Bisimilarity Problems Requiring Exponential Time (Extended Abstract)

Michele Boreale* and Luca Trevisan

Università di Roma "La Sapienza". Dipartimento di Scienze dell'Informazione. Via Salaria 113, 00198 Roma. Email {michele,trevisan}@dsi.uniroma1.it

Abstract. We study the complexity of deciding bisimilarity between non-deterministic processes. In particular, we consider a calculus with recursive definitions of processes, value passing (i.e. input/output of data) and an equality test over data. We show that the bisimilarity problem is EXP-complete over this calculus and thus that exponential time is *provably necessary* in order to solve it. We then prove that, if we add a *parallel composition* operator to the calculus, and we impose that parallel composition is never used inside recursive definitions, then the bisimilarity problem is still EXP-complete, thus no harder than in the fragment without parallel composition.

1 Introduction

A major field of research in theoretical computer science is concerned with the formal description and analysis of concurrent systems. Well-developed theories exist for calculi such as Milner's CCS [12], which permits naturally describing systems where different agents can interact via synchronization, without passing of data values involved. Recently, there has been much interest around extensions of CCS with explicit primitives for handling data values [7, 8, 15]. In these formalisms, referred to as *value-passing calculi*, data, beside being sent or received, can also be used as parameters in recursive definitions of processes and tested by means of certain predicates. As an example, the recursively defined process:

$$M(x) \Leftarrow [x < v]\big(\overline{r}x.M(x) \,+\, w(y).M(y)\big) + [x \geq v]E(x)$$

specifies an updatable memory cell, containing an initial value x. As long as x remains less than a certain value v, the memory can either output its content at channel r, $\overline{r}x.$, or input a new value at channel w,

* This research was done while the first author was at the Istituto per l'Elaborazione dell'Informazione of the CNR (Italian Research Council). Work partially supported by EEC, within HCM Project Express, and by CNR, within the project "Specifica ad Alto Livello e Verifica di Sistemi Digitali".

$w(y)$. (the symbol + represents non-deterministic choice). As soon as x equals or exceeds v, a recovery procedure E is called. A peculiar kind of value-passing calculus is the π-calculus [13, 15, 1], where the values being exchanged among processes are channels themselves: this makes it possible to dynamically reconfigure processes' communication topology.

In this setting, a central problem is that of *verification*, which consists in establishing whether two given descriptions (representing, e.g., a specification and an implementation) are "equivalent" or not, according to a chosen notion of *behavioural equivalence*. By now, the algebraic aspects of this problem have become well-understood also for value-passing calculi [7, 1, 2, 8, 13, 15], but much is left to do concerning computational complexity. A fundamental issue is that of classifying the relative computational power of the different mechanisms for handling data values, also in the presence of other primitives of process calculi, such as parallelism. This could in practice provide useful indications to build more effective verification tools.

In the present paper, we shall tackle some of these issues, for one of the most widely studied equivalence, Milner's bisimulation equivalence (also called "bisimilarity"), written $\sim$ [12].

Previous works about decidability and complexity of value-passing bisimilarity are [9] and [3]. In [9], Jonsson and Parrow consider a particular class of non-deterministic value-passing processes, the *data-independent* ones: here, data can be sent, received and used in parametric definitions, but no predicate or function over them is allowed. Jonsson and Parrow show the decidability of bisimilarity for these processes by reducing the problem to bisimilarity of certain non-value-passing programs, for which verification methods exist [14, 10]. In [3], the authors prove that bisimilarity is PSPACE-hard (in the syntactical size of the terms) for data-independent processes.

Having data values without being able to test them is, in practice, of little use. It is therefore natural to ask what happens to complexity when a simple form of predicate is added to data-independent processes. In [3], a simple equality predicate over data was considered. Equality is perhaps the most elementary form of predicate one would admit over data: not even negative tests, to check inequality of two values, are permitted. The computational power of equality was indirectly showed in [3] by proving that, relying on it, input-output primitives ($a(x).$ and $\overline{a}v.$) can be polynomially reduced to the remaining operators. Here, we prove that, when equality is added, value-passing bisimilarity is EXP-complete.

