entailment checking for a fragment of separation logic. In Garrigue [Gar14], pages 314–333.

[Ete02] Kousha Etessami. A hierarchy of polynomial-time computable simulations for automata. In *CONCUR*, volume 2421 of *LNCS*, pages 131–144, 2002.

[EWS05] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. *SIAM J. Comput.*, 34(5):1159–1175, 2005.

[FKWV13] Seth Fogarty, Orna Kupferman, Thomas Wilke, and Moshe Y. Vardi. Unifying Büchi complementation constructions. *Logical Methods in Computer Science*, 9(1), 2013.

[Gar14] Jacques Garrigue, editor. *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*. Springer, 2014.

[HHP14] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination analysis by learning terminating programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.

[HHR⁺11] Peter Habermehl, Lukás Holík, Adam Rogalewicz, Jirí Simácek, and Tomás Vojnar. Forest automata for verification of heap manipulation. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *LNCS*, pages 424–440, 2011.

[HKR02] Thomas A. Henzinger, Orna Kupferman, and Sriram K. Rajamani. Fair simulation. *Inf. Comput.*, 173(1):64–81, 2002.

[HLR⁺13] Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek, and Tomás Vojnar. Fully automated shape analysis based on forest automata. In *CAV*, volume 8044 of *LNCS*, pages 740–755, 2013.

[HLSV11] Lukás Holík, Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. Efficient inclusion checking on explicit and semi-symbolic tree automata. In *ATVA*, volume 6996 of *LNCS*, pages 243–258, 2011.

[Hol11] Lukás Holík. *Simulations and Antichains for Efficient Handling of Finite Automata*. PhD thesis, Faculty of Information Technology of Brno University of Technology, 2011.

[Hos10] Haruo Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press, 2010.

[HVP05] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[IRS13] Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013.

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Deciding entailments in inductive separation logic with tree automata. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2014.

[IRV15] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. SLIDE: Separation logic with inductive definitions. `http://www.fit.vutbr.cz/research/groups/verifit/tools/slide/`, 2015.

[KV98] Orna Kupferman and Moshe Y. Vardi. Verification of fair transition systems. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[KV01] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Logic*, 2(3):408–429, July 2001.

[KW08] Detlef Kähler and Thomas Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 724–735. Springer, 2008.

[Lan17] LanguageInclusion.org. RABIT: Ramsey-based Büchi automata inclusion testing. `http://languageinclusion.org/doku.php?id=tools`, Access date:27.04.2017.

[LSV12] Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2012.

[LSV15] Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. Libvata: highly optimised non-deterministic finite tree automata library. `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`, 2015.

[LSV17a] Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. Libvata: highly optimised non-deterministic finite tree automata library. `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`, Access date:24.05.2017.

[LSV$^+$17b] Ondrej Lengál, Jirí Simácek, Tomás Vojnar, Peter Habermehl, Lukás Holík, and Adam Rogalewicz. Forester: tool for verification of programs with pointers. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`, 2017.

[Pit06] Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 255–264. IEEE Computer Society, 2006.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Ram87] F.P. Ramsey. On a problem of formal logic. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, Modern Birkhäuser Classics, pages 1–24. Birkhäuser Boston, Boston, MA, 1987.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[Saf88] S. Safra. On the complexity of omega-automata. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:319–327, 1988.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.

[TCT$^+$08] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Wen-Chin Chan, and Chi-Jian Luo. Goal extended: Towards a research tool for omega automata and temporal logic. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 346–350. Springer Berlin Heidelberg, 2008.

[TFVT14] Ming-Hsien Tsai, Seth Fogarty, Moshe Y. Vardi, and Yih-Kuen Tsay. State of Büchi complementation. *Logical Methods in Computer Science*, 10(4), 2014.

[TTLH15] Yih-Kuen Tsay, Ming-Hsien Tsai, Chi-Shiang Liu, and Yu-Shiang Hwang. GOAL: graphical interactive tool for defining and manipulating Büchi automata and temporal logic formulae. `http://goal.im.ntu.edu.tw/wiki/doku.php`, 2015.

[TV07] Deian Tabakov and Moshe Y. Vardi. Model checking Buechi specifications. In Remco Loos, Szilárd Zsolt Fazekas, and Carlos Martín-Vide, editors, *LATA 2007. Proceedings of the 1st International Conference on Language and Automata Theory and Applications.*, volume Report 35/07, pages 565–576. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona, 2007.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Logic*, 51(1-2):79–98, 1991.

[vB08] Thomas von Bomhard. Minimization of tree automata, 2008.

[VW08] Moshe Y. Vardi and Thomas Wilke. Automata: from logics to algorithms. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].*, volume 2 of *Texts in Logic and Games*, pages 629–736. Amsterdam University Press, 2008.

[WDHfR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J. f. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *In Proc. of CAV 2006, LNCS 4144*, pages 17–30. Springer-Verlag, 2006.