Efficient Algorithms for Tree Automata

Ondřej Valeš*

Abstract
Tree automata and their languages find use in the field of formal verification and theorem proving but for many practical applications performance of existing algorithms for tree automata manipulation is unsatisfactory. In this work a novel algorithm for testing language equivalence and inclusion on tree automata is proposed and implemented as a module of the VATA library with a goal of creating algorithm that is comparatively faster than existing methods on at least a portion of real–world examples. First, existing approaches to equivalence and inclusion testing on both word and tree automata are examined. These existing approaches are then modified to create the *bisimulation up–to congruence* algorithm for tree automata. Efficiency of this new approach is compared with existing tree automata language equivalence and inclusion testing methods.

Keywords: finite automata — tree automata — language equivalence — language inclusion — bisimulation — antichains — bisimulation up–to congruence

Supplementary Material: Downloadable Code

*xvales03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Finite tree automata can be viewed as an extension of commonly used finite word automata that allow processing of languages with entities that do not have a strictly linear structure. These automata and languages they describe can be used in the field of verification to locate bugs in safety critical systems, theorem proving, database systems, and language manipulation based on XML schema [1, 2].

For instance, regular model checking of programs 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 solutions to WSkS formulae, which can be exploited to decide problems stated in terms of WSkS using language operations on tree automata [1, 3].

For the purposes of verification and theorem proving, performance of existing algorithms for manipulation of nondeterministic finite tree automata can be deemed unsatisfactory for many real–world problems. Because checking language equivalence and inclusion on nondeterministic tree automata has an exponential–time worst–case lower bound, there is a strong incentive to develop efficient algorithms that perform this task in as limited time as possible, trying to avoid, offset or limit an exponential growth even for larger automata [4].

Being a fundamentally exponential–time problem, it might not be possible to develop an algorithm that will perform satisfactorily on every possible automaton. The goal is rather to develop an algorithm that will outperform existing approaches on at least a subset of the tree automata that stem from real–world applications.

1.1 Previous work

This paper builds on previous work done in [5] where the first prototype of *bisimulation up–to congruence* for VATA library was presented. In this paper the *bisimulation up–to congruence* prototype is developed into a fully functioning algorithm, several optimizations are implemented, and its efficiency is compared with state–of–the–art approaches.

1.2 Goal

The goal of this paper is to modify *bisimulation up–to congruence* [6], an existing algorithm designed for testing language equivalence and inclusion on finite word automata for the use with finite tree automata.

Its main idea is based on optimizing the classical procedure for testing language equivalence, which is based on first determinizing both automata 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 ratio growing with automata size [6].

However, *bisimulation up–to congruence* has so far only been used with (and its soundness proven for) finite word automata. Substantial changes for both the algorithm and proof resulting from different structures used by word and tree automata are necessary to successfully adapt this approach for finite tree automata.

The main components of this paper therefore are: implementation of the modified *bisimulation up–to congruence* algorithm as an extension of the VATA library [7] and comparison with existing equivalence and inclusion testing approaches for tree automata already implemented in the VATA library, namely algorithms based on the so–called antichains [8, 9, 10].

2. Tree Automata

Finite tree automata are tools for describing languages containing entities with the tree structure. They are similar to finite word automata in both their components and operation but they require an extended alphabet, called ranked alphabet, to allow symbols to have more than one successor.

Tree automata can be divided based on determinism and a direction in which trees are processed. Processing can start with the root symbol and progress in the top-down direction expanding from the root symbol and assigning states to nodes as they are processed until all leaf nodes are reached. The alternative is to start with leaf nodes progressing in the bottom-up direction eventually reaching the root node [1].

In contrast to word automata where the processing direction largely does not matter, bottom-up deterministic tree automata are strictly stronger then their top-down counterparts [1]. This difference stems from the tree structure, where when traversing a tree 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 parent nodes must determine the states of all children nodes.

2.1 Ranked Alphabet

A ranked alphabet Σ_{arity} is a finite set of symbols together with a ranking function $rank : \Sigma \to \mathbb{N}$. Ranking function determines arity of each symbol (number of successor symbols).

