Fast equivalence-checking for normed context-free processes *

Wojciech Czerwiński and Sławomir Lasota

Institute of Informatics, University of Warsaw
Banacha 2, Warsaw, Poland
wczerwin,sl@mimuw.edu.pl

Abstract

Bisimulation equivalence is decidable in polynomial time over normed graphs generated by a context-free grammar. We present a new algorithm, working in time $\mathcal{O}(n^5)$, thus improving the previously known complexity $\mathcal{O}(n^8\text{polylog}(n))$. It also improves the previously known complexity $\mathcal{O}(n^6\text{polylog}(n))$ of the equality problem for simple grammars.

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2010.1

1 Introduction

Equivalence checking, that is determining whether two systems are equal under a given notion of equivalence, is an important verification problem with a long history. In this paper we consider systems described by context-free grammars. It is well known that language equivalence is undecidable in this class [1]. A decidability result was obtained by Korenjak and Hopcroft [12] for a restricted class of deterministic context-free grammars (*simple grammars*). Remarkably, the language containment is undecidable even for simple grammars [7].

In the context of process algebras, a grammar may be considered as a description of a transition graph rather than a language. The adequate concept of equivalence is then *bisimilarity* (bisimulation equivalence), a notion strictly finer than language equivalence. For graphs generated by context-free grammars, called *context-free processes*, bisimilarity is known to be decidable due to the result of [5]. It has also been demonstrated that bisimilarity is the only equivalence in van Glabbeek's spectrum [8] which is decidable for context-free processes. This places bisimilarity in a very favourable position.

Historically the first decision procedure for bisimilarity on infinite-state systems was given by [3] for a class of *normed* context-free processes, those defined by grammars in which, roughly, each nonterminal generates at least one word. Clearly, language equivalence is still undecidable in this class, as normedness assumption does not facilitate testing language equality. As language equivalence and bisimilarity coincide on deterministic graphs the result of [3] was a strict extension of [12]. Later, decidability was extended to all context-free processes [5].

In the normed case, a series of consecutive papers [4, 10, 11] lead finally to a remarkable polynomial-time algorithm of [9] for bisimilarity. The working time $\mathcal{O}(n^{13})$ was however not satisfactory and hence further results followed: an $\mathcal{O}(n^7 \mathit{polylog}\ n)$-time algorithm was proposed for equivalence of simple grammars [2] and an $\mathcal{O}(n^8 \mathit{polylog}\ n)$-time algorithm was given for bisimilarity [13]. The latter cut down to simple grammars works in time $\mathcal{O}(n^6 \mathit{polylog}\ n)$. We report a further progress: we give an $\mathcal{O}(n^5)$ time algorithm for bisimi-

* This paper is partially supported by Polish government grant no. N N206 356036 and by EU-FET-IP *Software Engineering for Service-Oriented Overlay Computers* SENSORIA.

IARCS Annual International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010.
Editors: Kamal Lodaya, Meena Mahajan; pp. 1–12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

larity on normed context-free processes, thus improving the previously known bound by a factor of $\mathcal{O}(n^3)$. We improve the case of simple grammars just as well.

Our approach is substantially different from all recent algorithms which, roughly, compute the base of the bisimulation equivalence by eliminating the incorrect decompositions from the initial 'overapproximating' base. Instead, we apply the refinement scheme used previously for the *commutative* context-free processes. Our starting point is the algorithm, given in [6], that works for both normed commutative and non-commutative processes. Intuitively, the algorithm combines the algorithmic theory of compressed strings, useful for non-commutative case, with the iterative approximation refinement scheme used in commutative case. In this paper we demonstrate, roughly speaking, that the latter scheme may be implemented efficiently for non-commutative processes in time $\mathcal{O}(n^5)$.

We start by defining the problem in Section 1. The following section introduces a necessary background material, namely pattern-matching in compressed strings and unique decomposition in a finitely-generated monoid, and explains the refinement scheme exploited in our algorithm. Then, in Sections 3 and 4 we present the algorithm itself. Section 3 provides a general outline and intentionally omits a number of details; in Section 4 we provide all the implementation details missing in Section 3.

Context-Free Processes and Bisimilarity.

Ingredients of a *process definition* Δ are a finite alphabet Σ, a finite set $\mathtt{V}$ of variables, and a finite set of rules

$$X \xrightarrow{a} \alpha, \tag{1}$$

with $a \in \Sigma$ and $\alpha \in \mathtt{V}^*$. Such process definitions are usually called in the literature *Basic Process Algebra*, or *Context-Free Processes*. The explanation of the latter is that each rule can be seen as a production $X \longrightarrow a\alpha$ of a context-free grammar in Greibach normal form. Elements of $\mathtt{V}^*$ are called here *processes*; a variable X can be seen as an elementary process.

Δ defines a transition system: its states are processes $\alpha \in \mathtt{V}^*$; and for each $a \in \Sigma$, there is a transition relation containing triples (α, a, β), where $a \in \Sigma$ and $\alpha, \beta \in \mathtt{V}^*$, written $\alpha \xrightarrow{a} \beta$. The transition relations are defined by a prefix rewriting: $X\beta \xrightarrow{a} \alpha\beta$ whenever Δ contains a rule $X \xrightarrow{a} \alpha$, and $\beta \in \mathtt{V}^*$.

