Complexity and Decidability of Some Equivalence-Checking Problems

Zdeněk Sawa

Ph.D. Thesis
Faculty of Electrical Engineering and Computer Science
Technical University of Ostrava
2005

Acknowledgements

I would like to thank my supervisor Petr Jančar for guidance, comments and fruitful discussions. I owe much to him.

I would also like to thank all co-authors that worked with me on papers for their cooperation: Antonín Kučera, Faron Moller, and Martin Kot.

I would also like to thank Philippe Schnoebelen for drawing my attention to the complexity of equivalence checking on finite-state systems composed from communicating subsystems and to Rabinovich's conjecture.

Last but not least I would like to thank my wife Silvie for her love and support.

Declaration

I declare that this thesis was composed by myself, and all presented results are my own, unless otherwise stated.

Some of the material has been previously published in [32], [53], [30], [52], [31], and [35].

Zdeněk Sawa

Abstract

The thesis presents results obtained by the author in the area of verification of finite-state and infinite-state systems. It concentrates on questions of complexity and decidability of equivalence checking, i.e., of deciding behavioural equivalences and preorders on transition systems.

It is shown that deciding of any relation between bisimulation equivalence and trace preorder is a PTIME-hard problem for finite-state systems that are given explicitly as a list of states and transitions. The problem becomes EXPTIME-hard for any such relation in the case of finite-state systems that are created as a composition of communicating finite-state components.

The other type of systems studied in the thesis are one-counter automata. It is shown that deciding simulation equivalence is an undecidable problem on one-counter automata. A general method for proving DP-hardness of some problems concerning one-counter automata is presented. Using this method is shown that deciding any relation between bisimulation equivalence and simulation preorder is DP-hard for one-counter nets (i.e., one-counter automata that cannot test for zero), and that deciding simulation equivalence and simulation preorder between a one-counter automaton and a finite-state system is DP-hard.

The last type of systems studied in the thesis are Basic Parallel Processes (BPP). Two polynomial time algorithms for BPP are presented. The first of these algorithms decides bisimulation equivalence between a BPP and a finite-state system, and the other decides distributed bisimilarity on BPP.

Abstrakt

Tato disertační práce prezentuje výsledky dosažené autorem v oblasti verifikace konečně stavových a nekonečně stavových systémů. Zaměřuje se na otázky výpočetní složitosti a rozhodnutelnosti equivalence checking, to jest problémů rozhodování behaviorálních ekvivalencí a kvaziuspořádání na přechodových systémech.

Je ukázáno, že rozhodování libovolné relace, která leží mezi bisimulační ekvivalencí a trace preorder je PTIME-těžký problém pro systémy, které jsou dány explicitně jako seznam stavů a přechodů. Tento problém se pro libovolnou z těchto relací stává EXPTIME-těžkým v případě konečně stavových systémů, které jsou vytvořeny z komunikujících konečně stavových komponent.

Dalším typem systémů zkoumaným v této práci jsou automaty s jedním čítačem. Je ukázáno, že rozhodování simulační ekvivalence pro automaty s jedním čítačem je nerozhodnutelné. Dále je prezentována obecná metoda pro dokazování DP-obtížnosti problémů týkajících se automatů s jedním čítačem. Použitím této metody je ukázáno, že rozhodování libovolné relace, která leží mezi bisimulační ekvivalencí a simulačním kvaziuspořádáním je DP-těžké pro one-counter nets (tj. pro automaty s jedním čítačem, které nemohou testovat nulu), a dále, že rozhodování simulační ekvivalence a simulačního kvaziuspořádání mezi automatem s jedním čítačem a konečně stavovým systémem je DP-těžké.

Posledním typem systémů zkoumaným v této práci jsou Basic Parallel Processes (BPP). Jsou ukázány dva polynomiální algoritmy pro BPP. První z těchto algoritmů rozhoduje bisimulační ekvivalenci mezi BPP a konečně stavovým systémem a druhý rozhoduje distribuovanou bisimilaritu na BPP.

Contents

Chapter 1

Introduction

We can find many examples of complicated software and hardware systems where bugs can have serious or even catastrophic consequences. Examples of such systems are operating systems, network communication protocols, microprocessors, and traffic control systems.

One of the main problems in the design of such complicated systems is to ensure the correctness of this design. Correctness means that the system fulfills the task for which it was designed. We usually have some *specification* of the desired behavior of the system and we want to ensure that the *implementation* of the system is correct with respect to this specification. The process of checking whether the given implementation satisfies the given specification is called *verification*.

Standard techniques used for verification are *testing* and *simulation*. In testing we run the system in different situations and with different inputs and observe the behavior of the system. Simulation is similar to testing but we do not test the actual system, but some model of it. It is used in situations when running the actual system is not possible or practical, for example due to enormous costs.

While testing and simulation are very useful in the early stages of development and allow to discover many bugs in the system, they have the important disadvantage that one can never be sure that there are not some other more subtle bugs in the system. Testing and simulation can never guarantee the correctness of the system due to their inherent limitations because they can explore only *some* of possible behaviors of the system, but the number of *all* possible behaviors is usually very large and often infinite.

The limitations of testing and simulation become even more severe in the design of systems composed of many components running concurrently that can interact with each other. The behavior of such systems is usually non-deterministic, and it can be difficult even to reproduce bugs in these systems since they can occur only under some rare circumstances.

It becomes obvious that some *formal methods* that ensure correctness of *all* possible behaviors should be used for verification.

The formal methods provide us with the necessary mathematical tools that can be used in the construction of rigorous mathematical proofs of the correctness of the system. The construction of such proofs can be done either by hand or may be automated, at least partially, by use of some sort of verification software tools. The latter approach is very attractive since the construction of proofs by hand is usually very tedious and error prone. The approach using automated tools is usually called *computer aided verification*.

Unfortunately this task can not be fully automated in full generality because many problems concerning behavior of computer programs are undecidable. For example, it is well known that Halting problem, i.e., the problem whether a program halts for a given input after some finite number of steps, is undecidable.

In general, there are two main approaches to verification that allow to ensure correctness for all possible behaviors of the system – *theorem proving* and techniques based on *model checking* and *equivalence checking*.

In theorem proving, we try to construct formal proofs of correctness of the system. Computer programs called *theorem provers* can assist the user in this task and do some simple steps automatically. However, the user has to guide the program and do the crucial steps of the proofs. The main disadvantage of this technique is that it requires a lot of knowledge, skill and practice from the user. This severely limits the practical applicability of theorem proving.

Other approaches are model checking and equivalence checking. These techniques are fully automatic and do not require any interaction from the user, however they can not be applied to arbitrary programs due to undecidability of Halting problem. Instead, some properties of models that do not have the expressive power of Turing machines are verified. These techniques have their origins in the automata and formal language theory where infinite languages are described finitely and some properties of these languages are decidable, for example, it is decidable whether two finite automata recognize the same language.

In *model checking* we have given a system (resp. description of the system) and some desired property of the system expressed as a formula of some temporal logic, and the question is whether the system satisfies the given property. See [15, 17, 57, 6] for more information about model checking and temporal logic.

Equivalence checking is type of problems where we have given (descriptions of) two systems and the question is whether these systems are equivalent with respect to some notion of equivalence. Usually one of these systems is a specification and the other is an implementation and we want to ensure that their behavior is identical. Equivalence checking problems are the main topic of this thesis.

There are many variants of model checking and equivalence checking problems that differ in the way how the systems and their properties are expressed. Models with a great expressive power cannot be verified automatically, and models that are too restrictive do not allow to model many aspects of real systems. This motivates the research that concentrates on decidability and complexity of verification problems. One active area of research concentrates on the question which model checking and equivalence checking problems are decidable, and where exactly lies the dividing line between decidable and undecidable problems. Another important question is what is the exact computational complexity of decidable verification problems, because many verification problems can be solved by some algorithm theoretically, but the algorithm can be used in practice only for small instances due to its computational complexity.

One of the most serious obstacles in the design of efficient verification algorithms is the phenomenon known as 'state explosion'. This problem appears when we have a system composed of many components. These components can have reasonably small state spaces, but the state space of the whole system can be exponentially larger with respect to the size of descriptions of its components. Unfortunately it shows up that the state explosion is unavoidable in many cases and that algorithms solving such problems require exponential time.

This thesis concentrates on complexity and decidability of some equivalence checking problems, and presents some results obtained by the author in this area. Some of the results presented here are joint work with other authors – Petr Jančar, Antonín Kučera, Faron Moller, and Martin Kot.

It is assumed that the reader is familiar with formal languages and the basics of complexity theory.

1.1 Goals of the Thesis

The main goal of the thesis was to contribute with some new results in the area of equivalence checking. The main focus was on the decidability and computational complexity of equivalence-checking problems. The thesis presents results obtained by the author in this area.