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

$$\Sigma_{arity} = \{+, -, \times, \div, n\}$$

$$\begin{aligned} rank: \quad & + \mapsto 2, \quad - \mapsto 2, \quad n \mapsto 0, \\ & \times \mapsto 2, \quad \div \mapsto 2, \end{aligned}$$

where all mathematical 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.

2.2 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 meaning $\forall u, v \in \mathbb{N}^* : uv \in Pos(t) \implies u \in Pos(t)$ and
- $\forall p \in Pos(t) : a = t(p) \implies \{j \in \mathbb{N} \mid pj \in Pos(t)\} = \{1, \ldots, rank(a)\}$.

2.3 Bottom-up Tree Automata

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

$$\mathscr{A} = (Q, \Delta, \Sigma_{arity}, F),$$

where

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

We use $(p_1, \ldots, p_n) \xrightarrow{a} P$ to denote that $((p_1, \ldots, p_n), a, P) \in \Delta$.

2.3.1 Determinism

Deterministic tree automata are a special class of nondeterministic tree automata, where

$$\forall (p_1, \ldots, p_n) \xrightarrow{a} P \in \Delta : |P| \leq 1.$$

2.4 Run

Let t be a tree and $\mathscr{A} = (Q, \Delta, \Sigma_{arity}, F)$ be a NFTA. A run of $\mathscr{A}$ over a tree t is a mapping $r_t : Pos(t) \to Q$ consistent with Δ, meaning $\forall p \in Pos(t) : r_t(p) = q \wedge 1 \le i \le rank(t(p)) : r_t(pi) = q_i \Leftrightarrow (q_1, \dots, q_n) \xrightarrow{a} q$. Run r_t is called accepting if $r(\varepsilon) \in F$.

2.5 Language

Language accepted by a NFTA $\mathscr{A} = (Q, \Delta, \Sigma_{arity}, F)$, denoted $L(\mathscr{A})$, is the set of all trees for which there exist an accepting run r_t compatible with Δ, formally $L(\mathscr{A}) = \{t \mid \exists r_t : r_t \text{ is a run of } \mathscr{A} \text{ on } t \wedge r_t(\varepsilon) \in F\}$.

2.5.1 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.

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.

3. Existing methods

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. Because every minimal tree automaton is unique (up to isomorphism), equivalence can be determined by directly comparing minimized versions of input automata [11, 1].

Using determinization can be complicated by the state space explosion [1]. 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 counterexample using only portion of the automata, the determinization and minimization will build the whole minimized automata first and check equivalence later, leading to poor performance on nonequivalent automata pairs.

An alternative is to use *on–the–fly* determinization. Techniques in this category do not work with automata as whole but rather with their macrostate (a set of states of the original automaton) pairs [10]. These algorithms start with pairs of macrostates that imply language equivalence (pairs of states of leaf nodes for tree automata) and use successor pairs generated during the run of algorithm. 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 [10, 8, 9]. The most basic *on–the–fly* subset construction technique is successor generation while constructing bisimulation relation.

These techniques can further be aided by tools that prune the searched macrostate pair space. The antichain approach, defined in [9], uses the identity relation, which implies language inclusion, to prune the search space [10, 9]. Simulation–based approaches 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) [10]. And, finally, the antichain approach can be combined with simulation to obtain a combined method [10].

4. Bisimulation up–to congruence for tree automata

Bisimulation up–to congruence is an extension of the regular language equivalence checking by relating automata states with a bisimulation relation [12] and using *on–the–fly* determinization. The input automata can be nondeterministic and determinization is done during the algorithm run only for macrostates that are encountered during the successor pairs generation. It is based on Hopcroft and Karp's algorithm [13], which builds a maximal bisimulation relating initial states of both automata [6, 13].

If all pairs in this relation are final state equivalent (meaning either both or neither states are final), then the first automaton can simulate any transition in the other one always ending in a final state if the first does and vice versa, therefore guarantying language equivalence of input automata.

Definition 1 (Bisimulation) *Bisimulation on tree automata* $\mathscr{A}_1 = (Q_1, \Delta_1, \Sigma, F_1)$ *and* $\mathscr{A}_2 = (Q_2, \Delta_2, \Sigma, F_2)$ *is a relation* $R \subseteq 2^{Q_1} \times 2^{Q_2}$*, such that if* $\forall a \in \Sigma : n = rank(a) \wedge \forall 1 \le i \le n : (X_i, Y_i) \in R$ *then*

1. $\forall 1 \le i \le n : X_i \cap F_1 = \emptyset \Leftrightarrow Y_i \cap F_2 = \emptyset$ *and*
2. $(\Delta_1(X_1, \dots, X_i, a), \Delta_2(Y_1, \dots, Y_i, a)) \in R$.