▶ **Definition 1.** Given a binary relation R over $\mathtt{V}^*$, we say that a pair (α, β) of processes satisfies expansion in R, written $(\alpha, \beta) \in \exp(R)$, if

- whenever $\alpha \xrightarrow{a} \alpha'$, there exists some β' with $\beta \xrightarrow{a} \beta'$ and $(\alpha', \beta') \in R$; and
- whenever $\beta \xrightarrow{a} \beta'$, there exists some α' with $\alpha \xrightarrow{a} \alpha'$ and $(\alpha', \beta') \in R$.

A binary relation S satisfies expansion in R if each pair $(\alpha, \beta) \in S$ does, i.e., $S \subseteq \exp(R)$. A relation R is a *bisimulation* if it satisfies expansion in itself. We say that α and β are *bisimilar*, denoted by $\alpha \sim \beta$, if (α, β) belongs to some bisimulation.

From now on we assume that Δ is *normed*, i.e., for each variable $X \in \mathtt{V}^*$ there is a finite sequence $X \xrightarrow{a_1} \alpha_1 \ldots \xrightarrow{a_k} \alpha_k = \varepsilon$ leading from X to the empty process ε. By $|X|$ denote the smallest length of such sequence and call it the *norm* of X (intuitively, $|X|$ is the length of the shortest word generated from X).

We consider the following Normed-BPA-Bisim Problem:

Instance: A normed process definition Δ and two variables $X, Y \in \mathtt{V}$.
Question: Is $X \sim Y$?

A more general problem of checking whether $\alpha \sim \beta$, for given $\alpha, \beta \in \mathtt{V}^*$, can be easily reduced to the above one. The size of Δ, denoted by N, is the sum of lengths of all the rules in Δ. Clearly $n \leq N$. Our main result is the following:

▶ **Theorem 2.** Normed-BPA-Bisim Problem *can be solved in time* $\mathcal{O}(N^5)$.

Thinking of Δ as of a grammar, call Δ a *simple grammar* if for each X and a, there is at most one rule (1) in Δ. As a direct corollary of Theorem 2 we obtain:

▶ **Corollary 3.** *Equivalence of simple grammars can be solved in time* $\mathcal{O}(N^5)$.

Our algorithm, similarly to previous ones [9, 13], builds a finite *base* of $\sim$ that, once constructed, allows to answer the Normed-BPA-Bisim Problem in constant time.

2 Preliminaries

Acyclic morphisms.

Let $\mathtt{A}$ be a finite set of terminal symbols, ranged over by $a, b, \ldots$, and let $\mathtt{S}$ be a finite set of non-terminal symbols, ranged over by $x, y, z, \ldots$. Assume a total ordering $<$ of non-terminal symbols. An *acyclic morphism* is a mapping $h : \mathtt{S} \to (\mathtt{S} \cup \mathtt{A})^*$ such that all symbols appearing in $h(x)$ are strictly smaller than x wrt. $<$. We implicitely extend the domain of h to $\mathtt{S} \cup \mathtt{A}$, assuming h to be identity on $\mathtt{A}$. Due to the acyclicity requirement, h induces a monoid morphism $h^* : (\mathtt{S} \cup \mathtt{A})^* \to \mathtt{A}^*$, as the limit of compositions $h, h^2 = h \circ h, \ldots$. Formally, $h^*(z) = h^k(z)$, for the smallest k with $h^k(z) \in \mathtt{A}^*$. Then the extension of h^* to all strings in $(\mathtt{S} \cup \mathtt{A})^*$ is as usual. Therefore each symbol $z \in \mathtt{S}$ *represents* a nonempty string $h^*(z)$ over A. Its length $\|h^*(z)\|$ may be exponentially larger than the size of h, written $\mathtt{size}(h)$, defined as the sum of lengths of all strings $h(z)$, $z \in \mathtt{S}$.

A relevant parameter of a symbol z, wrt. an acyclic morphism h, is its *depth*, written $\mathtt{depth}_h(z)$, and defined as the longest path in the derivation tree of z. A depth of h, written $\mathtt{depth}(h)$, is the greatest depth of a symbol.

An acyclic morphism h is *binary* if $\|h(z_i)\| = 2$, for all $z_i \in \mathtt{S}$. Any acyclic morphism h may be transformed to the equivalent binary one: replace each $h(z_i)$ of length greater than 2 with a balanced binary tree, using $\|h(z_i)\| - 2$ auxiliary symbols. Note that in consequence the depth of h may increase by a logarithmic factor, but the size of h may only increase by a constant factor. In the sequel we only consider binary morphisms.

In a word of length n we distinguish $n + 1$ *positions* $0 \ldots n$. If $h(z) = xy$, i.e., $h^*(z) = h^*(x)\, h^*(y)$, then by *the cutting position* in z we mean the position in $h^*(z)$ equal to the length of $h^*(x)$. We say that a substring *touches* a given position if this position is either inside this substring or on the border. The *occurrence table of* h stores, for each two symbols $x, y \in \mathtt{S}$, the set of starting positions of occurrences of $h^*(x)$ in $h^*(y)$ that touch the cutting position in y. The whole table may be stored compactly in $\mathcal{O}(|\mathtt{S}|^2)$ memory due to the following:

▶ **Lemma 4.** *(Basic Lemma [15]) The set of starting positions of occurences of* $h^*(x)$ *in* $h^*(y)$ *that touch the cutting position in* y*, if non-empty, is an arithmetic progression.*

▶ **Theorem 5.** *([14]) Given a binary acyclic morphism* h*, one may compute in time* $\mathcal{O}(\mathtt{size}(h)^2 \cdot \mathtt{depth}(h))$ *the occurence table of* h.

