BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EFFICIENT ALGORITHMS FOR TREE AUTOMATA

EFEKTIVNÍ ALGORITMY PRO STROMOVÉ AUTOMATY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **ONDŘEJ VALEŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Valeš Ondřej, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Efektivní algoritmy pro stromové automaty**
Efficient Algorithms for Tree Automata
Kategorie: Teoretická informatika

Zadání:

1. Seznamte se s (konečnými) stromovými automaty nad konečnými slovy.
2. Seznamte se s technikami antichainů a kongruencí pro efektivní testování inkluze jazyků konečných automatů.
3. Seznamte se s knihovnou libVATA pro práci se stromovými automaty.
4. Navrhněte a implementujte algoritmy pro efektivní testování inkluze a ekvivalence jazyků stromových automatů.
5. Navržené algoritmy testujte na sadě stromových automatů dle pokynů vedoucího.
6. Zhodnoťte dosažené výsledky a diskutujte možná pokračování práce.

Literatura:

- Tree Automata Techniques and Applications. http://tata.gforge.inria.fr
- knihovna libVATA. https://github.com/ondrik/libvata

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Lengál Ondřej, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstract

In this work a novel algorithm for testing language equivalence and inclusion on tree automata is proposed and implemented as a module in the VATA library. First, existing approaches to equivalence and inclusion testing on both word and tree automata are examined. These existing approaches are then modified to create bisimulation up-to congruence algorithm for tree automata and a formal proof of the soundness of the new algorithm is provided. Efficiency of this new approach is compared with existing language equivalence and inclusion testing methods for tree automata, showing the performance of our algorithm on hard cases is often superior.

Abstrakt

Cílem této práce je navržení efektivních algoritmů pro testování jazykové ekvivalence a inkluze stromových automatů a dále pak implementace těchto algoritmů jako rozšíření knihovny VATA. Nejprve je provedena rešerše existujících přístupů testování ekvivalence a inkluze slovních i stromových automatů. Z nich poté vychází návrh nového přístupu k testování ekvivalence a inkluze jazyků stromových automatů založený na bisimulaci vzhledem ke kongruenci, ke kterému je představen formální důkaz korektnosti. Součástí práce je také srovnání efektivity představeného algoritmu a již existujících přístupů, které ukazuje, že na obtížných případech je náš algoritmus často lepší než existující přístupy.

Keywords

word automata, tree automata, language equivalence, language inclusion, bisimulation, antichains, bisimulation up-to congruence

Klíčová slova

slovní automaty, stromové automaty, jazyková ekvivalence, jazyková inkluze, bisimulace, antichainy, bisimulation up-to congruence

Reference

VALEŠ, Ondřej. *Efficient Algorithms for Tree Automata.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Cílem této práce je navržení efektivních algoritmů pro testování jazykové ekvivalence a inkluze stromových automatů. Stromové automaty jsou rozšířením běžně užívaných slovních automatů, které umožňují zpracovávat stromy, což jsou acyklické strukturované entity.

Alternativou k využívání stromových automatů je serializace stromů do podoby lineárních slov a následné zpracování například pomocí zásobníkových automatů. Výhodou stromových automatů oproti alternativním přístupům, konkrétně oproti zásobníkovým automatům, je rozhodnutelnost problémů spojených s jazyky příjmanými stromovými automaty a uzavřenost vůči základním množinovým operacím nad jazyky. Nad jazyky zásobníkových automatů nelze rozhodnout například inkluzi a třída těchto jazyků není uzavřená vůči průniku ani komplementu (případně není uzavřená vůči sjednocení a průniku pro deterministické zásobníkové automaty). Všechny tyto problémy jsou ale rozhodnutelné pro stromové automaty.

Proto jsou stromové automaty využívány při verifikaci, když je potřeba popsat entity stromového charakteru a současně potom nad touto reprezentací provádět další operace, jako je sjednocení nebo průnik jazyků a nebo rozhodovat inkluzi. Konkrétně jsou stromové automaty využívány v abstraktním regulárním model checkingu pro verifikaci dynamických datových (nejen) struktur tvaru stromu na haldě.

Další oblastí s vazbou na stromové automaty je logika WSkS (*weak second-order logic of k successors*), zde existuje spojení mezi logickými formulemi a jazyky stromových automatů. Ke každé formuli ve WSkS existuje stromový automat popisující hodnoty proměnných ve formuli, kde každému konkrétnímu ohodnocení náleží strom a daný automat tento strom přijme právě tehdy, když je formule splnitelná s ohodnocením, které odpovídá tomuto stromu. Rozhodování splnitelnosti WSkS formulí se tedy dá redukovat na problém (ne)prázdnosti jazyka stromového automatu a otázka, zda je formule platná, zase na problém univerzality jazyka.

Nejprve je provedena rešerše existujících přístupů testování ekvivalence a inkluze slovních i stromových automatů. Z nich poté vychází návrh nového přístupu k testování ekvivalence a inkluze jazyků stromových automatů. K tomuto účelu byl vybrán algoritmus *bisimulation up-to congruence*, jenž už existoval ve variantě pro slovní automaty. Tento algoritmus byl v práci netriviálně rozšířen pro využití se stromovými automaty. Zakladní myšlenkou je synchronní procházení determinizovanými verzemi obou automatů a ověřování, jestli všechny přechody v jednom automatu lze simulovat v tom druhém a naopak. Tento algoritmus ve výsledku vytvoří bisimulaci mezi stavy vstupních automatů, což je relace spojující z pohledu zpracování stromů ekvivalentní stavy. Pokud je tato relace vytvořena úspěšně, znamená to, že vstupní automaty příjmají stejný jazyk a pokud není vytvořena úspěšně, byl nalezen protipříklad — strom příjímaný pouze jedním ze vstupních automatů — a automaty příjímají různé jazyky.

Hlavní rozšíření zlepšující efektivitu testování jazykové ekvivalence je využití kongruenčního uzávěru vytvářené relace bisimulace. Pokud je nějaký pár stavů, na který narazí algoritmus při synchronním procházení automatů, v kongruenčním uzávěru už známých párů, je možné takovýto pár vyloučit z dalšího prohledávání, protože je pokryt už prohledanými páry. To je možné udělat z toho důvodu, že pokud by vedl takto zahozený pár na protipříklad, musí nutně vést na protipříklad alespoň jeden z párů, které ho pokrývají. Takže zahození párů z kongručního uzávěru nemá vliv na výsledek testování ekvivalence. Tento algoritmus byl implementován jako rozšíření knihovny VATA.

K tomuto algoritmu je poté představen formální důkaz korektnosti založený na kompatibilních fukcích, pro které je dokázáno, že jejich aplikace na vytvářenou relaci bisimulace

neovlivňuje výsledek testu jazykové ekvivalence. Poté je dokázáno, že kongruenční uzávěr je kompatibliní funkce, což dokazuje korektnost celého algoritmu.

Součástí práce je také srovnání efektivity představeného algoritmu a již existujících přístupů, konkrétně srovnání s algoritmy založenými na *antichainech* (ve variantě *bottom-up* i *top-down*), které ukazuje, že na obtížných případech je náš algoritmus často lepší než tyto přístupy. Srovnání bylo provedeno na celkem 9025 testovacích případech, z nichž bylo 594 platných ekvivalencí (z toho 499 na neidentických automatech a 95 na identických automatech) a 8426 neplatných ekvivalencí. V případě neplatných ekvivalencí byl algoritmus *bisimulation up-to congruence* horší, protože má vyšší nároky na předzpracování vstupních automatů a většina neplatných ekvivalencí byla rozhodnuta dřív, než se projevila lepší schopnost ořezávat prohledávaný prostor algoritmu *bisimulation up-to congrunce*. Naopak pro platné ekvivalence, kde je potřeba prohledat celý stavový prostor obou automatů, protože není nalezen žádný protipříklad, dokázal algoritmus *bisimulation up-to congruence* doběhnout v lepším čase pro většinu případů. V několika případech byl algoritmus *bisimulation up-to cogruence* dokonce rychlejší o řád.

Cíl projektu se tedy podařilo splnit: byl navrhnut nový algoritmus na testování jazykové ekvivalence, který je alespoň pro část případů efektivnější, než existující přístupy.

Efficient Algorithms for Tree Automata

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Ondřej Lengál, Ph.D. The supplementary information was provided by Prof. Ing. Tomáš Vojnar, Ph.D. and Mgr. Lukáš Holík, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Ondřej Valeš
May 14, 2019

Acknowledgements

I would like to express my thanks to Ing. Ondřej Lengál, Ph.D. and Prof. Ing. Tomáš Vojnar, Ph.D. for extensive consultations they readily participated in, their willingness to share their knowledge of the topic, and for all the time and energy they invested into this thesis.

Contents

Chapter 1

Introduction

The field of formal languages in computer science is heavily based on language classes from Chomsky hierarchy. A common feature of all those languages is the assumption that the languages contain linear words only. Therefore each symbol in the word except the last one is followed by exactly one symbol. Limiting words to strictly linear structures introduces problems whenever a formal language has to describe entities with deeply inherent structure, for example the syntax of a programming language or mathematical formulae.

The usual solution uses linearization, which encodes structured entity into a linear word by means of adding parentheses or other symbols with similar functionality. Those symbols hold no information about the actual content of the entity, but only define its structure, artificially increasing the size of the encoding. A bigger issue is that linearization may break some desired properties of the encoded entities, such as their regular structure.

An alternative solution is to allow languages to contain non-linearly structured words (words where symbols can have multiple following symbols). Such words then have a tree structure where nodes represent symbols and edges represent structured information, thereby naturally and directly describing both content and structure of the underlying entities without introducing structure-encoding symbols.

For example, a language working with a Boolean algebra over the domain $\{0, 1\}$ can contain a word $(1*(0+1))$ (expressed using linearization with parentheses). This word has a high ratio of symbols encoding structure information (4 parentheses and 5 content symbols). If we enable symbols to have multiple successors, the same word can be expressed as the symbol `*` followed by symbols `1` and `+`, where `1` has zero successors and `+` has successors `0` and `1`.

Structured entities linearized with parentheses cannot be reliably parsed with finite word automata because languages consisting of such words generally belong into the context-free language family, therefore parsing with a pushdown automaton or a similar tool is required. But in doing so many desirable properties of regular languages are sacrificed, especially the loss of closure properties is problematic for many applications working with context-free languages.

Finite tree automata (FTA) can be then viewed as an extension of commonly used finite word automata that allows processing of languages with words that do not have a strictly linear structure. These automata (and languages they describe) are used in the fields of verification, theorem proving, database systems, and language manipulation based on XML schema [5, 11].

The main advantage of using tree automata is their ability to parse structured entities directly. Moreover, languages accepted by tree automata retain some desirable closure

properties that are lost if linearization with parentheses and pushdown automata are used. Thus tree automata can be more suited for many applications where using closure properties is necessary or advantageous.

For instance, regular model checking for verification of safety-critical systems with dynamic memory uses finite automata for representing heap objects, where language of the automaton encodes possible states or value of the corresponding heap object. Language inclusion and equivalence are important for checking whether a set of possible states of a given object encoded as an automaton language is closed under the transition relation.

Also there exists a connection between tree languages and models of WSkS formulae, which can be exploited to decide problems stated in terms of WSkS using language operations on tree automata [5, 12].

As with regular languages over finite words, regular tree languages (languages accepted by NFTA) can be tested for language equivalence and inclusion. Especially in the field of formal verification, equivalence and inclusion testing play a major role. Because determining equivalence and inclusion of NFTA is a problem with an exponential-time worst-case lower bound [15], there is a strong incentive to develop efficient heuristics that can perform the test fast on real-world use cases.

1.1 Previous Work

This thesis builds on previous work done in [16] where the first prototype of *bisimulation up-to congruence* over tree automata for the VATA library was presented. In this work the *bisimulation up-to congruence* prototype is developed into a fully functional algorithm, several optimizations are implemented, and its efficiency is compared with state-of-the-art approaches. Also a formal proof of the correctness of this algorithm is presented.

1.2 Project Goal

The goal of this thesis is to modify *bisimulation up-to congruence* [3], an existing algorithm for testing language equivalence over nondeterministic finite word automata, and adapt it for the use with finite tree automata.

Its main idea is based on optimizing the classical language equivalence testing procedure, which is based on first determinizing both NFAs and then trying to find a bisimulation between their initial states. *Bisimulation up-to congruence* extends this procedure by eliminating some of the checked state pairs based on information gained from already processed pairs and thus reduces the size of the search space, total number of processed pairs, and time required to perform language equivalence or inclusion check. This approach was shown to be an improvement over preceding techniques resulting in significant reduction in run times with relative reduction growing with automata size.

However *bisimulation up-to congruence* was only used with (and its soundness proven for) finite word automata. Substantial changes for both the algorithm and the proof resulting from different word structures used by word and tree automata are necessary to successfully adapt this approach for finite tree automata. This includes extension of the notion of transition function, which is used to compute successors, that would enable computing successors with regard to set operations (union and intersection) on nondeterministic automata states.

The main components of this thesis, therefore, are: extension of the transition function that enables to calculate successors after set operations are performed, implementation of the modified *bisimulation up-to congruence* algorithm as an extension of the VATA library [14], formal proof of the algorithm's soundness, and comparison with existing equivalence and inclusion testing approaches already implemented in the VATA library (namely algorithms based on antichains).

1.3 Document Structure

In Chapter 2 theory of finite word automata is examined. There, preliminary information that is necessary to understand *Bisimulation up-to congruence* for word automata, on which the work done in this thesis builds is introduced. This includes the definition of finite word automaton, regular languages, and examines closure properties of regular and context-free languages.

In Chapter 3 tree automata are introduced. There, formal definitions of a tree, ranked alphabet, tree automaton, run of a tree automaton on a tree, and a language accepted by a tree automaton are provided. Then, *bottom-up* automata, *top-down* automata and determinism are explored and closure properties of the class of regular tree languages are studied. This chapter also includes several examples of tree automata and their languages, together with examples of successful and unsuccessful runs and categorization of different types of tree automata. In Chapter 3.7 definitions for transition function on intersections and unions of macrostates are proposed and their correctness proven. Those notions are then used in the proof of *bisimulation up-to congruence* for tree automata.

Chapter 4 describes existing approaches to equivalence and inclusion testing of the languages of tree automata (minimization, simulations, and antichains). Those approaches constitute the state-of-the-art for language equivalence and inclusion testing on tree automata. Chapter 5 studies the *bisimulation up-to congruence* algorithm for finite word automata. This algorithm forms the basis on which this thesis builds and is later modified for use with tree automata.

In Chapter 6, a modified version of the bisimulation up-to congruence algorithm for tree automata is proposed and possible optimizations are discussed. First, the textbook algorithm for inclusion and equivalence testing is presented. This algorithm is then optimized by eliminating repeated calculations of successors. Lastly, this optimized textbook algorithm is combined with congruence closure for tree automata. Formal proof for this adapted algorithm is proposed in Chapter 7; it is build in a similar fashion to the proof of *bisimulation up-to congruence* for word automata presented in [3]. First, the correctness of the naive algorithm and the algorithm without repeated successors calculation is proven, then this proof is extended to cover *bisimulation up-to congruence* by introducing the notion of *compatible functions* and proving that congruence closure is in fact compatible.

Chapter 8 describes technical details of the implementation in the VATA library, focusing on added cache features and their influence on the overall performance. Lastly, Chapter 9 provides comparison of bisimulation up-to congruence with existing approaches for equivalence testing on NFTA implemented in the VATA library, namely algorithms based on antichains. Also metrics regarding the effectiveness of the congruence closure, its computational requirements and percentage of processed pairs found in the closure (and thus eliminated from further search) are presented.

Chapter 2

Word Automata

Finite word automaton is a type of state machine that takes words as inputs and determines whether the input has certain properties. The range of properties finite word automata can reliably check is somewhat limited, in fact, they are able to check exactly the same properties as those that can be expressed using regular expressions. Finite word automata therefore are one of the tools that can be used to define and recognize regular languages [17].

The class of regular languages is both prominent, forming a foundation of Chomsky hierarchy, and widely used in computer science. And because finite word automata are a natural tool for describing those languages, they are routinely used in many fields of computer science. Finite word automata are for example used for lexicographic analysis, modeling of state systems, text manipulation and matching, and representing dynamic heap objects in regular model checking.

To be able to recognize more complex languages than regular ones, it is necessary to use other tools. One possibility is to use pushdown automata but this sacrifices desirable closure properties, intersection and union for deterministic pushdown automata for example [17]. Again, many languages cannot be recognized by using pushdown automata, but considerable portion of languages arising from computer science problems can. Other possibility is to recognize languages by simulating Turing machines, but this approach can be extremely inefficient [17]. The last possibility, offering similar language recognizing abilities as pushdown automata and retaining closure properties of regular languages, are tree automata. However, this requires extending the notion of word and the resulting class of languages cannot be directly compared with regular or context free languages as a consequence.

Finite word automata can be viewed as a foundation tool for language recognition on which many others are build by extending their abilities.

2.1 Word Automata Preliminaries

Word automata are built on the notions of *alphabets*, *words*, and *languages*. This section therefore briefly introduces these concepts. The notions in this section are taken from [17].

Alphabet Σ is a finite nonempty set of elements called *symbols*. A word over alphabet Σ is a finite sequence of symbols from Σ. Words over Σ can be defined recursively as

1. ϵ, is a word (called the *empty word*) and
2. if w is a word over Σ and $a \in \Sigma$ then wa is a word over Σ.

These rules can be used do define the set of all words over Σ, denoted Σ^*:

$$\Sigma^* = \{w \mid w \text{ can be constructed using finitely many applications of rules 1 and 2}\}. \quad (2.1)$$

A language L over Σ is any set of words constructed using only symbols from Σ, i. e.,

$$L \subseteq \Sigma^*. \quad (2.2)$$

2.2 Finite Word Automata

A nondeterministic finite word automaton (NFA) is a quintuple

$$\mathcal{A} = (Q, \delta, \Sigma, s_0, F), \quad (2.3)$$

where

- Q is a finite non-empty set of states,
- δ is a transition function $\delta : Q \times \Sigma \to 2^Q$,
- Σ is an alphabet,
- s_0 is the initial state, $s_0 \in Q$ and
- F is the set of accepting states, $F \subseteq Q$.

The notation $p \xrightarrow{a} q$ is used to denote that $q \in \delta(p, a)$.

2.2.1 Determinism

Deterministic finite word automata (DFA) are a special class of nondeterministic finite word automata where the size of the right-hand side of transition rules is restricted. A deterministic finite word automaton is a quintuple $\mathcal{A} = (Q, \delta, \Sigma, s_0, F)$ where $Q, \delta, \Sigma, s_0, F$ have the same definition as in nondeterministic tree automata and

$$\forall p \in Q, \forall a \in \Sigma : |\delta(p, a)| \leq 1. \quad (2.4)$$

All NFAs can be determinized [17] and all DFAs are by definition also NFAs. Therefore, they recognize the same class of languages called regular languages. For convenience, deterministic version of word automata are used in definitions of a *run* and the *accepted language* of the NFA, but these definitions can be applied to nondeterministic word automata because they can be converted to equivalent deterministic automata.

To define a run of an automaton on a word, we use a multi-step version of the transition function $\hat{\delta} : Q \times \Sigma^* \to Q$. This multi-step transition function is defined recursively as

1. $\hat{\delta}(q, \epsilon) = q$,
2. $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

The language of an automaton $\mathcal{A} = (Q, \delta, \Sigma, s_0, F)$, denoted $L(\mathcal{A})$, is the set of all words over Σ for which there exists a multi-step transition into the accepting state, formally

$$L(\mathcal{A}) = \{w \mid \hat{\delta}(s_0, w) \in F\}. \quad (2.5)$$

Table 2.1: Closure properties of regular languages, deterministic context-free languages, and context-free languages. Source: [13]

	Class of languages		
Operation	Regular	Deterministic context-free	Context-free
Union	Yes	No	Yes
Intersection	Yes	No	No
Complement	Yes	Yes	No
Concatenation	Yes	No	Yes
Iteration	Yes	No	Yes

2.3 Closure Properties

Regular languages are closed (among others) under operations listed in Table 2.1. Closure properties are used for example in abstract regular model checking, where word automata representing dynamic heap objects undergo transformations that model operations performed on those objects, then, it is checked if the resulting object is not in some invalid state. This can be expressed using operations under which regular languages are closed and it guarantees that regular language representing dynamic heap object will stay regular during the whole process.

To illustrate why pushdown automata may not be the best choice for applications working with more complex languages (abstract regular model checking of programs containing tree structured heap objects for example), closure properties of deterministic and nondeterministic context-free languages are included.

Chapter 3

Tree Automata

Finite tree automata are tools for describing languages containing trees. They are similar to finite word automata in both their components and operations but they require an extended alphabet (called a ranked alphabet) to allow symbols to have more than one successor. Ranked alphabet is defined in Section 3.1.

Finite tree automata can be further divided based on determinism (Section 3.5) and direction in which trees are parsed.

