Bisimulation equivalence of a BPP and a finite-state system can be decided in polynomial time

Martin Kot [1], Zdeněk Sawa [2,3]

Department of Computer Science, FEI, Technical University of Ostrava
17. listopadu 15, 70833 Ostrava-Poruba, Czech Republic

Abstract

In this paper we consider the problem of deciding bisimulation equivalence of a BPP and a finite-state system. We show that the problem can be solved in polynomial time and we present an algorithm deciding the problem in time $O(n^4)$. The algorithm also constructs for each state of the finite-state system a 'symbolic' semilinear representation of the set of all states of the BPP system which are bisimilar with this state.

Key words: bisimulation equivalence, basic parallel processes, finite-state processes

1 Introduction

Bisimulation equivalence also called bisimilarity is one of the most important behavioral equivalences studied in the area of automatic verification. Basic parallel processes (BPP) are one type of infinite state systems on which deciding bisimilarity was studied. The problem of deciding bisimilarity on BPP was shown to be decidable in [2], but no complexity bounds were presented there. It was proven in [8] that the problem is *PSPACE*-hard and Jančar has recently shown in [4] that the problem is in *PSPACE*, so the problem is *PSPACE*-complete. Polynomial time algorithms are known for the case of normed BPP [3,6].

In this paper we present an algorithm for the special case of the problem where one of the (unnormed) processes is a finite-state process. The running time of the algorithm is $O(n^4)$ where n is the size of the instance. The result

[1] e-mail: martin.kot@vsb.cz

[2] e-mail: zdenek.sawa@vsb.cz

[3] Both authors supported by the Grant Agency of the Czech Republic, No. 201/03/1161

implies that it is possible to verify in polynomial time whether a system implemented as a finite-state automaton is equivalent to a 'specification' given as a BPP. The algorithm also generates for each state of the finite-state system a 'symbolic' semilinear representation of bisimilar BPP states.

The paper is organized as follows: In Section 2 we provide some basic definitions and formulate the main result of the paper. In Section 3 we describe the main ideas of the algorithm and show its correctness. In Section 4 we analyze the time complexity of the algorithm. Section 5 contains conclusion and directions of the future work.

2 Basic Definitions

A *labelled transition system* (LTS) is a tuple $(S, \mathcal{A}, \longrightarrow)$ where S is a (possibly infinite) set of *states*, $\mathcal{A}$ is a finite set of *actions*, and $\longrightarrow \subseteq S \times \mathcal{A} \times S$ is a *transition relation*. We write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \longrightarrow$. We extend this notation also to finite sequences of actions and for $w \in \mathcal{A}^*$ we write $s \xrightarrow{w} s'$ if $w = a_1 a_2 \ldots a_n$ and there are some states $s_0, s_1, \ldots, s_n$ such that $s = s_0$, $s' = s_n$ and $s_{i-1} \xrightarrow{a_i} s_i$ for each $0 < i \leq n$.

A binary relation $\mathcal{R}$ on S (of an LTS) is a *bisimulation* iff for each $(s_1, s_2) \in \mathcal{R}$ the following conditions hold:

- if $s_1 \xrightarrow{a} s'_1$ for some $a \in \mathcal{A}$ and $s'_1 \in S$ then there is $s'_2 \in S$ such that $s_2 \xrightarrow{a} s'_2$ and $(s'_1, s'_2) \in \mathcal{R}$, and
- if $s_2 \xrightarrow{a} s'_2$ for some $a \in \mathcal{A}$ and $s'_2 \in S$ then there is $s'_1 \in S$ such that $s_1 \xrightarrow{a} s'_1$ and $(s'_1, s'_2) \in \mathcal{R}$.

Union of bisimulations is also a bisimulation and so there is the maximal bisimulation (union of all bisimulations), called *bisimulation equivalence* or *bisimilarity* and denoted by $\sim$.

A *BPP system* is traditionally defined as a context free grammar in a Greibach normal form with a set of variables $V = \{X_1, \ldots, X_n\}$, a set of terminals $\mathcal{A}$ and with rules of the form $X \xrightarrow{a} Y_1 Y_2 \ldots Y_k$. For each sequence α of variables we define M-SET as the multiset of variables appearing in α (Parikh image of α). The associated LTS has multisets of variables (ranged over by M, M', ...) as states, and $M \xrightarrow{a} M'$ iff there is a rule $X \xrightarrow{a} \alpha$ such that $X \in M$ and $M' = (M - \{X\}) \cup \text{M-SET}(\alpha)$.