Generalized acyclic morphisms.

From now on assume that each terminal symbol $a \in \mathtt{A}$ has assigned its norm $|a|$, a positive integer. Norm extends additively to all strings from $\mathtt{A}^*$. For non-terminal symbols, we put $|z| := |h^*(z)|$.

▶ **Lemma 6.** *Given a binary acyclic morphism h, a symbol $z \in \mathtt{S}$, and $k < \|h^*(z)\|$, one may compute in time $\mathcal{O}(\mathtt{depth}_h(z))$ an acyclic morphism h' extending h, such that one of new symbols of h' represents the suffix of $h^*(z)$ of norm k (assumed that such exists), and $\mathtt{size}(h') \leq \mathtt{size}(h) + \mathcal{O}(\mathtt{depth}_h(z))$ and $\mathtt{depth}(h') = \mathtt{depth}(h)$.*

For efficiency reasons it is favourable *not* to compute explicitely the representation of the suffix of $h^*(z)$ of norm k. Instead, we will represent this suffix symbolically, as follows. By now, an acyclic morphism was defined by equations of the form $h(z) = x\,y$, meaning $h^*(z) = h^*(x)\,h^*(y)$. Now we will allow a more general form:

$$h(z) = x\,\mathrm{suffix}_k(y), \tag{2}$$

where $0 < k \leq |y|$, to mean that $h^*(z)$ is concatenation of $h^*(x)$ and the suffix of $h^*(y)$ of norm k. Note that (2) is well defined only when the required suffix exists. As a particular case, for $k = |y|$, one gets the standard definition $h(z) = x\,y$. As before we asume acyclicity in (2), i.e., $x, y < z$.

Theorem 5 may be adapted to the generalized acyclic morphisms, with a slightly worst time. Naively, one could get rid of all 'truncated' variables y in (2) using Lemma 6, ending with a quadratic blow-up of the size of the morphism, and then apply Theorem 5. We claim that one can do better:

▶ **Theorem 7.** *Given a generalized binary acyclic morphism h, one may compute in time $\mathcal{O}(\mathtt{size}(h)^3 \cdot \mathtt{depth}(h))$ the occurence table of h.*

From now on generalized acyclic morphisms are briefly called acyclic morphisms.

Unique decomposition.

Assume from now on a fixed normed process definition Δ, i.e., a finite alphabet Σ, a finite set $\mathtt{V} = \{X_1, \ldots, X_n\}$ of variables, and a finite set of rules of the form $X_i \xrightarrow{a} \alpha$, $a \in \Sigma, \alpha \in \mathtt{V}^*$. The complexity considerations in this section and later on are wrt. the size N of Δ.

Assume also that variables $\mathtt{V} = \{X_1, \ldots, X_n\}$ of a process definition are ordered according to non-decreasing norm: $|X_i| \leq |X_j|$ whenever $i < j$. We write $X_i < X_j$ if $i < j$. Note that $|X_1|$ is necessarily 1, and that norm of a variable is at most exponential wrt. the size of Δ, understood as the sum of lengths of all rules.

A congruence is *norm-preserving* if whenever α and β are related then $|\alpha| = |\beta|$. Let $\equiv$ be an arbitrary norm-preserving congruence in $\mathtt{V}^*$. Intuitively, an elementary process X_i is *decomposable* if $X_i \equiv \alpha\beta$ for some $\alpha, \beta \neq \epsilon$. Note that $|\alpha|$, $|\beta| < |X_i|$ then. For technical convenience we prefer to apply a slightly different definition. We say that X_i is *decomposable* wrt. $\equiv$, if $X_i \equiv \alpha$ for some process $\alpha \in \{X_1, \ldots, X_{i-1}\}^*$; otherwise, X_i is called *indecomposable*, or *prime* wrt. $\equiv$. In particular, X_1 is always prime.

Denote by $\mathtt{P}$ the set of primes wrt. $\equiv$. It is easy to show by induction on norm that for each process α there is some $\gamma \in \mathtt{P}^*$ with $\alpha \equiv \gamma$; in such case γ is called a *prime decomposition* of α. Note that a prime decomposition of X_i is either X_i itself, or it belongs to $\{X_1, \ldots, X_{i-1}\}^*$. We say that $\equiv$ has the *unique decomposition property* if each process has precisely one prime

decomposition. While the set $\mathtt{P}$ of primes depends on the chosen ordering of variables (in case $X_i \equiv X_j$, $i \neq j$), the unique decomposition property does not.

The following lemma is shown by considering the unique prime decompositions:

▶ **Lemma 8** (Left cancellation). *If $\equiv$ has the unique decomposition property and $\gamma\alpha \equiv \gamma\beta$ then $\alpha \equiv \beta$.*

The refinement step.