This definition of bisimulation is based on the definition found in [6] and deviates from the standard definition of bisimulation for tree automata found in [12].

Lemma 1 *Languages of two tree automata are equal if there exist a bisimulation relating them.*

```
    input  : $\mathcal{N} = (Q_N, \Delta_N, \Sigma_{arity}, F_N), \mathcal{M} = (Q_M, \Delta_M, \Sigma_{arity}, F_M)$
    output : $true$ iff $L(\mathcal{N}) = L(\mathcal{M})$, otherwise $false$
 1  todo $\leftarrow \{(\Delta_N(a), \Delta_M(a)) \mid a \in \Sigma_{arity}\}$
 2  done $\leftarrow \emptyset$
 3  while $todo \neq \emptyset$ do
 4      actual $\leftarrow (X, Y) \in$ todo
 5      done $\leftarrow$ done $\cup$ {actual}
 6      if $(X \cap F_N = \emptyset \not\Leftrightarrow Y \cap F_M = \emptyset)$ then
 7          return $false$
 8      end
 9      foreach $a \in \Sigma$ do
10          todo $\leftarrow$ (todo $\cup$ post(a, actual, done)) $\setminus$ c(done)
11      end
12  end
13  return $true$
```

Algorithm 1: Bisimulation up–to congruence for tree automata.

```
Function post(a: symbol, (X, Y): macrostate pair, done: set of pairs) : set of pairs is
    return
     $\{(\Delta_N(P_1, \ldots, X, \ldots P_m, a), \Delta_M(Q_1, \ldots, Y, \ldots Q_m, a)) \mid m = rank(a) - 1 \wedge 1 \leq i \leq m)(P_i, Q_i) \in done\}$
end
```

In order to improve performance of language equivalence checking, it is desirable to reduce the search space as much as possible. This is accomplished by using *bisimulation up–to congruence* (Algorithm 1), an approach that only adds a new macrostate pair into *todo* if it is not in the congruence closure of *done*, denoted c(done), thus removing all new pairs that cannot contribute to finding counterexample from further search.

Definition 2 (Bisimulation up–to) [6] *Given tree automata* $\mathcal{A}_1 = (Q_1, \Delta_1, \Sigma, F_1)$ *and* $\mathcal{A}_2 = (Q_2, \Delta_2, \Sigma, F_2)$ *and a function* $f : 2^{2^{Q_1} \times 2^{Q_2}} \rightarrow 2^{2^{Q_1} \times 2^{Q_2}}$ *a relation* R *on macrostates is a bisimulation up–to* f *if whenever* $\forall a \in \Sigma : n = rank(a) \wedge \forall 1 \leq i \leq n : (X_i, Y_i) \in R$ *then*

1. $\forall 1 \leq i \leq n : X_i \cap F_1 = \emptyset \Leftrightarrow Y_i \cap F_2 = \emptyset$ *and*
2. $(\Delta_1(X_1, \ldots, X_i, a), \Delta_2(Y_1, \ldots, Y_i, a)) \in f(R)$.

4.1 Congruence Closure

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

$$\frac{X\ R\ Y}{X\ c(R)\ Y}, \quad \frac{}{X\ c(R)\ X}, \quad \frac{X\ c(R)\ Y}{Y\ c(R)\ X},$$

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

Lemma 2 *Relation built in* `done` *over the run of Algorithm 1 is a bisimulation up–to congruence if line 13 is reached, otherwise a counterexample was found and* `actual` *holds a state pair that is reachable over the same tree in both automata and only one of the states is accepting.*

5. Implementation

Algorithm 1 for language equivalence checking was implemented as an extension of the C/C++ VATA library[1] [7]. It uses iterative calculation of successor macrostate pairs (Function *post*) and congruence closure membership checking using fixpoint calculation described in [13], which is based on applying inference rules described in Section 4.1.