Alternatively a BPP can be defined as a (special) labelled Petri net (BPP Petri net)

$$\Sigma = (P, \mathit{Tr}, \text{PRE}, F, \lambda)$$

where P is a finite set of *places*, Tr is a finite set of *transitions*, $\text{PRE} : \mathit{Tr} \rightarrow P$ is a function assigning the (only) input place to every transition, $F : (\mathit{Tr} \times P) \rightarrow \mathbb{N}$ is a function assigning the (multiset of) output places to each transition, and $\lambda : \mathit{Tr} \rightarrow \mathcal{A}$ is a labelling function. $\mathbb{N}$ denotes the set of nonnegative integers. We denote the set of output places of $t \in \mathit{Tr}$ by $\text{SUCC}(t) = \{p \in P \mid$

$F(t,p) > 0\}$. For $p \in P$ we define $\textsc{succ}(p) = \{t \in Tr \mid \textsc{pre}(t) = p\}$. We also extend the notation $\textsc{pre}(t)$ to set of transitions and for $T \subseteq Tr$ define $\textsc{pre}(T) = \{p \mid \exists t \in T : p = \textsc{pre}(t)\}$.

Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of places. A *marking* is a function $M : P \to \mathbb{N}$ assigning number of tokens to each place. Marking M can be viewed as a vector $(x_1, x_2, \ldots, x_k)$ where $x_i \in \mathbb{N}$ and $x_i = M(p_i)$. We use $\mathcal{M}$ to denote the set of all markings.

A transition t is *enabled* in $M \in \mathcal{M}$ iff $M(\textsc{pre}(t)) > 0$. A transition that is not enabled is *disabled*. An enabled transition can be performed, written $M \xrightarrow{t} M'$, where

$$M'(p) = \begin{cases} M(p) - 1 + F(t,p) & \text{if } p = \textsc{pre}(t) \\ M(p) + F(t,p) & \text{otherwise} \end{cases}$$

A BPP produces an LTS $(S, \mathcal{A}, \longrightarrow)$ where $S = \mathcal{M}$ and $M \xrightarrow{a} M'$ iff there is some $t \in Tr$ such that $M \xrightarrow{t} M'$ and $\lambda(t) = a$.

A *finite-state system* (FS) is traditionally defined as an LTS $(S, \mathcal{A}, \longrightarrow)$ where S is finite, but for technical convenience we define it as a BPP where for each $t \in Tr$ there is exactly one $p \in P$ such that $F(t,p) = 1$ and $F(t,p') = 0$ if $p' \neq p$. For $p \in P$ we define a marking M_p such that $M_p(p) = 1$ and $M_p(p') = 0$ for $p' \neq p$. We call such marking an *FS marking*.

Given BPPs $\Sigma_1 = (P_1, Tr_1, \textsc{pre}_1, F_1, \lambda_1)$ and $\Sigma_2 = (P_2, Tr_2, \textsc{pre}_2, F_2, \lambda_2)$ where P_1, P_2, Tr_1, and Tr_2 are disjoint, their *disjoint union* is a BPP $\Sigma = (P, Tr, \textsc{pre}, F, \lambda)$ where $P = P_1 \cup P_2$, $Tr = Tr_1 \cup Tr_2$, and $\textsc{pre}$, F, and λ are defined in an obvious manner. Markings of Σ_1 and Σ_2 can be extended to markings of Σ by setting all remaining elements to 0.

The problem $\textsc{bpp-bisim}$ can be formulated as follows: Given two BPP systems, Σ_1 and Σ_2, with distinguished markings M_1 and M_2 of Σ_1 and Σ_2, is $M_1 \sim M_2$? (The relation $\sim$ is defined over the disjoint union of Σ_1 and Σ_2.)

In this paper we consider the problem $\textsc{bpp-fs-bisim}$ which is a special case of $\textsc{bpp-bisim}$ where Σ_1 is a finite-state system and M_1 is an FS marking. We show that the problem $\textsc{bpp-fs-bisim}$ can be solved in time $O(n^4)$ where n is the size of the instance. We assume that BPPs in the instance are encoded as lists of places and transitions, where the encoding of each transition t contains a list of all $p \in \textsc{succ}(t)$ together with values $F(t,p)$. We assume that numbers are encoded in binary.

In the rest of this paper $\Sigma = (P, Tr, \textsc{pre}, F, \lambda)$ is the disjoint union of the BPP and the FS from the instance of $\textsc{bpp-fs-bisim}$, $\mathcal{M}$ denotes its set of markings, P_{FS} and Tr_{FS} are the sets of places and transitions of the FS from this instance ($P_{FS} \subseteq P$, $Tr_{FS} \subseteq Tr$), and M_p where $p \in P_{FS}$ denotes the marking such that $M_p \in \mathcal{M}$, $M_p(p) = 1$ and $M_p(p') = 0$ for $p' \neq p$. We define $\mathcal{M}_{FS} = \{M_p \mid p \in P_{FS}\}$.