Parsing can either start with the root symbol and progress in the top-down direction gradually expanding from the root symbol and assigning states to nodes as they are processed until all leaf nodes are reached, or start with leaf nodes progressing in the bottom-up direction, eventually reaching the root node.

In contrast to finite word automata, where the direction of parsing does not affect their ability to parse words (forward automata can be converted to backward automata by simply switching direction of all transitions, making initial state accepting, and creating a new starting state that can make a transition into previously accepting states), finite tree automata can only be reliably converted from one direction into another in a nondeterministic form, because bottom-up deterministic finite tree automata are strictly stronger than their top-down counterparts [5].

This difference stems from the tree structure, where when traversing in the bottom-up direction, states of all child nodes need to be used to determine the state of the parent node, whereas in the top-down direction a single state of the parent node must determine states of all children nodes.

3.1 Ranked Alphabet

A ranked alphabet Σ is a finite set of symbols, together with a total ranking function $rank : \Sigma \to \mathbb{N}$. The ranking function determines the arity of each symbol (number of successors).

For example, the following ranked alphabet can be used to construct syntax trees from the language of basic arithmetical expressions using symbols $+, -, \times, \div$, and numbers:

$$\Sigma_{arith} = \{+, -, \times, \div, n\} \quad rank = \{+ \mapsto 2, - \mapsto 2, n \mapsto 0, \times \mapsto 2, \div \mapsto 2\}, \tag{3.1}$$

where all arithmetical operators are binary (having a rank of 2) and n, representing a number, is nullary (having a rank of 0). Symbols with the rank 0 are called leaf symbols or leaves because they terminate branches of trees.

3.2 Tree

A *tree* t over a ranked alphabet Σ_{arity} is a function $t : Pos(t) \to \Sigma_{arity}$ where

- $Pos(t) \subseteq \mathbb{N}^*$,
- $Pos(t)$ is nonempty and prefix closed, i. e.,
  $\forall u, v \in \mathbb{N}^* : uv \in Pos(t) \implies u \in Pos(t)$, and
- $\forall p \in Pos(t) : a = t(p) \implies \{j \in \mathbb{N} \mid pj \in Pos(t)\} = \{1, \ldots, rank(a)\}$.

For example a formal definition of the tree t_1, depicted in Figure 3.1a, is

$$t_1 = \{\epsilon \mapsto +, 1 \mapsto n, 2 \mapsto \times, 21 \mapsto n, 22 \mapsto n\}. \tag{3.2}$$

3.3 Bottom-up Tree Automata

A *bottom-up nondeterministic finite tree automaton* (NFTA) is a quadruple

$$\mathcal{A} = (Q, \Delta, \Sigma, F), \tag{3.3}$$

where

- Q is a finite non-empty set of states,
- Δ is a finite partial transition function $\Delta : Q^* \times \Sigma_{arity} \to 2^Q$ such that if $\Delta((p_1, \ldots, p_n), a) = P$ then $rank(a) = n$,
- Σ is a ranked alphabet and
- F is the set of accepting states, $F \subseteq Q$.

The notation $(p_1, \ldots, p_n) \xrightarrow{a} q$ is used to denote that $q \in \Delta((p_1, \ldots, p_n), a)$.

3.3.1 Run

Let t be a tree and $\mathcal{A} = (Q, \Delta, \Sigma, F)$ be a NFTA. A run of $\mathcal{A}$ over a tree t is a mapping $r_t : Pos(t) \to Q$ consistent with Δ, i. e.,

$$\begin{aligned} &\forall p \in Pos(t), \exists (q_1, \ldots, q_n) \xrightarrow{a} q : \\ &\quad \big(n = rank(t(p)) \wedge q = r_t(p) \wedge \forall 1 \leq i \leq n : q_i = r_t(pi)\big). \end{aligned} \tag{3.4}$$

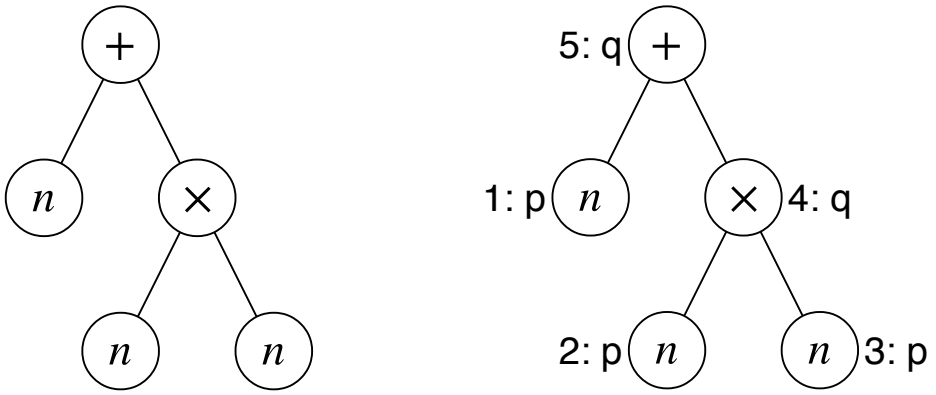
The run r_t is called accepting if $r_t(\epsilon) \in F$. An example of a successful run r_{t_1} on t_1 is depicted in Figure 3.1b and its formal definition is

$$r_{t_1} = \{\epsilon \mapsto q, 1 \mapsto p, 2 \mapsto q, 21 \mapsto p, 22 \mapsto p\}. \tag{3.5}$$

3.3.2 Language

Language accepted by a NFTA $\mathcal{A} = (Q, \Delta, \Sigma_{arity}, F)$, denoted $L(\mathcal{A})$, is the set of all trees for which there exist an accepting run r_t compatible with Δ, formally

$$L(\mathcal{A}) = \{t \mid \exists r_t : r_t \text{ is a run of } \mathcal{A} \text{ on } t \wedge r_t(\epsilon) \in F\}. \tag{3.6}$$



(a) A tree t_1 representing the arithmetical expression $(n + (n \times n))$.

(b) Bottom-up run r_{t_1} of automaton $\mathcal{A}_1$ from Section 3.3 over the tree t_1 from Figure 3.1a; the tree is accepted.

Figure 3.1: A tree structured entity and an example of a run.

3.3.3 Examples

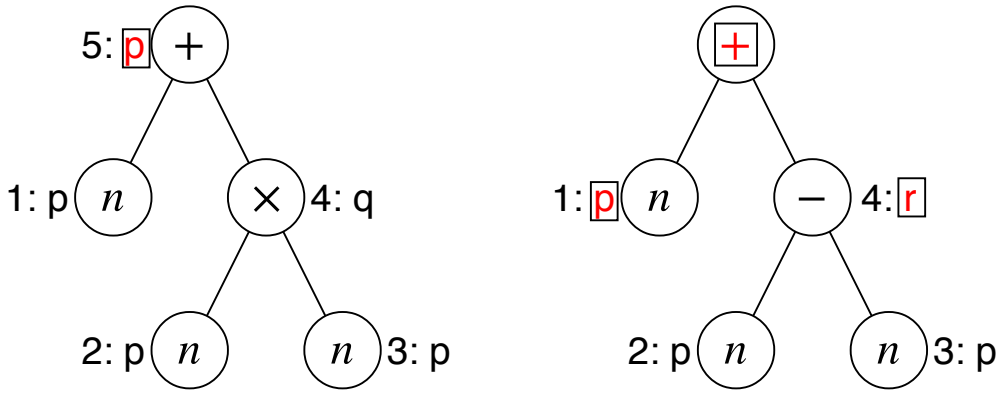
Let $\mathcal{A}_1 = (Q, \Delta, \Sigma_{arity}, F)$ be an automaton accepting the language of nontrivial arithmetical expressions (no expressions consisting of single number) composed using addition, subtraction, multiplication, and division defined as follows:

- $Q = \{p, q\}$,
- Δ : $(p,p) \xrightarrow{op} q \quad (q,p) \xrightarrow{op} q \quad () \xrightarrow{n} p$
  $(p,q) \xrightarrow{op} q \quad (q,q) \xrightarrow{op} q \quad$ for $op \in \{+, -, \times, \div\}$
- Σ_{arith} is the example alphabet from Section 3.1, and
- $F = \{q\}$.

An example of a tree accepted by $\mathcal{A}_1$ is in Figure 3.1a. The run r_{t_1} is constructed in five steps (Figure 3.1b). In the first three steps, the state p is assigned to leaves using the transition $() \xrightarrow{n} p$. In step 4 the state q is assigned to the node with $\times$ using the transition $(p,p) \xrightarrow{\times} q$. In the last step the state q is assigned to the node with $+$ using the transition $(p,q) \xrightarrow{+} q$. The root node is labelled with the state q and the labelling process is terminated. Because q is an accepting state, the tree t_1 belongs to the language of $\mathcal{A}_1$.

For examples of rejected trees, let $\mathcal{A}_2 = (Q, \Delta, \Sigma_{arity}, F)$ be an automaton accepting only expressions without subtraction and with multiplication as the root operation defined as follows:

- $Q = \{p, q, r\}$,
- Δ : $() \xrightarrow{n} p$
  $(p,p) \xrightarrow{op} p \quad (p,p) \xrightarrow{-} r \quad (p,p) \xrightarrow{\times} q$
  $(p,q) \xrightarrow{op} p \quad (p,q) \xrightarrow{-} r \quad (p,q) \xrightarrow{\times} q$
  $(q,p) \xrightarrow{op} p \quad (q,p) \xrightarrow{-} r \quad (q,p) \xrightarrow{\times} q$
  $(q,q) \xrightarrow{op} p \quad (q,q) \xrightarrow{-} r \quad (q,q) \xrightarrow{\times} q$
  for $op \in \{+, \div\}$



(a) Bottom-up run of $\mathcal{A}_2$ from Section 3.3 over t_1; the root node is reached with a non-accepting state and the tree is rejected.

(b) Bottom-up run of $\mathcal{A}_2$ from Section 3.3 over t_2; no transition with $(p, r) \xrightarrow{-}$ on the left-hand side exists, the tree is rejected.

Figure 3.2: Examples of rejected trees.

- Σ_{arith} is the example alphabet from Section 3.1,
- $F = \{q\}$.

Note that accepting the state q can only be reached immediately after processing a node with the multiplication symbol $\times$, processing the subtraction symbol $-$ assigns to the node the state r, for which there is no follow-up transition, and using any other transition assigns the non-accepting state p.

The example in Figure 3.2a labels the root node with state p in step 5. Because state p assigned to the root node is not an accepting state, the tree is rejected. Another example in Figure 3.2b shows a partially labelled tree after matching four transitions. In step 5 a transition with $(p, r) \xrightarrow{+}$ on the left-hand side is required to label the root node, however, no such transition exists in the automaton $\mathcal{A}_2$ so the tree is rejected.

3.4 Top-down Tree Automata

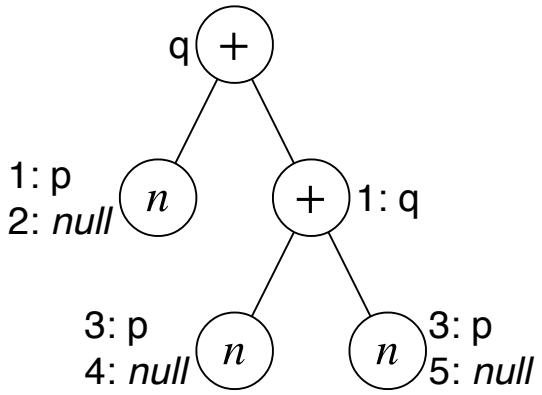
A top-down nondeterministic finite tree automaton is a quadruple

$$\mathcal{A} = (Q, \Delta, \Sigma, I), \tag{3.7}$$

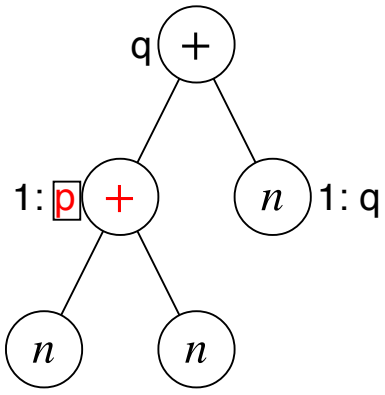
where

- Q is a finite non-empty set of states,
- Δ is a transition function $\Delta : Q \times \Sigma_{arity} \to 2^{Q^*}$ such that if $(p_1, \ldots, p_n) \in \Delta(p, a)$ then $rank(a) = n$,
- Σ is a ranked alphabet and
- I is the set of initial (root) states, $I \subseteq Q$.

The notation $p \xrightarrow{a} (p_1, \ldots, p_n)$ is used to denote that $(p_1, \ldots, p_n) \in \Delta(p, a)$. Note that the right-hand side of a transition can be an empty sequence i. e. $()$, denoted as *null* in this



(a) Successful top-down run of $\mathcal{A}_3$ over t_1; all leaf nodes labelled *null*.



(b) Failed top-down run of $\mathcal{A}_3$ over t_3; no transition with $p \xrightarrow{+}$ on the left-hand side.

Figure 3.3: Examples of top-down runs.

document. Transitions with null on the right-hand side represent termination of processing at leaf node level and correspond to leaf transitions in bottom-up NFTA. Top-down run construction is terminated when all leaf nodes are processed and are labelled *null* (tree is accepted) or when there is no applicable transition (tree is rejected).

3.4.1 Examples

Let $\mathcal{A}_3 = (Q, \Delta, \Sigma_{arith}, I)$ be an automaton accepting language of arithmetical addition expressions with right-to-left associativity where any other order of evaluation is rejected, defined as follows:

- $Q = \{p, q\}$,
- $\Delta : p \xrightarrow{n} null$, $q \xrightarrow{+} (p, q)$, $q \xrightarrow{+} (p, p)$,
- Σ_{arith} is the example alphabet from Section 3.1, and
- $I = \{p\}$.

The example run in Figure 3.3a reached and labelled all leaf nodes *null* (i. e. the tree t_1 is accepted) and the example run in Figure 3.3b reached a node + with the state p but no transition with $p \xrightarrow{+}$ on the left-hand side exists (the tree t_3 is rejected).

3.5 Determinism

As in the case of finite word automata, tree automata also exist in deterministic and nondeterministic variants. Both bottom-up tree automata and top-down tree automata in nondeterministic variants can recognize the same class of languages, called the class of the regular tree languages [5].

This class of languages is the same as the one recognized by deterministic bottom-up tree automata (every nondeterministic tree automaton can be converted to a deterministic bottom-up tree automaton accepting the same language [5]). Deterministic top-down automata recognize a strict subclass of that class.

A deterministic tree automaton differs from a nondeterministic one only in the definition of transition rules. A bottom-up deterministic tree automaton (DFTA) is a quadruple (Q, Δ, Σ, F) where Q, Σ, and F are the same as in the nondeterministic bottom-up automaton and the signature of the transition function Δ is as follows:

$$\Delta : Q^* \times \Sigma \to Q. \tag{3.8}$$

On the other hand a top-down deterministic tree automaton is a quadruple (Q, Δ, Σ, I) where Q, Σ, and I are the same as in the nondeterministic top-down automaton and the signature of the transition function Δ is as follows:

$$\Delta : Q \times \Sigma \to Q^*. \tag{3.9}$$

In other words deterministic tree automata are a special class of nondeterministic tree automata, where

$$\forall p_1, \ldots, p_n \in Q, \forall a \in \Sigma : |\Delta((p_1, \ldots, p_n), a)| \leq 1 \text{ for bottom-up automata or} \tag{3.10}$$
$$\forall p \in Q, \forall a \in \Sigma : |\Delta(p, a)| \leq 1 \text{ for top-down automata.} \tag{3.11}$$

Determinization is done by subset construction. Every set of transitions with the same left-hand side and symbol is merged into a single transition. The new transition retains the same left-hand side and symbol but on right-hand side it has a macrostate (a set of states of the original automaton) consisting of states that are on the right-hand side of individual transitions. Algorithm for subset construction on a bottom-up automaton is described in Section 4.2.2.

Many advanced methods for inclusion and equivalence testing use determinization *on the fly*, where the input automaton is nondeterministic and the subset construction is done during the algoritm run by constructing macrostates for nodes that are being examined, thus keeping the whole process deterministic even while working with a nondeterministic automaton.

Bisimulation up-to congruence (the technique that is being adapted to use with tree automata in this thesis) has nondeterministic automata as the input and relies on determinization *on the fly* to construct macrostate pairs of Algorithm 7.

3.6 Closure Properties

Closure properties of regular tree languages are similar to closure properties of regular word languages (Table 3.1). Therefore they are well suited for applications where repeatedly performing certain operations on tree structures is required (abstract regular tree model checking for example). Note that the notions of concatenation and iteration lose their meaning for tree languages and therefore are excluded from the table.

Table 3.1: Closure properties of regular word and tree languages. Source: [13, 5]

Operation	Regular word languages	Regular tree language
Union	Yes	Yes
Intersection	Yes	Yes
Complement	Yes	Yes

3.7 Transitions on Macrostates

This section defines new notions of transition functions on macrostates and their behaviour with regard to set operations performed on macrostates for bottom-up NTFAs. Notably how the result of a transition function Δ changes after application of a set operation on the parameters of Δ.

For the sake of simplicity the following definitions only consider symbols with arity 2. Constructs and operations used in those definitions are, however, not limited in any way by this choice only and can be extended to any other arity.

For all definitions in this section let $\mathcal{A} = (Q, \Delta, \Sigma, F)$ be a nondeterministic finite tree automaton and assume that $rank(a) = 2$ for some $a \in \Sigma$.

Definition 1 (Symbol Specific Successor)**.** *Symbol specific successor is a function that produces successor for a given pair of states reachable by using only the selected symbol a*

$$\Delta_a(p_1, p_2) = P \Leftrightarrow \Delta((p_1, p_2), a) = P. \tag{3.12}$$

Definition 2 (Macrostate Successor)**.** *Macrostate successor is an extension of regular transition function that allows macrostates to be present at the left-hand side of the rules. Formally $\Delta_a : (2^Q)^2 \to 2^Q$ is defined as*

$$\Delta_a(P_1, P_2) = \bigcup_{p_1 \in P_1, p_2 \in P_2} \Delta_a(p_1, p_2). \tag{3.13}$$

Definition 3 (Arbitrary State Successor)**.** *Arbitrary state, denoted _, is used to indicate that any state from Q is permited at the given position. It is in fact defined using macrostate successor of Q. The notation _ is used to remove explicit reference to Q in favour of arbitrary state independent from automaton definition. Arbitrary state successor is then defined as*

$$\Delta_a(P, \_) = \Delta_a(P, Q) \tag{3.14}$$
$$\Delta_a(\_, P) = \Delta_a(Q, P) \tag{3.15}$$
$$\Delta_a(\_, \_) = \Delta_a(Q, Q). \tag{3.16}$$

These definitions can be now used to describe behaviour of the transition function when set operations are applied. This is important mainly in Chapter 7 where set operations on macrostates are used in the proof of *bisimulation up-to congruence* correctness.

Lemma 1 (Macrostate Intersection)**.**

$$\Delta_a(P_1 \cap P_1', P_2 \cap P_2') = \Delta_a(P_1, P_2) \cap \Delta_a(P_1', P_2') \tag{3.17}$$

Proof.

$$\begin{aligned}
&\Delta_a(P_1 \cap P_1', P_2 \cap P_2') = \\
&\{p \mid (p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1 \cap P_1' \wedge p_2 \in P_2 \cap P_2'\} = \\
&\{p \mid (p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1 \wedge p_1 \in P_1' \wedge p_2 \in P_2 \wedge p_2 \in P_2'\} = \\
&\{p \mid (p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1 \wedge p_2 \in P_2 \wedge p_1 \in P_1' \wedge p_2 \in P_2'\} = \\
&\{p \mid ((p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1 \wedge p_2 \in P_2) \wedge ((p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1' \wedge p_2 \in P_2')\} = \\
&\{p \mid (p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1 \wedge p_2 \in P_2\} \cap \{p \mid (p_1, p_2) \xrightarrow{a} p \wedge p_1 \in P_1' \wedge p_2 \in P_2'\} = \\
&\Delta_a(P_1, P_2) \cap \Delta_a(P_1', P_2').
\end{aligned}$$

□

Macrostate intersection can be used to separate operations on the specific child node from operations performed on other child nodes. First, successors are calculated for every single child node using arbitrary state successor notation for surrounding child nodes and those successors are then combined using macrostate intersection.