6. Experiments

Bisimulation up–to congruence was compared with the bottom–up algorithm for inclusion checking based on antichains implemented in the VATA library [7]. Neither algorithm was combined with the simulation approach. 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

[1] https://github.com/Iorethan/libvata-congruence

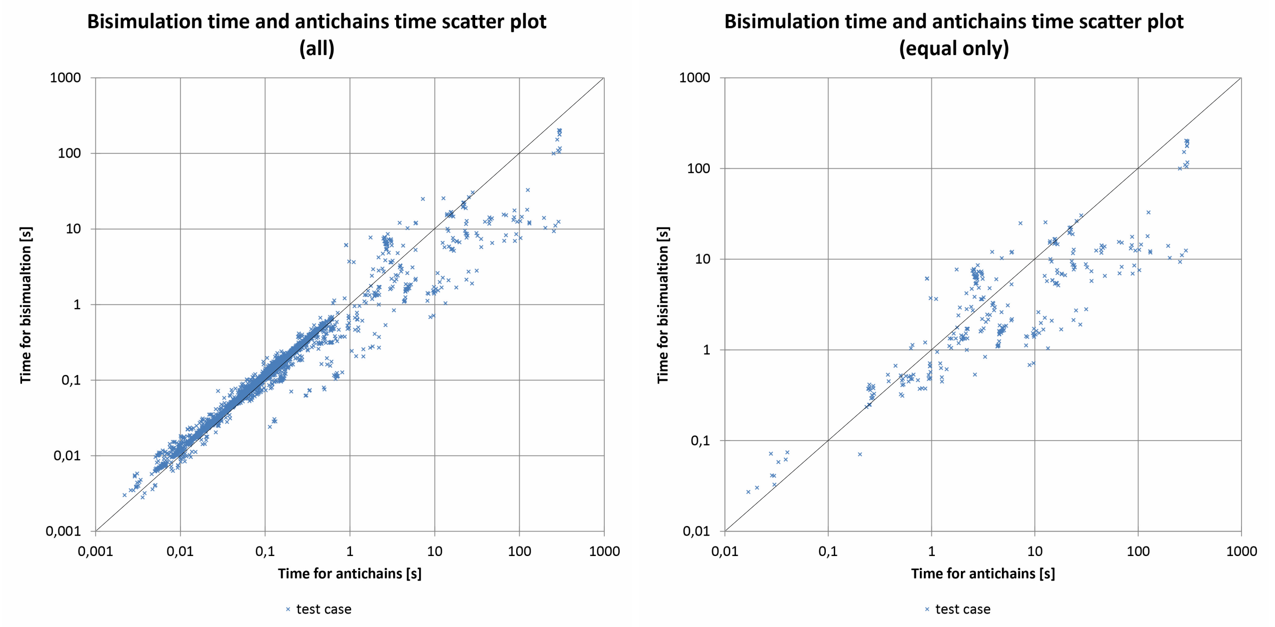

Figure 1. Scatter plot of the times needed by the bisimulation up–to congruence and the algorithm based on antichains to check equivalence.