We also consider adding other primitives typical of process calculi, such as parallelism [12]. It is known that, for full CCS, bisimilarity is undecidable; however, it is decidable for certain meaningful restricted formats, such as *finite control* processes, where parallelism does not occur inside the scope of recursive definitions. For processes in such format, we prove that, in the presence of equality, the overall computational complexity does not increase, i.e. bisimilarity remains EXP-complete. It is worth to stress that EXP-complete problems are *provably intractable*, that is, any algorithm solving an EXP-complete problem must have an exponential worst-case running time, and this can be shown without relying on unproven complexity-theoretic conjectures (conversely, NP-complete and PSPACE-complete problems are just *supposed* to be intractable). Meaningful EXP-complete problems are quite rare in the literature. In particular, we are not aware of any other EXP-complete problem concerned with verification of process equivalences.

The used proof techniques are also worth to mention. We rely on the characterization of the class EXP in terms of *Alternating Turing Machines (ATM)*. (see [5]). We show how to define processes that somehow *simulate* an arbitrary linear-space ATM. The more difficult technical step consists in simulating *alternation*.

The rest of the paper is organized as follows. In Section 2 we define the syntax and the operational and bisimulation semantics of the basic calculus, without parallel composition. In Section 3 we recall some basics facts about the class EXP and its characterization in terms of ATM's. Section 4 contains the EXP-completeness proofs. In Section 5 we deal with the parallel composition operator. A few conclusive remarks are contained in Section 6.

2 The Language

Below, we present first the syntax and then operational and bisimulation semantics of the calculi. The notation we use is that of value-passing CCS [11, 12] and of π-calculus [13]. We assume the following disjoint sets:

- a countable set Act of *pure actions* or *communications ports*, ranged over by $a, a', \ldots$;
- a countable set Var of *variables*, ranged over by $x, y, \ldots$;
- a countable set Val of *values*, ranged over by $v, v', \ldots$;
- a countable set Ide of *identifiers* each having a non-negative *arity*. Ide is ranged over by Id and capital letters.

A *value expression* is either a variable or a value. Value expressions are ranged over by $e, e', \ldots$. We also consider the set $\overline{Act} = \{\overline{a} \mid a \in Act\}$ of *co-actions*, which represent output synchronizations. The set $Act \cup \overline{Act}$ will be ranged over by c.

Following the notation of [3], we let $\mathcal{L}_{v,r}$ be the set of *terms* (ranged over by $P, Q, \ldots$) given by the operators of *pure synchronization prefix, input prefix, output prefix, non-determinism, matching* and *identifier*, according to the following grammar:

$$P ::= c.P \quad | \quad a(x).P \quad | \quad \overline{a}e.P \quad | \quad \sum_{i \in I} P_i \quad | \quad [e_1 = e_2]P \quad | \quad Id(e_1, \ldots, e_k)$$

where k is the arity of Id. We also let $\mathcal{L}_r$ be the restriction of $\mathcal{L}_{v,r}$ to terms without input and output prefixes. We always assume that the index set I in $\sum_{i \in I} P_i$ is finite and sometimes write $P_1 + \cdots + P_n$ for $\sum_{i \in \{1,\ldots,n\}} P_i$. When I is empty, we use the symbol $\mathbf{0}$: $\mathbf{0} \stackrel{\text{def}}{=} \sum_{i \in \emptyset} P_i$.

An occurrence of a variable x in a term P is said to be *bound* if it is within the scope of an input prefix $a(x)$; otherwise it is said a *free* occurrence. The set of variables which have a free occurrence in P is denoted by $fvar(P)$. The *size* of a term P, indicated by $|P|$, is the number of symbols appearing in it; e.g., if $P = a(x).\overline{a}x.a'.\mathbf{0} + Id(x)$ then $|P| = 9$.