Symbol ω denotes infinity. We stipulate that for each $x \in \mathbb{N}$, $x < \omega$, $\omega + x = x + \omega = \omega + \omega = \omega - \omega = -\omega + \omega = \omega$, $\omega \cdot 0 = 0 \cdot \omega = 0$, and for each

$x \geq 1$, $\omega \cdot x = x \cdot \omega = \omega$. We define $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$.

Let U be a set. $|U|$ denotes the cardinality of U. *Partition* $\mathcal{U}$ of U is a set $\mathcal{U} = \{U_1, U_2, \dots, U_l\}$ of disjoint non-empty *classes* whose union is U.

3 The Algorithm

In the proof we use a method of Jančar used in [4] for showing that BPP-BISIM is in *PSPACE*. The basic idea is to construct a series of norm functions that are used for approximation of the bisimulation equivalence. The construction stops when no other functions can be added, and at this point the approximation is exact.

At first we recall some ideas from [4]. Let $(S, \mathcal{A}, \longrightarrow)$ be an LTS, and let $\mathcal{C} : S \to \mathcal{D}$ be a mapping assigning to each state a value from some domain $\mathcal{D}$. We say the mapping $\mathcal{C}$ is *bis-necessary* if for each $s, s' \in S$, $s \sim s'$ implies $\mathcal{C}(s) = \mathcal{C}(s')$. If we have a set of functions $\{\mathcal{C}_1, \mathcal{C}_2, \dots \mathcal{C}_l\}$ where $\mathcal{C}_i : S \to \mathcal{D}_i$, we say the set is *bis-necessary* iff every $\mathcal{C}_i$ is bis-necessary. A predicate $\mathcal{P}$ on S can be viewed as a mapping $\mathcal{P} : S \to \{0, 1\}$, and so we can also talk about *bis-necessary predicate*. Note that if $\mathcal{P}$ is bis-necessary, then $\neg\mathcal{P}$ is also bis-necessary.

Let $\mathcal{P}$ be a predicate on S. We define the mapping $\textsc{dist}(\mathcal{P}) : S \to \mathbb{N}_\omega$ where $\textsc{dist}(\mathcal{P})(s)$ is the length of the shortest w such that $s \xrightarrow{w} s'$ and $\mathcal{P}(s')$, and if there is no such w, $\textsc{dist}(\mathcal{P})(s) = \omega$. Intuitively, $\textsc{dist}(\mathcal{P})$ represents 'distance' to $\mathcal{P}$.

Claim 3.1 *If $\mathcal{P}$ is bis-necessary then* $\textsc{dist}(\mathcal{P})$ *is bis-necessary.*

Proof. Let us assume without loss of generality that there are states s_1, s_2 such that $s_1 \sim s_2$ and $\textsc{dist}(\mathcal{P})(s_1) < \textsc{dist}(\mathcal{P})(s_2)$. Then there is some shortest $w \in \mathcal{A}^*$ such that $s_1 \xrightarrow{w} s_1'$ and $\mathcal{P}(s_1')$. Because $s_1 \sim s_2$, there must be some s_2' such that $s_2 \xrightarrow{w} s_2'$ and $s_1' \sim s_2'$. But $|w| < \textsc{dist}(\mathcal{P})(s_2')$, and so $\neg\mathcal{P}(s_2')$, which means that $\mathcal{P}$ is not bis-necessary. $\square$

Let us now consider the BPP Σ from the instance of BPP-FS-BISIM. Let $T \subseteq Tr$. We say T is *disabled* in M if every $t \in T$ is disabled in M. Notice that if T is the set of all transitions t such that $\lambda(t) = a$ for some $a \in \mathcal{A}$, then 'T is disabled' is a bis-necessary predicate. Notice also that T is disabled iff each place in $\textsc{pre}(T)$ is empty. These leads to the following formal definitions. Let $Q \subseteq P$ be a set of places. We define the predicate $\textsc{zero}(Q)$ on $\mathcal{M}$ such that $\textsc{zero}(Q)(M)$ iff $\forall p \in Q : M(p) = 0$. We define *norm* of Q as the function $\textsc{norm}(Q) = \textsc{dist}(\textsc{zero}(Q))$.