Lemma 2 (Node-wise Successor)**.**

$$\Delta_a(P_1, P_2) = \Delta_a(P_1, \_) \cap \Delta_a(\_, P_2) \tag{3.18}$$

Proof.

$$\begin{aligned}
&\Delta_a(P_1, P_2) = \ \text{/* because } P_1, P_2 \subseteq Q \text{ */} \\
&\Delta_a(P_1 \cap Q, Q \cap P_2) = \ \text{/* Lemma 1 */} \\
&\Delta_a(P_1, Q) \cap \Delta_a(Q, P_2) = \ \text{/* Definition 3 */} \\
&\Delta_a(P_1, _) \cap \Delta_a(_, P_2).
\end{aligned}$$

□

Lemma 3 (Node-wise Union)**.**

$$\Delta_a(P_1 \cup P_2, \_) = \Delta_a(P_1, \_) \cup \Delta_a(P_2, \_) \tag{3.19}$$

Proof.

$$\begin{aligned}
&\Delta_a(P_1 \cup P_2, _) = \\
&\{p' \mid (p, q) \xrightarrow{a} p' \wedge p \in P_1 \cup P_2 \wedge q \in Q\} = \\
&\{p' \mid ((p, q) \xrightarrow{a} p' \wedge q \in Q) \wedge (p \in P_1 \vee p \in P_2)\} = \\
&\{p' \mid ((p, q) \xrightarrow{a} p' \wedge q \in Q \wedge p \in P_1) \vee ((p, q) \xrightarrow{a} p' \wedge q \in Q \wedge p \in P_2)\} = \\
&\{p' \mid (p, q) \xrightarrow{a} p' \wedge q \in Q \wedge p \in P_1\} \cup \{p' \mid (p, q) \xrightarrow{a} p' \wedge q \in Q \wedge p \in P_2\} = \\
&\Delta_a(P_1, _) \cup \Delta_a(P_2, _).
\end{aligned}$$

□

Lemma 4 (Macrostate union)**.**

$$\Delta_a(P_1 \cup P_1', P_2 \cup P_2') = \bigcup_{(R,S) \in \{P_1, P_2\} \times \{P_1', P_2'\}} \Delta_a(R, S) \tag{3.20}$$

Proof.

$$\begin{aligned}
&\Delta_a(P_1 \cup P_1', P_2 \cup P_2') = \ \text{/* Lemma 2 */} \\
&\Delta_a(P_1 \cup P_1', _) \cap \Delta_a(_, P_2 \cup P_2') = \ \text{/* Lemma 3 */} \\
&(\Delta_a(P_1, _) \cup \Delta_a(P_1', _)) \cap (\Delta_a(_, P_2) \cup \Delta_a(_, P_2')) = \\
&((\Delta_a(P_1, _) \cap \Delta_a(_, P_2)) \cup ((\Delta_a(P_1, _) \cap \Delta_a(_, P_2')) \cup \\
&((\Delta_a(P_1', _) \cap \Delta_a(_, P_2)) \cup ((\Delta_a(P_1', _) \cap \Delta_a(_, P_2')) = \ \text{/* Lemma 2 */} \\
&\Delta_a(P_1, P_2) \cup \Delta_a(P_1, P_2') \cup \Delta_a(P_1', P_2) \cup \Delta_a(P_1', P_2') = \\
&\bigcup_{(R,S) \in \{P_1, P_2\} \times \{P_1', P_2'\}} \Delta_a(R, S).
\end{aligned}$$

□

Chapter 4

Current Status of Equivalence and Inclusion Testing on Tree Automata

Testing inclusion and equivalence on tree automata can be done using several different techniques. First, there is the approach based on removal of unreachable states, determinization, and minimization described in Section 4.2. Because every minimal tree automaton is unique (up to isomorphism), equivalence can be determined by directly comparing minimized versions of input automata for isomorphism [4, 5].

Using determinization can be complicated by the state space explosion because determinized automaton can be exponentially larger than the original automaton [5]. Thus performing determinization before equivalence checking is resource-intensive, decreasing the overall performance of this approach. Moreover, even if equivalence can be disproven by finding a counterexample using only portion of the input automaton, the determinization and minimization will build the whole minimized automaton first and check equivalence later, leading to poor performance on nonequivalent automata pairs.

An alternative is to use *on the fly* determinization. This approach does not generate the whole determinized automaton right away. Instead, only a partial determinization for macrostates that were encountered is computed [2]. In the worst case, where all macrostates were encountered, the whole automaton is determinized.

Because two automata accept the same language if there is a bisimulation relating them, determinization *on-the-fly* and bisimulation is used to decide language equivalence. This algorithm starts with pairs of macrostates of leaf nodes and there is an assumption that a bisimulation relating them exists. Iteratively, successor pairs are generated for known pairs. Those successors constitute proof obligations that need to be satisfied to prove the initial assumption. This process is repeated until a failed obligation is found (counterexample) or no unprocessed proof obligation is left (the initial assumption holds). This allows for early termination of the whole process if a counterexample is found [2, 6, 9].

These techniques can further be aided by tools that prune the searched macrostate pair space. The antichain approach (Section 4.3), defined in [9], uses the identity relation, which implies language equivalence, to prune the search space [2, 9]. Simulation-based approaches (Section 4.3.1) are generalized versions of antichains that allow the use of any relation that implies language inclusion, but they are not complete (simulation implies language inclusion but not vice versa) [2]. And, finally, the antichain approach can be combined with simulation to obtain a combined method (Section 4.4) [2].

input : $\mathcal{A} = (Q_A, \Delta_A, \Sigma, F_A)$ a nondeterministic tree automaton
output: $\mathcal{B} = (Q_B, \Delta_B, \Sigma, F_B)$ a NTFA without bottom-up unreachable states such that $L(\mathcal{A}) = L(\mathcal{B})$

$reachable \leftarrow \emptyset$
$previous \leftarrow \emptyset$
do
  $previous \leftarrow reachable$
  $reachable \leftarrow \{q \in Q_A \mid a \in \Sigma \wedge q_1, \ldots, q_n \in previous \wedge (q_1, \ldots, q_n) \xrightarrow{a} q \in \Delta_A\}$
while $previous \neq reachable$
$Q_B \leftarrow reachable$
$F_B \leftarrow F_A \cap reachable$
$\Delta_B \leftarrow \{(q_1, \ldots, q_n) \xrightarrow{a} q \mid (q_1, \ldots, q_n) \xrightarrow{a} q \in \Delta_A \wedge q, q_1, \ldots, q_n \in reachable\}$

Algorithm 1: Removal of bottom-up unreachable states from a nondeterministic tree automaton. Source: [5]

4.1 Inclusion and Equivalence

Even though asking whether two languages are the same (equivalence) or one is a subset of the other (inclusion) may seem as two distinct questions, there is a mathematical connection between these two problems.

Because $A \subseteq B \wedge B \subseteq A \Leftrightarrow A = B$ and $A \cup B = B \Leftrightarrow A \subseteq B$, being able to test inclusion can also be used to determine equivalence and vice versa (provided language union can be computed [5]).

Therefore, algorithms described in this document may be specifically designed to solve only one of these problems, but can also be utilized to solve the other by exploiting this connection.

4.2 Direct Comparison

For every regular tree language, there exists a unique minimal (in the number of states and up to isomorphism) automaton that recognizes the given language [5]. Because of this, a viable way to test language equivalence is to convert the input automata into minimal automata accepting the same languages. These minimal automata can then be directly compared with each other. If there is a one-to-one correspondence between states and transitions in both automata (there exists an isomorphism relating the two) then they accept the same language. The process of minimization consists of the following steps: removal of unreachable states, determinization, and, finally, the minimization itself.

4.2.1 Removal of Bottom-up Unreachable States

In the first step, unreachable states and corresponding transitions are removed from the input nondeterministic tree automaton. This process is described in Algorithm 1.

Starting with an empty set of reachable states, new states are added to the set iteratively if there is a transition rule with the left-hand side consisting of states already in the set. Note that leaf rules have no state on their left-hand side and states they produce are added into the set *reachable* in the first iteration forming the basis for the following iterations

input : $\mathcal{B} = (Q_B, \Delta_B, \Sigma, F_B)$ an automaton without bottom-up unreachable states
output: $\mathcal{C} = (Q_C, \Delta_C, \Sigma, F_C)$ a deterministic automaton without bottom-up unreachable states such that $L(\mathcal{B}) = L(\mathcal{C})$

$Q_C \leftarrow \emptyset$
$\Delta_C \leftarrow \emptyset$
$\Delta'_C \leftarrow \emptyset$
do
  $\Delta'_C \leftarrow \Delta_C$
  $\Delta_C \leftarrow \Delta_C \cup \{(P_1, \ldots, P_n) \xrightarrow{a} P \mid a \in \Sigma \wedge P_1, \ldots, P_n \in Q_C \wedge P = \{q \mid \exists q_1 \in P_1, \ldots q_n \in P_n, (q_1, \ldots q_n) \xrightarrow{a} q \in \Delta_B\}\}$
  $Q_C \leftarrow \{P \mid \exists P_1, \ldots, P_n \in Q_C, (P_1, \ldots, P_n) \xrightarrow{a} P \in \Delta_C\}$
while $\Delta_C \neq \Delta'_C$
$F_C \leftarrow \{P \mid P \in Q_C \wedge P \cap F_B \neq \emptyset\}$

Algorithm 2: Determinization of a tree automaton. Source: [5]

input : $\mathcal{C} = (Q_C, \Delta_C, \Sigma, F_C)$ a deterministic automaton without bottom-up unreachable states
output: $\mathcal{D} = (Q_D, \Delta_D, \Sigma, F_D)$ a minimal tree automaton such that $L(\mathcal{C}) = L(\mathcal{D})$

$E \leftarrow \{F_C, Q_C \setminus F_C\}$
$E' \leftarrow \emptyset$
while $E \neq E'$ **do**
  $E' \leftarrow E$
  $qEq' \Leftrightarrow pE'p' \wedge \forall a \in \Sigma, (q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) \xrightarrow{a} p \in \Delta_C \wedge (q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n) \xrightarrow{a} p' \in \Delta_C$
end
$Q_D \leftarrow \{[q] \mid [q] \text{ is an equivalence class in } E\}$
$\Delta_D \leftarrow \{([q_1], \ldots, [q_n]) \xrightarrow{a} [q] \mid (q_1, \ldots, q_n) \xrightarrow{a} q \in \Delta_C\}$
$F_D \leftarrow \{[q] \mid q \in F_C\}$

Algorithm 3: Minimization of deterministic tree automaton. Source: [5]

to grow from. This process is repeated until no new state can be added into the set. An automaton without bottom-up unreachable states is constructed using only the states from the *reachable*.

4.2.2 Determinization

The next step is to determinize the automaton (Algorithm 2). Determinization is the step in which the exponential state explosion can happen and therefore contributes the most of all steps to the overall poor performance of this approach.

The determinization is done using the subset construction, where a set of states that are reachable with rules with same left-hand side form a single macrostate (new macrostates are in the powerset of initial states). Algorithm 2 constructs only macrostates that are bottom-up reachable, keeping the state explosion limited, however, in the worst case scenario the whole powerset will be constructed.

4.2.3 Minimization

The last step is minimization (Algorithm 3), which produces a unique automaton representing the accepted language. This automaton can be compared with other minimal automata to determine language equivalence.

Algorithm 3 constructs an equivalence relating indistinguishable states. First, accepting and non-accepting states form the initial equivalence classes, this relation is then iteratively refined by splitting states in existing classes into subclasses based on differences in groups that are reachable by them. The final automaton is constructed using the refined equivalence classes as states and modifying transition rules correspondingly [4, 5].

4.3 Antichains

A language inclusion problem can be transformed into emptiness problem on a product automaton because $L(\mathcal{A}) = L(\mathcal{B}) \Leftrightarrow L(\mathcal{A}) \cap \overline{L(\mathcal{B})} = \emptyset$. Let $\mathcal{A}$ and $\mathcal{B}$ be NTFAs, every state in the product automaton is a pair (p, Q) such that p is a state in $\mathcal{A}$ and Q is a macrostate in $\mathcal{B}$. This transformation is used in antichain-based algorithms in [2].

Antichains as a means of testing language inclusion were introduced for finite word automata first [2, 6, 9]. This approach exploits the mathematical concept of an antichain, which is a set of objects that are pairwise incomparable, and applies it to subsets of the set of automaton states Q. An antichain C is formed by a set of states where each set is not a subset of any other set in the antichain ($C \subseteq 2^Q : \forall P, R \in C, P \not\subset R \wedge R \not\subset P$) [6].

Antichain structure is then used to store encountered product states. All newly encountered product states, that can not be added to the antichain without breaking it can be safely discarded from further search. Antichain algorithm from [2] does not need to continue search if it encounters a product state (p, P) that is larger than some product state (q, Q) it already visited, i. e. if $p = q \wedge P \subseteq Q$) . This is sound because antichain algorithm searches for counterexamples in the product automaton and if a product state (p, P) takes the automaton to accepting state then so does the (q, Q).

This can be used during language inclusion checking with *on the fly* determinization to decide on relation between languages of currently processed pair allowing to prematurely terminate an algorithm run if a counterexample is found [2, 6, 9].

4.3.1 Simulations

Another approach to testing language inclusion is to utilize a simulation relation (one of the relations that imply language inclusion). A simulation relation is calculated independently before the language inclusion checking. It was shown in [2] that there exist simulations (e. g. the maximal downward simulation) that can be computed in a reasonable time.

A simulation can be then used during the *on the fly* determinization to check if the language of one state is included in the language of an other state while processing one macrostate pair, thereby removing the necessity to check the successor pairs [2].

It is important to note that a simulation only *implies* language inclusion. Therefore, testing language equivalence using simulation is an incomplete method because there can be pairs of states whose languages are included one in another, but are not related by a simulation. Simulation is, however, often used together with other complete methods, for example antichains.

4.4 Top-down Antichain with Simulation

A similar technique to antichains relying on combining antichains and simulation for language inclusion testing was proposed for top-down tree automata in [9]. It uses the maximal downward simulation $\preceq \subseteq Q \times Q$ defined as

$$p \preceq q \Leftrightarrow \forall (p_1, \ldots, p_n) \xrightarrow{a} p, \exists (q_1, \ldots, q_n) \xrightarrow{a} q \wedge \forall i, p_i \preceq q_i \tag{4.1}$$

to prune unnecessary parts of the search space. There exists an extension of maximal downward simulation for macrostates $\preceq^{\forall\exists} \subseteq 2^Q \times 2^Q$ defined as

$$P \preceq^{\forall\exists} Q \Leftrightarrow \forall p \in P : \exists q \in Q : p \preceq q \tag{4.2}$$

which is used in function `expand`.

The unction `expand` checks whether the language of a single state is included in the language of a set of states ($L(q) \subseteq L(R)$) by checking that q is accepting only if there is some $r \in R$ that is also accepting and recursively calling function `expand` for all possible successors of this product state. When checking whether $L(\mathcal{A}) \subseteq L(\mathcal{B})$ in Algorithm 4, `expand` is initially called for every pair (q, R) where $q \in F_A \wedge R = F_B$. The function `expand` evaluates all transitions that can be taken from state q and for every leaf transition checks whether there is a leaf transition over the same symbol for at least one state from R or it recursively calls itself for queries (q', R') on which the result of (q, R) depends. The function returns *true* if all leaf transitions can be imitated by R and all recursive calls returned *true*, otherwise it returns *false* [9].

To avoid endless recursion, the algorithm keeps track of processed pairs in `workset`. Every pair for which `expand` is called is added to `workset`. If at any point `expand` is called for a pair that is already in `workset`, such a call immediately returns *true* because the result is then dependent only on the result of the other branches of the search [9].

The language inclusion holds if for every pair for which `expand` was initially called *true* is returned, and does not hold otherwise.

input : $\mathcal{N} = (Q_N, \Delta_N, \Sigma_N, F_N), \mathcal{M} = (Q_M, \Delta_M, \Sigma_M, F_M), \preceq \subseteq Q_N \times Q_M$,
output: *true* iff $L(N) \subseteq L(M)$, otherwise *false*
data : $NN = \emptyset$

```
1 foreach f ∈ F_N do
2 |   if ¬expand(q, F_M, ∅) then
3 |   |   return false
4 |   end
5 end
6 return true
```

Algorithm 4: Downward inclusion checking for tree automata using antichains and maximal downward simulation as the preorder $\preceq$. Source: [9]

```
1  Function expand(p_s, P_b, workset) is
2      /* N = (Q_N, Δ_N, Σ_N, F_N), M = (Q_M, Δ_M, Σ_M, F_M), ⪯ ⊆ Q_N × Q_M */
3      if ∃(p'_s, P'_b) ∈ workset : p_s ⪯ p'_s ∧ P'_b ⪯^{∀∃} P_b then
4          return true
5      end
6      if ∃(p'_s, P'_b) ∈ NN : p'_s ⪯ p_s ∧ P_b ⪯^{∀∃} P'_b then
7          return false
8      end
9      if ∃p ∈ P_b : p_s ⪯ p then
10         return true
11     end
12     workset ← workset ∪ {(p_s.P_b)}
13     n ← rank(a)
14     foreach a ∈ Σ do
15         if n = 0 then
16             if Δ_N(p_s, a) ≠ ∅ ∧ Δ_M(P_b, a) = ∅ then
17                 return false
18             end
19         else
20             W ← Δ_M(P_b, a)
21             foreach (r_1, ..., r_n) ∈ Δ_N(p_s, a) do
22                 foreach f ∈ {W → {1, ..., n}} do
23                     found ← false
24                     foreach 1 ≤ i ≤ n do
25                         S ← {q_i | (q_1, ..., q_n) ∈ W ∧ f((q_1, ..., q_n)) = i}
26                         if expand(r_i, S, workset) then
27                             found ← true
28                             break
29                         end
30                         if ∄(r', H) ∈ NN : r' ⪯ r_i ∧ S ⪯^{∀∃} H then
31                             NN ← (NN \ {(r', H) | r_i ⪯ r' ∧ H ⪯^{∀∃} S}) ∪ {(r_i, S)}
32                         end
33                     end
34                     if ¬found then
35                         return false
36                     end
37                 end
38             end
39         end
40     end
41     return true
42 end
```

Chapter 5

Bisimulation up-to Congruence for Finite Word Automata

Bisimulation up-to congruence for nondeterministic finite word automata was introduced by Bonchi and Pous [3]. This algorithm is an optimization of the textbook language equivalence checking algorithm (Algorithm 5) that relates states of deterministic word automata with a bisimulation relation and belongs to the category of algorithms based on *coinductive proofs*. A characteristic for this class of algorithms is *on the fly* determinization, where input automata can be nondeterministic and the determinization is done during the algorithm run only for macrostates that are encountered during the computation. It is based on Hopcroft and Karp's algorithm [10], which over its run builds a minimal bisimulation relating both automata that contains their initial states [3, 10]. Let $\mathcal{N} = (Q_N, \delta_N, \Sigma, s_N, F_N)$ and $\mathcal{M} = (Q_M, \delta_M, \Sigma, s_M, F_M)$ be two deterministic finite word automata.

Definition 4 (Bisimulation for word automata)**.** *Bisimulation for word automata is a relation $R \subseteq Q_N \times Q_M$ on states, such that if $(x, y) \in R$ then*

1. *$x \in F_N \Leftrightarrow y \in F_M$ and*
2. *$\forall a \in \Sigma : (\delta_N(x, a), \delta_M(y, a)) \in R$* [3].

Proposition 1 (Bisimulation, Language equivalence for FAs, Proposition 1 of [3])**.** *Two FAs are language equivalent iff there exist a bisimulation relating their initial states.*

Bisimulation up-to congruence improves efficiency of this technique by building only a portion of the bisimulation built by Algorithm 5, which can be extended into a full bisimulation by computing its congruence closure. In the worst-case scenario, the relation built will be the minimal bisimulation containing initial states, degrading this algorithm to Hopcroft and Karp algorithm. However, in the majority of cases, a relation built by bisimulation up-to congruence will be a subset of the minimal bisimulation containing initial states [3].

Definition 5 (Bisimulation up-to for word automata)**.** *Given a function $f : 2^{Q_N \times Q_M} \to 2^{Q_N \times Q_M}$ a relation $R \subseteq Q_N \times Q_M$ on macrostates is a bisimulation up-to function f if whenever $(x, y) \in R$ then*

1. *$x \in F_N \Leftrightarrow y \in F_M$ and*
2. *$\forall a \in \Sigma : (\delta_N(x, a), \delta_M(y, a)) \in f(R)$* [3].

input : DFAs $\mathcal{N} = (Q_N, \delta_N, \Sigma, s_N, F_N)$, $\mathcal{M} = (Q_M, \delta_M, \Sigma, s_M, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

```
1  todo ← (s_N, s_M)
2  done ← ∅
3  while todo ≠ ∅ do
4  |   actual ← (x, y) ∈ todo
5  |   done ← done ∪ {actual}
6  |   if (x ∈ F_N ⇎ y ∈ F_M) then
7  |   |   return false
8  |   end
9  |   todo ← {(δ_N(x, a), δ_M(y, a)) | a ∈ Σ} \ done
10 end
11 return true
```

Algorithm 5: Hopcroft and Karp's algorithm for testing language equivalence of deterministic finite word automata using bisimulation. Source: [3]

5.1 Technique Basics

The ability to build only a partial bisimulation is based on the observation of the connection between union of languages and union of macrostates on nondeterministic finite word automata.