A transition $\alpha \xrightarrow{a} \beta$ is called *norm-reducing* if $|\beta| < |\alpha|$; we write $\alpha \xrightarrow{a}_{\text{n-r}} \beta$ in such case. We will need a concept of norm-reducing bisimulation (n-r-bisimulation, in short), i.e., a bisimulation over the transition system restricted to only norm-reducing transitions. The appropriate norm-reducing expansion wrt. R (cf. Def. 1) will be written as n-r-exp(R). Every bisimulation is a n-r-bisimulation (as a norm-reducing transition must be matched in a bisimulation by a norm-reducing one) but the converse does not hold in general.

▶ **Proposition 1.** *Each n-r-bisimulation, and hence each bisimulation, is norm-preserving.*

For a norm-preserving equivalence $\equiv$ over processes, let $\sim^{\equiv}_{\text{n-r}}$ denote the union of all n-r-bisimulations contained in $\equiv$. It witnesses most of typical properties of bisimulation equivalence. Being the union of n-r-bisimulations, $\sim^{\equiv}_{\text{n-r}}$ is a n-r-bisimulation itself, in fact the greatest n-r-bisimulation that is contained in $\equiv$. It admits the following fix-point characterization:

▶ **Proposition 2.** $(\alpha, \beta) \in \sim^{\equiv}_{n\text{-}r}$ *iff* $\alpha \equiv \beta$ *and* $(\alpha, \beta) \in$ *n-r-exp*$(\sim^{\equiv}_{n\text{-}r})$.

Moreover $\sim^{\equiv}_{\text{n-r}}$ is clearly an equivalence as $\equiv$ is assumed to be so. The relation $\sim^{\equiv}_{\text{n-r}}$ may be thus seen as the bisimulation equivalence relativized to pairs of processes related by $\equiv$ and to norm-reducing moves only.

The relativized bisimulation equivalence will play a crucial role in the algorithm, that will work by consecutive refinements of a current congruence until it finally stabilizes. Instead of the classical refinement step $\equiv \mapsto \equiv \cap \exp(\equiv)$, we prefer to use the following one:

$$\equiv \;\mapsto\; \sim^{\equiv \cap \exp(\equiv)}_{\text{n-r}} .$$

This transformation will be refered to as *the refinement step*, and $\sim^{\equiv \cap \exp(\equiv)}_{\text{n-r}}$ will be called *the refinement of* $\equiv$.

By the results of [6] specialized to normed BPA, it follows:

▶ **Lemma 9.** *([6]) If a norm-preserving congruence $\equiv$ has the unique decomposition property then the refinement of $\equiv$ is a congruence with the unique decomposition property.*

3 Outline of the algorithm

Overall idea.

We describe the algorithm in a top-down manner, introducing the implementation details incrementally. The overall idea is as follows: we start with the initial congruence $\equiv$, given simply by the norm equality (cf. Prop. 1), and then perform the fixpoint computation by refining $\equiv$ until it finally stabilizes:

Initialize $\equiv$ as the norm equality.
REPEAT
    replace $\equiv$ by its refinement
UNTIL $\equiv$ coincides with its refinement.

Note that the approximating congruence $\equiv$ always subsumes bisimulation equivalence $\sim$ in the course of the algorithm, $\sim \subseteq \equiv$. Moreover, if $\equiv$ and its refinement coincide, then $\equiv \subseteq \exp(\equiv)$ and thus the opposite implication $\equiv \subseteq \sim$ follows. Thus the approximation scheme is correct wrt. the bisimulation equivalence. At the end of this section we will argue that the algorithm always terminates after at most n iterations.

Now we will outline a way of implementing the scheme above.

Representation by an acyclic morphism.

Clearly, instead of the whole (infinite!) congruence $\equiv$, the algorithm should maintain a finite representation of $\equiv$. As a succint representation we choose an acyclic morphism $h : \mathtt{S} \to (\mathtt{S} \cup \mathtt{A})^*$. The set $\mathtt{A}$ of terminal symbols will consist of all variables X_i that are currently prime wrt. $\equiv$, and the set of non-terminal symbols $\mathtt{S}$ will contain the variables currently decomposable wrt. $\equiv$, together with some other auxiliary symbols, to be defined later on. We assume that the ordering $<$ on $\mathtt{S}$ is consistent with the ordering $X_1 < \ldots < X_n$. For any variable X_i, $h^*(X_i) \in \mathtt{A}^* \subseteq \mathtt{V}^*$ will describe the prime decomposition of X_i. The morphism h will represent $\equiv$ in the following sense: $\alpha \equiv \beta \iff h^*(\alpha) = h^*(\beta)$. Below we prefer to write $=_h$ instead of $\equiv$ to emphesize the role of h.

Recall that due to Lemma 9 the congruence $\equiv$ invariantly has the unique decomposition property, hence always some h exists that represents $\equiv$ (e.g., take as $h(X_i)$ the prime decomposition of X_i). Note however that the same congruence may be represented by many different acyclic morphisms. A key ingredient of the algorithm will be an efficient construction of a sufficiently succint one.

As the congruence $\equiv$ is norm-preserving, norm of X_i is always equal to norm of $h(X_i)$, and hence to norm of $h^*(X_i)$ as well.

Leftmost prime factors.