The solved problems are more or less independent of each other and can be divided into three main groups:

- problems concerning finite-state systems – either given explicitly or as a parallel composition of communicating components
- problems concerning one-counter automata and a variant of them called one-counter nets
- problems concerning Basic Parallel Processes (BPP)

The next section gives an overview of the results presented in this theses. See Chapter 2 for formal definitions of terms used in this section.

1.2 Overview of the Results

The following subsections describe shortly the main results presented in this thesis.

1.2.1 Finite-State Systems

Finite-State Systems are one of the simplest type of systems. They can be given either explicitly (as an explicit list of states and transitions), or they can be described as a composition of explicitly given communicating finite-state systems. The systems of the former type are called *explicit* finite-state system in the thesis, and the latter are called *composed* finite-state systems. An example of a composed finite-state system is a parallel composition of explicit finite-state systems that synchronize on common actions and that use hiding of actions (they are called Parallel Composition with Hiding (PCH) in this thesis), or 1-safe Petri nets.

Chapter 3 contains proofs of lower bounds of the complexity of equivalence-checking on explicit and composed systems. These lower bounds are not specific for one type of equivalence, but apply to whole spectrum of relations between bisimulation equivalence and trace preorder. This spectrum includes almost all types of equivalences considered in the literature that are useful in practice.

It is shown at first that deciding any relation between bisimulation equivalence and trace preorder on explicit systems is PTIME-hard. This result was proved in [53], however the proof presented in this thesis uses a different and simpler construction that was published in [52]. This construction was used in [52] as one part of the proof concerning composed systems. It was shown there that deciding any relation between bisimulation equivalence and trace preorder is EXPTIME-hard for many types of composed systems including PCH and 1-safe Petri nets. This result was conjectured for PCH by A. Rabinovich in [51], and the result from [52] approves his conjecture. To simplify the proof, new model of computation called Reactive Linear Bounded Automata (RLBA) was introduced. It was shown that deciding any relation between bisimulation equivalence and trace preorder is EXPTIME-hard for RLBA. Since RLBA can be "implemented" (i.e., its behavior can be easily simulated) by different types of composed systems, the EXPTIME-hardness result extends to all such types of systems. (However there is one notable exception, parallel composition of finite-state systems that synchronize on common actions where hiding of actions is not allowed. Such systems are not powerful enough to simulate RLBA, and deciding for example trace equivalence is PSPACE-complete for them. [54])

1.2.2 One-Counter Automata

One-Counter Automata (OCA) are like finite-state systems extended with a counter containing a non-negative integer value. This counter can be incremented and decremented by one. This kind of machines can test whether the value of the counter is zero or non-zero and perform actions depending on that. There is a variant of one-counter automata called One-Counter Nets (OCN). One-counter nets can not test for zero value of the counter, they can only test for non-zero value. One-counter nets are equivalent (up to isomorphism) with Petri nets with (at most) one unbounded place.

It is shown in Chapter 4 that deciding simulation equivalence and simulation preorder is undecidable problem for one-counter automata. This result was proved in [32]. The rest of the chapter is devoted to presentation of a general method for proving DP-hardness of different problems concerning one-counter automata published in [30] and [31]. This method uses a fragment of Presburger arithmetic called One Counter Logic (OCL). This fragment is chosen in such a way that it is possible to reduce the problem of deciding the truth of a formula of OCL to some problems concerning one-counter automata. The constructions in these reductions proceed by induction on the structure of a formula. It is proved that the problem of deciding truth of an OCL formula is DP-hard. This implies that the problems to which this problem can be reduced are also DP-hard.

This technique was used to show DP-hardness of deciding any relation between bisimulation equivalence and simulation preorder on one-counter nets, and to show DP-hardness of deciding simulation preorder and simulation equivalence of a one-counter automaton and a finite-state system. (The same technique was also used in [31] by A. Kučera to show DP-hardness of model checking the logic EF (fragment of CTL) on one-counter nets, however this result is not presented in this thesis, see [31] for more information.)

1.2.3 Basic Parallel Processes

Basic Parallel Processes (BPP) are a very natural subclass of infinite state systems. It was proved by P. Jančar in [25] that deciding bisimulation equivalence on BPP is PSPACE-complete. The technique used in this paper was used to show two results concerning BPP presented in this thesis in Chapter 5.

The first of these results is a polynomial time algorithm for deciding bisimulation equivalence between a BPP and a finite-state system with time complexity $O(n^4)$ where n is the size of the instance. This result was presented in [35].

The second of these results is a polynomial time algorithm for deciding distributed bisimilarity on BPP with time complexity $O(n^3)$. A polynomial time algorithm for this problem was already presented by Lasota in [40], however the algorithm presented here is simpler, more efficient and provides an explicit upper bound on the complexity of the problem. (No degree of the polynomial was specified in the proof in [40].) The new algorithm for this problem presented in this thesis was not published yet.

1.3 Layout of the Thesis

Chapter 2 provides necessary basic definitions. Problems concerning finite-state systems (both explicit and composed) are studied in Chapter 3. Chapter 4 concentrates on problems concerning one-counter automata, and Chapter 5 on problems concerning Basic Parallel Processes. Chapter 6 contains conclusion and an overview of results presented in this thesis. Appendix A contains a list of publications of the author.

Because the types of problems solved in Chapters 3, 4, and 5 are more or less independent, each of these chapters contains its own section called 'State of the Art' which describes known results that are relevant for the given chapter.

Chapter 2

Definitions

This chapter contains some basic definitions that are used in the remaining chapters. Section 2.1 presents notation conventions used in the thesis. Section 2.2 describes the notion of labelled transition systems, a formalism that underlies different types of models described in the following sections – process rewrite systems (Section 2.3), process algebra, BPA and BPP (Section 2.4), Petri nets (Section 2.5), one-counter automata (Section 2.6), and explicit and composed finite state systems (Section 2.7). The remaining two sections describe different types of behavioural equivalences on labelled transition systems. Section 2.8 describes equivalences from linear time – branching time spectrum, and Section 2.9 describes distributed bisimilarity – one of non-interleaving equivalences.

2.1 Notation Conventions

The following notation conventions are used in the rest of the thesis.

$\mathbb{N}$ denotes the set of non-negative integers, i.e., the set $\{0, 1, 2, \ldots\}$. The symbol ω denotes infinity. We define $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$.

$[x, y]$, where $x, y \in \mathbb{N}$, denotes the set of integers between (and including) x and y, i.e., the set $\{z \mid x \leq z \leq y\}$.

Let X be a set. $|X|$ denotes the cardinality of X.

$X \subsetneq Y$ denotes that X is a *proper* subset of Y, while $X \subseteq Y$ allows equality. $\mathcal{P}(X)$ denotes the power-set of X, i.e., the set $\{Y \mid Y \subseteq X\}$.

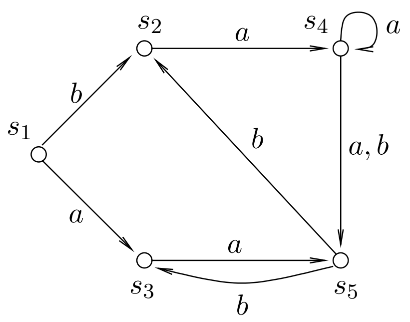


Figure 2.1: Example of an LTS

X^* denotes the set of *finite* sequences of elements from X. Let $x \in X^*$ be a sequence. The *length* of x is the number of its elements denoted $|x|$. We use $x(i)$ to denote i-th element of x, i.e., $x(i) = x_i$ for $x = x_1 x_2 x_3 \cdots$.

Let X be a set. *Partition* $\mathcal{X}$ of X is a set $\mathcal{X} = \{X_1, X_2, \ldots, X_l\}$ of disjoint non-empty *classes* whose union is X.

2.2 Labelled Transition Systems

There are many possible ways how systems can be described, for example different types of automata, process rewriting systems, process algebras, or Petri nets. However there is one common concept underlying all these formalisms. It is the concept of a *labelled transition system* (LTS).

Formally, a labelled transition system is a triple $(S, \mathit{Act}, \longrightarrow)$ where:

- S is a set of *states*,
- Act is a finite set of *actions*, and
- $\longrightarrow \subseteq S \times \mathit{Act} \times S$ is a *transition relation.*

Informally, S is the set of all possible states of the system, Act is a set of names of externally observable actions which can be performed by the system, and the transition relation represents behaviour of the system. Instead of $(s, a, s') \in \longrightarrow$ we usually write $s \xrightarrow{a} s'$, and this can be interpreted as that the system in the state s can perform the action a and go to the state s'.