We presuppose an arbitrarely fixed *finite* set Eq of *identifiers definitions*, each of the form

$$Id(x_1, \ldots, x_k) \Leftarrow P$$

where $k \geq 0$ is the arity of the identifier Id, the x_i's are pairwise distinct and $fvar(P) \subseteq \{x_1, \ldots, x_k\}$. In Eq, each identifier has a single definition. The requirement for the set Eq to be finite is motivated by the fact that we are only interested in syntactically finite processes.

A process term P is said to be *closed* if $fvar(P) = \emptyset$; in this case, P is said to be a *process*. Processes are the terms we are most interested in. As we shall see, bisimulation semantics will be defined only over the set of processes.

The operational behaviour of our processes is defined by means of a transition relation. Its elements are triples (P, μ, P') written as $P \xrightarrow{\mu} P'$. Here, μ can be of three different forms: c, $\overline{a}v$ or $a(v)$. A *pure action* c represents a synchronization through the port c, without passing of data involved. An *output action* $\overline{a}v$ means transmission of the datum v through the port a. An *input action* $a(v)$ represents receipt of the datum v through the port a. We let μ range over actions. The transition relation is defined

by the inference rules in Table 2. Note that $\xrightarrow{\mu}$ leads processes into processes. On the top of the transition relation $\xrightarrow{\mu}$, we define *strong bisimulation equivalence* $\sim$, [12, 13, 15] as usual:

$$(Sync)\ c.P \xrightarrow{c} P$$

$$(Inp)\ a(x).P \xrightarrow{a(v)} P\{v/x\},\ v \in Val \qquad (Out)\ \overline{a}v.P \xrightarrow{\overline{a}v} P$$

$$(Match)\frac{P \xrightarrow{\mu} P'}{[v=v]P \xrightarrow{\mu} P'} \qquad (Sum)\frac{P_j \xrightarrow{\mu} P'}{\sum_{i\in I} P_i \xrightarrow{\mu} P'}\ j \in I$$

$$(Ide)\frac{P\{\tilde{v}/\tilde{x}\} \xrightarrow{\mu} P'}{Id(\tilde{v}) \xrightarrow{\mu} P'} \quad \text{if } Id(\tilde{x}) \Leftarrow P \text{ is in } Eq$$

Table 1. Inference rules for the transition relation $\xrightarrow{\mu}$.

Definition 1 ((Strong bisimulation equivalence)). *A binary symmetric relation $\mathcal{R}$ over processes is a* bisimulation *if, whenever $P\mathcal{R}Q$ and $P \xrightarrow{\mu} P'$, there exists Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\mathcal{R}Q'$. We let $P \sim Q$, and say that P* is bisimilar to *Q, if and only if $P\mathcal{R}Q$, for some bisimulation $\mathcal{R}$.*

From now on, we will omit the adjective "strong".

3 Alternating Turing Machines and the Class EXP

In this paper, we will measure the complexity of deciding bisimilarity between P and Q with a set of identifier definitions Eq, in function of the sum of the syntactical sizes of P, Q and of the terms occurring in Eq. We will deal with the complexity classes P, LIN-EXP and EXP and with the notions of *polynomial-time reducibility*, *hardness* and *completeness*.

It is known that $\mathsf{P} \subset \mathsf{LIN\text{-}EXP} \subset \mathsf{EXP}$ and that the these three classes are provably distinct. A problem is hard for a class $\mathcal{C}$ if every problem in $\mathcal{C}$ is polynomial-time reducible to it; a $\mathcal{C}$-hard problem is said to be $\mathcal{C}$-complete if it belongs to $\mathcal{C}$. It is easy to show that a problem is LIN-EXP-hard if and only if it is EXP-hard. See e.g. [4] for a more complete introduction to complexity classes. Here we recall the following result due to

Hartmanis and Stearns that states the provable intractability of LIN-EXP-hard problem.