Let $L(X)$ be a language recognized by a macrostate (a set of states) X. Then for any macrostates X and Y it holds that

$$L(X) \cup L(Y) = L(X \cup Y). \tag{5.1}$$

This allows for deriving conclusions about language equivalence of composite macrostates in the deterministic word automata without explicitly checking them. For example, given a bisimulation R relating macrostate $\{a, b\}$ with $\{x\}$ and macrostate $\{c\}$ with $\{y\}$, it can be immediately concluded that the maximal bisimulation would relate $\{a, b, c\}$ with $\{x, y\}$.

Therefore, if a macrostate pair that is a composite of already checked pairs is encountered during a run, this pair can be safely discarded without checking because its language equivalence can be determined based solely on the contents of a partial bisimulation R. Note that by computing the closure with regard to this property, it is possible to construct a bisimulation that is equal to or a superset of the minimal bisimulation containing initial states (exploiting union can introduce macrostates pairs that are language equivalent but may not have been encountered during the *on the fly* determinization, for example unreachable macrostates). Exploiting this property leads to bisimulation up-to context and is one of the building blocks of bisimulation up-to congruence.

Another exploitable property is equivalence. Because not all bisimulations are equivalence relations [3] they can relate macrostates X with Y and X with Z but not necessarily Y with Z. But because $L(X) = L(Y)$ and $L(X) = L(Z)$, then $L(Y) = L(Z)$ must be also true. As with context, equivalence can be used to build a partial bisimulation that can be expanded into a bisimulation by computing the appropriate closure. Using this technique is called bisimulation up-to equivalence.

Bisimulation up-to congruence is created by combining bisimulation up-to context with bisimulation up-to equivalence [3].

input : NFAs $\mathcal{N} = (Q_N, \delta_N, \Sigma, s_N, F_N)$, $\mathcal{M} = (Q_M, \delta_M, \Sigma, s_M, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

```
1  todo ← ({s_N}, {s_M})
2  done ← ∅
3  while todo ≠ ∅ do
4  |  actual ← (X, Y) ∈ todo
5  |  done ← done ∪ {actual}
6  |  if (X ∩ F_N = ∅ ⇎ Y ∩ F_M = ∅) then
7  |  |  return false
8  |  end
9  |  todo ← {(δ_N(X, a), δ_M(Y, a)) | a ∈ Σ} \ c(done)
10 end
11 return true
```

Algorithm 6: Hopcroft and Karp's algorithm for testing language equivalence of nondeterministic finite word automata using bisimulation up-to congruence. Modification of Bonchi and Pous from [3].

5.2 Bisimulation up-to Congruence Algorithm

The implementation of bisimulation up-to congruence algorithm for finite word automata (Algorithm 6) is similar to standard bisimulation. The only difference is the exclusion of processing of macrostate pairs for which language equivalence can be deduced from congruence closure (line 9).

The algorithm keeps the set of visited macrostate pairs in *done* and the set of pairs to be checked in *todo*. At the beginning, the pair representing starting states of the input automata is inserted into *todo*. Then the algorithm loops by processing a single pair from *todo* in each iteration. The processing consists of checking if macrostates in the pair are finite state equivalent (line 6) and generating successor pairs (line 9).

Definition 6 (Congruence closure)**.** *Congruence closure is a symmetric, transitive, and reflexive closure of the original relation combined with the union function. Let $R \subseteq 2^Q \times 2^Q$, the congruence closure of R, denoted $c(R)$ can be defined inductively using following rules:*

$$\frac{X\ R\ Y}{X\ c(R)\ Y}, \quad \frac{}{X\ c(R)\ X}, \quad \frac{X\ c(R)\ Y}{Y\ c(R)\ X}, \\ \frac{X\ c(R)\ Y\ Y\ c(R)\ Z}{X\ c(R)\ Z} \text{ and } \frac{X_1\ c(R)\ Y_1\ X_2\ c(R)\ Y_2}{X_1 \cup X_2\ c(R)\ Y_1 \cup Y_2}. \tag{5.2}$$

Proposition 2. *The relation built in* `done` *during the run of Algorithm 6 is a bisimulation up-to congruence if line 11 is reached, otherwise a counterexample was found and* `actual` *holds a pair of macrostates that is reachable over the same word in determinized versions of both automata and only one of the macrostates is accepting* [3].

Chapter 6

Bisimulation up-to Congruence for Tree Automata

First, we propose an algorithm that utilizes simple bisimulation with no *up-to* relation for language equivalence testing on nondeterministic bottom-up tree automata (Algorithm 7). This algorithm can be viewed as an implementation of the textbook approach to equivalence checking on deterministic finite tree automata, where it is checked whether corresponding pairs of states in both automata can imitate all possible transitions of each other and assert that either both or none of the states are accepting. The only modification is that the algorithm operates on nondeterministic automata and performs determinization *on the fly*.

This algorithm is then improved with iterative successor macrostates generation (Algorithm 8), which is an improvement on the implementation level that prevents repeated calculations of successors for states that were already processed, and *up-to congruence* relation (Algorithm 9), which is an algorithmic improvement that greatly reduces the size of the search space that is necessary to explore by inferring information from already processed pairs and discarding branches of the search space that cannot yield counterexamples.

6.1 Naive Algorithm

The naive algorithm for language equivalence testing on nondeterministic tree automata (Algorithm 7) is an adaptation of the naive algorithm for testing language equivalence on word automata described in Bonchi and Pous (Chapter 5). Algorithms performs determinization *on the fly*. This is achieved by performing computations with sets of states, called macrostates, instead of individual states. Macrostates on nondeterministic automata represent states of their determinized counterparts, which are computed on demand only for macrostates that are actually visited.

The naive algorithm asserts that all pairs of macrostates that are reachable over the same leaves are bisimulating each other [1], in other words, every sequence of transitions from the same set of leaves in one automaton can be imitated in the other automaton. This is ensured by iteratively adding successors of known pairs that constitute proof obligations for initial pairs. For every macrostate pair, it is checked whether the macrostates are either both accepting or non-accepting. If at any point a mixed pair is found, one of the proof obligations have failed and automata are not related by a bisimulation, therefore their languages are not equal. If all pairs were processed without finding a mixed pair, proof obligations are satisfied and languages of the input automata are equal.

input : NFTAs $\mathcal{N} = (Q_N, \Delta_N, \Sigma, F_N),\ \mathcal{M} = (Q_M, \Delta_M, \Sigma, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

todo $\leftarrow \Delta(\emptyset)$
done $\leftarrow \emptyset$
while *todo* $\neq \emptyset$ **do**
    actual $\leftarrow (X, Y) \in$ todo
    done $\leftarrow$ done $\cup$ {actual}
    if $(X \cap F_N = \emptyset \nLeftrightarrow Y \cap F_M = \emptyset)$ **then**
        return *false*
    end
    todo $\leftarrow \Delta$(done) $\setminus$ done
end
return *true*

Algorithm 7: The naive algorithm for testing language equivalence on tree automata.

Function $\Delta(done)$ **is**
    /* NFTAs $\mathcal{N} = (Q_N, \Delta_N, \Sigma_N, F_N),\ \mathcal{M} = (Q_M, \Delta_M, \Sigma_M, F_M)$ */
    /* Returns the set of macrostates reachable from *done*. */
    return $\{(\Delta_N(P_1, \ldots, P_n, a), \Delta_M(Q_1, \ldots Q_n, a)) \mid \forall a \in \Sigma : n = rank(a) \wedge \forall 1 \leq i \leq n : (P_i, Q_i) \in done\}$
end

Definition 7 (Bisimulation)**.** *Bisimulation on tree automata is a relation $R \subseteq Q_N \times Q_M$ on macrostates, such that if $\forall (x, y) \in R : \forall a \in \Sigma : \forall 1 \leq i \leq rank(a) - 1 : (x_i, y_i) \in R$ then*

1. $x \in F_N \Leftrightarrow y \in F_M$ *and*
2. $(\Delta_N((x_1, \ldots, x, \ldots, x_n), a), \Delta_M((y_1, \ldots, y, \ldots, y_n), a)) \in R,\ n = rank(a) - 1.$

Proposition 3 (Bisimulation, Language equivalence)**.** *Two DFTAs are language equivalent iff there exist a bisimulation relating sates of their leaf transitions.*

Definition 8 (Bisimulation step)**.** *Given a relation R on macrostates, bisimulation step*

$$\Delta(R) = \{(\Delta_N(X_1, \ldots, X_n, a), \Delta_M(Y_1, \ldots, Y_n, a)) \mid \forall a \in \Sigma : n = rank(a) \wedge \forall 1 \leq i \leq n : (P_i, Q_i) \in R\}. \tag{6.1}$$

6.1.1 Calculating Successor Macrostates on Tree Automata

In contrast to word automata, calculating a set of successor macrostates for tree automata — a set of macrostates that are reachable from already processed macrostates using transitions of the given tree automaton — has to take into account the context of transitions, that represents other parts of the tree the transition is traversing.

There, by context, we mean other macrostates occuring in potentially usable transitions, because in order to satisfy all conditions of the bisimulation, it must be ensured that other branches of trees can be traversed by both automata, therefore macrostates figuring as a context must be pairwise bisimulating each other. To be able to use a transition in

input : NFTAs $\mathcal{N} = (Q_N, \Delta_N, \Sigma, F_N)$, $\mathcal{M} = (Q_M, \Delta_M, \Sigma, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

```
1  todo ← Δ(∅)
2  done ← ∅
3  while todo ≠ ∅ do
4  |   actual ← (X, Y) ∈ todo
5  |   done ← done ∪ {actual}
6  |   if (X ∩ F_N = ∅ ⇔ Y ∩ F_M = ∅) then
7  |   |   return false
8  |   end
9  |   todo ← (todo ∪ Δ'(actual, done)) \done
10 end
11 return true
```

Algorithm 8: The naive algorithm with iterative successors generation.

```
1  Function Δ'((X, Y), done) is
2  |   /* NFTAs N = (Q_N, Δ_N, Σ_N, F_N), M = (Q_M, Δ_M, Σ_M, F_M) */
3  |   /* Returns the set of macrostates reachable from done using (X, Y). */
4  |   return {(Δ_N(P_1, ..., X, ... P_n, a), Δ_M(Q_1, ..., Y, ... Q_n, a)) | ∀a ∈ Σ :
   |    n = rank(a) − 1 ∧ ∀1 ≤ i ≤ n : (P_i, Q_i) ∈ done}
5  end
```

successor list generation, the whole context of the rule must be known. This means that every macrostate in the context must itself be reachable from leaf states (or intermediate macrostates that are themselves reachable from leaf states).

Note that the context in function Δ is represented by the set *done*, which (as defined in Algorithm 7) at first holds only leaf states and is incrementally expanded with macrostates that are reachable from the previous iteration of *done*.

6.2 The Naive Algorithm with Iterative Successors

Computing the whole set of possible successors in each iteration of Algorithm 7 is redundant because possible successors from previous iteration are already stored in the sets *todo* and *done*. Therefore, it is desirable to compute only newly reachable successors that are created by adding currently processed pair *actual* to *done*.

Algorithm 8 implements iterative successors generation, where in each iteration only successors of the currently processed pair are computed. This modifications significantly reduces the difficulty of computing $\Delta(done)$ because only a portion of the set *todo* has to be computed in every iteration.

6.2.1 Iterative Calculation of Successor Macrostates

Function Δ' is a modification of function Δ that generates only successors that are made reachable by the addition of *actual* to *done*. This is achieved by mandating that at least one position on the left-hand side of a transition is filled with a macrostate from *actual*.

input : NFTAs $\mathcal{N} = (Q_N, \Delta_N, \Sigma, F_N)$, $\mathcal{M} = (Q_M, \Delta_M, \Sigma, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

```
1  todo ← Δ(∅)
2  done ← ∅
3  while todo ≠ ∅ do
4  |  actual ← (X, Y) ∈ todo
5  |  done ← done ∪ {actual}
6  |  if (X ∩ F_N = ∅ ⇔ Y ∩ F_M = ∅) then
7  |  |  return false
8  |  end
9  |  todo ← (Δ(done)) \ c(done)
10 end
11 return true
```

Algorithm 9: The bisimulation up-to congruence on nondeterministic tree automata.

6.3 The Bisimulation up-to Congruence Algorithm

In order to improve the performance of equivalence checking algorithm even further, it is desirable to reduce the search space as much as possible. This is achieved in Algorithm 9 by using *bisimulation up-to congruence*, an approach that only adds a new macrostate pair into *todo* if it is not in the congruence closure of *done*, denoted as c(done). Thus, all new pairs that cannot contribute to finding a counterexample are removed from further search.

In the Algorithm 9, the congruence closure of a relation R is defined in terms of the functions introduced in Chapter 7.2.1 as a reflexive, symmetric and transitive closure coupled with the union function, formally

$$c(R) = (id \cup r \cup s \cup t \cup u^{\omega})^{\omega}. \tag{6.2}$$

Because $(id \cup r \cup s \cup t \cup u^{\omega})^{\omega} = (id \cup r \cup s \cup t \cup u)^{\omega}$, the same inductive rules as in Definition 6 can be used. This slightly different definition from the one found in [3] is used because the function u for tree automata is lacking certain properties required by the technique used in the formal proof and iterating the function u ensures those properties. Difference of the proof for word and tree automata is further discussed in Section 7.2.3.

6.3.1 Bisimulation up-to

An algorithm based on a bisimulation that does not explore the whole search space but introduces a mechanism (in the form of some function f) that will prune parts of the search space can be considered a bisimulation *up-to* f. Bisimulation up-to can be defined using the notion of a progression [3].

Definition 9 (Bisimulation up-to)**.** *Given a function* $f : 2^{2^{Q_N} \times 2^{Q_M}} \to 2^{2^{Q_N} \times 2^{Q_M}}$*, a relation* R *on macrostates is a bisimulation up-to function* f *if whenever* $\forall (X, Y) \in R : \forall a \in \Sigma : \forall 1 \leq i \leq rank(a) - 1 : (X_i, Y_i) \in R$ *then*

1. $X \cap F_N = \emptyset \Leftrightarrow Y \cap F_M = \emptyset$ *and*
2. $(\Delta_N((X_1, \ldots, X, \ldots, X_n), a), \Delta_M((Y_1, \ldots, Y, \ldots, Y_n), a)) \in f(R)$, $n = rank(a) - 1$.

input : NFTAs $\mathcal{N} = (Q_N, \Delta_N, \Sigma, F_N)$, $\mathcal{M} = (Q_M, \Delta_M, \Sigma, F_M)$
output: *true* iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise *false*

```
1  todo ← Δ(∅)
2  done ← ∅
3  while todo ≠ ∅ do
4  |  actual ← (X, Y) ∈ todo
5  |  done ← done ∪ {actual}
6  |  if (X ∩ F_N ≠ ∅ ∧ Y ∩ F_M = ∅) then
7  |  |  return false
8  |  end
9  |  todo ← (Δ(done)) \ c_i(done) )
10 end
11 return true
```

Algorithm 10: The bisimulation up-to congruence algorithm for inclusion checking on nondeterministic tree automata.

6.4 Bisimulation up-to Congruence for Inclusion Testing

In order to test language inclusion directly without reliance on the $A \subseteq B \Leftrightarrow A \cup B = B$ equivalence, we propose a modification of *bisimulation up-to congruence* optimized for direct inclusion checking (Algorithm 10). First, there is no need too compute the union automaton, saving time during preprocessing, second, optimizations directly in Algorithm 9 can be made in the congruence closure calculation. This optimization is based on the observation that an empty state set always produces the empty language and that the empty language is a subset of any other language.

Correctness of the modification of *bisimulation up-to congruence* for inclusion testing is not proven in this thesis. But experiments on 400 inclusion tests were performed and the results were matching the results obtained with the algorithm based on antichains.

Definition 10 (Congruence closure for inclusion checking)**.** *Congruence closure for inclusion checking is a symmetric, transitive, and reflexive closure of the original relation that includes $\emptyset$ in relation with all other elements. Let $R \subseteq 2^Q \times 2^Q$, the congruence closure of R, denoted $c_i(R)$, can be defined inductively using the following rules:*

$$\frac{X\ R\ Y}{X\ c_i(R)\ Y}, \frac{}{X\ c_i(R)\ X}, \frac{}{\emptyset\ c_i(R)\ X}, \frac{X\ c_i(R)\ Y}{Y\ c_i(R)\ X}, \frac{X\ c_i(R)\ Y\ Y\ c_i(R)\ Z}{X\ c_i(R)\ Z} \text{ and } \frac{X_1\ c_i(R)\ Y_1\ X_2\ c_i(R)\ Y_2}{X_1 \cup X_2\ c_i(R)\ Y_1 \cup Y_2}. \tag{6.3}$$

Chapter 7

Proof of Correctness

In this chapter, a proof of the bisimulation up-to congruence algorithm for NFTAs is gradually built from the proof of the naive algorithm (Section 7.1), its equivalence with iterative successors calculation algorithm (Section 7.3), and by defining compatible functions that satisfy the definition of *bisimulation up-to* that can be used to construct a congruence relation.

First, we show that the naive algorithm is sound and returns *true* only if the languages of the input automata are the same. Then, we introduce a notion of compatible functions and show that any *bisimulation up-to compatible function* is contained in a *bisimulation* constructed by the naive algorithm. This is sufficient to prove that *bisimulation up-to compatible function* checks language equivalence. Finally, we prove that congruence closure is a compatible function by constructing it from elementary compatible functions.

7.1 Soundness of the Naive Algorithm

The naive algorithm's soundness is ensured by proving that the relation built in done over the run of Algorithm 7 is a bisimulation. There is an invariant $\Delta(\mathit{done}) \subseteq \mathit{done} \cup \mathit{todo}$ corresponding to line 3 of Algorithm 7. This invariant holds by definition of todo (line 9), in this case the invariant is even stronger, as $\Delta(\mathit{done}) = \mathit{done} \cup \mathit{todo}$ also holds in line 3:

$$\begin{aligned} \text{todo} &= \Delta(\text{done}) \setminus \text{done} && /\text{line } 9 \\ \text{todo} \cup \text{done} &= \Delta(\text{done}) \setminus \text{done} \cup \text{done} && / \cup \text{done} \\ \text{todo} \cup \text{done} &= \Delta(\text{done}). \end{aligned}$$

Lemma 5 (Bisimulation, Language equivalence)**.** *Algorithm 7 with input automata $\mathcal{N}, \mathcal{M}$ returns true iff $L(\mathcal{N}) = L(\mathcal{M})$.*

Proof. If Algorithm 7 reaches line 11 and returns *true* then the set *todo* is empty and $\mathit{done} = \Delta(\mathit{done})$, making the relation built over the run of Algorithm 7 a bisimulation and $L(\mathcal{N}) = L(\mathcal{M})$ (Proposition 3).

If Algorithm 7 reaches line 7 and returns *false*, then there exist a set of trees $\{t_1, \ldots, t_n\}$ over Σ and a symbol a such that there exists a run of $\mathcal{N}$ (and $\mathcal{M}$) on each of $t_1, \ldots, t_n$ that assigns macrostates $X_1, \ldots, X_n$ (macrostates $Y_1, \ldots, Y_n$ for runs of $\mathcal{M}$) to the root nodes of given trees, $(X_1, Y_1), \ldots, (X_n, Y_n) \in$ done, and $\Delta_N((X_1, \ldots, X_n), a) \cap F_N = \emptyset \nLeftrightarrow \Delta_M((Y_1, \ldots, Y_n), a) \cap F_M = \emptyset$. We have found a counterexample tree composed of subtrees $t_1, \ldots, t_n$ with root a that one automaton accepts and the other does not, therefore $L(\mathcal{N}) \neq L(\mathcal{M})$. □

The same conclusion can be reached by using the notion of progression. Progression is better suited for proving *up-to* techniques and is mentioned here to establish a connection between bisimulation and progression on naive algorithm example.

Definition 11 (Progression)**.** *Given two relations* $R, R' \subseteq 2^{Q_N} \times 2^{Q_M}$ *on macrostates,* R *progresses to* R'*, denoted* $R \rightarrowtail R'$*, if, whenever* $\forall (X, Y) \in R : \forall a \in \Sigma : n = rank(a) - 1 \wedge \forall 1 \leq i \leq n : (X_i, Y_i) \in R$ *then*

1. $X \cap F_N = \emptyset \Leftrightarrow Y \cap F_M = \emptyset$ *and*
2. $(\Delta_N(X_1, \ldots, X, \ldots, X_n, a), \Delta_M(Y_1, \ldots, Y, \ldots, Y_n, a)) \in R'$.

Note that bisimulation is a relation for which $R \rightarrowtail R$.

There is an invariant $done \rightarrowtail done \cup todo$ corresponding to line 3 of Algorithm 7. This invariant holds because at first, *done* is empty and at each step single element from *todo* is checked for finality equivalence and moved to *done*, thus satisfying condition 1 of Definition 11 and all successor macrostate pairs reachable with the context given by *done* are added to *todo*, thus satisfying condition 2 of Definition 11.