See Figure 2.1 for an example of an LTS where $S = \{s_1, s_2, s_3, s_4, s_5\}$, $Act = \{a, b\}$, and the transition relation contains the following transitions:

$$
\begin{array}{lll}
s_1 \xrightarrow{a} s_3 & s_3 \xrightarrow{a} s_5 & s_4 \xrightarrow{b} s_5 \\
s_1 \xrightarrow{b} s_2 & s_4 \xrightarrow{a} s_4 & s_5 \xrightarrow{b} s_2 \\
s_2 \xrightarrow{a} s_4 & s_4 \xrightarrow{a} s_5 & s_5 \xrightarrow{b} s_3
\end{array}
$$

The set of states S in a labelled transition system can be finite or infinite. Labelled transition systems where S is finite are called *finite-state* labelled transition systems, and labelled transition systems where S is infinite are called *infinite-state* labelled transition systems. Finite-state systems are the simplest type of systems. The sets of states and transitions in such systems are finite and can be given explicitly. On the other hand, it is not possible to work directly with infinite-state systems. Instead we must work with some finite representations of them. Examples of such representations are different kinds of automata, Petri nets, process algebras, and process rewrite systems.

The notation $s \xrightarrow{a} s'$ can be extended in a natural way to sequences of actions. Let $w \in Act^*$. We write $s \xrightarrow{w} s'$ iff there is a sequence of states $s_0, s_1, \ldots, s_n$ such that $s = s_0$, $s' = s_n$, and $s_{i-1} \xrightarrow{w(i)} s_i$ for each i such that $1 \leq i \leq n$. Recall that $w(i)$ denotes i-the symbol of w.

A state s' is *reachable* from a state s, written $s \longrightarrow^* s'$, iff there is some $w \in Act^*$ such that $s \xrightarrow{w} s'$.

A labelled transition system is *deterministic* if for every $s \in S$ and every $a \in Act$ there is at most one s' such that $s \xrightarrow{a} s'$. System that is not deterministic is *nondeterministic*.

A labelled transition systems is *image-finite* iff for every $s \in S$ and every $a \in Act$ the set $\{s' \mid s \xrightarrow{a} s'\}$ is finite.

Sometimes a labelled transition system has a distinguished *initial* state or more generally a set of initial states.

Besides observable actions in Act, there can be a *invisible* action denoted τ with a special semantics. We define $\tau \notin Act$ for any set of actions Act. In labelled transition systems where τ actions occur, the transition relation is defined as a subset of $S \times (Act \cup \{\tau\}) \times S$ where S is the set of states. Again we use the notation $s \xrightarrow{\tau} s'$ instead of $(s, \tau, s') \in \longrightarrow$.

2.3 Process Rewrite Systems

Process Rewrite Systems (PRS) defined by Mayr in [43] provide a unified view of many formalisms presented in the following sections.

Process rewrite systems are defined as follows. Let $Act = \{a, b, c, \ldots\}$ be a countably infinite set of atomic actions and $Var = \{X, Y, Z, \ldots\}$ be a countably infinite set of process variables. Process terms are defined by the following abstract grammar

$$P \ ::= \ \varepsilon \ \mid \ X \ \mid \ P_1.P_2 \ \mid \ P_1 \parallel P_2$$

where ε is the empty term, X is a process variable, and where '.' denotes sequential composition and '$\parallel$' parallel composition. Sequential composition is associative and parallel composition is associative and commutative. We always work with equivalence classes of terms modulo associativity of sequential composition and modulo associativity and commutativity of parallel composition. We also define that $\varepsilon.P = P.\varepsilon = P$ and $P \parallel \varepsilon = P$.

Process rewrite system is a finite set of rules Δ containing rules of the form $t_1 \xrightarrow{a} t_2$ where t_1 and t_2 are process terms and $a \in Act$ is an atomic action. Let $Var(\Delta)$ be the set of process variables occurring in Δ and let $Act(\Delta)$ be the set of atomic actions occurring in Δ.

Process rewrite system Δ produces a corresponding labelled transition system $(S, Act', \longrightarrow)$ where S is the set of process terms that contain only variables from $Var(\Delta)$, $Act' = Act(\Delta)$, and the transition relation is the smallest relation satisfying the following inference rules where t_1, t_2, t_1', t_2' are process terms:

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Delta}{t_1 \xrightarrow{a} t_2}$$

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'}$$

Note that $Var(\Delta)$ and $Act(\Delta)$ are finite. Since Δ is finite, the generated labelled transition system is finitely branching, which means that the branching-degree is finite in every state, however it can be be arbitrarily high, i.e., it is possible that there is no finite constant depending only on Δ bounding the branching-degree in all states.

It is worth mentioning that process rewrite systems are not Turing powerful because for example the reachability problem is decidable for them [43].

Note also that there is no operator for non-deterministic choice ('+'), because nondeterminism can be encoded in the set of rules Δ which can contain more rules with the same term on the left side.

There can be defined different types of subclasses of process rewrite systems. At first we distinguish four classes of process terms:

1 – terms consisting of a single process variable (e.g., X),

$\mathcal{S}$ – terms consisting of ε, a single variable, or a sequential composition of process variables (e.g., $X.Y.Z$),

$\mathcal{P}$ – terms consisting of ε, a single variable, or a parallel composition of process variables (e.g., $X \parallel Y \parallel Z$),

$\mathcal{G}$ – any process terms without any restriction (e.g., $(X \parallel Y).Z$).

Obviously $1 \subsetneq \mathcal{S}$, $1 \subsetneq \mathcal{P}$, $\mathcal{S} \subsetneq \mathcal{G}$, and $\mathcal{P} \subsetneq \mathcal{G}$. Classes $\mathcal{S}$ and $\mathcal{P}$ are incomparable and $\mathcal{S} \cap \mathcal{P} = 1 \cup \{\varepsilon\}$.

Let $\alpha, \beta \in \{1, \mathcal{S}, \mathcal{P}, \mathcal{G}\}$ be classes of process terms such that $\alpha \subseteq \beta$. We define (α, β)-PRS as a finite set of rules Δ where in every rewrite rule $(l \xrightarrow{a} r) \in \Delta$ the term l is from class α and $l \neq \varepsilon$ and the term r is from class β (and r can be ε).

The hierarchy of (α, β)-PRS models is depicted in Figure 2.2. Each model in the hierarchy has a name shown also in the figure and many of these (α, β)-PRS correspond to well-known classes of infinite state systems studied in the literature. A line from a higher model to a lower model means that the higher model is more general than the lower one. It is known that the hierarchy is strict with respect to bisimilarity [43].

The classes of process rewrite systems correspond to the following following formalisms:

FS – finite-state systems,

BPA – Basic Process Algebra [7], also called *context-free processes*,

BPP – Basic Parallel Processes [13],

PDA – Pushdown Automata, also called *pushdown processes* or *pushdown systems*,

PA – Process Algebra [4],

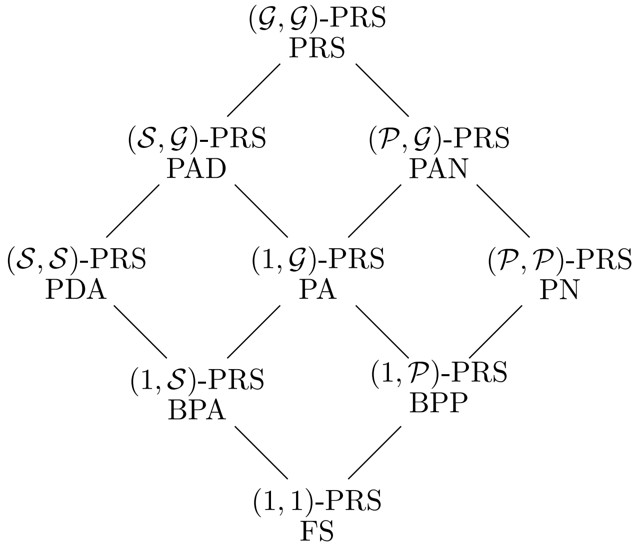


Figure 2.2: Hierarchy of process rewrite systems

PN – Petri nets

PRS – Process Rewrite Systems

Class PAD was introduced in [42] as the 'smallest' common generalization of classes PDA and PA. Similarly class PAN was introduced in [41] as the 'smallest' common generalization of classes PA and PN.

2.4 Process Algebra, BPA, and BPP

Process Algebra (PA) was introduced in [4]. It is defined as follows. Let $Act = \{a, b, c, \ldots\}$ be a countably infinite set of atomic actions and let $Var = \{X, Y, Z, \ldots\}$ be a countably infinite set of process variables. The class of PA expressions is defined by the following abstract syntax:

$$P \ ::= \ \mathbf{0} \ | \ X \ | \ a.P \ | \ P_1 + P_2 \ | \ P_1.P_2 \ | \ P_1 \parallel P_2$$

where $\mathbf{0}$ denotes the empty process, X is a process variable, $a.P$ is an action prefix, '+' denotes non-deterministic choice, '.' sequential composition, and '$\parallel$' parallel composition.

We work with expressions modulo associativity and commutativity of non-deterministic choice and parallel composition, and modulo associativity of sequential composition. We also define $P.\mathbf{0} = \mathbf{0}.P = P$ and $P \parallel \mathbf{0} = P$.