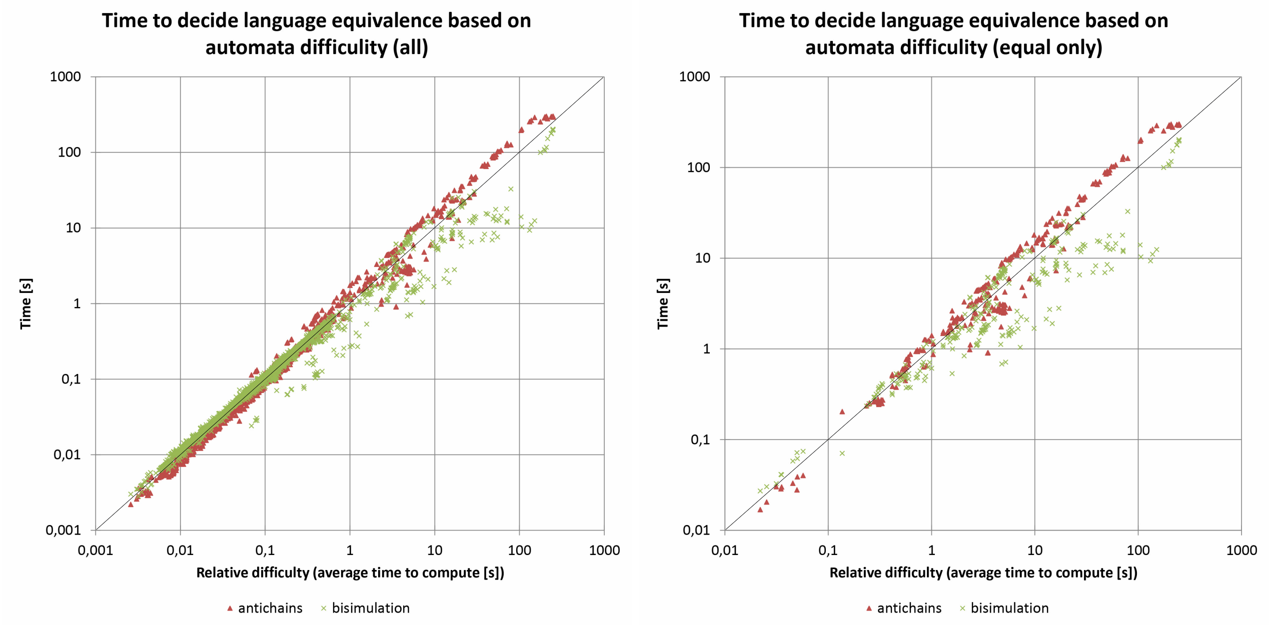

Figure 2. Comparison of the time needed to decide language equivalence for antichain based algorithm and bisimulation up–to congruence. Relative difficulty is average time needed by both algorithms to decide the given problem.

Table 1. Required time [s] (all results)

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 2. Required time [s] (valid equivalences only)

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

the other (artificially inflating the time two inclusion checks need to find a counterexample), 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 in total 95 automata 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 non–trivial) and 8426 were invalid equivalences.

Percentile times for both algorithms can be seen in Table 1 for all comparisons and in Table 2 for valid equivalences only. For problems that could be decided relatively quickly (most of the invalid equivalences fall into this category) 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.

Scatter plots of the times both algorithms took to check equivalence are in Figure 1 (all cases and valid equivalences only). Another comparison of time required to check equivalence, this time based on relative difficulty of individual test cases, is in Figure 2 (all cases and valid equivalences only).

7. Conclusion

In this paper, *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 main goal of this paper to develop an algorithm for language inclusion and equivalence testing on tree automata and outperform existing approaches on real-world examples was accomplished.

7.1 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 bottom–up approach (for both *bisimulation up–to congruence* and antichains 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.

Results of the comparison done in this paper are not corresponding to those in [14]. 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.

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 paper.

Secondly, I would also like to thank Mgr. Lukáš Holík, Ph.D. and Bc. Petr Žufan for their pioneer work on *bisimulation up–to congruence* for tree automata.

References

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

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

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

[4] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[5] P. Žufan. Inkluze jazyků nedeterministických stromových automatů. In *Proc. of Excel@FIT '16*, 2016.

[6] F. Bonchi and D. Pous. Checking nfa equivalence with bisimulations up to congruence. In *Proc. of POPL '13*, pages 457–468, New York, NY, USA, 2013. ACM.

[7] O. Lengál, J. Šimáček, and T. Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS '12*, pages 79–94, Berlin, Heidelberg, 2012. Springer.

[8] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV '06*, pages 17–30, Berlin, Heidelberg, 2006. Springer.

[9] L. Holík, O. Lengál, J .Šimáček, and T. Vojnar. Efficient inclusion checking on explicit and semi-symbolic tree automata. In *Proc. of ATVA '11*, pages 243–258, Berlin, Heidelberg, 2011. Springer.

[10] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. of TACAS '10*, pages 158–174, Berlin, Heidelberg, 2010. Springer.

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

[12] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Composed bisimulation for tree automata. *Int. J. Found. Comput. Sci.*, 20(4):685–700, 2009.

[13] J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical report, Dept. of Computer Science, Cornell U, December 1971.

[14] Ch. Fu, Y. Deng, D. N. Jansen, and L. Zhang. On equivalence checking of nondeterministic finite automata. In *Dependable Software Engineering. Theories, Tools, and Applications*, pages 216–231, Cham, 2017. Springer International Publishing.