Every norm can be expressed as a *linear function* $L : \mathcal{M} \to \mathbb{N}_\omega$ of the form

$$L(x_1, x_2, \dots, x_k) = c_1x_1 + c_2x_2 + \cdots + c_kx_k$$

where $c_i \in \mathbb{N}_\omega$ and k is the number of places, see [4] for details. Coefficients $c_1, c_2, \dots, c_k$ of L for the given Q can be computed by the algorithm in Figure 1. Intuitively, c_i is the minimal number of transitions that remove one token in

for each $p \in P$ **do**
  if $p \in Q$ **then** $c_p := \omega$ **else** $c_p := 0$
$Q' := Q$
$T := \{t \in Tr \mid \text{PRE}(t) \in Q'\}$
while $Q' \neq \emptyset$ **do**
  let p_{min} refer to some $p \in Q'$ with minimal c_p
  for each $t \in T$ such that $\text{SUCC}(t) \cap Q' = \emptyset$ **do**
    remove t from T
    $p := \text{PRE}(t)$; $R := \text{SUCC}(t)$
    $d_t := 1 + \sum_{q \in R} c_q \cdot F(t,q)$
    if $d_t < c_p$ **then** $c_p := d_t$
    if $c_p < c_{p_{min}}$ **then** $p_{min} := p$
  end for
  if $c_{p_{min}} = \omega$ **then break**;
  $Q' := Q' - \{p_{min}\}$
  remove from T every t such that $\text{PRE}(t) = p_{min}$
end while

Fig. 1. Computing coefficients of $\text{NORM}(Q)$ function

p_i from Q. In the algorithm, Q' is the set of unprocessed places and T is the set of unprocessed transitions. We write c_p instead of c_i where $p = p_i$. Places that are not in Q' are places for which c_p was already determined. The algorithm computes for each unprocessed transition t that stores tokens only to places out of Q' the value d_t, a possible candidate for c_p where $p = \text{PRE}(t)$, and chooses between these candidates the one with the minimal value.

We define $\Omega\text{-CARR}(L) = \{p_i \in P \mid c_i = \omega\}$. Note that $L(M) = \omega$ iff $M(p) > 0$ for some $p \in \Omega\text{-CARR}(L)$. It is not hard to show that $\Omega\text{-CARR}(L)$ is a trap. Recall that a set of places $R \subseteq P$ is a *trap* iff

$$\forall t : \text{PRE}(t) \in R \Rightarrow (R \cap \text{SUCC}(t) \neq \emptyset)$$

Intuitively this means that every t removing tokens from a trap also adds some tokens to it, so 'marked' trap, i.e., a trap with at least one token, can not get unmarked. From this follows the following claim:

Claim 3.2 *If $L = \text{NORM}(Q)$ for some $Q \subseteq P$ and $L(M) = \omega$, then $L(M') = \omega$ for every M' such that $\exists w \in \mathcal{A}^* : M \xrightarrow{w} M'$.*

For a linear function L we can compute for each $t \in Tr$ the value

(1) $$\delta_t^L = -c_i + \sum_{1 \leq j \leq k} c_j \cdot F(t, p_j)$$

where $\text{PRE}(t) = p_i$. The value δ_t^L represents the 'change' on the value of L when the transition t is performed.

Claim 3.3 *If $M \xrightarrow{t} M'$ then $L(M) + \delta_t^L = L(M')$. If $L(M) < \omega$ and $\delta \neq \delta_t^L$ then $L(M) + \delta \neq L(M')$.*

Now we come to the description of the algorithm. The algorithm constructs a set of linear functions $\mathcal{L} = \{L_1, L_2, \ldots\}$ such that each L_i represents norm of some set of places and where each L_i is bis-necessary. The algorithm starts with $\mathcal{L} = \emptyset$, successively adds linear functions to $\mathcal{L}$ and stops when no new linear function can be added. For $\mathcal{L}$ we define the equivalence $\equiv_{\mathcal{L}}$ on $\mathcal{M}$ such that $M \equiv_{\mathcal{L}} M'$ iff $\forall L \in \mathcal{L} : L(M) = L(M')$. Since each $L \in \mathcal{L}$ is bis-necessary, $\mathcal{L}$ is also bis-necessary, and $M \not\equiv_{\mathcal{L}} M'$ implies $M \not\sim M'$. On the other hand, we show that if $M \in \mathcal{M}_{FS}$ and $M' \in \mathcal{M}$ then $M \equiv_{\mathcal{L}} M'$ implies $M \sim M'$.

The main algorithm looks as follows:

1. Set $\mathcal{L} = \emptyset$.
2. For each $p \in P_{FS}$ perform Step described below.
3. If $\mathcal{L}$ has changed in the previous step, go to 2.