If $X_i \equiv X_j\gamma$, $j < i$ and X_j is prime wrt. $\equiv$ we say that X_j is the *leftmost prime factor* of X_i wrt. $\equiv$. Note that due to the unique decomposition property of $\equiv$, X_i may have at most one leftmost prime factor wrt. $\equiv$. Moreover, Lemma 8 guarantees that if $X_i \equiv X_j\alpha$ and $X_i \equiv X_j\beta$ are two decompositions of X_i, starting with the same variable X_j, then $\alpha \equiv \beta$ and thus $\alpha \equiv \beta$ is determined uniquely up to $\equiv$. In consequence, it is crucial just to know, for each variable X_i, which variable X_j, $j < i$, if any, is the leftmost prime factor of X_i wrt. the current congruence $\equiv$. In the algorithm, this information will be maintained using the indices $\mathbf{lpf}(i) \in \{1 \ldots n-1\}$, for $i \in \{2 \ldots n\}$, with the following meaning: if the variable X_i is currently decomposable wrt. $\equiv$, then $X_{\mathbf{lpf}(i)}$ is the leftmost prime factor of X_i. Clearly

$$\mathbf{lpf}(i) < i. \tag{3}$$

As $\equiv$ is represented by an acyclic morphism h, $X_{\mathbf{lpf}(i)}$ always belongs to $\mathtt{A}$ and is the first letter in $h^*(X_i)$ (and the first letter in $h(X_i)$ as well, which will become apparent shortly).

Outline of the algorithm.

FOR $i \in \{2 \dots n\}$ DO let $\mathbf{lpf}(i) := 1$;
$\mathtt{A} := \{X_1\}$; $\mathtt{A}' := \mathtt{A}$;
initialize h;

REPEAT
  FOR $X_i \in \{X_2 \dots X_n\} \setminus \mathtt{A}$ DO (∗)
    IF $\neg$ **lpfactor**$(X_i, X_{\mathbf{lpf}(i)})$ THEN (∗∗)
      FOR $X_j \in \{X_{\mathbf{lpf}(i)+1} \dots X_{i-1}\} \cap (\mathtt{A}' \setminus \mathtt{A})$ DO (∗ ∗ ∗)
        IF **lpfactor**(X_i, X_j) THEN
          let $\mathbf{lpf}(i) := j$;
          ($h'(X_i)$ is defined as a side-effect of **lpfactor**(X_i, X_j))
          break the inner for loop;
        FI
      IF the inner for loop not broken THEN add X_i to $\mathtt{A}'$ FI
    FI
  $\mathtt{A} := \mathtt{A}'$; $h := h'$;
UNTIL A does not change

In the FOR loops (∗) and (∗∗∗), the variables are assumed to be inspected in the increasing order $X_1 \leq \ldots \leq X_n$.

Each iteration of the REPEAT loop, as outlined above, corresponds to a single refinement step of $\equiv$: given the current acyclic morphism h it computes a new acyclic morphism, say h', representing the refinement of $=_h$ (the latter has unique decomposition property by Lemma 9). The subroutine **lpfactor**(X_i, X_j), invoked several times in the algorithm, is to check whether X_j is the leftmost prime factor of X_i wrt. the refinement of $=_h$. The acyclic morphism h' is computed as a side-effect of the invocations of **lpfactor**; thus **lpfactor**(X_i, X_j) is invoked under assumption that the value of h' is already known for the variables from $\{X_1 \dots X_{i-1}\}$. Description of this subroutine and other implementation details related to the computation of h' are deferred to the next section. Here, we merely focus on the scheme of updating the indices $\mathbf{lpf}(i)$.

In the inner FOR loop (∗ ∗ ∗), the variable X_j ranges over $\{X_{\mathbf{lpf}(i)+1} \dots X_{i-1}\} \cap (\mathtt{A}' \setminus \mathtt{A})$. The rationale behind restricting this range so is the following.

As the refinement of $=_h$ is clearly finer than $=_h$ (we may write $=_{h'} \subseteq =_h$), a decomposition wrt. the refinement of $=_h$ is automatically a decomposition wrt. $=_h$. Thus, if some X_i is decomposable wrt. $=_h$, then there are just two possibilities for a new value of $\mathbf{lpf}(i)$ (denote it by $\mathbf{lpf}(i)'$): either it remains unchanged, $\mathbf{lpf}(i)' = \mathbf{lpf}(i)$ (if the **lpfactor** (∗∗) succeeds), or it changes. In the latter case, its new value $\mathbf{lpf}(i)'$ may be only chosen from $\mathtt{A}' \setminus \mathtt{A}$, as otherwise X_i would have two different leftmost prime factors wrt. (the coarser) $=_h$. Moreover, the new value of $\mathbf{lpf}(i)$ must be necessarily bigger than the old one. This follows from the observation that the 'fresh' prime variable $X_{\mathbf{lpf}(i)'} \in \mathtt{A}' \setminus \mathtt{A}$ was not so in the previous iteration, and again by the unique decomposition property its leftmost prime factor was necessarily the same as the previous leftmost prime factor of X_i:

$$\mathbf{lpf}(\mathbf{lpf}(i)') = \mathbf{lpf}(i). \tag{4}$$

Thus $\mathbf{lpf}(i) < \mathbf{lpf}(i)'$ by (3).

The size of the set $\mathtt{A}$ of terminal symbols (containing exactly the prime variables wrt. $=_h$) increases in each iteration of the REPEAT loop (except for the very last iteration). Thus the total number of iterations is at most n.