A *PA-process* is defined by a finite family of recursive equations

$$\Delta = \{X_i := P_i \mid 1 \leq i \leq n\}$$

where all X_i are distinct and all P_i are PA expressions containing variables only from $Var(\Delta)$, where $Var(\Delta)$ denotes the set $\{X_1, X_2, \ldots, X_n\}$. The set of actions occurring in Δ is denoted $Act(\Delta)$. It assumed that every occurrence of a variable in the P_i is *guarded*, i.e., that it is within the scope of an action prefix.

The set of equations Δ produces a labelled transition system $(S, Act', \longrightarrow)$ where S is the set of PA expressions, $Act' = Act(\Delta)$, and transition relation is the least relation satisfying the following inference rules:

$$a.P \xrightarrow{a} P \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q} \qquad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$$

$$\frac{P \xrightarrow{a} P'}{X \xrightarrow{a} P'} \text{ (where } (X := P) \in \Delta)$$

Sometimes also the 'left merge' operator $\mathbin{\|\!\!\lfloor}$ is included in the definition with the following semantics:

$$\frac{P \xrightarrow{a} P'}{P \mathbin{\|\!\!\lfloor} Q \xrightarrow{a} P' \parallel Q}$$

It resembles parallel composition but only the 'left' of the processes can proceed by performing an action.

A PA-process is in *normal form* if all its equations are of the form

$$X_i = \sum_{j=1}^{n_i} a_{ij} P_{ij}$$

where $1 \leq i \leq n$, $n_i \in \mathbb{N}$, $a_{ij} \in Act$, and P_i are PA expression not containing any non-deterministic choice ('+') or action prefix. Any PA-process can be effectively transformed to the normal form as was proved in [8].

Remark. The term 'Process Algebra' is nowadays also used in a much wider sense denoting also other formalisms such as for example CCS [44].

There are two natural subclasses of PA-processes – Basic Process Algebra (BPA) [7] and Basic Parallel Processes (BPP) [13].

BPA is a subclass of PA where no parallel composition ('$\|$') is allowed. Such systems are also called *context-free processes* since the equations of a BPA in the normal form can viewed as a set of rules of a context-free grammar in Greibach normal form (GNF) where each rule is of the form

$$X \xrightarrow{a} Y_1 Y_2 \ldots Y_k$$

and where only left derivations are allowed. Note that states of the produced labelled transition system are sequences of variables from Var, i.e., elements of Var^*.

BPP is a subclass of PA where no sequential composition ('.') is allowed. Processes of the form $X_1 \parallel X_2 \parallel \cdots \parallel X_n$ where each $X_i \in Var$ and $n \geq 0$ are called *basic processes*. We identify the basic process where $n = 0$ with $\mathbf{0}$.

It is known that every BPP process Δ can be transformed to *normal form* where if all its equations are of the form

$$X_i = \sum_{j=1}^{n_i} a_{ij} P_{ij}$$

where $1 \leq i \leq n$, $a_{ij} \in Act$, and every P_i is a basic process.

The order of variables in a basic process is not important due to associativity and commutativity of '$\|$', so we can identify a basic process with a multiset of variables. We use $Var^{\oplus}$ to denote the set of all multisets of Var. For $P \in Var^{\oplus}$ and $X \in Var$ we use $P(X)$ to denote number of occurrences of X in P. The relation $\leq$ on $Var^{\oplus}$ is defined as $P \leq Q$ iff $P(X) \leq Q(X)$ for every $X \in Var$. Other relations such as $\geq$, $<$ and $>$ and defined analogously. We use $P \oplus Q$ to denote the union of $P, Q \in Var^{\oplus}$, i.e., $(P \oplus Q)(X) = P(X) + Q(X)$ for each $X \in Var$. We use $P \ominus Q$ to denote the difference of $P, Q \in Var^{\oplus}$ such that $P \geq Q$, i.e., $(P \ominus Q)(X) = P(X) - Q(X)$ for each $X \in Var$.

Equivalently we can represent Δ as a finite set of *rules* of the form

$$X \xrightarrow{a} P$$

where $X \in Var$, $a \in Act$, and $P \in Var^{\oplus}$. For each $t = (X \xrightarrow{a} P) \in \Delta$, we define $\textsc{pre}(t) = X$, $\lambda(t) = a$, and we use $F(t, X)$ to denote $P(X)$. We write $P \xrightarrow{t} P'$.

Remark. Note that the rules representing BPP in the normal form can be viewed as a set of rules of a context-free grammar in Greibach normal form (GNF) where i any derivation is allowed, not only the left-most.

2.5 Petri Nets

A *net* is a triple $\mathcal{N} = (P, Tr, F)$, where P is a finite set of *places*, Tr is a finite set of *transitions*, and

$$F : (P \times Tr) \cup (Tr \times P) \to \mathbb{N}$$

is the *flow function*.

Let $X = P \cup Tr$. For a place or transition $x \in X$ we define sets $\textsc{pre}(x) = \{y \in X \mid F(y, x) > 0\}$ and $\textsc{succ}(x) = \{y \in X \mid F(x, y) > 0\}$. This notation can be extended to sets of places and transitions in the natural way, and so for $X \subseteq P \cup Tr$

$$\textsc{pre}(X) = \bigcup_{x \in X} \textsc{pre}(x) \qquad \textsc{succ}(X) = \bigcup_{x \in X} \textsc{succ}(x)$$

For a transition $t \in Tr$, the sets $\textsc{pre}(t)$ and $\textsc{succ}(t)$ are called its *input* places and *output* places, respectively.

A *marking* is a mapping $M : P \to \mathbb{N}$. If $P = \{s_1, s_2, \ldots, s_k\}$, the marking M can be identified with a vector $(x_1, x_2, \ldots, x_k)$ where $x_i = M(s_i)$ for each $i \in [1, k]$.

A *(Place/Transition) Petri net* is a pair $N = (\mathcal{N}, M_0)$ where $\mathcal{N}$ is a net and M_0 is the *initial marking*.

A transition t is *enabled* at a marking M if $M(p) \geq F(p, t)$ for every $p \in \textsc{pre}(t)$. A transition t that is not enabled is *disabled*. If t is enabled at M, then it can *fire* or *occur*, and its firing leads to the successor marking M' such that

$$M'(p) = M(p) - F(p, t) + F(t, p)$$

for every $p \in P$. The expression $M \xrightarrow{t} M'$ denotes that t is enabled in M, and M' is reached from M after firing of t.

For a sequence $\sigma = t_1 t_2 \dots t_n$ of transitions, $M \xrightarrow{\sigma} M'$ denotes that there is a sequence of markings $M_0, M_1, \dots, M_n$ such that $M_0 = M$, $M_n = M'$, and $M_{i-1} \xrightarrow{t_i} M_i$ for every $i \in [1, n]$. A marking M' *reachable from* a marking M, written $M \longrightarrow M'$, iff there is some sequence of transitions σ such that $M \xrightarrow{\sigma} M'$. A marking M is *reachable* iff it is reachable from the initial marking, i.e., when $M_0 \longrightarrow M$. We use $\mathcal{M}(N)$ to denote the set of reachable markings of a Petri net N.

A *labelled net* is a fourtuple (P, Tr, F, λ), where (P, Tr, F) is a net and

$$\lambda : P \to Act$$

is a mapping that associates to each $t \in Tr$ a label $\lambda(t)$ from a set of actions Act. A *labelled Petri net* is a pair $(\mathcal{N}, M_0)$, where $\mathcal{N}$ is a labelled net and M_0 is the initial marking.

To a labelled Petri net N we associate a labelled transition system

$$(S, Act, \longrightarrow)$$

where $S = \mathcal{M}(N)$, and $M \xrightarrow{a} M'$ iff there is some transition t such that $\lambda(t) = a$ and $M \xrightarrow{t} M'$.

A Petri net is *1-safe* iff $M(p) \leq 1$ for every reachable marking M and every place p.

A Petri net is *communication-free* if for each transition t there is exactly one place p such that $F(p, t) = 1$ and $F(p', t) = 0$ for each $p' \neq p$. Labelled transition systems produced by labelled communication-free Petri nets are isomorphic to labelled transition systems produced by BPP processes.

Let us have a BPP Δ in normal form. Let $V = Var(\Delta)$. We can construct the communication-free Petri net with the set of places V, where for each equation

$$X_i = \sum_{j=1}^{n_i} a_{ij} \alpha_{ij}$$

from Δ where $\alpha_{ij} \in V^*$ we add for each $j \in [1, n_i]$ a new transition t such that $\lambda(t) = a_{ij}$, $F(X_i, t) = 1$, $F(X', t) = 0$ for each $X' \neq X_i$, and $F(t, X)$ is set to number of occurrences of X in α_{ij} for each $X \in V^*$. It is obvious that the labelled transition system produced by the constructed Petri net is isomorphic to the labelled transition system produced by the BPP. The construction of the corresponding BPP for a given communication-free Petri net is similar.