The Step looks as follows: For the given p we define the set $\mathcal{F} \subseteq \mathcal{L}$ such that $L \in \mathcal{F}$ iff $L(M_p) < \omega$. For $\mathcal{F}$ we define the equivalence $\cong_{\mathcal{F}}$ on Tr such that $t \cong_{\mathcal{F}} t'$ iff $\lambda(t) = \lambda(t')$ and $\forall L \in \mathcal{F} : \delta^L_t = \delta^L_{t'}$. Let $[t]$ denote the equivalence class of $\cong_{\mathcal{F}}$ containing t. Let $\mathcal{T}_1 = \{[t] \mid t \in \textsc{succ}(p)\}$, and let $T_0 = Tr - \bigcup_{T \in \mathcal{T}_1} T$. We define the set $\mathcal{T}$ as $\mathcal{T} = \mathcal{T}_1 \cup \{T_0\}$, respectively as $\mathcal{T} = \mathcal{T}_1$ when T_0 is empty. Note $\mathcal{T}$ is a partition of Tr. We extend the definition of Ω-carr to sets of linear functions and define

$$\Omega\text{-}\textsc{carr}(\mathcal{F}) = \bigcup_{L \in \mathcal{F}} \Omega\text{-}\textsc{carr}(L)$$

The algorithm now computes for each $T \in \mathcal{T}$ the function $L = \textsc{norm}(\textsc{pre}(T) \cup \Omega\text{-}\textsc{carr}(\mathcal{F}))$ and adds it to $\mathcal{L}$.

We now show that the algorithm is correct.

Lemma 3.4 *Every L added to $\mathcal{L}$ by the algorithm is bis-necessary.*

Proof. We proceed by induction on the number of steps. The proposition is trivially true at the start. Assume now the algorithm performs Step for some $p \in P_{FS}$ and adds $\textsc{norm}(Q)$ to $\mathcal{L}$ for some $T \in \mathcal{T}$ where $Q = \textsc{pre}(T) \cup \Omega\text{-}\textsc{carr}(\mathcal{F})$. Due to Claim 3.1 it is sufficient to show that $\textsc{zero}(Q)$ is bis-necessary. Let us assume without loss of generality that $M_1 \sim M_2$, $\neg\textsc{zero}(Q)(M_1)$, and $\textsc{zero}(Q)(M_2)$. By induction hypothesis, $\forall L \in \mathcal{L} : L(M_1) = L(M_2)$. Let $R = \Omega\text{-}\textsc{carr}(\mathcal{F})$. Since $\textsc{zero}(R)(M_2)$, we have $\forall L \in \mathcal{F} : L(M_2) < \omega$, and $\textsc{zero}(R)(M_1)$, since otherwise there is some $L \in \mathcal{F}$ such that $L(M_1) = \omega \neq L(M_2)$. From this and $\neg\textsc{zero}(Q)(M_1)$ we have $\neg\textsc{zero}(\textsc{pre}(T))(M_1)$. This means there is some transition $t \in T$ such that $M_1 \xrightarrow{t} M_1'$. Because $M_1 \sim M_2$ there is some t' such that $M_2 \xrightarrow{t'} M_2'$ where $M_2 \sim M_2'$ and $\lambda(t) = \lambda(t')$, but necessarily $t' \notin T$. This means there is some $L \in \mathcal{F}$ such that $\delta^L_t \neq \delta^L_{t'}$. By Claim 3.3 this implies $L(M_1') \neq L(M_2')$, a contradiction. □

Since every L added to $\mathcal{L}$ is $\textsc{norm}(Q)$ for some $Q \subseteq P$, and P is finite, it is obvious that the algorithm stops after some finite number of steps. The following lemma shows that $\equiv_{\mathcal{L}}$ corresponding to $\mathcal{L}$ computed by the algorithm coincides with $\sim$ on pairs of markings where one of markings is from $\mathcal{M}_{FS}$.

Lemma 3.5 *Let $\mathcal{L}$ be the set computed by the algorithm. Then for every $M_1 \in \mathcal{M}_{FS}$ and $M_2 \in \mathcal{M}$, $M_1 \equiv_{\mathcal{L}} M_2$ implies $M_1 \sim M_2$.*

Proof. We show that $\equiv_{\mathcal{L}} \cap (\mathcal{M}_{FS} \times \mathcal{M})$ is a bisimulation. Let us consider $M_1 \in \mathcal{M}_{FS}$ and $M_2 \in \mathcal{M}$ such that $M_1 \equiv_{\mathcal{L}} M_2$. Let $\mathcal{F} = \{L \in \mathcal{L} \mid L(M_1) < \omega\}$ and let $R = \Omega\text{-}\textsc{carr}(\mathcal{F})$. Note that $M_1 = M_p$ for some $p \in P_{FS}$ and the same $\mathcal{F}$ would be produced when the algorithm would perform Step for p. Notice that $\textsc{zero}(R)(M_1)$ since otherwise there is some $L \in \mathcal{F}$ such that $L(M_1) = \omega$. Also $\textsc{zero}(R)(M_2)$ is true, because otherwise there is some $L \in \mathcal{F}$ such that $L(M_2) = \omega$ which means $L(M_1) \neq L(M_2)$. Let $\mathcal{T}$ be defined for $\mathcal{F}$ correspondingly as in Step.