Proof of Lemma 5 using progression. If Algorithm 7 reaches line 11 and returns *true*, then the set *todo* is empty and $done \rightarrowtail done$, making the relation built over the run of Algorithm 7 a bisimulation. The rest of the proof is identical to the proof using bisimulation in Lemma 5. □

7.2 Soundness of Bisimulation up-to Congruence

Soundness of bisimulation up-to congruence algorithm for NFTAs is proven by defining compatible functions that can be exploited by *up-to* techniques to reduce the search space and establishing containment relation between the bisimulation built by the naive algorithm and the relation built by bisimulation up-to compatible function algorithm.

7.2.1 Compatible Functions

To prove the correctness of Algorithm 9, we define a set of compatible functions, i. e. functions that preserve progression.

Definition 12 (Compatible function)**.** *A function* $f : 2^{2^{Q_N} \times 2^{Q_M}} \rightarrow 2^{2^{Q_N} \times 2^{Q_M}}$ *is compatible if it is monotone and preserves progression for all* R, R'*:*

$$R \rightarrowtail R' \quad \Longrightarrow \quad f(R) \rightarrowtail f(R')[3].$$

Lemma 6 (Containment)**.** *Let* f *be a compatible function, any bisimulation up-to* f *is contained in a bisimulation.*

Proof. Let R be a bisimulation up-to f (defined in the terms of a progression $R \rightarrowtail f(R)$). Using the compatibility of f and induction on n we get: $\forall n, f^n(R) \rightarrowtail f^{n+1}(R)$. Therefore, we have:

$$\bigcup_n f^n(R) \rightarrowtail \bigcup_n f^n(R)$$

and $\bigcup_n f^n(R)$ is a bisimulation. This bisimulation contains R by taking $n = 0$. □

Lemma 7. *The following functions are compatible.*

id : *the identity function,*
$f \circ g$: *function composition,*
$\bigcup F$: *pointwise union of a set of compatible functions* F*,*
f^ω : *iteration of a compatible function,*
r : *the constant reflexive function* $r(R) = \{(X, X) \mid X \in 2^{Q_N} \vee X \in 2^{Q_M}\}$*,*
s : *the converse function* $s(R) = \{(X, Y) \mid (Y, X) \in R\}$*,*
t : *the squaring function* $t(R) = \{(X, Y) \mid \exists Z, (X, Z) \in R \wedge (Z, Y) \in R\}$*, and*
u^ω : *the iterative union function* $u(R) = \{(X_1 \cup X_2, Y_1 \cup Y_2) \mid (X_1, Y_1), (X_2, Y_2) \in R\}^\omega$*.*

Proof. Identity, function composition. The identity function preserves progression because $id(R) = R$, therefore if $R \rightarrowtail R'$ then $id(R) \rightarrowtail id(R)$. Similar reasoning can be used with composition. If f and g are compatible and $R \rightarrowtail R$ then $f(R) \rightarrowtail f(R')$ (because f is compatible) and $g(f(R)) \rightarrowtail g(f(R'))$ (because g is compatible) [3].

Iteration. Compatibility of the iteration function can be proven by induction: $f^0 = id$ is compatible, we assume that f^n is compatible. By definition of composition $f^{n+1} = f \circ f^n$, then by induction f^m is compatible $\forall m \geq 0$. By definition $f^\omega = \bigcup_m f^m$ [3].

Pointwise union. Pointwise union of the set of functions F is compatible because if $\forall f \in F$, f is compatible, and $R \rightarrowtail R'$, then $f(R) \rightarrowtail f(R')$, therefore $\bigcup_{f \in F} f(R) \rightarrowtail \bigcup_{f \in F} f(R')$ [3].

Reflexive function. The reflexive relation $Ref = \{(X, X) \mid X \in 2^Q \cup 2^Q\}$ is always a bisimulation (i. e. $Ref \rightarrowtail Ref$ holds). Because $r(R) = Ref = r(R')$ then $r(R) = Ref \rightarrowtail Ref = r(R')$ making r compatible [3].

Converse function. Because the definition of a progression is completely symmetric (with regard to respective automata) then if $R \rightarrowtail R'$ then also $s(R) \rightarrowtail s(R')$ [3].

Squaring function. To prove compatibility of the squaring function, let us assume $R \rightarrowtail R'$. By Definition 11 it holds that $\forall (X, Y) \in R : X \cap F = \emptyset \Leftrightarrow Y \cap F = \emptyset$ and $\forall a \in \Sigma : \forall 1 \leq i \leq rank(a) : (X_i, Y_i) \in R : (\Delta(X_1, \ldots, X_i, a), \Delta(Y_1, \ldots, Y_i, a)) \in R'$. The squaring function $t(R) = \{(X, Y) \mid \exists Z, (X, Z) \in R \wedge (Z, Y) \in R\}$ is compatible because if $X \cap F = \emptyset \Leftrightarrow Z \cap F = \emptyset$ and $Z \cap F = \emptyset \Leftrightarrow Y \cap F = \emptyset$ than also $X \cap F = \emptyset \Leftrightarrow Y \cap F = \emptyset$, satisfying condition 1 of Definition 11.

And because $\forall a \in \Sigma : \forall 1 \leq i \leq rank(a) : (X_i, Z_i) \in R \wedge (Z_i, Y_i) \in R : (\Delta(X_1, \ldots, X_i, a), \Delta(Z_1, \ldots, Z_i, a)) \in R' \wedge (\Delta(Z_1, \ldots, Z_i, a), \Delta(Y_1, \ldots, Y_i, a)) \in R'$, then also $(\Delta(X_1, \ldots, X_i, a), \Delta(Y_1, \ldots, Y_i, a)) \in t(R')$, thus satisfying condition 2 of Definition 11.

Iterative union function. For the iterative union function compatibility proof, let us assume $R \rightarrowtail R'$ and choose $(X_1, Y_1), (X_2, Y_2) \in R$; now there are two possible options. Either $X_1 \cap F = \emptyset \Leftrightarrow X_2 \cap F = \emptyset$, then by Definition 11 also $Y_1 \cap F = \emptyset \Leftrightarrow Y_2 \cap F = \emptyset$, meaning X_1, X_2, Y_1, and Y_2 are either all accepting or all non-accepting, then condition 1 of Definition 11 is satisfied. This also means that $(X_1, Y_2), (X_2, Y_1) \in R$, therefore $\{X_1, X_2\} \times \{Y_1, Y_2\} \subseteq R$ and condition 2 of Definition 11 is satisfied by using Lemma 4.

The other option is that $X_1 \cap F \neq \emptyset \Leftrightarrow X_2 \cap F = \emptyset$. Condition 1 of Definition 11 is still satisfied because now $(X_1 \cup X_2) \cap F \neq \emptyset$ and $(Y_1 \cup Y_2) \cap F \neq \emptyset$ and condition 2 of Definition 11 is satisfied because now $(X_1, Y_2), (X_2, Y_1) \notin R$, therefore $\Delta(u^\omega(R)) \subseteq u^\omega(\Delta(R))$, and every successor reachable from $u^\omega(R)$ can be found in $u^\omega(\Delta(R))$. Note that condition 2 in progression definition does not necessitate $\Delta(u^\omega(R)) = u^\omega(\Delta(R))$ and $\Delta(u^\omega(R)) \subseteq u^\omega(\Delta(R))$ is sufficient. □

7.2.2 Correctness of Bisimulation up-to Congruence

To prove that bisimulation up-to congruence can be used to determine language equivalence we use an invariant $\mathit{done} \rightarrowtail c(\mathit{done} \cup \mathit{todo})$ corresponding to line 3 of Algorithm 9. To prove this, it is sufficient to show that c is a compatible function. Then, to prove the equality with language equivalence checking, it is sufficient to show that bisimulation up-to equivalence is contained in a bisimulation.

Lemma 8 (Congruence closure compatibility)**.** *The congruence closure of a relation, defined as* $c = (id \cup r \cup s \cup t \cup u^\omega)^\omega$ *is a compatible function.*

Proof. The congruence closure $c = (id \cup r \cup s \cup t \cup u^\omega)^\omega$ is a compatible function, because it is constructed strictly by using elementary compatible functions from Lemma 7. □

Lemma 9 (Bisimulation up-to congruence containment)**.** *Bisimulation up-to congruence is contained in a bisimulation.*

Proof. Bisimulation up-to congruence is contained in a bisimulation because it is a compatible function (Lemma 8) and because every bisimulation up-to compatible function is contained in a bisimulation (Lemma 6). □

Lemma 10 (Bisimulation up-to congruence, Language equivalence)**.** *Algorithm 9 with input automata* $\mathcal{N}, \mathcal{M}$ *returns true iff* $L(\mathcal{N}) = L(\mathcal{M})$*.*

Proof. By using Lemma 9 we can conclude that the relation built by the bisimulation up-to congruence is contained in a bisimulation. Then, by using Proposition 3, equality with language equivalence checking is proven. □

7.2.3 Differences in Proofs for Word and Tree Automata

The standard definition of congruence closure from [3] is $c = (id \cup r \cup s \cup t \cup u)^\omega$, but this definition cannot be directly used for tree automata because the function u is not compatible by itself. To illustrate this, we apply Lemma 4 to successors generated by two pairs (X_1, Y_1) and (X_2, Y_2) from the bisimulation R for some a where $\mathit{rank}(a) = 2$. One can see that $\Delta_a(X_1 \cup X_2, Y_1 \cup Y_2) = \Delta_a(X_1, Y_1) \cup \Delta_a(X_1, Y_2) \cup \Delta_a(X_2, Y_1) \cup \Delta_a(X_2, Y_2) \neq \Delta_a(X_1, Y_1) \cup \Delta_a(X_2, Y_2)$. Therefore $\Delta_a(X_1 \cup X_2, Y_1 \cup Y_2)$ cannot be reliably found in $u(\Delta(R))$ because it is composed by applying union up to three times (possibly less if some of the sub-results are empty or overlapping, but this cannot be guaranteed) and the definition of u permits only one use.

Therefore, a similar definition of congruence closure for tree automata $c = (id \cup r \cup s \cup t \cup u^\omega)^\omega$ remedies this by allowing the function u to be iterated by itself. By simple equality $(id \cup r \cup s \cup t \cup u^\omega)^\omega = (id \cup r \cup s \cup t \cup u)^\omega$, it can be seen that the congruence closure as a whole is not affected by this change and all approaches to calculating congruence closure membership from [3] can still be used. This change is made solely for the purpose of proving *bisimulation up-to congruence* correctness using progression and compatible functions, where compatibility of each individual function is required. If we were to use the definition $c = (id \cup r \cup s \cup t \cup u)^\omega$ the resulting closure will be the same but the function c as a whole will no longer be compatible and therefore could not be used to prove the correctness of Algorithm 9.

7.3 Naive and Iterative Successors Calculation Equivalence

Lastly, we need to prove that iterative successors calculation in Algorithm 8 is sound. This modification changes the way successors of currently processed macrostate are calculated and greatly improves efficiency of the whole algorithm. Note that this modification is independent from congruence closure calculation and therefore can be proven for naive algorithm only.

To prove the equivalence of the algorithm using naive successors calculation (Algorithm 7) and the algorithm using iterative successors calculation (Algorithm 8) we show that the sets *todo* and *done* hold exactly the same pairs at each iteration step in both algorithms. First we prove that this is true for *done* and use this in our proof that the same holds for *todo*. The proof of equivalence of the sets *todo* is further split into two parts to keep the induction steps simple.

Let us assign superscript *naive* to entities from Algorithm 7 and *iter* to entities from Algorithm 8. Subscript denotes the iteration of the algorithm. We use induction on the number of iterations of Algorithm 7 and Algorithm 8.

Lemma 11 (Equivalence of sets *done*)**.**

$$done^{naive} = done^{iter} \tag{7.1}$$

Proof.

base case:	$\text{done}_0^{naive} = \text{done}_0^{iter}$	/by definition
hypothesis:	$\text{done}_n^{naive} = \text{done}_n^{iter}$	
step:	$\text{done}_{n+1}^{naive} = \text{done}_n^{naive} \cup \{\text{actual}\} = \text{done}_{n+1}^{iter}$	/same *actual* is selected □

To prove equality of the sets *todo* (with omitted superscripts for *done* as we have already proven they are equal), we use induction to show that at each iteration both $todo^{naive}$ and $todo^{iter}$ will hold exactly the same macrostate pairs. This proof is split into two parts to keep it more readable. We assume that $\Delta(\text{done}_n) \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1}) = \Delta(\text{done}_{n+1})$. The usage of this assumption is marked in proof of Lemma 12 and is proven separately (Lemma 13).

Lemma 12 (Equivalence of sets *todo*)**.**

$$todo^{naive} = todo^{iter} \tag{7.2}$$

Proof.

base case:	$\text{todo}_0^{naive} = \text{todo}_0^{iter}$	/by definition
hypothesis:	$\text{todo}_n^{naive} = \text{todo}_n^{iter}$	
step:	$\text{todo}_{n+1}^{iter} = (\text{todo}_n^{iter} \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1})) \setminus \text{done}_{n+1}$	
	$\text{todo}_{n+1}^{iter} = (\text{todo}_n^{naive} \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1})) \setminus \text{done}_{n+1}$	
	$\text{todo}_{n+1}^{iter} = (\Delta(\text{done}_n) \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1})) \setminus \text{done}_{n+1}$	
	$\text{todo}_{n+1}^{iter} = (\Delta(\text{done}_{n+1})) \setminus \text{done}_{n+1}$	
	$\text{todo}_{n+1}^{iter} = \text{todo}_{n+1}^{iter}.$	□

Now we prove that $\Delta(\text{done}_n) \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1}) = \Delta(\text{done}_{n+1})$. For this part of the proof we need to work with the definitions of the input tree automata. Lets assume that $\mathcal{N} = (Q_N, \Delta_N, \Sigma, F_N)$ and $\mathcal{M} = (Q_M, \Delta_M, \Sigma, F_M)$ are the input tree automata. The proof that $\Delta(\text{done}_n) \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1}) = \Delta(\text{done}_{n+1})$ is again done using induction on iteration step.

Lemma 13 (Auxiliary)**.**

$$\Delta(\mathit{done}_n) \cup \Delta'(\mathit{actual}_{n+1}, \mathit{done}_{n+1}) = \Delta(\mathit{done}_{n+1}) \tag{7.3}$$

Proof.

base case: $\Delta(\text{done}_0) \cup \Delta'(\text{actual}_1, \text{done}_1) = \Delta(\text{done}_1)$

/* holds, there is only one macrostate */

hypothesis: $\Delta(\text{done}_n) \cup \Delta'(\text{actual}_{n+1}, \text{done}_{n+1}) = \Delta(\text{done}_{n+1})$

step: $\Delta(\text{done}_{n+1}) \cup \Delta'(\text{actual}_{n+2}, \text{done}_{n+2}) =$

$\Delta(\text{done}_{n+1}) \cup \Delta'(\text{actual}_{n+2}, \text{done}_{n+1} \cup \{\text{actual}_{n+2}\}) =$

$\{(\Delta_N(P_1,\ldots,P_n,a), \Delta_M(Q_1,\ldots Q_n,a)) \mid a \in \Sigma \wedge n = \mathit{rank}(a) \wedge \forall 1 \leq i \leq n :$
$(P_i, Q_i) \in \text{done}_{n+1}\} \cup \{(\Delta_N(P_1,\ldots,P_j,\ldots P_n,a), \Delta_M(Q_1,\ldots,Q_j,\ldots Q_n,a)) \mid$
$a \in \Sigma \wedge n = \mathit{rank}(a) \wedge \forall 1 \leq i \leq n : (P_i, Q_i) \in \text{done}_{n+1} \cup \{\text{actual}_{n+2}\} \wedge$
$\text{actual}_{n+2} = (P_j, Q_j)\} =$

$\{(\Delta_N(P_1,\ldots,P_j,\ldots P_n,a), \Delta_M(Q_1,\ldots,Q_j,\ldots Q_n,a)) \mid a \in \Sigma \wedge$
$n = \mathit{rank}(a) \wedge \forall 1 \leq i \leq n : (P_i, Q_i) \in \text{done}_{n+1} \vee ((P_i, Q_i) \in \text{done}_{n+1} \cup$
$\{\text{actual}_{n+2}\} \wedge \text{actual}_{n+2} = (P_j, Q_j))\} =$

$\{(\Delta_N(P_1,\ldots,P_j,\ldots P_n,a), \Delta_M(Q_1,\ldots,Q_j,\ldots Q_n,a)) \mid a \in \Sigma \wedge$
$n = \mathit{rank}(a) \wedge \forall 1 \leq i \leq n : ((P_i, Q_i) \in \text{done}_{n+1} \vee (P_i, Q_i) \in \text{done}_{n+1} \cup$
$\{\text{actual}_{n+2}\}) \wedge ((P_i, Q_i) \in \text{done}_{n+1} \vee \text{actual}_{n+2} = (P_j, Q_j))\} =$

$\Delta(\text{done}_{n+2}) \cap \{(\Delta_N(P_1,\ldots,P_j,\ldots P_n,a), \Delta_M(Q_1,\ldots,Q_j,\ldots Q_n,a)) \mid$
$a \in \Sigma \wedge n = \mathit{rank}(a) \wedge \forall 1 \leq i \leq n : (P_i, Q_i) \in \text{done}_{n+1} \vee$
$\text{actual}_{n+2} = (P_j, Q_j)\}$
$= \Delta(\text{done}_{n+2})$ □

This is sufficient to prove that iterative successors generation is sound. Moreover, we have shown that contents on *todo* and *done* are not dependent on chosen successors generation method. Therefore, iterative successors generation can be freely combined with *bisimulation up-to congruence* without any impacting on the result of the equivalence check.

Chapter 8

Implementation

The *bisimulation up-to congruence* algorithm for testing language equivalence of NFTAs introduced in Chapter 6 is implemented as an extension of libVATA [14] C++ library. It is designed to work with the explicit representation of tree automata in the Timbuk [8] format. The algorithm is incorporated into explicit tree automata equivalence and inclusion checking and is accessible using the standard libVATA command line interface. Its behaviour and output are conforming to the standard libVATA format[1].

Bisimulation up-to congruence implementation is in class `BisimulationBase`, which contains general purpose methods shared by inclusion and equivalence checking. Classes `BisimulationEquivalence` and `BisimulationInclusion` then implement specific methods that are used for equivalence and inclusion checking respectively.

To fully integrate this extension with libVATA, minor changes had to be made to the existing libaVATA code that allow the new algorithm to be called using the existing interface. These changes are limited to extending argument option `alg` and adding option `congr`, parsing of those arguments and a block that executes the new algorithm if libVATA is run with those arguments.

Interface

Bisimulation up-to congruence is accessible using the main libVATA interface. It is called by specifying that either inclusion or equivalence check should be performed and then selecting options `dir=up` (bottom-up tree processing), `alg=bisimulation` (check using bisimulation) and `congr=yes` (use congruence closure).

For inclusion checking, automata are passed in the order *smaller, bigger* and the algorithm calculates if $L(smaller) \subseteq L(bigger)$. For equivalence checking the order does not matter. Example executions on files *aut_file1* and *aut_file2* for equivalence and inclusion check respectively:

```
$ vata equiv -o "dir=up,alg=bisimulation,congr=yes" 'aut_file1' 'aut_file2'
$ vata incl -o "dir=up,alg=congr" 'aut_file1' 'aut_file2'
```

Note the extra argument for equivalence testing `congr=yes`, this is an argument that enables and disables using of the congruence closure. Setting `congr=no` degrades the algorithm to the plain bisimulation and was used during experimenting with the implementation. This option is not available for inclusion testing and is disabled in the released code.

[1] More information about libVATA C++ library, its interface, and input and output formats can be found at https://github.com/ondrik/libvata/blob/master/README.md

Input

Equivalence and inclusion check using bisimulation up-to congruence is performed with two tree automata as an input. The algorithm uses the default libVATA parser and the default data structures for storing the loaded automata. It only supports automata in explicit Timbuk format and does not support equivalence and inclusion checking on automata in the semi-symbolic representation.

Custom Data Structures

The *bisimulation up-to congruence* implementation uses standard libVATA data structures with a single exception. At the beginning of every equivalence check, *arbitrary state successors* are pre-calculated for every symbol-position-state triplet and stored in an array `successors`. Because alphabet size, arity, and number of states are fixed, this information can be stored in a fixed size array. Successor calculation for macrostate is then done by exploiting set operations proposed in Section 3.7. Let $\mathcal{A} = (Q, \Delta, \Sigma, F)$ be a tree automaton, $a \in \Sigma$, $q \in Q$, and assume $rank(a) = 2$, then

$$\texttt{successors[a][0][q]} = \Delta_a(\{q\}, \_) \text{ and} \tag{8.1}$$
$$\texttt{successors[a][1][q]} = \Delta_a(\_, \{q\}). \tag{8.2}$$

By using Lemma 1 and 3, we can calculate successor macrostate for some macrostates $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots y_m\}$ over a symbol a, $rank(a) = 2$ as

$$\Delta_a(X, \_) = \bigcup_{1 \leq i \leq n} \Delta_a(x_i, \_), \tag{8.3}$$
$$\Delta_a(\_, Y) = \bigcup_{1 \leq i \leq m} \Delta_a(\_, y_i), \text{ and} \tag{8.4}$$
$$\Delta_a(X, Y) = \Delta_a(X, \_) \cap \Delta_a(\_, Y). \tag{8.5}$$

Note that $\Delta_a(X, Y)$ can be calculated by using set operations on the elements of the `successors` array. Moreover, if either $\Delta_a(X, \_)$ or $\Delta_a(\_, Y)$ is empty, then $\Delta_a(X, Y)$ is also empty and the successor can be determined without the need to compute all *node-wise successors*.