Theorem 1 ([6]). *For any* LIN-EXP*-hard problem A, a constant c_A exists such that no algorithm can solve A with a worst case running time smaller than $2^{n^{c_A}}$.*

In the following we shall outline the characterization of LIN-EXP as the class of languages decided by *alternating Turing machines* (in short, ATM) working with linear space. This characterization will be exploited in Section 4 in order to prove the EXP-hardness of bisimilarity in $\mathcal{L}_r$.

Definition 2 (Alternating Turing Machine). *An alternating Turing machine (ATM in short) AT is a five-tuple $AT = (Q, q_0, g, \Sigma, \delta)$ where*

- *Q is the set of* states*;*
- *q_0 is the* initial state*;*
- *$g : Q \to \{\wedge, \vee, \mathbf{accept}, \mathbf{reject}\}$;*
- *Σ is the* tape alphabet*;*
- *$\delta \subseteq Q \times (\Sigma \cup \{\Box\}) \times Q \times \Sigma \times \{L, R\}$ is the* next move relation*.*

Here $\Box$ is a distinguished *blank* symbol, not belonging to Σ that represents unused parts of the tape. The function g partitions Q into four sets: the set $Q_U = \{q \in Q | g(q) = \wedge\}$ of *universal states*, the set $Q_E = \{q \in Q | g(q) = \vee\}$ of *existential states*, the set $Q_A = \{q \in Q | g(q) = \mathbf{accept}\}$ of *accepting states*, and the set $Q_R = \{q \in Q | g(q) = \mathbf{reject}\}$ of *rejecting states*.

Definition 3 (Configuration). *A* configuration *of an ATM AT is a string*

$$\bar{c} = (q_1, s_1, \ldots, q_n, s_n) \in ((Q \cup \{\bot\}) \cdot \Sigma)^*$$

such that exactly one index $j \in \{1, \ldots, n\}$ exists such that $q_j \neq \bot$.

Intuitively, $\bar{c} = (\bot, s_1, \ldots, \bot, s_{j-1}, q, s_j, \bot, s_{j+1}, \ldots, \bot, s_n)$ represents the global state of machine AT when n cells of the tape have been used, the head is on the j-th cell, the content of the tape is $s_1, \ldots, s_n$, and the finite control is in state q. We will denote by $\mathcal{GC}_{AT}$ the set of configurations of machine AT. A configuration is said to be *halting* (respectively, *existential*, *universal*) if it contains a halting (respectively, existential, universal) state. The *initial* configuration of AT with input $x = (x_1, \ldots, x_k)$ is $(q_0, x_1, \bot, x_2, \ldots, \bot, x_k)$.

With a slight abuse of notation, we will denote by $\delta(\bar{c})$ the set of configurations $\bar{c}$' such that $\bar{c}$ can evolve in one step into $\bar{c}$' according to the

relation δ. Whenever $\bar{c}' \in \delta(\bar{c})$ we will write $\bar{c} \vdash \bar{c}'$; let $\vdash^*$ be the transitive and reflexive closure of $\vdash$, let $\bar{c}_0(x)$ be the initial configuration of machine AT with input x, we will denote by $\mathcal{GC}_{AT(x)}$ the set $\{\bar{c} \in \mathcal{GC}_{AT} |\ \bar{c}_0(x) \vdash^* \bar{c}\}$ and call it the *computation tree* of AT with input x. In this paper we shall only consider *time-bounded* ATM's, that is, machines having a finite computation tree for any input.

Acceptance is defined in a quite involved way for general ATM's (see [5]). In the case of time bounded ATM's, however, a much simpler inductive definition can be given.

Definition 4 (Acceptance). *Let AT be a time-bounded ATM, x be a string, $\bar{c} \in \mathcal{GC}_{AT(x)}$ be a configuration.*

1. *If $\bar{c}$ is a halting configuration, then we say that $\bar{c}$ is an* accepting *configuration if it contains an accepting state, otherwise we say that it is a* rejecting *configuration.*
2. *If $\bar{c}$ is a universal configuration, then we say that it is accepting if all the configurations in $\delta(\bar{c})$ are accepting, otherwise we say that it is rejecting.*
3. *If $\bar{c}$ is an existential configuration, then we say that it is accepting if at least one configuration in $\delta(\bar{c})$ is accepting, otherwise we say that it is rejecting.*

We say that AT *accepts* input x if the initial configuration of AT with input x is accepting. A language L is *decided* by an alternating Turing machine AT if AT accepts x if and only if $x \in L$. The following theorem has been proved by Chandra, Kozen, and Stockmeyer.