At first consider a transition $M_1 \xrightarrow{t} M_1'$. Let T be the class from $\mathcal{T}$ such that $t \in T$. Obviously $T \in \mathcal{T}_1$. Consider now the function $L_1 = \textsc{norm}(R \cup \textsc{pre}(T))$. It must be the case that $L_1 \in \mathcal{L}$, otherwise L_1 could be added to $\mathcal{L}$ and the algorithm has not finished yet. So $L_1(M_1) = L_1(M_2)$. From this, from $\neg\textsc{zero}(\textsc{pre}(T))(M_1)$, and from $\textsc{zero}(R)(M_2)$ we have $\neg\textsc{zero}(\textsc{pre}(T))(M_2)$ and there is some $t' \in T$ such that $M_2 \xrightarrow{t'} M_2'$, $\lambda(t) = \lambda(t')$, and $\forall L \in \mathcal{F} : \delta_t^L = \delta_{t'}^L$. From this and Claim 3.3 we obtain $\forall L \in \mathcal{F} : L(M_1') = L(M_2')$. For each $L \in \mathcal{L} - \mathcal{F}$ is $L(M_1) = L(M_2) = \omega$, and, by Claim 3.2, $L(M_1') = L(M_2') = \omega$. This means $M_1' \equiv_{\mathcal{L}} M_2'$.

Now consider a transition $M_2 \xrightarrow{t'} M_2'$. This case similar to the previous case, but we must also consider the possibility $t' \in T_0$. Let $L_0 = \textsc{norm}(R \cup \textsc{pre}(T_0))$. Since $L_0 \in \mathcal{L}$ (otherwise the algorithm has not finished yet), $L_0(M_1) = L_0(M_2)$. Because $L_0(M_1) = 0$, we obtain $L_0(M_2) = 0$, and $\textsc{zero}(\textsc{pre}(T_0))(M_2)$, so t' is not enabled in M_2, a contradiction. $\square$

4 Time Complexity of the Algorithm

In this section we show that the running time of the algorithm is $O(n^4)$. In the rest of the paper n denotes the size of the input instance.

The running time of the algorithm depends on implementation details of Step. In Section 3 we described how to, for the given $p \in P_{FS}$, compute in Step sets $\mathcal{F}$, $\Omega\text{-}\textsc{carr}(\mathcal{F})$, and $\mathcal{T}$. It is more efficient not to recompute these sets every time, but instead to store their values and perform necessary changes on them when new L is added to $\mathcal{L}$. So the algorithm maintains for each $p \in P_{FS}$ values of the corresponding $\Omega\text{-}\textsc{carr}(\mathcal{F})$ and $\mathcal{T}$. Note that $\mathcal{T}$ always contains at most $|\textsc{succ}(p)| + 1$ equivalence classes. The algorithm also maintains for each $T \in \mathcal{T}$ and for $\Omega\text{-}\textsc{carr}(\mathcal{F})$ a boolean flag indicating whether it has changed since the last invocation of Step and adds a new function $L = \textsc{norm}(\Omega\text{-}\textsc{carr}(\mathcal{F}) \cup T)$ to $\mathcal{L}$ only when $\Omega\text{-}\textsc{carr}(\mathcal{F})$ or T is new or has actually changed.

Addition of L to $\mathcal{L}$ includes the following steps:

1. Compute coefficients $c_1, c_2, \ldots, c_k$ of L.
2. Compute δ_t^L for each $t \in Tr$.
3. Partition Tr according to values of δ_t^L and $\lambda(t)$.
4. For each $p \in P_{FS}$ such that $L(M_p) < \omega$:
   - Add Ω-CARR(L) to the corresponding Ω-CARR$(\mathcal{F})$.
   - Modify the corresponding $\mathcal{T}$ using the partition computed in step 3.