See [50] for more information about Petri nets.

2.6 One-Counter Automata

One-counter automata are nondeterministic finite-state automata operating on a single counter variable which takes values from $\mathbb{N}$. Formally this is a tuple

$$A = (Q, Act, \delta^{=}, \delta^{>}, q_0)$$

where Q is a finite set of *control states*, Act is a finite set of *actions*,

$$\begin{array}{rl} \delta^{=}: & Q \times Act \to \mathcal{P}(Q \times \{0,1\}) \qquad\qquad \text{and} \\ \delta^{>}: & Q \times Act \to \mathcal{P}(Q \times \{-1,0,1\}) \end{array}$$

are *transition functions*, and $q_0 \in Q$ is a distinguished *initial* control state. The function $\delta^{=}$ represents the transitions which are enabled when the counter value is zero, and the function $\delta^{>}$ represents the transitions which are enabled when the counter value is positive.

A one-counter automaton A is a *one-counter net* if and only if for all pairs $(q, a) \in Q \times Act$ we have that $\delta^{=}(q, a) \subseteq \delta^{>}(q, a)$.

The set of (global) states of A is the set $Q \times \mathbb{N}$. States from $Q \times \mathbb{N}$ are written as $p(n)$ instead of (p, n).

To the one-counter automaton A we associate the labelled transition system $(S, Act, \longrightarrow)$, where $S = \{p(n) \mid p \in Q, n \in \mathbb{N}\}$, and $\longrightarrow$ is defined as follows:

$$p(n) \xrightarrow{a} q(n+i) \quad \text{iff} \quad \begin{cases} n = 0, \text{ and } (q, i) \in \delta^{=}(p, a); \quad \text{or} \\ n > 0, \text{ and } (q, i) \in \delta^{>}(p, a). \end{cases}$$

Note that any transition increments, decrements, or leaves unchanged the counter value, and a decrementing transition is only possible if the counter value is strictly positive. Also observe that when $n > 0$ the immediate transitions of $p(n)$ do not depend on the actual value of n. Finally note that a one-counter *net* can in a sense test if its counter is nonzero (that is, it can perform some transitions only on the proviso that its counter is nonzero), but it cannot test in any sense if its counter is zero.

Finite-state systems can be viewed as one-counter nets where $\delta^{=} = \delta^{>}$ and where the counter is never changed. Thus, the parts of the labelled transition system produced by the automaton reachable from $p(i)$ and $p(j)$ are isomorphic and finite for all $p \in Q$ and $i, j \in \mathbb{N}$.

Remark. The class of transition systems generated by one-counter automata is the same (up to isomorphism) as that generated by the class of realtime

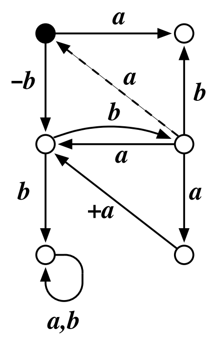


Figure 2.3: An example of a one-counter automaton

pushdown automata (i.e., pushdown automata without ε-transitions) with a single stack symbol (apart from a special bottom-of-stack marker). The class of transition systems generated by one-counter nets is the same (up to isomorphism) as that generated by the class of labelled Petri nets with (at most) one unbounded place.

One-counter automata can be depicted 'graphically' as finite graphs with two kinds of edges (solid and dashed ones) which are labelled by pairs of the form $(a, i) \in Act \times \{-1, 0, 1\}$. Instead of $(a, -1)$, $(a, 1)$, and $(a, 0)$ we write simply $-a$, $+a$, and a, respectively. A *solid* edge from p to q labelled by (a, i) indicates that the represented one-counter automaton can make a transition $p(k) \xrightarrow{a} q(k + i)$ whenever $i \geq 0$ or $k > 0$. A *dashed* edge from p to q labelled by (a, i) (where i must not be -1) represents a zero-transition $p(0) \xrightarrow{a} q(i)$. Hence, graphs representing one-counter nets do not contain any dashed edges, and graphs corresponding to finite-state systems use only labels of the form $(a, 0)$ (remember that finite-state systems can be viewed as special one-counter nets). Also observe that the graphs cannot represent non-decrementing transitions which are enabled *only* for positive counter values. This does not matter since we do not need such transitions in our proofs. The distinguished initial control states are indicated by black circles. See Figure 2.3 for an example of a graph representing an one-counter automaton.

2.7 Explicit and Composed Finite-State Systems

We call a finite-state transition system that is given explicitly a *explicit* transition system. A *composed* system is a system given as a composition of interacting explicit systems. The set of global states of a composed system can be exponentially larger than the sum of sizes of its parts. This phenomenon is known as a state explosion and presents the main challenge in the design of efficient algorithms for verification of composed systems.

There are several different types of parallel composition. One of these types is the parallel composition where systems synchronize on common actions and where actions can be 'hidden'. Synchronization on common actions means that a visible action a is executed iff every LTS that has a in its alphabet executes it. Invisible actions are not synchronized, that is, when an LTS executes the invisible action τ, other LTSs do nothing.

Formally, the *parallel composition*

$$\mathcal{T}_1 \parallel \mathcal{T}_2 \parallel \cdots \parallel \mathcal{T}_n$$

of LTSs $\mathcal{T}_1, \ldots, \mathcal{T}_n$ where $\mathcal{T}_i = (S_i, Act_i, \longrightarrow_i)$ for each $i \in I$ where $I = \{1, 2, \ldots, n\}$, produces the LTS $(S, Act, \longrightarrow)$ where:

- $S = S_1 \times S_2 \times \cdots \times S_n$,
- $Act = Act_1 \cup Act_2 \cup \cdots \cup Act_n$,
- $\longrightarrow$ contains a transition $(s_1, \ldots, s_n) \xrightarrow{a} (s'_1, \ldots, s'_n)$ iff either
  - $a \in Act$ and for every $i \in I$: if $a \in Act_i$, then $s_i \xrightarrow{a} s'_i$, and if $a \notin Act_i$, then $s_i = s'_i$, or
  - $a = \tau$ and $s_i \xrightarrow{\tau} s'_i$ for some $i \in I$, and $s_j = s'_j$ for each $j \neq i$.

Tuples from $S_1 \times S_2 \times \cdots \times S_n$ are called *global states*.

Hiding of actions removes a set of visible actions from the alphabet of an LTS and relabels corresponding transitions with the invisible action τ. Formally, *hide* $\mathcal{B}$ *in* $\mathcal{T}_1$, where $\mathcal{T}_1$ is an LTS $(S_1, Act_1, \longrightarrow_1)$ and $\mathcal{B} \subseteq Act_1$, denotes the LTS $(S, Act, \longrightarrow)$ where $S = S_1$, $Act = Act_1 - \mathcal{B}$, and $s \xrightarrow{a} s'$ iff there is some $a' \in (Act_1 \cup \{\tau\})$ such that $s \xrightarrow{a'} s'$ and either $a \notin \mathcal{B}$ and $a = a'$, or $a' \in \mathcal{B}$ and $a = \tau$.

A *parallel composition with hiding* (PCH) is an LTS $\mathcal{T}$ given in the form

$$\textit{hide } \mathcal{B} \textit{ in } (\mathcal{T}_1 \parallel \cdots \parallel \mathcal{T}_n)$$

where $\mathcal{T}_1, \dots, \mathcal{T}_n$ are explicit finite-state systems. The size $|\mathcal{T}|$ of PCH $\mathcal{T}$ is $|\mathcal{T}_1| + \cdots + |\mathcal{T}_n| + |\mathcal{B}|$.

There are also other types of parallel composition defined in the literature, see, e.g., [59], however, most of them are more general than parallel composition with hiding described in this section. An example is a parallel composition where renaming of actions is allowed.

Another formalism that can be included in composed systems are 1-safe Petri nets, as individual places of a 1-safe Petri net can be viewed as finite-state systems (with 2 states) that communicate through the transitions of the Petri net.

2.8 Behavioural Equivalences

The *equivalence-checking* approach to the formal verification of systems is based on the following scheme: the specification S (i.e., the intended behaviour) and the actual implementation I of a system are defined as states in transition systems, and then it is shown that S and I are *equivalent*.

There are many possible ways how equivalence of processes can be defined. The most prominent of equivalences defined in the literature were organized by van Glabbeek into the hierarchy called linear time – branching time spectrum [60]. The hierarchy is shown in Figure 2.4. Arrows in the diagram represent strict inclusion of equivalences, i.e., an arrow from a relation $\mathcal{R}$ to a relation $\mathcal{R}'$ means that states related by $\mathcal{R}$ must be related by $\mathcal{R}'$, but the converse is not true in general. As can be seen in the diagram, bisimulation equivalence is the finest of these equivalences and trace equivalence is the coarsest.