Theorem 2 ([5]). *Every language $L \in$ LIN-EXP is decidable by an alternating Turing machine AT_L working with linear space and exponential time.*

With standard techniques from the theory of Turing machines, we may assume without loss of generality that AT_L is such that, for any input x of size n, only the cells of the tape containing x are accessed, all the computation paths of $AT(x)$ have the same length and if $\bar{c} \in \mathcal{GC}_{AT(x)}$ is a universal (respectively, existential) configuration, then all the configurations in $\delta(\bar{c})$ are existential (respectively, universal). In the following, we shall call such a machine a *canonical linear-space alternating machine*.

4 The EXP-completeness Result

In the following we shall prove the following result.

Lemma 1. *Let AT be a canonical linear space ATM. Then, for any string x of length n, we can compute, in time polynomial in n, two processes P and Q of $\mathcal{L}_r$ such that $P \sim Q$ if and only if x is accepted by AT.*

In order to prove the above lemma, we shall define three identifiers A, S, F[1]. As a first approximation, if $\bar{c}$ is a configuration of AT, then $A(\bar{c})$ is a process that *simulates* the computation of AT starting from configuration $\bar{c}$. In particular, if $\bar{c}_0$ is the starting configuration of AT with input x, then the labeled transition system of $A(\bar{c}_0)$ is " isomorphic" to $\mathcal{GC}_{AT(x)}$: there is a correspondence between configurations $\bar{c} \in \mathcal{GC}_{AT(x)}$ and processes $A(\bar{c})$, and between nondeterministic branching of the ATM and nondeterministic choice in the process. Furthermore, the processes corresponding to halting configurations $\bar{c} \in \mathcal{GC}_{AT(x)}$ can do a single action: a if $\bar{c}$ is accepting and b if $\bar{c}$ is rejecting. S (respectively, F) is defined to be identical to A, except that states corresponding to halting configurations always do a (respectively b). Thus, intuitively, A would be bisimilar to S in case AT accepts, and bisimilar to F otherwise.

Indeed, the above straightforward construction fails to express alternation of quantifiers in terms of bisimulation, and has to be slightly modified. For example, assume that an existential configuration $\bar{c}$ of $AT(x)$ branches into two configurations, one accepting and one rejecting. Then $\bar{c}$ is accepting, and we would like the corresponding process $A(\bar{c})$ to be bisimilar to $S(\bar{c})$. But $A(\bar{c})$ branches into both a rejecting and an accepting state, while $S(\bar{c})$ branches into two accepting states: thus $A(\bar{c})$ and $S(\bar{c})$ could not be bisimilar. This inconvenience can be overcome if we assume that each state corresponding to an existential (respectively, universal) configuration always branches into at least one state corresponding to a rejecting (respectively, accepting) configuration.

The actual definition of identifiers A, S, and F is given in Tables 2–4. It is convenient to split into three lemmas the proof that those processes indeed exhibit the desired behaviour.

Since no confusion can arise, in the following we will use $\mathcal{GC}$ as a shorthand for $\mathcal{GC}_{AT(x)}$.