In the proof we need the following well-known fact:

Fact 4.1 *Let U be a non-empty finite set, and let $\mathcal{U}_1, \mathcal{U}_2, \ldots$ be a sequence of partitions of U such that each $\mathcal{U}_{i+1}$ is a refinement of $\mathcal{U}_i$. Then the total number of different classes in all these partitions is less then $2 \cdot |U|$.*

Proof idea. Use induction on $|U|$. □

Lemma 4.2 *The number of functions added to $\mathcal{L}$ is in $O(n^2)$.*

Proof. Let us consider all invocations of STEP for one $p \in P_{FS}$. In invocations where Ω-CARR$(\mathcal{F})$ has changed, the algorithm adds a new function to $\mathcal{L}$ for each $T \in \mathcal{T}$. If Ω-CARR$(\mathcal{F})$ has not changed, a new function is added for each $T \in \mathcal{T}$ that has changed.

Ω-CARR$(\mathcal{F})$ can only grow, so the number of invocations of STEP when Ω-CARR$(\mathcal{F})$ has changed is $O(|P|)$. Because $|\mathcal{T}|$ is at most $h+1$ where $h =$ SUCC(p), the number of functions added in such invocations is at most $(h+1) \cdot O(|P|)$.

Consider now the possible changes of $\mathcal{T}$. Either some t was added to T_0, or some $T \in \mathcal{T}_1$ was split, or some combination of these possibilities has occurred. Since T_0 can only grow, the first possibility can occur only $O(|Tr|)$ times. It remains to estimate the total number of classes from $\mathcal{T}_1$. It is in $O(|Tr|)$ as follows from Fact 4.1, since sequence of values of $\mathcal{T}_1$ in subsequent invocations of STEP can be extended to a sequence of refined partitions by adding some classes to each $\mathcal{T}_1$.

Let us sum now the numbers of functions added to $\mathcal{L}$ for all $p \in P_{FS}$. In invocations where Ω-CARR$(\mathcal{F})$ has changed it is at most

$$\sum_{p \in P_{FS}} (|\textsc{succ}(p)| + 1) \cdot O(|P|) = O(|P| \cdot |Tr_{FS}|)$$

The number of function added in the remaining invocations is in $O(|P_{FS}| \cdot |Tr|)$, so we obtain that the total number of functions is in $O(|P| \cdot |Tr|)$. □

Now we consider the complexity of computation of coefficients of $L =$ NORM(Q) for some $Q \subseteq P$. For $x \in \mathbb{N}_\omega$, $size(x)$ denotes the number of bits of x when encoded in binary. We suppose that $size(x+y) = 1 + \max\{size(x), size(y)\}$, $size(x \cdot y) = size(x) + size(y)$, and $size(\omega) = O(1)$.

Proposition 4.3 *For each $p \in P$, $size(c_p)$ is in $O(n)$.*

Proof. Let $p_1, p_2, \ldots, p_l$ be the sequence of places from Q ordered by the order in which the algorithm determines their coefficients, let c_i be the coefficient of p_i, and let t_i be the transition used for computation of c_i, i.e., the transition such that $\textsc{pre}(t_i) = p_i$ and $c_i = d_i$, where we write d_i instead of d_{t_i}. Let *size*(t) be the number of bits of representation of $t \in T$, i.e.,

$$size(t) = O((1 + |R|) \cdot size(|P|)) + \sum_{p \in R} size(F(t, p))$$

where $R = \textsc{succ}(t)$.

By induction on i we prove the following proposition from which the result directly follows: For each i, $1 \leq i \leq l$, $size(c_i) \leq \sum_{1 \leq j \leq i} size(t_j)$. This holds trivially for $i = 1$ because c_1 is always 1 or ω, so suppose $i > 1$. Let $R = \textsc{succ}(t_i)$. Note that

$$d_i = 1 + \sum_{q \in R} c_q \cdot F(t_i, q) \leq 1 + \sum_{j=1}^{i-1} c_j \cdot F(t_i, p_j)$$

because when d_i is computed, each c_q is known and finite, and so it is either 0 or one from c_1 to c_{i-1}.

$size(c_j \cdot F(t_i, p_j)) = size(c_j) + size(F(t_i, p_j))$. The sum of all such products can be written in the size of maximal of them plus some number less then their count (overflow caused by addition). This size is less then $size(\max\{c_j \mid 1 \leq j < i\}) + \sum_{j=1}^{i-1} size(F(t_i, p_j))$. The second summand (the sum) is less then $size(t_i)$. By induction hypothesis maximal c_j can be written in the count of bits needed for first $i - 1$ transitions. Therefore d_i (and hence c_i too) can be written in the space needed for representations of transitions $t_1, \ldots, t_i$. $\square$

Proposition 4.4 *All coefficients of* $L = \textsc{norm}(Q)$ *are computed in* $O(n^2)$.

Proof. The most time-consuming step is computation of all d_i. In computation of this, multiplications are more time-consuming than additions. Hence it suffices to show that aggregated complexity of all multiplications is in $O(n^2)$.