As all equivalences except bisimilarity are defined as a symmetric closure of a preorder, there is also a similar hierarchy of preorders, see e.g. [21].

Bisimulation equivalence and *simulation equivalence* [44, 49] are of special importance, as their accompanying theory has found its way into many practical applications. Another important equivalence is *trace equivalence* due to its direct correspondence to language equivalence in formal language theory.

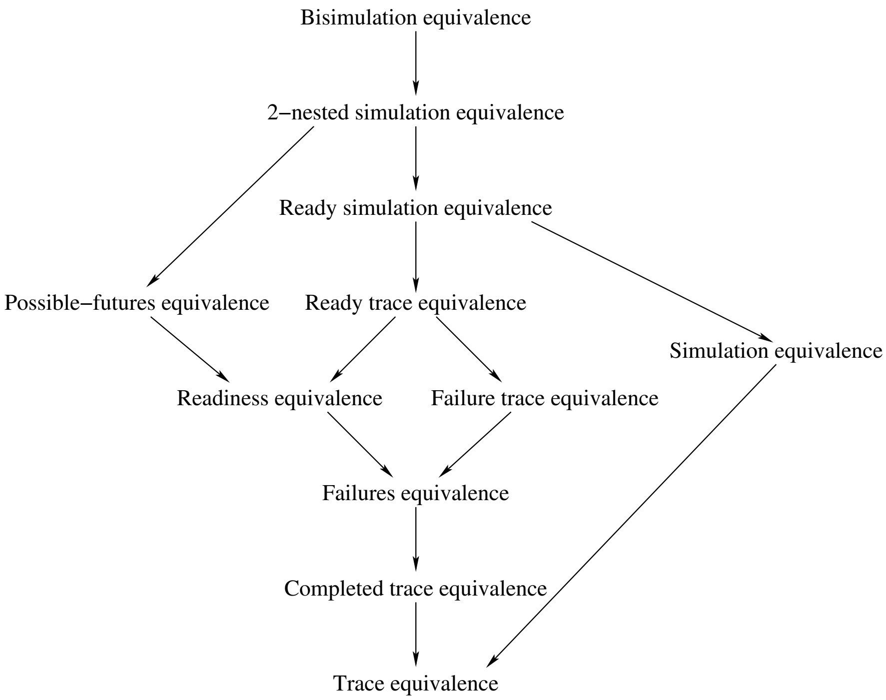


Figure 2.4: Linear time – branching time spectrum

Let $(S, Act, \longrightarrow)$ be a labelled transition system. A binary relation $\mathcal{R} \subseteq S \times S$ is a *bisimulation* iff for every pair of states $(s,t) \in \mathcal{R}$ and every action $a \in Act$ the following conditions hold:

- If there is some $s' \in S$ such that $s \xrightarrow{a} s'$, then there is some $t' \in S$ such that $t \xrightarrow{a} t'$ and $(s', t') \in \mathcal{R}$.
- If there is some $t' \in S$ such that $t \xrightarrow{a} t'$, then there is some $s' \in S$ such that $s \xrightarrow{a} s'$ and $(s', t') \in \mathcal{R}$.

(It is said that $s \xrightarrow{a} s'$ is *matched* by $t \xrightarrow{a} t'$, resp. $t \xrightarrow{a} t'$ is matched by $s \xrightarrow{a} s'$.) States s, t are *bisimilar*, written $s \sim t$, iff there exists some bisimulation $\mathcal{R}$ such that $(s,t) \in \mathcal{R}$. The relation $\sim$ is called *bisimulation equivalence* or *bisimilarity*.

It is not difficult to show that $\sim$ is reflexive, symmetric and transitive. Notice that a union of a family of bisimulation relations is also a bisimulation relation. This implies that $\sim$ which is the union of all bisimulations is the maximal bisimulation.

A binary relation $\mathcal{R} \subseteq S \times S$ is a *simulation* iff for every pair of states $(s,t) \in \mathcal{R}$ and every action $a \in Act$ the following condition holds:

- If there is some $s' \in S$ such that $s \xrightarrow{a} s'$, then there is some $t' \in S$ such that $t \xrightarrow{a} t'$ and $(s', t') \in \mathcal{R}$.

State s is *simulated* by state t, written $s \sqsubseteq t$, iff $(s,t) \in \mathcal{R}$ for some simulation $\mathcal{R}$. States s and t are *simulation equivalent*, written $s \simeq t$, iff $s \sqsubseteq t$ and $t \sqsubseteq s$. The relation $\sqsubseteq$ is called *simulation preorder* and the the relation $\simeq$ is called *simulation equivalence*.

Note that the union of a family of simulation relations is itself a simulation relation, hence, simulation preorder, being the union of all simulation relations, is in fact the maximal simulation relation.

A *trace* from $s \in S$ is any $w \in Act^*$ such that there is a sequence of states $s_0, s_1, \ldots, s_n$ where $s_0 = s$ and $s_{i-1} \xrightarrow{w(i)} s_i$ for every $1 \leq i \leq n$. The set of all traces from s is denoted $\mathit{Traces}(s)$. States s, t are in *trace preorder*, written $s \sqsubseteq_{tr} t$, iff $\mathit{Traces}(s) \subseteq \mathit{Traces}(t)$. States s, t are *trace equivalent* iff $s \sqsubseteq_{tr} t$ and $t \sqsubseteq_{tr} s$.

Let $\mathcal{R}_1, \mathcal{R}_2$ be binary relations over S such that $\mathcal{R}_1 \subseteq \mathcal{R}_2$. We say the relation $\mathcal{R}$ is *between* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ iff $\mathcal{R}_1 \subseteq \mathcal{R} \subseteq \mathcal{R}_2$.

Any relation relating states of a labelled transition system can also relate states of *different* transition systems, because we can consider two transition systems to be a single one by taking the disjoint of them.

Let Δ_1, Δ_2 be (descriptions of) labelled transition systems with distinguished initial states s and t, and let let $\leftrightarrow$ be a binary relation relating states of these systems. Systems Δ_1, Δ_2 are related by $\leftrightarrow$ iff their initial states are related by $\leftrightarrow$, formally $\Delta_1 \leftrightarrow \Delta_2$ iff $s \leftrightarrow t$.

Let $\mathcal{P}$ and $\mathcal{Q}$ be classes of labelled transition systems and let $\leftrightarrow$ be a binary relation relating states of these systems. The problem of deciding whether given systems $\Delta_1 \in \mathcal{P}$ and $\Delta_2 \in \mathcal{Q}$ with distinguished initial states are related by $\leftrightarrow$ is denoted by $\mathcal{P} \leftrightarrow \mathcal{Q}$. For example the problem whether a two systems from class $\mathcal{P}$ are bisimilar is denoted by $\mathcal{P} \sim \mathcal{P}$. Similarly the problem whether a given process from class $\mathcal{P}$ is simulated by a process from class $\mathcal{Q}$ is denoted by $\mathcal{P} \sqsubseteq \mathcal{Q}$.

Equivalence checking problem is any problem of the form $\mathcal{P} \leftrightarrow \mathcal{Q}$ where $\mathcal{P}$, $\mathcal{Q}$, and $\leftrightarrow$ are fixed. We abuse the terminology a little bit, since the term "equivalence checking" is used even for problems where $\leftrightarrow$ is not an equivalence relation.

See [46, 9, 55] for an overview of known results about decidability and complexity of equivalence-checking problems for different types of systems and different types of equivalences.

2.9 Distributed Bisimilarity and BPP

Distributed bisimilarity is one of *non-interleaving* equivalences also called *true concurrency* equivalences. It was introduced in [10]. Examples of other non-interleaving equivalences are location equivalence [11], causal equivalence [16], history preserving bisimilarity [61], or performance equivalence [19].

In this thesis we concentrate on deciding distributed bisimilarity on BPP and we use the definition from [13]. However, it is known that distributed bisimilarity coincides on BPP with many other non-interleaving equivalences, see [40] for details. This means that an algorithm that decides distributed bisimilarity on BPP can be used also for deciding any such equivalence on BPP.

When considering distributed bisimilarity we use the following definition of BPP. Let $Act = \{a, b, c, \ldots\}$ be a countably infinite set of atomic actions and let $Var = \{X, Y, Z, \ldots\}$ be a countably infinite set of process variables. The class of BPP expressions over Act and Var is defined by the following abstract syntax:

$$P \ ::= \ \mathbf{0} \ \mid \ X \ \mid \ a.P \ \mid \ P_1 + P_2 \ \mid \ P_1 \parallel P_2 \ \mid \ P_1 \mathbin{|\!\lfloor} P_2$$

where $\mathbf{0}$ denotes the empty process, X is a process variable, $a.\_$ is an action prefix, $\_ + \_$ denotes non-deterministic choice, $\_ \parallel \_$ parallel composition, and $\_ \mathbin{|\!\lfloor} \_$ left merge.