Lemma 2. *Let $\bar{c}_1, \bar{c}_2 \in \mathcal{GC}$ be any two (not necessarily distinct) configurations that halt within the same number of steps, then the following holds.*

[1] A stands for ATM, S for *success* and F for *failure*.

$$
\begin{aligned}
A(x_1, y_1, \ldots, x_n, y_n) \Leftarrow\ & \\
& \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_E \\ \langle q,s,q',s',L\rangle\in\delta}} [x_i = s][y_i = q]a.A(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
+ & \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_E \\ \langle q,s,q',s',R\rangle\in\delta}} [x_i = s][y_i = q]a.A(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_U \\ \langle q,s,q',s',L\rangle\in\delta}} [x_i = s][y_i = q]a.A(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
+ & \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_U \\ \langle q,s,q',s',R\rangle\in\delta}} [x_i = s][y_i = q]a.A(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{q\in Q_A} [y_i = q]a \\
+ & \sum_{q\in Q_R} [y_i = q]b
\end{aligned}
$$

Table 2. Definition of A.

$$
\begin{aligned}
S(x_1, y_1, \ldots, x_n, y_n) \Leftarrow & \\
& \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_E \\ \langle q,s,q',s',L\rangle\in\delta}} [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
+ & \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_E \\ \langle q,s,q',s',R\rangle\in\delta}} [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{\substack{i\in\{1,\ldots,n\},\ q\in Q_U \\ \langle q,s,q',s',L\rangle\in\delta}} [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
& \quad + [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{q\in Q_A\cup Q_R} [y_i = q]a
\end{aligned}
$$

Table 3. Definition of S.

1. $S(\bar{c}_1) \not\sim F(\bar{c}_2)$.
2. $S(\bar{c}_1) \sim S(\bar{c}_2)$ *and* $F(\bar{c}_1) \sim F(\bar{c}_2)$.

Lemma 3. *For any $\bar{c} \in \mathcal{GC}$, either $A(\bar{c}) \sim S(\bar{c})$ or $A(\bar{c}) \sim F(\bar{c})$.*

The above two lemmas can be proved by induction on the number of steps required to move from $\bar{c}_1$ (respectively, $\bar{c}$) to a halting configurations. Canonicity plays an essential role in the proofs.

Lemma 4. *For any $\bar{c} \in \mathcal{GC}$, $A(\bar{c}) \sim S(\bar{c})$ if and only if $\bar{c}$ is an accepting configuration.*

Proof. We proceed again by induction on the number of residual steps. If $\bar{c}$ is a halting configuration, then the proof is trivial.

Otherwise, let us assume that, for any $\bar{c}' \in \delta(\bar{c})$, $\bar{c}'$ is an accepting configuration if and only if $A(\bar{c}') \sim S(\bar{c}')$. We have to distinguish two cases: either $\bar{c}$ is existential or it is universal. We show only the first case, as the second one is very similar.

Since $\bar{c}$ is existential, then it is accepting if and only if $\delta(\bar{c})$ contains at least one accepting configuration. By induction hypothesis, the latter

$$
\begin{aligned}
F(x_1, y_1, \ldots, x_n, y_n) \Leftarrow \\
& \sum_{\substack{i \in \{1,\ldots,n\},\ q \in Q_E \\ \langle q,s,q',s',L\rangle \in \delta}} [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
+ & \sum_{\substack{i \in \{1,\ldots,n\},\ q \in Q_E \\ \langle q,s,q',s',R\rangle \in \delta}} [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{\substack{i \in \{1,\ldots,n\},\ q \in Q_U \\ \langle q,s,q',s',L\rangle \in \delta}} [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
& \qquad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, q', s', \bot, x_{i+1}, y_{i+1}, \ldots, x_n, y_n) \\
+ & \sum_{\substack{i \in \{1,\ldots,n\},\ q \in Q_U \\ \langle q,s,q',s',R\rangle \in \delta}} [x_i = s][y_i = q]a.S(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
& \qquad + [x_i = s][y_i = q]a.F(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, s', \bot, x_{i+1}, q', \ldots, x_n, y_n) \\
+ & \sum_{q \in Q_A \cup Q_R} [y_i = q]b
\end{aligned}
$$

Table 4. Definition of F.

statement holds if and only if:

$$\text{there is } \bar{c}' \in \delta(\bar{c}) \text{ s.t. } A(\bar{c}') \sim S(\bar{c}'). \tag{1}$$

We now prove that (1) holds if and only if $A(\bar{c}) \sim S(\bar{c})$. Let us assume that (1) holds. There are two non-trivial cases.