Concerning the correctness: if $\mathtt{A}$ does not change in one iteration of the REPEAT loop, it clearly follows that $=_h$ coincides with its refinement, which guarantees that $=_h$ reaches the bisimulation equivalence, as discussed in the beginning of this section.

4 Implementation of the algorithm

Now we explain how the acyclic morphism h is initialized and refined during one iteration of the REPEAT loop; and how the subroutine **lpfactor** is implemented. Total running time is discussed at the end of this section.

Initialization of h.

For any right-hand side β of a production in Δ we introduce a non-terminal symbol Y_β, and define $h(Y_\beta) = \beta$. Then we transform h into binary form, if necesssary. This will possibly introduce further auxiliary symbols; in the sequel these further symbols will not be mentioned explicitly. The symbols Y_β (together with the other auxiliary symbols) will be continuously contained in $\mathtt{S}$ during the algorithm and their definition will never change. The number of the symbols is $\mathcal{O}(N)$.

Distinguish one fixed *norm-reducing* transition rule

$$X_i \xrightarrow{a_i}_{\text{n-r}} \alpha_i \tag{5}$$

for every variable X_i, $2 \le i \le n$; clearly $|X_i| = |\alpha_i| + 1$. These distinguished rules will be fixed in the algorithm. For $i \ge 2$, we initilize h by

$$h(X_i) = X_1\, Y_{\alpha_i}, \tag{6}$$

which gives a decomposition of X_i wrt. the norm equality, i.e., $h^*(X_i) = X_1^{|X_i|}$.

Initially, $\mathtt{A} = \{X_1\}$ and $\mathtt{S} = \{Y_\beta\}_\beta \cup \{X_2 \dots X_n\}$.

Invariant.

In the algorithm, the acyclic morphism h is determined by the leftmost prime factors and by the distinguished norm-reducing rules (5). Recall that $X_{\mathbf{lpf}(i)}$ is the leftmost prime factor of X_i wrt. $=_h$. The following invariant will be respected by h after each iteration of the REPEAT loop (note that (6) is a special case when $\mathbf{lpf}(i) = 1$):

$$h(X_i) = X_{\mathbf{lpf}(i)}\, \mathrm{suffix}_{|X_i|-|X_{\mathbf{lpf}(i)}|}(Y_{\alpha_i}) \quad \text{for every} \quad X_i \notin \mathtt{A}. \tag{7}$$

Why is (7) correct? It follows from the claim below.

▶ **Claim 1.** *Let h be an acyclic morphism such that the congruence $=_h$ is a norm-reducing bisimulation. If $X_i =_h X_j\delta$ then $h^*(\delta)$ is a suffix of $h^*(\alpha_i)$.*

Indeed: consider a norm-reducing transition $X_i \xrightarrow{a_i} \alpha_i$ and a matching transition of $X_j\delta$, say $X_j\delta \xrightarrow{a_i} \gamma\delta$; as X_j is an active variable, the process δ stays unchanged, and thus $\alpha_i =_h \gamma\delta$ as required.

From h to h'.

As the refinement of $=_h$ is included in $=_h$, a decomposition wrt. the refinement of $=_h$ is automatically a decomposition wrt. $=_h$. Thus a variable prime wrt. $=_h$ is still prime wrt.$=_{h'}$: $\mathtt{A} \subseteq \mathtt{A}'$. We do not introduce any new symbols for h': $\mathtt{A} \cup \mathtt{S} = \mathtt{A}' \cup \mathtt{S}'$, and hence $\mathtt{S} \supseteq \mathtt{S}'$. Thus $\mathtt{S}' = \{Y_\beta\}_\beta \cup \{X_2 \dots X_n\} \setminus \mathtt{A}'$.

We assume that the occurrence table for h is available prior to each succesive iteration of the REPEAT loop (it was constructed in the previous iteration). As a side-effect of the consecutive invocations of **lpfactor** during one iteration of the REPEAT loop, the algorithm computes incrementally the occurrence table for h'. We assume that prior to each invocation of **lpfactor**(X_i, X_j) the occurrence table for h' is already computed for variables $\{X_1 \dots X_{i-1}\}$.

Implementation of lpfactor(X_i, X_j).

The subroutine **lpfactor** is given, as its input, two variables X_i, X_j. Its task is to check whether X_j is the leftmost prime factor of X_i wrt. $\sim_{\text{n-r}}^{=_h \cap \exp(=_h)}$, the refinement of $=_h$.

It is assumed that prior to the call of **lpfactor**(X_i, X_j), all variables $X_2, \dots, X_{i-1}$ have been processed by the outer FOR loop and either qualified to $\mathtt{A}'$, or have already a definition in h'. We thus assume that on $\{X_1 \dots X_{i-1}\}$ the congruence $=_{h'}$ represented by h' agrees with $\sim_{\text{n-r}}^{=_h \cap \exp(=_h)}$. In particular $(h')^*(\alpha_i)$ is already defined, as all variables appearing in α_i are in $\{X_1 \dots X_{i-1}\}$. It is also assumed that $X_i \notin \mathtt{A}'$ and $X_j \in \mathtt{A}'$. Recalling (7), the aim of **lpfactor**(X_i, X_j) is to check whether substituting **lpf**$(i) = j$ in (7) is correct for the refinement of h. I.e., the aim is to check the 'candidate':

$$h'(X_i) = X_j \,\mathrm{suffix}_{|X_i|-|X_j|}(Y_{\alpha_i}). \tag{8}$$

The right-hand side of (8) is meaningful only when $(h')^*(Y_{\alpha_i}) = (h')^*(\alpha_i)$ has a suffix of norm $|X_i| - |X_j|$, say $\bar\alpha \in (\mathtt{A}')^*$; this is verified in point 1 below. The 'candidate' (8) is acceptable if the following condition holds:

$$(X_i,\ X_j\,\bar\alpha) \quad \in \quad \sim_{\text{n-r}}^{=_h \cap \exp(=_h)}, \tag{9}$$