Remark. The left merge operator $\_ \mathbin{|\!\lfloor} \_$ is similar to parallel composition $\_ \parallel \_$, but in $\_ \mathbin{|\!\lfloor} \_$ an action must be performed first in the first argument.

A *BPP process definition* is a finite family of recursive equations

$$\Delta = \{X_i := P_i \mid 1 \leq i \leq n\}$$

where all X_i are distinct and all P_i are BPP expressions where every occurrence of a variable in P_i is *guarded*, i.e., it is within the scope of an action prefix. The sets of actions and variables occurring in Δ are denoted $Act(\Delta)$ and $Var(\Delta)$, respectively.

A *BPP process* is a pair (P, Δ) where Δ is a BPP process definition and P is a process expression containing only actions and variables from $Act(\Delta)$ and $Var(\Delta)$. We usually write just P instead of (P, Δ) when Δ is obvious from the context.

Distributed bisimilarity is a binary relation defined over BPP expressions. Informally, for each BPP expression there is a set of possible transitions going out of this expression to a pair of expressions called *local* derivative and *concurrent* derivative. The intuition behind this definition is that processes are distributed in space, and local and concurrent derivatives are two parts of the whole process. Local derivative records a location at which the action is observed, and concurrent derivative records the rest of the process. We write transitions as $P \xrightarrow{a} [P', P'']$ where P is the original process, P' and P'' are its local and concurrent derivatives, and a is the performed action.

Let us assume we have a fixed BPP process definition Δ. Then the possible transitions are defined by the following set of rules:

$$\frac{}{a.P \xrightarrow{a} [P, \mathbf{0}]} \qquad \frac{P_j \xrightarrow{a} [P', P''] \text{ for some } j \in I}{\sum_{i \in I} P_i \xrightarrow{a} [P', P'']}$$

$$\frac{P \xrightarrow{a} [P', P'']}{P \parallel Q \xrightarrow{a} [P', P'' \parallel Q]} \qquad \frac{Q \xrightarrow{a} [Q', Q'']}{P \parallel Q \xrightarrow{a} [Q', P \parallel Q'']}$$

$$\frac{P \xrightarrow{a} [P', P'']}{P \mathbin{\|\!\!\lfloor} Q \xrightarrow{a} [P', P'' \parallel Q]} \qquad \frac{P \xrightarrow{a} [P', P'']}{X \xrightarrow{a} [P', P'']}((X \stackrel{\text{def}}{=} P) \in \Delta)$$

A relation $\mathcal{R}$ is a *distributed bisimulation* iff for each $(P, Q) \in \mathcal{R}$ and each $a \in Act$ two following conditions hold:

- if $P \xrightarrow{a} [P', P'']$ then $Q \xrightarrow{a} [Q', Q'']$ for some Q', Q'' such that $(P', Q') \in \mathcal{R}$ and $(P'', Q'') \in \mathcal{R}$, and
- if $Q \xrightarrow{a} [Q', Q'']$ then $P \xrightarrow{a} [P', P'']$ for some P', P'' such that $(P', Q') \in \mathcal{R}$ and $(P'', Q'') \in \mathcal{R}$.

Processes P and Q are distributed bisimilar, denoted $P \sim Q$, iff there is a distributed bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. The relation $\sim$ is called *distributed bisimulation equivalence* or *distributed bisimilarity*.

Remark. Distributed bisimilarity and (normal) bisimilarity are both denoted by the symbol $\sim$ in this thesis. The convention is that $\sim$ represents (normal) bisimilarity unless otherwise stated, the only exception is Section 5.3 that concentrates on distributed bisimilarity and where $\sim$ represents distributed bisimilarity.

It is not difficult to show that non-deterministic choice $(\_ + \_)$ and parallel composition $(\_ \parallel \_)$ are associative and commutative with respect to $\sim$, so we can work with them modulo associativity and commutativity.

Processes of the form $X_1 \parallel X_2 \parallel \cdots \parallel X_n$, where $n \geq 0$ and each $X_i \in Var$, are called *basic processes*. We identify the basic process where $n = 0$ with $\mathbf{0}$.

Every BPP process definition Δ can be transformed to equivalent *normal form* where all equations are of the form

$$X \stackrel{\text{def}}{=} \sum_{i \in I} ((a.P_i) \mathbin{\|\!\!\lfloor} Q_i)$$

where each P_i and Q_i are basic processes and where the relation $\prec$ defined below is irreflexive. The relation $\prec \subseteq Var(\Delta) \times Var(\Delta)$ is defined such that $Y \prec X$ holds iff $X \stackrel{\text{def}}{=} \sum_{i \in I}((a.P_i) \mathbin{\|\!\!\lfloor} Q_i)$ and there is some Q_i such that Y occurs in Q_i. It is an easy task to verify if some such relation $\prec$ exists and to find it.

Since $\prec$ is irreflexive, it can be easily extended to some (arbitrary) linear order. In the thesis we use $\prec$ to denote this linear order.

Remark. Note that some variables in the normal form are not guarded, and so in this sense the normal form is not correct BPP process definition. However, note that although these variables are not guarded in syntactical sense, they are guarded in semantic sense – it is not possible to rewrite a variable X to some expression without going through some action prefix.

See [40] for a polynomial time algorithm that transforms BPP process definition to normal form.

Due to associativity and commutativity of parallel composition $\_ \parallel \_$, the order of variables in a basic process is not important, and so we can identify a basic process with a multiset of variables.

We use $\mathit{Var}^{\oplus}$ to denote the set of all multisets of $\mathit{Var}(\Delta)$. For $P \in \mathit{Var}^{\oplus}$ and $X \in \mathit{Var}$ we use $P(X)$ to denote the number of occurrences of X in P. The relation $\geq$ on $\mathit{Var}^{\oplus}$ is defined as $P \geq Q$ iff $P(X) \geq Q(X)$ for every $X \in \mathit{Var}(\Delta)$. We use $P \oplus Q$ to denote the union of $P, Q \in \mathit{Var}^{\oplus}$, i.e., $(P \oplus Q)(X) = P(X) + Q(X)$ for each $X \in \mathit{Var}(\Delta)$. We use $P \ominus Q$ to denote the difference of $P, Q \in \mathit{Var}^{\oplus}$ such that $P \geq Q$, i.e., $(P \ominus Q)(X) = P(X) - Q(X)$ for each $X \in \mathit{Var}(\Delta)$.

For technical convenience we use a little bit different notation for BPP process definitions in the thesis. We represent a BPP process definition Δ as a finite set of *rules* of the form

$$X \xrightarrow{a} (P, Q)$$

where $X \in \mathit{Var}(\Delta)$, $a \in \mathit{Act}$, and $P, Q \in \mathit{Var}^{\oplus}$. Note that there can be more than one rule with the same variable X on the left hand side.

For each $t = (X \xrightarrow{a} (P, Q)) \in \Delta$, we define $\textsc{pre}(t) = X$, $\lambda(t) = a$, and we use $F(t, X)$ and $G(t, X)$ to denote $P(X)$ and $Q(X)$, respectively. We write $P \xrightarrow{t} [P', P'']$ iff a process P goes to a pair of processes $[P', P'']$ (denoted $P \xrightarrow{a} [P', P''])$ using a rule $t \in \Delta$, i.e., iff t is of the form $X \xrightarrow{a} (P', Q')$ where $P'' = (P \ominus \{X\}) \oplus Q'$.

Chapter 3

Finite-State Processes

In this chapter we consider the lower bounds of the complexity of equivalence checking problems for finite-state systems. At first we will discuss finite-state systems where states and transitions are explicitly given – *explicit* systems, and then *composed* finite-state systems produced by a parallel composition of explicit systems.

The main results presented in this chapter are that the equivalence checking is PTIME-hard for explicit systems and $\mathsf{EXPTIME}$-hard for composed systems. These results hold for any relation between bisimilarity and trace preorder.

At first we define families of problems $\textsc{fs-eq}_{\mathcal{R}}$, $\textsc{pch-eq}_{\mathcal{R}}$, and $\textsc{pn-eq}_{\mathcal{R}}$. Let $\mathcal{R}$ be a binary relation between $\sim$ and $\sqsubseteq_{tr}$ defined over states of labelled transition systems (i.e., such that $s \sim s'$ implies $s\mathcal{R}s'$, and $s\mathcal{R}s'$ implies $s \sqsubseteq_{tr} s'$). The problems $\textsc{fs-eq}_{\mathcal{R}}$, $\textsc{pch-eq}_{\mathcal{R}}$, $\textsc{pn-eq}_{\mathcal{R}}$ are defined as follows:

Problem: $\textsc{fs-eq}_{\mathcal{R}}$

Instance: An FS $\mathcal{T}$ and its two states s and s'.

Question: Is $s\,\mathcal{R}\,s'$?

Problem: $\textsc{pch-eq}_{\mathcal{R}}$

Instance: A PCH $\mathcal{T}$ and its two global states s and s'.