In our algorithm, each d_i is computed only once. During computation of d_i we need to determine all products $F(t_i, p_j) \cdot c_j$ where $p_j \in \textsc{succ}(t)$. From Proposition 4.3 we know that $size(c_j)$ is in $O(n)$ for every c_j. Hence one product is computed in $O(n \cdot size(F(t_i, p_j)))$. If we sum complexities of such products for all transitions and places to which transitions give tokens, we get the aggregated complexity of all multiplications

$$O(\sum_{i,j} (n \cdot size(F(t_i, p_j)))) = O(n \cdot \sum_{i,j} size(F(t_i, p_j))) = O(n^2)$$

. $\square$

Proposition 4.5 *For given* $L = \textsc{norm}(Q)$*, changes* δ_t^L *caused by all transitions can be computed in time* $O(n^2)$.

Proof. For each transition t, the value δ_t^L is computed using expression 1 from Section 3. If some c_j is infinite then δ_t^L is infinite too. Hence we do computation of the sum only for finite values. The complexity of additions is dominated by the complexity of multiplications $c_j \cdot F(t, p_j)$. Each such product is computed only once. From Proposition 4.3 we know that each c_j is in $O(n)$. Each $F(t, p_j)$ is used only once and is part of our representation of BPP. Hence we can similarly as in Proposition 4.4 for coefficients deduce that aggregated complexity of all multiplications is in $O(n^2)$, from which the result follows. □

Lemma 4.6 *The algorithm adds one L to $\mathcal{L}$ in time $O(n^2)$.*

Proof. As follows from Propositions 4.4 and 4.5, the running time of steps 1 and 2 is $O(n^2)$. The running time of step 3 is also $O(n^2)$ using one of standard algorithms for lexicographic sorting of strings (see [1,7]), because values of δ_t^L can be represented as binary numbers, i.e., as strings of 0 and 1. Also the running time of step 4 is $O(n^2)$ if it is implemented carefully. □

Theorem 4.7 *There is an algorithm solving* BPP-FS-BISIM *with running time $O(n^4)$.*

Proof. We have described the algorithm. Lemmas 3.4 and 3.5 ensure the correctness of the algorithm. Since the addition of new L to $\mathcal{L}$ is the most time consuming operation of the algorithm, it follows from Lemmas 4.2 and 4.6 that the running time of the algorithm is $O(n^4)$. □

5 Conclusion and Future Work

In this paper we have presented an algorithm for deciding bisimilarity between a BPP and a finite-state system with time complexity $O(n^4)$. However it is possible that the time complexity of the algorithm is not optimal and can be further improved.

Other problem where the techniques from [4] used in this paper can be used is the problem of deciding regularity of a BPP system, i.e., the problem whether for a given BPP process there exists a bisimilar finite-state process. This problem is known to be decidable [5] and *PSPACE*-hard [8], but no upper bound is known for the problem. We conjecture that the problem is in *PSPACE* and we plan to show it in the future.

References

[1] Aho, A. V., J. E. Hopcroft and J. D. Ullman, “Design and Analysis of Computer Algorithms,” Addison-Wesley Reading, 1974.

[2] Christensen, S., Y. Hirsfeld and F. Moller, *Bisimulation is decidable for all basic parallel processes*, in: *Proc. CONCUR’93*, LNCS **715** (1993), pp. 143–157.

[3] Hirsfeld, Y., M. Jerrum and F. Moller, *A polynomial algorithm for deciding bisimulation equivalence of normed basic parallel processes*, Mathematical Structures in Computer Science **6** (1996), pp. 251–259.

[4] Jančar, P., *Strong bisimilarity on basic parallel processes is PSPACE-complete*, in: *Proc. 18th LiCS* (2003), pp. 218–227.

[5] Jančar, P. and J. Esparza, *Deciding finiteness of petri nets up to bisimulation*, in: *Proc. of ICALP'96*, LNCS **1099** (1996), pp. 478–489.

[6] Jančar, P. and M. Kot, *Bisimilarity on normed basic parallel processes can be decided in time $O(n^3)$*, in: R. Bharadwaj, editor, *Proceedings of the Third International Workshop on Automated Verification of Infinite-State Systems – AVIS 2004*, 2004.

[7] Paige, R. and R. E. Tarjan, *Three partition refinement algorithms*, SIAM Journal on Computing **16** (1987), pp. 973–989.

[8] Srba, J., *Strong bisimilarity and regularity of basic parallel processes is PSPACE-hard*, in: *Proc. STACS'02*, LNCS **2285** (2002), pp. 535–546.