First, consider the case when $A(\bar{c}) \stackrel{a}{\longrightarrow} A(\bar{c}_1)$, for any $\bar{c}_1 \in \delta(\bar{c})$. From canonicity and Lemma 3, it is $A(\bar{c}_1) \sim S(\bar{c}_1)$ or $A(\bar{c}_1) \sim F(\bar{c}_1)$: in the first case the matching move for $S(\bar{c})$ is $A(\bar{c}) \stackrel{a}{\longrightarrow} S(\bar{c}_1)$, in the second case it is $S(\bar{c}) \stackrel{a}{\longrightarrow} F(\bar{c}_1)$. Consider the case when $A(\bar{c})$ has to match a transition $S(\bar{c}) \stackrel{a}{\longrightarrow} S(\bar{c}_1)$ from $A(\bar{c})$. Then we have:

$$A(\bar{c}) \stackrel{a}{\longrightarrow} A(\bar{c}') \sim S(\bar{c}') \sim S(\bar{c}_1)$$

where the first $\sim$ follows from (1) and the second one from canonicity and Lemma 2.

Conversely, assume that $A(\bar{c}) \sim S(\bar{c})$. Take any $c_1 \in \delta(\bar{c})$ and consider the transition $S(\bar{c}) \stackrel{a}{\longrightarrow} S(\bar{c}_1)$. Then we must have for some $\bar{c}'$ $A(\bar{c}) \stackrel{a}{\longrightarrow} K(\bar{c}') \sim S(\bar{c}_1)$, where $K = F$ or $K = A$. The case $K = F$ cannot arise, due to canonicity and Lemma 2. Thus it must be $K = A$. From canonicity and Lemma 2 it follows $A(\bar{c}') \sim S(\bar{c}_1) \sim S(\bar{c}')$, which validates (1). □

Proof. (Of Lemma 1) Let $\bar{c}_0$ be the initial configuration of AT with input x. Since the definition of the identifiers can be clearly constructed in time polynomial in n, the lemma follows by setting $P \stackrel{\text{def}}{=} A(\bar{c}_0)$ and $Q \stackrel{\text{def}}{=} S(\bar{c}_0)$. □

Lemma 1 and the results recalled in the previous section immediately imply the EXP-hardness of $\mathcal{L}_r$.

Theorem 3. *The bisimilarity problem in the language $\mathcal{L}_r$ is* EXP*-hard.*

We shall indeed see in the next section that the problem is in EXP. From the above theorem and from Theorem 1 the intractability of bisimilarity in $\mathcal{L}_r$ follows.

Corollary 1. *A constant $c > 0$ exists such that any algorithm that decides the bisimilarity problem in $\mathcal{L}_r$ has a worst-case running time no better than 2^{n^c}, where n is the size of the input.*

5 The Parallel Composition Operator

In this section we consider adding the *parallel composition* operator $\mid$ (see e.g. [12]) to the language described in Section 2. We will show that, for a certain restricted syntactic format, the bisimilarity problem with parallel composition is decidable and EXP-complete. As a consequence, the bisimilarity problem is in EXP for all the fragments we have considered in the paper.

The syntax of the language $\mathcal{L}_{v,r}$ is extended with the clause

$$P ::= P \mid P \,.$$

All definitions and notions given for $\mathcal{L}_{v,r}$ (such as free variables, subterms etc.) are extended to the new language in the expected way. Following [12], the operational semantics of the new operator is given by the rules:

$$(Par)\frac{P_1 \stackrel{\mu}{\longrightarrow} P_1'}{P_1 \mid P_2 \stackrel{\mu}{\longrightarrow} P_1' \mid P_2} \qquad (Com)\frac{P_1 \stackrel{\mu}{\longrightarrow} P_1',\ P_2 \stackrel{\mu'}{\longrightarrow} P_2'}{P_1 \mid P_2 \stackrel{\tau}{\longrightarrow} P_1' \mid P_2'}$$

with ($\mu = a$ and $\mu' = \overline{a}$) or ($\mu = a(v)$ and $\mu' = \overline{a}v$), *plus* the symmetric versions of the above rules, where the roles of P_1 and P_2 are exchanged. Here $\tau \notin Act$ is a new kind of action, called the *silent* action.