Cache

To further improve performance of this implementation, a cache for storing *arbitrary state successors* for macrostates is added. This cache stores successors for already encountered macrostate pair and position combinations and prevents repeated computation of the same successors. It is implemented as an array of maps `set_successors`. The array is indexed with a symbol and a position and contains a map from macrostates to their successors. Let $\mathcal{A} = (Q, \Delta, \Sigma, F)$ be a tree automaton, $a \in \Sigma$, $X \subseteq Q$, and assume $rank(a) = 2$, then

$$\texttt{set\_successors[a][0].find(X)} = \Delta_a(X, \_) \text{ and}$$
$$\texttt{set\_successors[a][1].find(X)} = \Delta_a(\_, X).$$

Node-wise successors can be calculated with the information retrieved from the array `set_successors` by using Lemma 2.

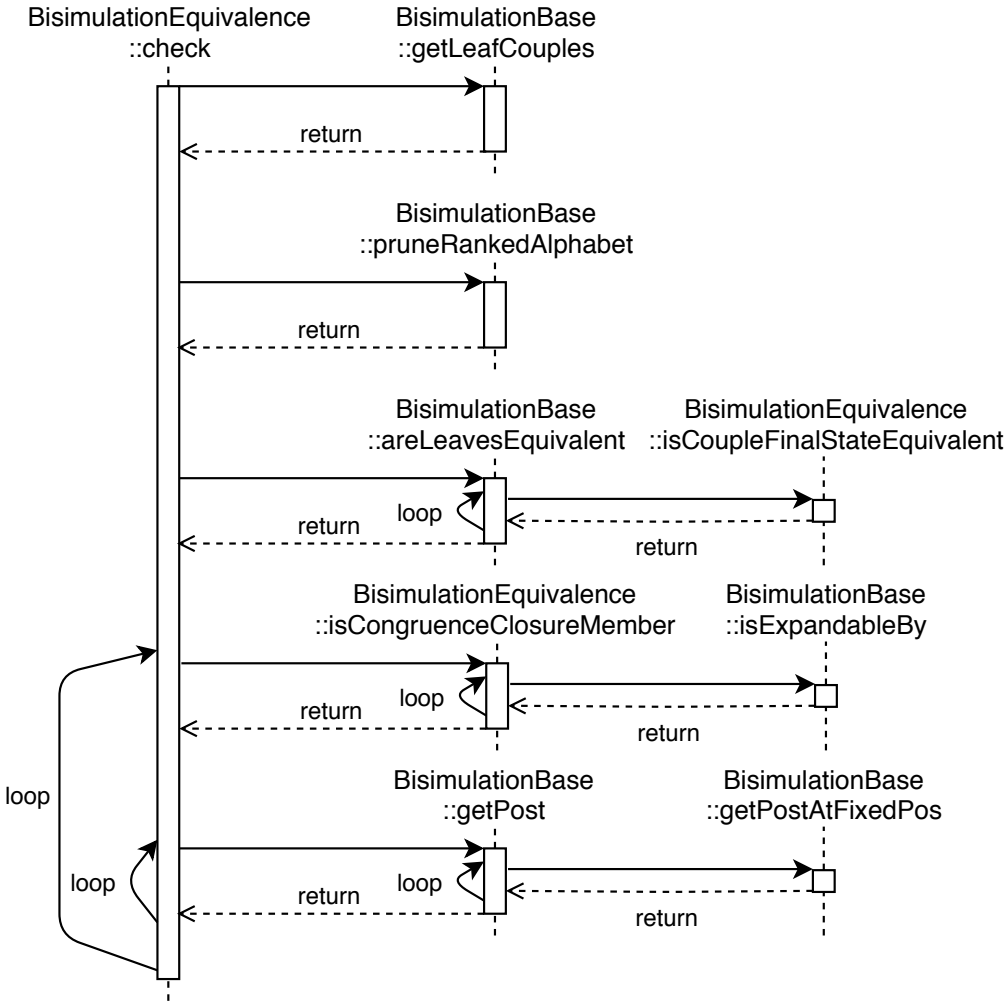


Figure 8.1: Graph depicting methods implemented in `BisimulationEquivalence` and `BisimulationBase` classes with caller-callee relationships.

8.1 Structure and Methods

The implementation roughly follows structuring from Algorithm 9 and is illustrated in Figure 8.1. the method `check` corresponds with the whole Algorithm 9. First, `getLeafCouples` is called, this method initializes the set *todo*. Because this is the only place where leaf rules are used, `pruneRankedAlphabet` is called afterward to remove symbols with the rank 0 from the alphabet.

Leaf pairs are then checked for final state equivalence. In Algorithm 9, this is done when pairs are removed from *todo*, but during implementation this was moved to the inserting phase for performance improvement. Leaf pairs are checked separately before the main loop is entered.

The main loop consists of selecting a single pair *actual*, it is checked if *actual* is a member of the congruence closure of $todo \cup done$ (`isCongruenceClosureMember`). Finally, successors are calculated for *actual* by looping over the pruned alphabet and calling `getPost` for each symbol. This method is also responsible for checking final state equivalence and inserting successors into *todo*.

Chapter 9

Experiments

Bisimulation up-to congruence for NFTAs was compared with the bottom-up algorithm for inclusion checking based on antichains implemented in the VATA library [14]. In our evaluation, neither algorithm was combined with the simulation approach. The inclusion checking algorithm based on antichains was used to test equivalence by checking inclusion in both directions. To take care of the possibility that inclusion would hold in one direction and not in the other (artificially inflating the time two inclusion checks need to find a counterexample), the lower measured time for both directions was taken as the result.

Experiments were conducted on a set of NFTA obtained from Abstract Regular Tree Model Checking (ARTMC). There were 95 automata[1] in total ranging in size up to automata with approximately 100 symbols in the alphabet, 1000 states, and over 20000 transitions. Every automaton was tested for equivalence with every other automaton (including itself), totaling 9025 comparisons of which 594 were valid equivalences (499 on not identical automata) and 8426 were invalid equivalences.

Percentile times for both algorithms can be seen in Table 9.1 for all comparisons and in Table 9.2 for valid equivalences only. For problems that could be decided relatively quickly (most of the invalid equivalences fall into this category), the algorithm based on antichains performed better than *bisimulation up-to congruence*, but with increasing difficulty this reversed and *bisimulation up-to congruence* outperformed antichains on the majority of the difficult examples.

Table 9.1: Required time [s] (all results, timeout 300 s)

Algorithm	50%	90%	95%	99%	100%
Antichains	0.100	0.327	0.670	23.406	-
Bisimulation	0.112	0.336	0.533	10.333	202.674

Table 9.2: Required time [s] (valid equivalences only, timeout 300 s)

Algorithm	50%	90%	95%	99%	100%
Antichains	3.87	69.28	131.23	297.24	-
Bisimulation	2.73	15.13	22.39	178.25	202.67

[1] The set of 95 automata used in experiments can be found in the folder `reference_automata/` of enclosed medium (see Appendix A).

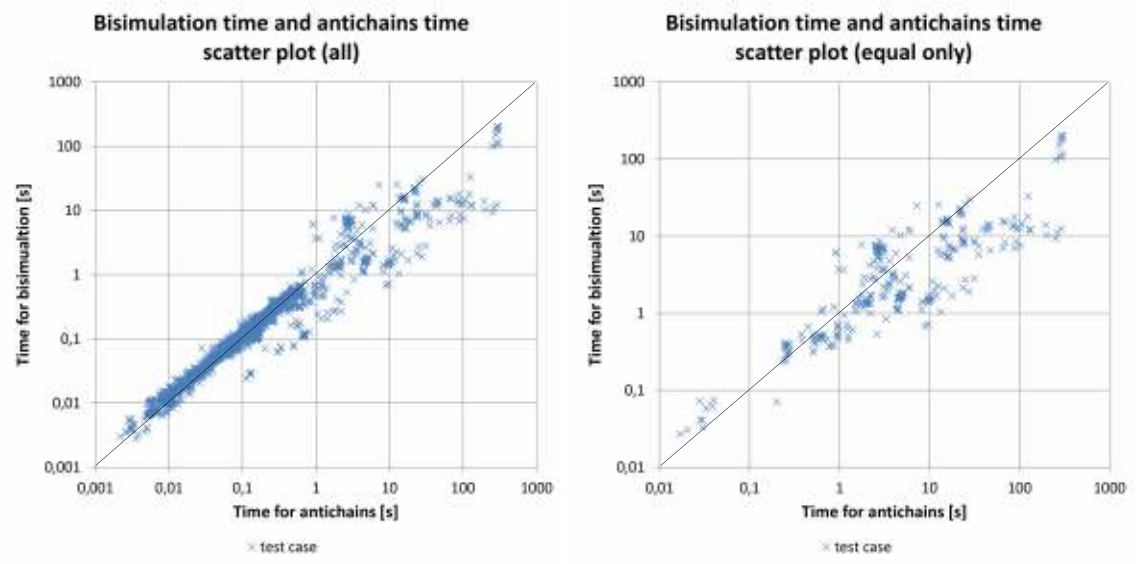


Figure 9.1: Scatter plot of the times needed by the bisimulation up-to congruence and the algorithm based on antichains to check equivalence.

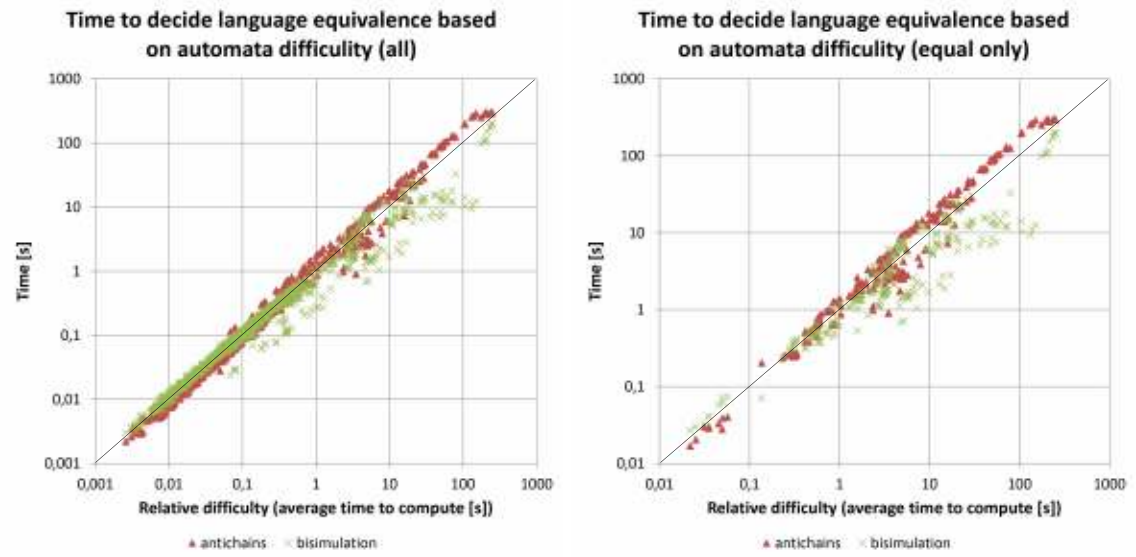


Figure 9.2: Comparison of the time needed to decide language equivalence for the algorithm based on antichains and bisimulation up-to congruence. Relative difficulty is the average time needed by both algorithms to decide the given problem.

Scatter plots of the times both algorithms took to check equivalence are in Figure 9.1 (all cases and valid equivalences only). It can be seen that for simpler test cases *bisimulation up-to congruence* performs worse because majority of the test cases in the bottom left quarter are above the diagonal. For average test cases results are inconclusive, there are test cases both above and bellow the diagonal around the center and neither algorithm performs clearly better. Lastly, for difficult cases, *bisimulation up-to congruence* starts to outperform the antichain-based algorithm (test cases bellow the diagonal in the upper right quarter).

Table 9.3: Percentage of macrostate pairs discarded from further search by *bisimulation up-to congruence* and antichain-based algorithm. All test cases (left), valid equivalences only (middle), and invalid equivalences only (right).

	Equvalence check result		
Algorithm	All	Equal	Not equal
Antichains	13.62%	12.94%	20.11%
Bisimulation	33.79%	39.07%	5.24%

Another comparison of time required to check equivalence, this time based on relative difficulty of individual test cases, is in Figure 9.2 (all cases and valid equivalences only). Again, it can be seen that for simpler test cases the antichain-based algorithm performs better, but *bisimulation up-to congruence* eventually outperforms it for more difficult cases.

The *bisimulation up-to congruence* algorithm has a larger overhead but its results follow a trend with gentler slope than the trend of results from the antichain-based algorithm. This leads to poor performance of *bisimulation up-to congruence* for simpler cases and increasingly better results with growing difficulty of the test cases, where the efficiency of the pruning abilities out-weights a larger overhead.

9.1 Congruence Closure Efficiency

An attempt to measure implementation-independent efficiency of both algorithms was made. There were two possible properties that could be considered for this role: the total number of processed pairs and the percentage of pairs discarded from further search.

Because antichain-based algorithms operates on product states (state-macrostate pairs) and *bisimulation up-to congruence* operates on macrostate-macrostate pairs, the total number of processed pairs is not directly comparable. This makes measuring the number of processed pairs unsuitable for comparing the efficiency of these algorithms.

The percentage of discarded pairs roughly represents an algorithm's ability to prune the search space and is measured relative to the total number of processed pairs. This metric therefore takes into account different sizes of the search space for both algorithms and is more suited to comparing efficiency of these algorithms.

The percentage of discarded pairs for both algorithms is in Table 9.3. In this regard, *bisimulation up-to congruence* clearly outperforms the antichain-based algorithm for valid equivalences and is outperformed for invalid equivalences. Because tests of valid equivalence generally produce larger search spaces, this difference of performance can be interpreted as a congruence closure being progressively more powerful with a growing number of pairs for which the closure is calculated.

For all test cases, *bisimulation up-to congruence* is able to discard approximately 34% of all pairs because they are found in the congruence closure. This includes only pairs that are not trivial, meaning they are not a duplicate of an already encountered pair. This value rises to 39% for valid equivalence only, again conforming the hypothesis that efficiency of the congruence closure grows with the size of the search space. For valid equivalences, the antichain-based algorithm is able to discard only 13% of encountered pairs. This corresponds with the better performance of *bisimulation up-to congruence* for valid equivalence test cases.

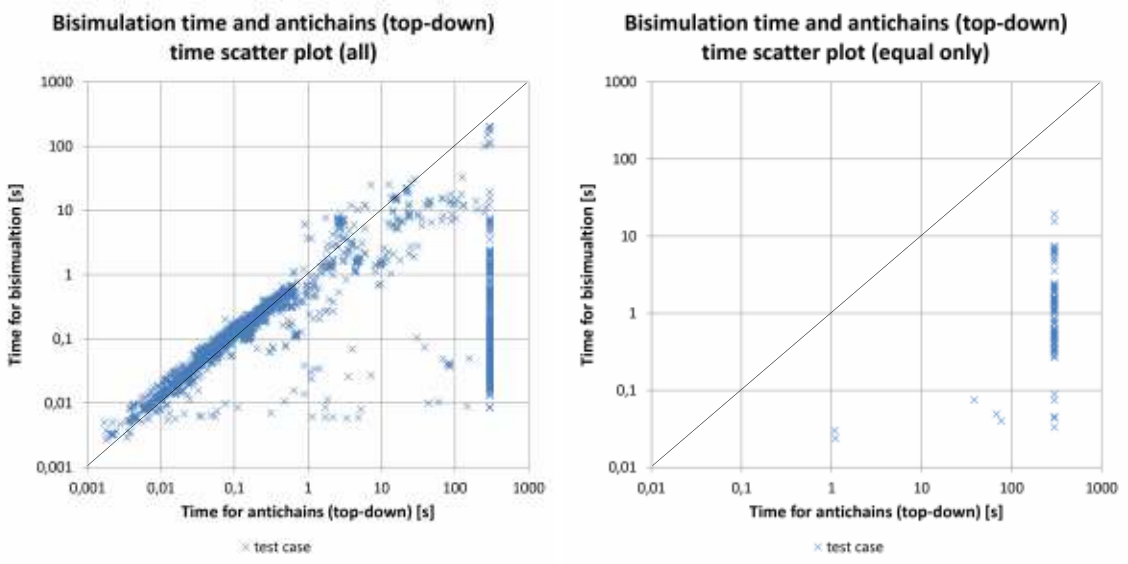


Figure 9.3: Scatter plot of the times needed by the bisimulation up-to congruence and the algorithm based on antichains (top-down) to check equivalence.

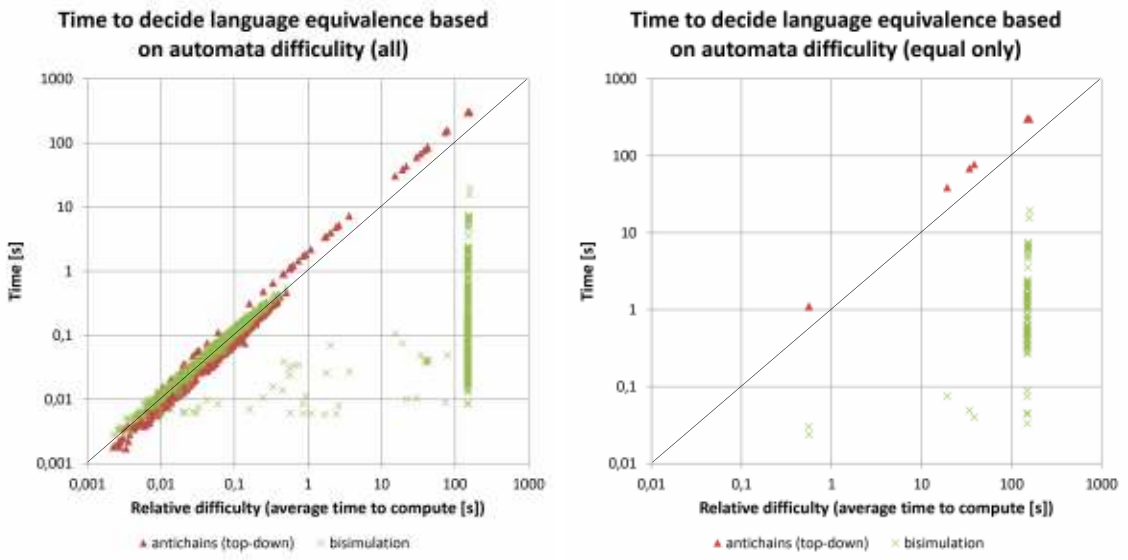


Figure 9.4: Comparison of the time needed to decide language equivalence for the algorithm based on antichains (top-down) and bisimulation up-to congruence. Relative difficulty is the average time needed by both algorithms to decide the given problem.

For invalid equivalences, only 5% of the pairs are discarded for *bisimulation up-to congruence* and 20% for the antichain-based algorithm. This suggests that the antichain-based approach should outperform *bisimulation up-to congruence* for these test cases and is in accordance with earlier findings from time measurements from this section.

9.2 Comparison with Top-down Approach

Another comparison was made with top-down-antichain-based approach. Scatter plots of the times *bisimulation up-to congruence* and top-down antichains took to check equivalence are in Figure 9.3 (all cases and valid equivalences only). The top-down algorithm preforms significantly worse compared to its bottom-up counterpart and *bisimulation up-to congruence* has shorter execution times for majority of the test cases.

Similarly, in Figure 9.4, execution times based on test case difficulty are shown. It can be seen that the trend *bisimulation up-to congruence* follows is much less steep than that of the top-down antichains approach and larger overhead of *bisimulation up-to congruence* is compensated by efficiency difference almost immediately.

Note that because of the poor performance of the top-down-antichain-based algorithm, this comparison was done only on a subset of the test cases used with bottom-up antichains approach because performing equivalence check using top-down on all 9025 test cases required more than 48 hours of execution time and, because of that, was terminated prematurely.

Chapter 10

Conclusion

In this thesis, *bisimulation up-to congruence*, a novel algorithm for testing language equivalence and inclusion on tree automata was presented. This algoritm operates on nondeterministic tree automata and performs *on the fly* determinization to try to offset state explosion connected to determinization. Moreover, it tries to build only a fraction of a bisimulation relation that would usually be required to check language equivalence by exploiting properties of the congruence closure to prune the search space.

In comparison with the algorithm based on antichains, *bisimulation up-to congruence* has a larger overhead, thus performing worse on simpler examples and invalid equivalences where counterexample can be found relatively quickly, but it outperforms the algorithm based on antichains if the problems become complex enough and effectiveness of the search space pruning outweighs larger overhead. The difference between efficiency of *bisimulation up-to congruence* and the algorithm based on antichains seems to grow with increasing difficulty of test cases.