By referring directly to Prop. 2, one sees that (9) is equivalent to

$$X_i =_h X_j\bar\alpha \quad \text{and} \quad (X_i, X_j\,\bar\alpha) \in \exp(=_h) \quad \text{and} \quad (X_i, X_j\,\bar\alpha) \in \text{n-r-exp}(=_{h'}).$$

(In the last condition we use the assumption that $=_{h'}$ agrees with $\sim_{\text{n-r}}^{=_h \cap \exp(=_h)}$ on variables from $\{X_1 \dots X_{i-1}\}$.) These three conditions are verified in points 2, 3 and 4 below.

subroutine **lpfactor**$(X_i,\ X_j)$*:*

1. Check if $(h')^*(Y'_{\alpha_i})$ has a suffix of norm $l := |X_i| - |X_j|$. If no, return **false**. As $\mathtt{A} \subseteq \mathtt{A}'$ we conclude that $h^*(Y_{\alpha_i})$ has a suffix of norm l too, the fact to be needed in the following points.

2. Test if

$$X_i \ =_h \ X_j \operatorname{suffix}_l(Y_{\alpha_i}) \qquad (\text{in } \mathtt{A}^*). \tag{10}$$

If this is not the case, return **false**.

3. For each $a \in \Sigma$, let $C_a := \{\alpha : X_i \xrightarrow{a} \alpha\}$ and $D_a := \{\beta : X_j \xrightarrow{a} \beta\}$; then for all $\alpha \in C_a$ and $\beta \in D_a$, test if

$$Y_\alpha \ =_h \ Y_\beta \operatorname{suffix}_l(Y_{\alpha_i}) \qquad (\text{in } \mathtt{A}^*). \tag{11}$$

If the bisimulation expansion condition (for each $\alpha \in C_a$, there exists $\beta \in D_a$ such that (11) holds; and for each $\beta \in D_a$, there exists $\alpha \in C_a$ such that (11) holds) is not satisfied return **false**.

4. For each $a \in \Sigma$, let $C_a := \{\alpha : X_i \xrightarrow{a}_{\text{n-r}} \alpha\}$ and $D_a := \{\beta : X_j \xrightarrow{a}_{\text{n-r}} \beta\}$; then for all $\alpha \in C_a$ and $\beta \in D_a$, test if

$$Y_\alpha \ =_{h'} \ Y_\beta \operatorname{suffix}_l(Y_{\alpha_i}) \qquad (\text{in } (\mathtt{A}')^*). \tag{12}$$

If the bisimulation expansion condition, with (12) in place of (11), is not satisfied return **false**.

5. Extend h' by $h'(X_i) = X_j \operatorname{suffix}_{|X_i|-|X_j|}(Y_{\alpha_i})$ and return **true**.

In case when the 'candidate' (9) is checked succesfully, as a side-effect of the invocation of **lpfactor** the acyclic morphism h' is extended in point 5.

It remains now to explain how the equality tests (10), (11) and (12) are implemented. Basing on the the same insight as in Theorem 7 we prove:

▶ **Lemma 10.** *Each of equality tests* (10), (11) *may be solved in time* $\mathcal{O}(N)$*. All equality tests* (12) *during one iteration of the* REPEAT *loop require* $\mathcal{O}(N^4)$ *time.*

Total running time.

The time needed for a single iteration of the REPEAT loop is devoted to two tasks: (I) construction of the occurrence table for h', as a side-effect of solving tests (12), and (II) solving the equality tests (10) and (11). Task (I) requires $\mathcal{O}(N^4)$ total time, by Lemma 10 and Theorem 7, knowing that the depth of h is at most n (note that the depth is so even after increasing by a logarithmic factor when transforming h to the binary form). Concerning (II), there are $\mathcal{O}(N^2)$ equality tests in a single iteration of the REPEAT loop and each of them requires $\mathcal{O}(N)$ time by Lemma 10, hence the latter task is time-dominated by the former one. As the number of iterations is at most n, we get total time $\mathcal{O}(N^5)$, as stated in Theorem 2.

Example.

As an example, we analyze a run of our algorithm for the following input process definition:

$$X \xrightarrow{a} \epsilon \qquad Y \xrightarrow{a} \epsilon \qquad Y \xrightarrow{b} Y \qquad Y \xrightarrow{b} X \qquad Z \xrightarrow{a} X$$
$$Z \xrightarrow{b} Z \qquad Z \xrightarrow{b} YY \qquad T \xrightarrow{a} YY \qquad W \xrightarrow{a} ZZ \qquad W \xrightarrow{b} W$$

Variables are ordered as follows: $X < Y < Z < T < W$. The compression issue, i.e., representation of the approximating congruences by acyclic morphisms, are completely omitted here for simplicity. We fix:

$$\alpha_X = \epsilon \qquad 1\alpha_Y = \epsilon \qquad \alpha_Z = X \qquad \alpha_T = YY \qquad \alpha_W = ZZ.$$

The initial decomposition is: $Y \equiv X$, $Z \equiv XX$, $T \equiv XXX$, $W \equiv XXXXX$; variable X is the only prime.