An important class of parallel processes is that *finite-control* processes, where parallel composition never occurs inside recursive definitions. The corresponding sub-language is indicated with $\mathcal{L}_{v,r,p}$, while the sublanguage of $\mathcal{L}_{v,r,p}$ without input/output primitives is indicated by $\mathcal{L}_{r,p}$. By confining ourselves to finite-control processes, we are able to extend a reduction from $\mathcal{L}_{v,r}$to $\mathcal{L}_r$ given in [3] and prove:

Proposition 1. *The bisimilarity problem in $\mathcal{L}_{v,r,p}$ is equivalent to the bisimilarity problem in $\mathcal{L}_{r,p}$, up to polynomial-time reduction.*

Observe that every process $P \in \mathcal{L}_{r,p}$, defined with respect to a set of identifier definitions Eq, has a finite transition system. More precisely, first note that the size of every term reachable from P cannot exceed k, where

$$k \stackrel{\text{def}}{=} |P| * \max\{|R| \,:\, R \text{ appears in } Eq\} \cup \{1\} \,.$$

Therefore, there are at most n^k different states in the transition system, where n is the number of distinct values appearing in P and in Eq. Both n and k are easily seen to be at most polynomial functions of the size of the problem. It follows that the bisimilarity problem in $\mathcal{L}_{r,p}$ can be solved

in exponential time using, for example, the algorithm by Page and Tarjan [14]. Putting together the latter fact, Theorem 3, the cited result of [3] and the above Proposition 1, we have the following result of equivalence between languages.

Theorem 4. *The bisimilarity problems for the languages* $\mathcal{L}_r$, $\mathcal{L}_{v,r}$, $\mathcal{L}_{r,p}$ *and* $\mathcal{L}_{v,r,p}$ *are all* EXP*-complete.*

It is worth to notice that, even in the absence of values, the presence of parallel composition implies an exponential blow-up of the number of states. This is implicitly present, for example, in the so-called "expansion law" [12]: $a \mid b \sim a.b + b.a$. The above theorem tells us that the computational complexity due to parallel composition itself is not greater than that caused by the handling of data-values.

6 Conclusions

We studied the complexity of deciding bisimulation equivalence over calculi with recursive definitions, value-passing, and/or limited parallel composition. All the considered calculi are EXP-complete, and thus provably intractable and computationally equivalent.

It would be interesting to consider a calculus $\mathcal{L}_{v,p}$ with value-passing and unrestricted parallel composition, but with no recursion. Classifying the computational complexity of this calculus would give some insight into the intrinsic hardness of parallel composition, in a setting where it would not be overwhelmed by the expressiveness of recursive definitions.

References

1. M. Boreale and R. De Nicola. A symbolic semantics for the π-calculus. Short version in *Proc. of CONCUR'94*, LNCS, Springer Verlag. Full version to appear on *Information and Computation*.
2. M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 2(120):279–303, 1995.
3. M. Boreale and L. Trevisan. On the complexity of bisimilarity for value-passing processes. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*. LNCS, Springer Verlag, 1995.
4. D.P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1993.
5. A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
6. J. Hartmanis and R.E. Stearns. On the computational complexity of algorithms. *Transactions of the AMS*, 117:285–306, 1965.
7. M. Hennessy and A. Ingolfsdottir. A theory of communicating processes with value passing. *Information and Computation*, 2(107):202–236, 1993.
8. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
9. B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite state programs. *Information and Computation*, 107:272–302, 1993.
10. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite sate processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
11. R. Milner. *A Calculus of Communicating Systems*. LNCS, 92. Springer-Verlag, Berlin, 1980.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
13. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 1 and 2. *Information and Computation*, 100:1–78, 1992.
14. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
15. J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1995.