Therefore, the first main goal of this thesis, i. e., to develop an algorithm for language inclusion and equivalence testing on tree automata and outperform existing approaches on a portion of real-world examples, was accomplished.

This thesis also presented extensions to transition functions used in tree automata for macrostates. Transition functions on macrostates were then used to define behaviour of transitions with regard to set operations performed on macrostates, an important achievement that generalized and formalized the relation between operations on macrostates and results of transition functions. To the best of the author's knowledge, those relations were not formalized before as cited literature merely acknowledges that certain properties of transition functions in word automata cannot be applied to tree automata (for example relation between union on macrostates and outcome of the transition), but does not elaborate on what proper generalizations of those properties are.

Lastly, a formal proof of correctness of *bisimulation up-to congruence* was presented. It was built by adapting an existing notion of compatible functions to tree automata with modifications to those compatible functions that could not be directly adapted. Notably, the union function had to be redefined by using knowledge gained from study of relation between union on macrostates and outcome of the transition function.

First, it was proven that the naive algorithm is sound and can be used to check language equivalence by asserting that a bisimulation relation is build over a successful run of the algorithm and cannot be build otherwise. Second, it was proven that successors can be calculated iteratively for newly discovered macrostates by asserting that the sets *todo* and *done* contain exactly the same elements in each step of calculation regardless of whether

the complete or iterative successors calculation is used. And finally, it was shown that *bisimulation up-to congruence* is equivalent to naive algorithm by asserting that congruence closure is a compatible function and by showing that any bisimulation up-to compatible function is contained in a bisimulation calculated by naive algorithm.

10.1 Future Work

Because the direction of processing trees has a lot of impact on tree automata, even restricting the set of recognizable languages for deterministic top-down automata, it will be interesting to study the effects of parsing direction on the performance of language equivalence and inclusion checking algorithms. Therefore, modifying *bisimulation up-to congruence* for top-down automata and comparing its effectiveness with the bottom-up approach (for both *bisimulation up-to congruence* and the antichain-based algorithm) can yield some insight into this issue.

Another possibility is to augment *bisimulation up-to congruence* with a simulation relation. Language equivalence and inclusion checking based on simulation relation can be extremely efficient, but this technique is not complete. Combining *bisimulation up-to congruence* with a simulation relation could possibly exploit effectiveness of a simulation for cases where it is sufficient and use *bisimulation up-to congruence* for cases where simulation fails.

The results of the comparison done in this thesis are not corresponding to those in [7]. Therefore, further tests to determine efficiency of *bisimulation up-to congruence* and antichains approach on larger and more diverse automata sets should be made. To rule out possible distortion of results stemming from different optimization level of *bisimulation up-to congruence* and antichains implementations, a new measure to directly compare search space sizes and not run times should be developed.

Lastly, experimental modification of bisimulation up-to congruence for direct inclusion checking was presented in this thesis, but no attempt for a formal proof was made. Therefore deeper study of bisimulation up-to congruence for direct inclusion checking should be made, including thorough testing on larger and more diverse tree automata set and formal proof of correctness of this algorithm (or further modification of this algorithm should this modification be proven unsound).

Bibliography

[1] Abdulla, P. A.; Bouajjani, A.; Holík, L.; et al.: Composed Bisimulation for Tree Automata. *Int. J. Found. Comput. Sci.*. vol. 20, no. 4. 2009: pp. 685–700.

[2] Abdulla, P. A.; Chen, Y.-F.; Holík, L.; et al.: When Simulation Meets Antichains. In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by J. Esparza; R. Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg. 2010. ISBN 978-3-642-12002-2. pp. 158–174.

[3] Bonchi, F.; Pous, D.: Checking NFA Equivalence with Bisimulations Up to Congruence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. New York, NY, USA: ACM. 2013. ISBN 978-1-4503-1832-7. pp. 457–468. doi:10.1145/2429069.2429124. Retrieved from: http://doi.acm.org/10.1145/2429069.2429124

[4] Brainerd, W. S.: The minimalization of tree automata. 1968.

[5] Comon, H.; Dauchet, M.; Gilleron, R.; et al.: Tree Automata Techniques and Applications. Retrieved from: http://www.grappa.univ-lille3.fr/tata. 2007. release October, 12th 2007.

[6] De Wulf, M.; Doyen, L.; Henzinger, T. A.; et al.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verification*, edited by T. Ball; R. B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg. 2006. ISBN 978-3-540-37411-4. pp. 17–30.

[7] Fu, C.; Deng, Y.; Jansen, D. N.; et al.: On Equivalence Checking of Nondeterministic Finite Automata. In *Dependable Software Engineering. Theories, Tools, and Applications*. Cham: Springer International Publishing. 2017. ISBN 978-3-319-69483-2. pp. 216–231.

[8] Genet, T.: Timbuk/Taml: A Tree Automata Library. 2003. Retrieved from: http://doi.acm.org/10.1145/2429069.2429124

[9] Holík, L.; Lengál, O.; Šimáček, J.; et al.: Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata. In *Automated Technology for Verification and Analysis*, edited by T. Bultan; P.-A. Hsiung. Berlin, Heidelberg: Springer Berlin Heidelberg. 2011. ISBN 978-3-642-24372-1. pp. 243–258.

[10] Hopcroft, J.; Karp, R.: A Linear Algorithm for Testing Equivalence of Finite Automata. Technical report. Dept. of Computer Science, Cornell U. December 1971.

[11] Hosoya, H.: *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press. 2010. doi:10.1017/CBO9780511762093.

[12] Klarlund, N.; Møller, A.: *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University. January 2001.

[13] Kozen, D. C.: *Automata and Computability*. Berlin, Heidelberg: Springer-Verlag. first edition. 1997. ISBN 0387949070.

[14] Lengál, O.; Šimáček, J.; Vojnar, T.: VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by C. Flanagan; B. König. Berlin, Heidelberg: Springer Berlin Heidelberg. 2012. ISBN 978-3-642-28756-5. pp. 79–94.

[15] Seidl, H.: Deciding equivalence of finite tree automata. *SIAM Journal on Computing*. vol. 19, no. 3. 1990: pp. 424–437.

[16] Žufan, P.: *Congruences for Tree Automata*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=19744

[17] Češka, M.; Vojnar, T.: Theoretical Computer Science Textbook. Technical report. Faculty of Information Technology, Brno University of Technology. February 2009.

Appendix A

Enclosed Medium

- `libvata-congruence/` — root directory of libVATA library with *bisimulation up-to congruence* algorithm implemented
- `reference_automata/` — the set of automata used in experiments in Section 9
- `src_code/` — directory with files implementing the core of equivalence and inclusion algorithms described in this thesis (the same files can be found in libvata-congruence/src/), not standalone, can only be compiled with libVATA
- `src_thesis/` — directory with the source code of this thesis
- `thesis.pdf` — digital version of this document
- `README.md` — instructions for compiling and running libVATA with *bisimulation up-to congruence* algorithm implemented
- `reference_results/equiv_test.log` — reference results from performance testing of equivalence check (Appendix B)
- `reference_results/incl_test.log` — reference results from performance testing of inclusion check

Appendix B

Reference results

The reference results from performance testing of equivalence check using *bisimulation up-to congruence* (times in column congr-up) and double inclusion using *antichains* (times in columns dincl and dincld, for both directions of comparing double inclusion). Absolute times may vary depending on machine used for testing but relative values should be the same (e. g. if *bisimulation up-to congruence* took half the time as *antichains* for specific test in reference results it should take half the time on any machine).

This test was performed using standard libVATA script on standard set of tree automata and can be replicated by executing

```
$ ./tests/equiv_test.sh ./tests/aut_timbuk_smaller/
```