Let us analyze the first iteration of the REPEAT loop; the refinement of $\equiv$ computed in this iteration we denote by $\equiv'$. We check that **lpfactor**(Y, X) yields **false**, because $(Y, X) \notin \exp(\equiv)$: Y has a b-move, X has not. So Y becomes prime. Next we check that **lpfactor**(Z, X) yields **false** because $(Z, XX = X\alpha_Z) \notin \exp(\equiv)$. We check that **lpfactor**(Z, Y) yields **true**, so we put $Z \equiv' YX = Y\alpha_Z$. Next we check that **lpfactor**(T, X) yields **true**, hence $T \equiv' XYY = X\alpha_T$. Now we proceed with processing of the last variable W. We check that **lpfactor**(W, X) yields **false**, because $(W, XYXYX \equiv' X\alpha_W) \notin \exp(\equiv)$: W has a b-move, X has not. Finally, we check that **lpfactor**(W, Y) yields **true**, so now $W \equiv' Y\alpha_W \equiv' YZZ \equiv' YYXYX$.

After the first iteration, the decomposition is: $Z \equiv YX$, $T \equiv XYY$, $W \equiv YYXYX$; the primes are $\mathtt{A} = \{X, Y\}$.

Now we proceed with the analysis of the second iteration of the REPEAT loop. We check that **lpfactor**(Z, Y) yields **false**, because $(Z, YX) \notin \exp(\equiv)$: there is no good Duplicator's response to the move $Z \xrightarrow{b} YY$. In consequence, Z becomes prime. Next we check that **lpfactor**(T, X) yields **true**. In consequence, $T \equiv' XYY$. Finally, we process variable W. We check that **lpfactor**(W, Y) yields **false**, as $(W, YZZ) \notin \exp(\equiv)$: there is no good response to the move $Y \xrightarrow{b} X$. The next candidate for the left-most prime factor is Z, a 'fresh' prime. We check that **lpfactor**(W, Z) yields **false**, but the reason is different than before: $\alpha_W = ZZ$ has no suffix of norm $|W| - |Z| = 3$.

After the second iteration, we have four primes $\mathtt{A} = \{X, Y, Z, W\}$ and the decomposition is $T \equiv XYY$.

The third iteration is the last one as $\mathtt{A}$ does not change any more.

Acknowledgements.

We are very grateful to Wojtek Rytter and Sibylle Fröschle for many helpful discussions.

References

1 Y. Bar-Hillel, M. Perles, and S. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift fuer Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, 14:143–177, 1961.

2 C. Bastien, J. Czyżowicz, W. Fraczak, and W. Rytter. Prime normal form and equivalence of simple grammars. In *Proc. CIAA'05*, volume 3845 of *LNCS*, pages 79–90. Springer-Verlag, 2005.

3 J. Beaten, J. Bergstra, and J. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proc. PARLE'87*, volume 259 of *LNCS*, pages 94–113. Springer-Verlag, 1987.

4 D. Caucal. Graphes canoniques des graphes algébraiques. *Informatique Théoretique et Applications (RAIRO)*, 24(4):339–352, 1990.

5 S. Christensen, Y. Hirshfeld, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 12(2):143–148, 1995.

6 W. Czerwiński, S. Fröschle, and S. Lasota. Partially-commutative context-free processes. In *Proc. CONCUR'09*, volume 5710 of *LNCS*, pages 259–273. Springer-Verlag, 2009.

7 E.P. Friedman. The inclusion problem for simple languages. *Theoretical Computer Science*, 1:297–316, 1976.

8 R. v. Glabbeek. The linear time - branching time spectrum. In *Proc. CONCUR'90*, pages 278–297, 1990.

9 Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity on normed context-free processes. *Theoretical Computer Science*, 15:143–159, 1996.

10 H. Huettel and C. Stirling. Actions speak louder than words: proving bisimilarity for context-free processes. In *Proc. LICS'91*, pages 376–386. IEEE Computer Society Press, 1991.

11 D. Huynh and L. Tian. Deciding bisimilarity of normed context-free processes is in Σ_2^P. *Theoretical Computer Science*, 123:183–197, 1994.

12 A. Korenjak and J. Hopcroft. Simple deterministic languages. In *Proc. 7th Annual IEEE Symposium on Switching and Automata Theory*, pages 36–46, 1966.

13 S. Lasota and W. Rytter. Faster algorithm for bisimulation equivalence of normed context-free processes. In *Proc. MFCS'06*, volume 4162 of *LNCS*, pages 646–657. Springer-Verlag, 2006.

14 Y. Lifshits. Solving classical string problems an compressed texts. In *Combinatorial and Algorithmic Foundations of Pattern and Association Discovery*, 2006.

15 A. Shinohara, M. Miyazaki, and M. Takeda. An improved pattern-matching algorithm for strings in terms of straight-line programs. *Journal of Discrete Algorithms*, 1(1):187–204, 2000.