in libVATA root directory (see README.md on enclosed medium).

```
======== Checking equivalence of automata =======
Automata directory:   ./tests/aut_timbuk_smaller/
Timeout:              300 s
reading files ...
-----------------------------------------------
aut1;      aut2;     dincl;     dincld;  congr-up;      '==';     match;
A0053;     A0053;    0.0228;    0.0228;    0.0319;         1;        ok;
A0053;     A0054;    0.0069;    0.0065;    0.0081;         0;        ok;
A0053;     A0055;    0.0204;    0.0060;    0.0115;         0;        ok;
A0053;     A0056;    0.0066;    0.0074;    0.0081;         0;        ok;
A0053;     A0057;    0.0068;    0.0068;    0.0083;         0;        ok;
A0053;     A0058;    0.0073;    0.0071;    0.0085;         0;        ok;
A0053;     A0059;    0.0071;    0.0068;    0.0086;         0;        ok;
A0053;     A0060;    0.0181;    0.0068;    0.0083;         0;        ok;
A0053;     A0062;    0.0146;    0.0071;    0.0088;         0;        ok;
A0053;     A0063;    0.0104;    0.0109;    0.0124;         0;        ok;
A0053;     A0064;    0.0104;    0.0107;    0.0124;         0;        ok;
A0053;     A0065;    0.0103;    0.0111;    0.0125;         0;        ok;
A0053;     A0070;    0.0109;    0.0118;    0.0131;         0;        ok;
A0053;     A0080;    0.0120;    0.0123;    0.0144;         0;        ok;
A0053;     A0082;    0.0127;    0.0128;    0.0147;         0;        ok;
A0053;     A0083;    0.0127;    0.0129;    0.0149;         0;        ok;
A0053;     A0086;    0.0207;    0.0213;    0.0234;         0;        ok;
```

```
A0053;    A0087;    0.0157;    0.0259;    0.0187;        0;        ok;
A0053;    A0088;    0.0162;    0.0188;    0.0208;        0;        ok;
A0053;    A0089;    0.0156;    0.0169;    0.0187;        0;        ok;
A0054;    A0053;    0.0069;    0.0102;    0.0119;        0;        ok;
A0054;    A0054;    0.0387;    0.0387;    0.0422;        1;        ok;
A0054;    A0055;    0.0076;    0.0070;    0.0085;        0;        ok;
A0054;    A0056;    0.0075;    0.0131;    0.0137;        0;        ok;
A0054;    A0057;    0.0103;    0.0090;    0.0116;        0;        ok;
A0054;    A0058;    0.0082;    0.0095;    0.0097;        0;        ok;
A0054;    A0059;    0.0083;    0.0084;    0.0098;        0;        ok;
A0054;    A0060;    0.0106;    0.0076;    0.0115;        0;        ok;
A0054;    A0062;    0.0087;    0.0080;    0.0100;        0;        ok;
A0054;    A0063;    0.0114;    0.0156;    0.0136;        0;        ok;
A0054;    A0064;    0.0137;    0.0218;    0.0132;        0;        ok;
A0054;    A0065;    0.0112;    0.0196;    0.0132;        0;        ok;
A0054;    A0070;    0.0118;    0.0547;    0.0138;        0;        ok;
A0054;    A0080;    0.0141;    0.0131;    0.0161;        0;        ok;
A0054;    A0082;    0.0134;    0.0136;    0.0159;        0;        ok;
A0054;    A0083;    0.0136;    0.0137;    0.0158;        0;        ok;
A0054;    A0086;    0.0219;    0.0227;    0.0249;        0;        ok;
A0054;    A0087;    0.0165;    0.0335;    0.0194;        0;        ok;
A0054;    A0088;    0.0168;    0.0210;    0.0195;        0;        ok;
A0054;    A0089;    0.0170;    0.0180;    0.0196;        0;        ok;
A0055;    A0053;    0.0062;    0.0161;    0.0073;        0;        ok;
A0055;    A0054;    0.0074;    0.0073;    0.0084;        0;        ok;
A0055;    A0055;    0.0297;    0.0295;    0.0391;        1;        ok;
A0055;    A0056;    0.0069;    0.0081;    0.0085;        0;        ok;
A0055;    A0057;    0.0071;    0.0084;    0.0088;        0;        ok;
A0055;    A0058;    0.0074;    0.0127;    0.0126;        0;        ok;
A0055;    A0059;    0.0074;    0.0078;    0.0089;        0;        ok;
A0055;    A0060;    0.0204;    0.0080;    0.0090;        0;        ok;
A0055;    A0062;    0.0175;    0.0084;    0.0095;        0;        ok;
A0055;    A0063;    0.0106;    0.0116;    0.0127;        0;        ok;
A0055;    A0064;    0.0107;    0.0116;    0.0127;        0;        ok;
A0055;    A0065;    0.0107;    0.0133;    0.0132;        0;        ok;
A0055;    A0070;    0.0112;    0.0434;    0.0134;        0;        ok;
A0055;    A0080;    0.0125;    0.0136;    0.0146;        0;        ok;
A0055;    A0082;    0.0130;    0.0142;    0.0149;        0;        ok;
A0055;    A0083;    0.0131;    0.0139;    0.0152;        0;        ok;
A0055;    A0086;    0.0212;    0.0217;    0.0233;        0;        ok;
A0055;    A0087;    0.0160;    0.0280;    0.0190;        0;        ok;
A0055;    A0088;    0.0162;    0.0193;    0.0192;        0;        ok;
A0055;    A0089;    0.0197;    0.0172;    0.0231;        0;        ok;
A0056;    A0053;    0.0077;    0.0069;    0.0083;        0;        ok;
A0056;    A0054;    0.0089;    0.0074;    0.0090;        0;        ok;
A0056;    A0055;    0.0086;    0.0069;    0.0083;        0;        ok;
A0056;    A0056;    0.0398;    0.0402;    0.0536;        1;        ok;
A0056;    A0057;    0.0262;    0.0075;    0.0091;        0;        ok;
```

```
A0056;    A0058;    0.0307;    0.0101;    0.0092;        0;        ok;
A0056;    A0059;    0.0294;    0.0078;    0.0096;        0;        ok;
A0056;    A0060;    0.0143;    0.0076;    0.0145;        0;        ok;
A0056;    A0062;    0.0099;    0.0121;    0.0144;        0;        ok;
A0056;    A0063;    0.0111;    0.0114;    0.0131;        0;        ok;
A0056;    A0064;    0.0119;    0.0118;    0.0131;        0;        ok;
A0056;    A0065;    0.0120;    0.0114;    0.0164;        0;        ok;
A0056;    A0070;    0.0118;    0.0121;    0.0140;        0;        ok;
A0056;    A0080;    0.0130;    0.0130;    0.0151;        0;        ok;
A0056;    A0082;    0.0135;    0.0136;    0.0158;        0;        ok;
A0056;    A0083;    0.0135;    0.0136;    0.0158;        0;        ok;
A0056;    A0086;    0.0217;    0.0228;    0.0247;        0;        ok;
A0056;    A0087;    0.0174;    0.0370;    0.0206;        0;        ok;
A0056;    A0088;    0.0179;    0.0228;    0.0256;        0;        ok;
A0056;    A0089;    0.0177;    0.0187;    0.0204;        0;        ok;
A0057;    A0053;    0.0071;    0.0073;    0.0105;        0;        ok;
A0057;    A0054;    0.0092;    0.0078;    0.0092;        0;        ok;
A0057;    A0055;    0.0089;    0.0073;    0.0087;        0;        ok;
A0057;    A0056;    0.0082;    0.0287;    0.0094;        0;        ok;
A0057;    A0057;    0.0448;    0.0504;    0.0777;        1;        ok;
A0057;    A0058;    0.0345;    0.0100;    0.0124;        0;        ok;
A0057;    A0059;    0.0328;    0.0091;    0.0096;        0;        ok;
A0057;    A0060;    0.0117;    0.0097;    0.0094;        0;        ok;
A0057;    A0062;    0.0097;    0.0080;    0.0100;        0;        ok;
A0057;    A0063;    0.0114;    0.0122;    0.0132;        0;        ok;
A0057;    A0064;    0.0115;    0.0122;    0.0134;        0;        ok;
A0057;    A0065;    0.0113;    0.0124;    0.0132;        0;        ok;
A0057;    A0070;    0.0121;    0.0559;    0.0139;        0;        ok;
A0057;    A0080;    0.0131;    0.0131;    0.0151;        0;        ok;
A0057;    A0082;    0.0137;    0.0137;    0.0158;        0;        ok;
A0057;    A0083;    0.0137;    0.0139;    0.0160;        0;        ok;
A0057;    A0086;    0.0218;    0.0223;    0.0241;        0;        ok;
A0057;    A0087;    0.0167;    0.0362;    0.0194;        0;        ok;
A0057;    A0088;    0.0169;    0.0209;    0.0195;        0;        ok;
A0057;    A0089;    0.0169;    0.0179;    0.0195;        0;        ok;
A0058;    A0053;    0.0073;    0.0069;    0.0085;        0;        ok;
A0058;    A0054;    0.0100;    0.0080;    0.0096;        0;        ok;
A0058;    A0055;    0.0140;    0.0071;    0.0122;        0;        ok;
A0058;    A0056;    0.0079;    0.0349;    0.0144;        0;        ok;
A0058;    A0057;    0.0103;    0.0357;    0.0134;        0;        ok;
A0058;    A0058;    0.0546;    0.0528;    0.0736;        1;        ok;
A0058;    A0059;    0.0339;    0.0093;    0.0100;        0;        ok;
A0058;    A0060;    0.0099;    0.0080;    0.0097;        0;        ok;
A0058;    A0062;    0.0102;    0.0082;    0.0100;        0;        ok;
A0058;    A0063;    0.0124;    0.0128;    0.0136;        0;        ok;
A0058;    A0064;    0.0116;    0.0127;    0.0137;        0;        ok;
A0058;    A0065;    0.0116;    0.0125;    0.0133;        0;        ok;
A0058;    A0070;    0.0122;    0.0667;    0.0144;        0;        ok;
```

```
A0058;    A0080;    0.0200;    0.0201;    0.0154;        0;        ok;
A0058;    A0082;    0.0139;    0.0176;    0.0160;        0;        ok;
A0058;    A0083;    0.0138;    0.0145;    0.0161;        0;        ok;
A0058;    A0086;    0.0240;    0.0252;    0.0260;        0;        ok;
A0058;    A0087;    0.0170;    0.0374;    0.0195;        0;        ok;
A0058;    A0088;    0.0206;    0.0226;    0.0223;        0;        ok;
A0058;    A0089;    0.0173;    0.0185;    0.0205;        0;        ok;
A0059;    A0053;    0.0075;    0.0074;    0.0114;        0;        ok;
A0059;    A0054;    0.0088;    0.0082;    0.0098;        0;        ok;
A0059;    A0055;    0.0116;    0.0073;    0.0127;        0;        ok;
A0059;    A0056;    0.0124;    0.0290;    0.0142;        0;        ok;
A0059;    A0057;    0.0087;    0.0321;    0.0096;        0;        ok;
A0059;    A0058;    0.0097;    0.0337;    0.0100;        0;        ok;
A0059;    A0059;    0.0529;    0.0531;    0.0941;        1;        ok;
A0059;    A0060;    0.0089;    0.0080;    0.0099;        0;        ok;
A0059;    A0062;    0.0092;    0.0085;    0.0105;        0;        ok;
A0059;    A0063;    0.0160;    0.0128;    0.0180;        0;        ok;
A0059;    A0064;    0.0118;    0.0132;    0.0136;        0;        ok;
A0059;    A0065;    0.0114;    0.0128;    0.0137;        0;        ok;
A0059;    A0070;    0.0124;    0.0710;    0.0142;        0;        ok;
A0059;    A0080;    0.0135;    0.0135;    0.0160;        0;        ok;
A0059;    A0082;    0.0193;    0.0194;    0.0195;        0;        ok;
A0059;    A0083;    0.0145;    0.0153;    0.0175;        0;        ok;
A0059;    A0086;    0.0229;    0.0228;    0.0247;        0;        ok;
A0059;    A0087;    0.0170;    0.0390;    0.0204;        0;        ok;
A0059;    A0088;    0.0176;    0.0219;    0.0206;        0;        ok;
A0059;    A0089;    0.0171;    0.0183;    0.0201;        0;        ok;
A0060;    A0053;    0.0111;    0.0179;    0.0126;        0;        ok;
A0060;    A0054;    0.0092;    0.0095;    0.0093;        0;        ok;
A0060;    A0055;    0.0083;    0.0240;    0.0124;        0;        ok;
A0060;    A0056;    0.0078;    0.0087;    0.0092;        0;        ok;
A0060;    A0057;    0.0079;    0.0141;    0.0144;        0;        ok;
A0060;    A0058;    0.0082;    0.0096;    0.0099;        0;        ok;
A0060;    A0059;    0.0082;    0.0087;    0.0099;        0;        ok;
A0060;    A0060;    0.0385;    0.0370;    0.1005;        1;        ok;
A0060;    A0062;    0.0370;    0.0151;    0.0255;        0;        ok;
A0060;    A0063;    0.0118;    0.0130;    0.0142;        0;        ok;
A0060;    A0064;    0.0124;    0.0133;    0.0140;        0;        ok;
A0060;    A0065;    0.0116;    0.0134;    0.0141;        0;        ok;
A0060;    A0070;    0.0126;    0.0622;    0.0149;        0;        ok;
A0060;    A0080;    0.0140;    0.0142;    0.0162;        0;        ok;
A0060;    A0082;    0.0142;    0.0150;    0.0159;        0;        ok;
A0060;    A0083;    0.0141;    0.0148;    0.0160;        0;        ok;
A0060;    A0086;    0.0222;    0.0224;    0.0250;        0;        ok;
A0060;    A0087;    0.0170;    0.0311;    0.0199;        0;        ok;
A0060;    A0088;    0.0170;    0.0206;    0.0202;        0;        ok;
A0060;    A0089;    0.0168;    0.0182;    0.0199;        0;        ok;
A0062;    A0053;    0.0077;    0.0159;    0.0087;        0;        ok;
```

```
A0062;    A0054;    0.0082;    0.0085;    0.0099;        0;        ok;
A0062;    A0055;    0.0088;    0.0173;    0.0098;        0;        ok;
A0062;    A0056;    0.0080;    0.0094;    0.0099;        0;        ok;
A0062;    A0057;    0.0103;    0.0099;    0.0104;        0;        ok;
A0062;    A0058;    0.0084;    0.0101;    0.0103;        0;        ok;
A0062;    A0059;    0.0087;    0.0091;    0.0124;        0;        ok;
A0062;    A0060;    0.0146;    0.0274;    0.0160;        0;        ok;
A0062;    A0062;    0.0387;    0.0377;    0.0542;        1;        ok;
A0062;    A0063;    0.0117;    0.0152;    0.0139;        0;        ok;
A0062;    A0064;    0.0118;    0.0190;    0.0144;        0;        ok;
A0062;    A0065;    0.0116;    0.0194;    0.0140;        0;        ok;
A0062;    A0070;    0.0124;    0.0495;    0.0150;        0;        ok;
A0062;    A0080;    0.0135;    0.0138;    0.0161;        0;        ok;
A0062;    A0082;    0.0140;    0.0144;    0.0163;        0;        ok;
A0062;    A0083;    0.0143;    0.0146;    0.0170;        0;        ok;
A0062;    A0086;    0.0220;    0.0227;    0.0250;        0;        ok;
A0062;    A0087;    0.0173;    0.0313;    0.0199;        0;        ok;
A0062;    A0088;    0.0175;    0.0205;    0.0203;        0;        ok;
A0062;    A0089;    0.0170;    0.0184;    0.0202;        0;        ok;
A0063;    A0053;    0.0114;    0.0107;    0.0124;        0;        ok;
A0063;    A0054;    0.0154;    0.0111;    0.0132;        0;        ok;
A0063;    A0055;    0.0119;    0.0106;    0.0126;        0;        ok;
A0063;    A0056;    0.0116;    0.0110;    0.0132;        0;        ok;
A0063;    A0057;    0.0125;    0.0113;    0.0132;        0;        ok;
A0063;    A0058;    0.0128;    0.0115;    0.0135;        0;        ok;
A0063;    A0059;    0.0130;    0.0115;    0.0135;        0;        ok;
A0063;    A0060;    0.0129;    0.0113;    0.0135;        0;        ok;
A0063;    A0062;    0.0153;    0.0117;    0.0139;        0;        ok;
A0063;    A0063;    0.3324;    0.3265;    0.3280;        1;        ok;
A0063;    A0064;    0.3676;    0.3699;    0.4377;        1;        ok;
A0063;    A0065;    0.3453;    0.3486;    0.4246;        1;        ok;
A0063;    A0070;    0.0170;    0.0163;    0.0196;        0;        ok;
A0063;    A0080;    0.1983;    0.0167;    0.0198;        0;        ok;
A0063;    A0082;    0.1682;    0.0172;    0.0203;        0;        ok;
A0063;    A0083;    0.1611;    0.0175;    0.0210;        0;        ok;
A0063;    A0086;    0.0268;    0.0262;    0.0294;        0;        ok;
A0063;    A0087;    0.0206;    0.0802;    0.0232;        0;        ok;
A0063;    A0088;    0.0214;    0.0259;    0.0233;        0;        ok;
A0063;    A0089;    0.0211;    0.0261;    0.0239;        0;        ok;
A0064;    A0053;    0.0113;    0.0108;    0.0126;        0;        ok;
A0064;    A0054;    0.0200;    0.0117;    0.0137;        0;        ok;
A0064;    A0055;    0.0123;    0.0113;    0.0137;        0;        ok;
A0064;    A0056;    0.0116;    0.0110;    0.0132;        0;        ok;
A0064;    A0057;    0.0160;    0.0113;    0.0170;        0;        ok;
A0064;    A0058;    0.0129;    0.0115;    0.0137;        0;        ok;
A0064;    A0059;    0.0160;    0.0115;    0.0161;        0;        ok;
A0064;    A0060;    0.0130;    0.0114;    0.0135;        0;        ok;
A0064;    A0062;    0.0201;    0.0124;    0.0138;        0;        ok;
```

```
A0064;    A0063;    0.3602;    0.3583;    0.3914;        1;        ok;
A0064;    A0064;    0.3378;    0.3546;    0.3497;        1;        ok;
A0064;    A0065;    0.3632;    0.3706;    0.5549;        1;        ok;
A0064;    A0070;    0.0176;    0.0169;    0.0197;        0;        ok;
A0064;    A0080;    0.2441;    0.0191;    0.0228;        0;        ok;
A0064;    A0082;    0.2061;    0.0188;    0.0220;        0;        ok;
A0064;    A0083;    0.2089;    0.0182;    0.0216;        0;        ok;
A0064;    A0086;    0.0293;    0.0289;    0.0326;        0;        ok;
A0064;    A0087;    0.0212;    0.0825;    0.0237;        0;        ok;
A0064;    A0088;    0.0213;    0.0266;    0.0236;        0;        ok;
A0064;    A0089;    0.0207;    0.0217;    0.0232;        0;        ok;
A0065;    A0053;    0.0109;    0.0102;    0.0122;        0;        ok;
A0065;    A0054;    0.0215;    0.0113;    0.0150;        0;        ok;
A0065;    A0055;    0.0121;    0.0105;    0.0125;        0;        ok;
A0065;    A0056;    0.0117;    0.0111;    0.0132;        0;        ok;
A0065;    A0057;    0.0136;    0.0116;    0.0143;        0;        ok;
A0065;    A0058;    0.0145;    0.0117;    0.0147;        0;        ok;
A0065;    A0059;    0.0136;    0.0170;    0.0193;        0;        ok;
A0065;    A0060;    0.0132;    0.0120;    0.0145;        0;        ok;
A0065;    A0062;    0.0215;    0.0122;    0.0149;        0;        ok;
A0065;    A0063;    0.3436;    0.3529;    0.3919;        1;        ok;
A0065;    A0064;    0.3627;    0.3611;    0.5094;        1;        ok;
A0065;    A0065;    0.3061;    0.3067;    0.3003;        1;        ok;
A0065;    A0070;    0.0168;    0.0165;    0.0195;        0;        ok;
A0065;    A0080;    0.1985;    0.0169;    0.0197;        0;        ok;
A0065;    A0082;    0.1872;    0.0173;    0.0201;        0;        ok;
A0065;    A0083;    0.1960;    0.0175;    0.0204;        0;        ok;
A0065;    A0086;    0.0269;    0.0257;    0.0299;        0;        ok;
A0065;    A0087;    0.0205;    0.0758;    0.0230;        0;        ok;
A0065;    A0088;    0.0208;    0.0266;    0.0233;        0;        ok;
A0065;    A0089;    0.0205;    0.0217;    0.0229;        0;        ok;
A0070;    A0053;    0.0116;    0.0109;    0.0130;        0;        ok;
A0070;    A0054;    0.0549;    0.0117;    0.0139;        0;        ok;
A0070;    A0055;    0.0454;    0.0112;    0.0135;        0;        ok;
A0070;    A0056;    0.0123;    0.0116;    0.0139;        0;        ok;
A0070;    A0057;    0.0593;    0.0119;    0.0149;        0;        ok;
A0070;    A0058;    0.0696;    0.0127;    0.0154;        0;        ok;
A0070;    A0059;    0.0737;    0.0121;    0.0144;        0;        ok;
A0070;    A0060;    0.0694;    0.0134;    0.0156;        0;        ok;
A0070;    A0062;    0.0502;    0.0123;    0.0147;        0;        ok;
A0070;    A0063;    0.0164;    0.0168;    0.0194;        0;        ok;
A0070;    A0064;    0.0165;    0.0171;    0.0196;        0;        ok;
A0070;    A0065;    0.0163;    0.0168;    0.0197;        0;        ok;
A0070;    A0070;    0.2748;    0.2749;    0.0926;        1;        ok;
A0070;    A0080;    0.0183;    0.0175;    0.0208;        0;        ok;
A0070;    A0082;    0.0190;    0.0179;    0.0211;        0;        ok;
A0070;    A0083;    0.0187;    0.0183;    0.0212;        0;        ok;
A0070;    A0086;    0.1434;    0.0264;    0.0295;        0;        ok;
```

```
A0070;    A0087;    0.0212;    0.0843;    0.0243;        0;        ok;
A0070;    A0088;    0.0216;    0.0276;    0.0247;        0;        ok;
A0070;    A0089;    0.0213;    0.0223;    0.0244;        0;        ok;
A0080;    A0053;    0.0125;    0.0119;    0.0141;        0;        ok;
A0080;    A0054;    0.0136;    0.0127;    0.0151;        0;        ok;
A0080;    A0055;    0.0140;    0.0121;    0.0145;        0;        ok;
A0080;    A0056;    0.0133;    0.0125;    0.0152;        0;        ok;
A0080;    A0057;    0.0134;    0.0128;    0.0153;        0;        ok;
A0080;    A0058;    0.0137;    0.0130;    0.0155;        0;        ok;
A0080;    A0059;    0.0137;    0.0131;    0.0155;        0;        ok;
A0080;    A0060;    0.0145;    0.0130;    0.0151;        0;        ok;
A0080;    A0062;    0.0142;    0.0132;    0.0158;        0;        ok;
A0080;    A0063;    0.0169;    0.1976;    0.0197;        0;        ok;
A0080;    A0064;    0.0169;    0.2149;    0.0199;        0;        ok;
A0080;    A0065;    0.0168;    0.2049;    0.0197;        0;        ok;
A0080;    A0070;    0.0175;    0.0180;    0.0204;        0;        ok;
A0080;    A0080;    0.2512;    0.2513;    0.4183;        1;        ok;
A0080;    A0082;    0.1337;    0.0203;    0.0239;        0;        ok;
A0080;    A0083;    0.1352;    0.0201;    0.0238;        0;        ok;
A0080;    A0086;    0.0288;    0.0274;    0.0304;        0;        ok;
A0080;    A0087;    0.0223;    0.0769;    0.0257;        0;        ok;
A0080;    A0088;    0.0225;    0.0271;    0.0259;        0;        ok;
A0080;    A0089;    0.0225;    0.0232;    0.0259;        0;        ok;
A0082;    A0053;    0.0133;    0.0127;    0.0146;        0;        ok;
A0082;    A0054;    0.0139;    0.0133;    0.0156;        0;        ok;
A0082;    A0055;    0.0145;    0.0127;    0.0148;        0;        ok;
A0082;    A0056;    0.0139;    0.0132;    0.0156;        0;        ok;
A0082;    A0057;    0.0140;    0.0134;    0.0197;        0;        ok;
A0082;    A0058;    0.0142;    0.0136;    0.0158;        0;        ok;
A0082;    A0059;    0.0143;    0.0137;    0.0161;        0;        ok;
A0082;    A0060;    0.0152;    0.0136;    0.0157;        0;        ok;
A0082;    A0062;    0.0149;    0.0138;    0.0161;        0;        ok;
A0082;    A0063;    0.0175;    0.1574;    0.0203;        0;        ok;
A0082;    A0064;    0.0175;    0.1941;    0.0203;        0;        ok;
A0082;    A0065;    0.0173;    0.1928;    0.0203;        0;        ok;
A0082;    A0070;    0.0182;    0.0185;    0.0209;        0;        ok;
A0082;    A0080;    0.0206;    0.1371;    0.0236;        0;        ok;
A0082;    A0082;    0.3289;    0.3383;    0.4709;        1;        ok;
A0082;    A0083;    0.3324;    0.3311;    0.4913;        1;        ok;
A0082;    A0086;    0.0295;    0.0282;    0.0311;        0;        ok;
A0082;    A0087;    0.0227;    0.4435;    0.0263;        0;        ok;
A0082;    A0088;    0.0230;    0.4018;    0.0269;        0;        ok;
A0082;    A0089;    0.0226;    0.3908;    0.0264;        0;        ok;
A0083;    A0053;    0.0131;    0.0124;    0.0150;        0;        ok;
A0083;    A0054;    0.0141;    0.0133;    0.0155;        0;        ok;
A0083;    A0055;    0.0142;    0.0129;    0.0150;        0;        ok;
A0083;    A0056;    0.0140;    0.0131;    0.0157;        0;        ok;
A0083;    A0057;    0.0144;    0.0135;    0.0157;        0;        ok;
```

```
A0083;    A0058;    0.0143;    0.0138;    0.0161;        0;        ok;
A0083;    A0059;    0.0143;    0.0137;    0.0161;        0;        ok;
A0083;    A0060;    0.0152;    0.0134;    0.0159;        0;        ok;
A0083;    A0062;    0.0149;    0.0137;    0.0164;        0;        ok;
A0083;    A0063;    0.0175;    0.1604;    0.0204;        0;        ok;
A0083;    A0064;    0.0176;    0.1943;    0.0205;        0;        ok;
A0083;    A0065;    0.0175;    0.1940;    0.0206;        0;        ok;
A0083;    A0070;    0.0183;    0.0184;    0.0212;        0;        ok;
A0083;    A0080;    0.0204;    0.1373;    0.0238;        0;        ok;
A0083;    A0082;    0.3247;    0.3291;    0.4896;        1;        ok;
A0083;    A0083;    0.3222;    0.3203;    0.4868;        1;        ok;
A0083;    A0086;    0.0292;    0.0281;    0.0315;        0;        ok;
A0083;    A0087;    0.0229;    0.4617;    0.0266;        0;        ok;
A0083;    A0088;    0.0231;    0.3946;    0.0266;        0;        ok;
A0083;    A0089;    0.0229;    0.3968;    0.0263;        0;        ok;
A0086;    A0053;    0.0217;    0.0205;    0.0235;        0;        ok;
A0086;    A0054;    0.0227;    0.0214;    0.0240;        0;        ok;
A0086;    A0055;    0.0223;    0.0208;    0.0233;        0;        ok;
A0086;    A0056;    0.0224;    0.0211;    0.0242;        0;        ok;
A0086;    A0057;    0.0226;    0.0215;    0.0241;        0;        ok;
A0086;    A0058;    0.0229;    0.0216;    0.0247;        0;        ok;
A0086;    A0059;    0.0229;    0.0215;    0.0246;        0;        ok;
A0086;    A0060;    0.0229;    0.0214;    0.0242;        0;        ok;
A0086;    A0062;    0.0232;    0.0220;    0.0246;        0;        ok;
A0086;    A0063;    0.0265;    0.0269;    0.0318;        0;        ok;
A0086;    A0064;    0.0259;    0.0264;    0.0292;        0;        ok;
A0086;    A0065;    0.0261;    0.0268;    0.0290;        0;        ok;
A0086;    A0070;    0.0267;    0.1409;    0.0295;        0;        ok;
A0086;    A0080;    0.0280;    0.0278;    0.0306;        0;        ok;
A0086;    A0082;    0.0286;    0.0290;    0.0318;        0;        ok;
A0086;    A0083;    0.0287;    0.0291;    0.0317;        0;        ok;
A0086;    A0086;    0.5891;    0.5861;    0.8778;        1;        ok;
A0086;    A0087;    0.0314;    0.1074;    0.0349;        0;        ok;
A0086;    A0088;    0.0321;    0.0361;    0.0358;        0;        ok;
A0086;    A0089;    0.0315;    0.3926;    0.0350;        0;        ok;
A0087;    A0053;    0.0266;    0.0155;    0.0183;        0;        ok;
A0087;    A0054;    0.0338;    0.0164;    0.0191;        0;        ok;
A0087;    A0055;    0.0306;    0.0158;    0.0189;        0;        ok;
A0087;    A0056;    0.0355;    0.0162;    0.0190;        0;        ok;
A0087;    A0057;    0.0366;    0.0165;    0.0193;        0;        ok;
A0087;    A0058;    0.0364;    0.0166;    0.0194;        0;        ok;
A0087;    A0059;    0.0390;    0.0166;    0.0195;        0;        ok;
A0087;    A0060;    0.0309;    0.0166;    0.0194;        0;        ok;
A0087;    A0062;    0.0318;    0.0171;    0.0199;        0;        ok;
A0087;    A0063;    0.0807;    0.0204;    0.0229;        0;        ok;
A0087;    A0064;    0.0789;    0.0205;    0.0231;        0;        ok;
A0087;    A0065;    0.0762;    0.0205;    0.0228;        0;        ok;
A0087;    A0070;    0.0811;    0.0210;    0.0242;        0;        ok;
```

```
A0087;    A0080;    0.0808;    0.0222;    0.0257;        0;        ok;
A0087;    A0082;    0.4332;    0.0229;    0.0264;        0;        ok;
A0087;    A0083;    0.4485;    0.0250;    0.0267;        0;        ok;
A0087;    A0086;    0.0992;    0.0311;    0.0346;        0;        ok;
A0087;    A0087;    1.1866;    1.1893;    0.4841;        1;        ok;
A0087;    A0088;    1.0377;    1.0436;    0.4811;        1;        ok;
A0087;    A0089;    0.1508;    0.5903;    0.0305;        0;        ok;
A0088;    A0053;    0.0191;    0.0158;    0.0186;        0;        ok;
A0088;    A0054;    0.0210;    0.0166;    0.0195;        0;        ok;
A0088;    A0055;    0.0195;    0.0163;    0.0190;        0;        ok;
A0088;    A0056;    0.0207;    0.0166;    0.0193;        0;        ok;
A0088;    A0057;    0.0210;    0.0170;    0.0200;        0;        ok;
A0088;    A0058;    0.0216;    0.0169;    0.0199;        0;        ok;
A0088;    A0059;    0.0216;    0.0170;    0.0222;        0;        ok;
A0088;    A0060;    0.0209;    0.0168;    0.0199;        0;        ok;
A0088;    A0062;    0.0208;    0.0171;    0.0203;        0;        ok;
A0088;    A0063;    0.0261;    0.0207;    0.0234;        0;        ok;
A0088;    A0064;    0.0262;    0.0209;    0.0234;        0;        ok;
A0088;    A0065;    0.0263;    0.0208;    0.0231;        0;        ok;
A0088;    A0070;    0.0275;    0.0213;    0.0246;        0;        ok;
A0088;    A0080;    0.0285;    0.0222;    0.0262;        0;        ok;
A0088;    A0082;    0.3869;    0.0229;    0.0265;        0;        ok;
A0088;    A0083;    0.3798;    0.0231;    0.0266;        0;        ok;
A0088;    A0086;    0.0377;    0.0315;    0.0353;        0;        ok;
A0088;    A0087;    1.0177;    1.0250;    0.4818;        1;        ok;
A0088;    A0088;    0.8987;    0.8955;    0.5169;        1;        ok;
A0088;    A0089;    0.0332;    0.4683;    0.0311;        0;        ok;
A0089;    A0053;    0.0173;    0.0155;    0.0184;        0;        ok;
A0089;    A0054;    0.0185;    0.0164;    0.0193;        0;        ok;
A0089;    A0055;    0.0176;    0.0157;    0.0187;        0;        ok;
A0089;    A0056;    0.0181;    0.0161;    0.0191;        0;        ok;
A0089;    A0057;    0.0181;    0.0165;    0.0192;        0;        ok;
A0089;    A0058;    0.0184;    0.0167;    0.0196;        0;        ok;
A0089;    A0059;    0.0185;    0.0165;    0.0195;        0;        ok;
A0089;    A0060;    0.0184;    0.0168;    0.0195;        0;        ok;
A0089;    A0062;    0.0188;    0.0168;    0.0200;        0;        ok;
A0089;    A0063;    0.0218;    0.0204;    0.0231;        0;        ok;
A0089;    A0064;    0.0219;    0.0203;    0.0230;        0;        ok;
A0089;    A0065;    0.0218;    0.0258;    0.0228;        0;        ok;
A0089;    A0070;    0.0224;    0.0209;    0.0245;        0;        ok;
A0089;    A0080;    0.0236;    0.0222;    0.0262;        0;        ok;
A0089;    A0082;    0.3906;    0.0225;    0.0262;        0;        ok;
A0089;    A0083;    0.3958;    0.0226;    0.0265;        0;        ok;
A0089;    A0086;    0.4022;    0.0312;    0.0348;        0;        ok;
A0089;    A0087;    0.5784;    0.1565;    0.0304;        0;        ok;
A0089;    A0088;    0.4613;    0.0320;    0.0308;        0;        ok;
A0089;    A0089;    0.9123;    0.9117;    0.4907;        1;        ok;
```