Question: Is $s\,\mathcal{R}\,s'$?

Problem: PN-EQ$_{\mathcal{R}}$

INSTANCE: A labelled net $\mathcal{N}$ with two markings M, M', such that $(\mathcal{N}, M)$ and $(\mathcal{N}, M')$ are 1-safe Petri nets.

QUESTION: Is $M \,\mathcal{R}\, M'$?

We show that FS-EQ$_{\mathcal{R}}$ is PTIME-hard, and PCH-EQ$_{\mathcal{R}}$ and PN-EQ$_{\mathcal{R}}$ are $\mathsf{EXPTIME}$-hard for any $\mathcal{R}$ satisfying $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$.

3.1 State of the Art

In the case of explicit finite-state systems we can easily derive PSPACE-hardness of deciding trace equivalence from standard language theory results. On the other hand, there is a polynomial time algorithm for deciding bisimilarity [47, 34]. The paper [21] is a survey of results in this area. Loosely speaking, 'trace-like' equivalences on the bottom part of the spectrum turn out to be PSPACE-complete, and the 'simulation-like' equivalences on the top of the spectrum are in PTIME. Balcázar, Gabarró and Sántha [5] have considered the question of an efficient parallelization of the algorithm for bisimilarity, and they have shown that the problem is PTIME-complete.

We recall that a problem P is PTIME-hard if any problem in PTIME can be reduced to P by a logspace reduction. Recall that a Turing machine performing such a reduction uses work space of size at most $O(\log n)$, where n denotes the size of the input on a read-only input tape, and writes the output on a write-only output tape. A problem P is PTIME-complete if P is PTIME-hard and $P \in \mathsf{PTIME}$.

From the practical point of view, PTIME-hardness of a problem P means that the problem is hardly parallelisable, that there exist no efficient parallel algorithm for P unless $\mathsf{PTIME} = \mathsf{NC}$. The complexity class NC is the class of problems solvable by an efficient parallel algorithm, i.e., a parallel algorithm with time complexity in $O(\log^k n)$ for some constant k, while the number of used processors must be bounded by a polynomial in the size n of the input instance. It is known that $\mathsf{NC} \subseteq \mathsf{PTIME}$, and it is generally conjectured that the inclusion is proper, however it was not proved. See [18] for more detailed discussion of PTIME-hardness and the NC complexity class.

A. Rabinovich [51] considered a composition of finite-state systems that synchronize on identical actions and where some actions may be 'hidden' in the sense that they are replaced with invisible τ actions, i.e., the model called *parallel composition with hiding* (PCH) in this thesis. He proved that equivalence checking is PSPACE-hard for such systems for any relation between bisimilarity and trace equivalence, and that the problem is EXPSPACE-complete for trace equivalence. He also mentioned that the problem is EXPTIME-complete for bisimilarity and conjectured that the problem is in fact EXPTIME-hard for any relation between bisimilarity and trace equivalence.

Laroussinie and Schnoebelen [39] approved the Rabinovich's conjecture for all relations that lie between bisimilarity and simulation preorder. The composed systems, used in their proof, synchronize on identical actions and do not use hiding. It is not possible to extend their result to all equivalences between bisimilarity and trace equivalence, because for example trace equivalence can be decided in PSPACE for this model, as was proved in [54]. See also [59] for results for other types of 'trace-like' equivalences and composed systems. Other type of composed systems are 1-safe Petri nets. See [33] for some results concerning them, in particular, deciding of bisimilarity is EXPTIME-complete for 1-safe Petri nets.

3.2 Own Contribution

The result of Balcázar, Gabarró and Sántha [5] was extended in [53] to whole linear time – branching time spectrum, i.e., to all relations between bisimilarity and trace preorder. This means that deciding any such relation is a PTIME-hard problem on explicit finite-state systems. The proof presented in this thesis comes from [52] and uses a different (simpler) construction.

The Rabinovich's conjecture from [51] was approved in [52] for *all* relations between bisimilarity and trace preorder, not only for relations between bisimilarity and simulation preorder. It was shown that equivalence checking is EXPTIME-hard for any such relation for parallel composition with hiding (PCH), the model for which Rabinovich formulated his conjecture.

To simplify the proof, a new auxiliary model called reactive linear bounded automaton (RLBA) was introduced. Reactive linear bounded automata can be easily modeled by different types of composed systems, for example by parallel compositions of finite-state systems with hiding, or by 1-safe Petri nets. The EXPTIME-hardness result is shown for RLBA first, and then it is extended to other types of composed systems that are able to model RLBA, such as PCH and 1-safe Petri nets.

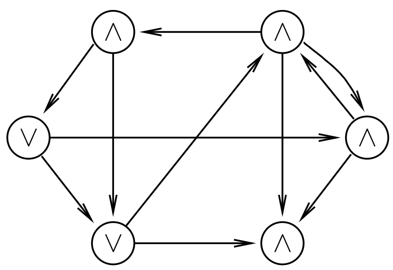


Figure 3.1: Alternating graph

3.3 Explicit Finite-State Systems

In the proof of PTIME-hardness of the problem FS-EQ$_{\mathcal{R}}$ we show a logspace reduction from the *Alternating Graph Problem* (AGP) that is defined in Subsection 3.3.1 to problem FS-EQ$_{\mathcal{R}}$.

Note that FS-EQ$_{\mathcal{R}}$ is in fact a whole family of problems. The reduction described in Subsection 3.3.2 is the same for any problem in this family. The basic idea is to construct an explicit finite-state system with two distinguished states s, s', such that $s \not\sqsubseteq_{tr} s'$ if the answer to the original problem (AGP) is YES, and $s \sim s'$ otherwise. The same construction can be used for every $\mathcal{R}$, because $s \not\sqsubseteq_{tr} s'$ implies that not $s\mathcal{R}s'$, and $s \sim s'$ implies $s\mathcal{R}s'$. The same idea was also used for example in [24] and [51]. We can conclude that the complement of FS-EQ$_{\mathcal{R}}$ is PTIME-hard for any $\mathcal{R}$, and so also FS-EQ$_{\mathcal{R}}$ is PTIME-hard because PTIME is closed under complement.

3.3.1 Alternating Graphs

Before definition of AGP we need some definitions.

An *alternating graph* is a directed graph $G = (V, E, t)$ where V is a finite set of nodes, $E \subseteq V \times V$ is a set of edges, and $t : V \to \{\wedge, \vee\}$ is a labelling function that partitions V into sets

$$V_\wedge = \{v \in V \mid t(v) = \wedge\} \qquad\qquad V_\vee = \{v \in V \mid t(v) = \vee\}$$

of conjunctive and disjunctive nodes. We use $\textsc{succ}(v)$ to denote the set of successors of a node v, i.e., $\textsc{succ}(v) = \{v' \in V \mid (v, v') \in E\}$. See Figure 3.1 for an example of an alternating graph.

The set of *successful* nodes W is the least subset of V such that two following properties hold for each $v \in V$:

- if $v \in V_\wedge$ and $\textsc{succ}(v) \subseteq W$, then $v \in W$, (i.e., if *all* successors of a conjunctive node v are successful, then v is successful),
- if $v \in V_\vee$ and $\textsc{succ}(v) \cap W \neq \emptyset$, then $v \in W$, (i.e., if there *exists* a successful successor of a disjunctive node v, then v is successful).

The AGP is the problem whether a given node is successful in a given alternating graph:

Problem: AGP

INSTANCE: An alternating graph $G = (V, E, t)$ and a node $v \in V$.

QUESTION: Is v successful?

Intuitively, an alternating graph can be viewed as a game played by two players – Player 1 and Player 2. The game starts in some node v. If $v \in V_\vee$ then Player 1 chooses the next move, otherwise (when $v \in V_\wedge$) Player 2 chooses the next move. The player who chooses the move must select some node $v' \in \textsc{succ}(v)$. The game then continues in v'. The game stops when one of players is stuck, i.e., when a node v such that $\textsc{succ}(v) = \emptyset$ is reached. The player who is stuck loses. The successful nodes in alternating graph are nodes where Player 1 has a (history-free) winning strategy in this game.

Let us have a node v with no successors. The node v is called *accepting* node if $v \in V_\wedge$, and it is called *rejecting* node if $v \in V_\vee$. Notice that accepting nodes are always successful and that rejecting nodes are never successful and the game ends in these nodes. Also notice that the set of successful nodes is nonempty iff G contains at least one accepting node, because otherwise the empty set is the least set satisfying the given conditions.

The set of successful nodes W can be computed as the least fixed point of a function $f : \mathcal{P}(V) \to \mathcal{P}(V)$ such that for $U \subseteq V$ we define $f(U)$ as the set of all nodes v such that either:

- $v \in V_\wedge$ and $\textsc{succ}(v) \subseteq U$, or
- $v \in V_\vee$ and $\textsc{succ}(v) \cap U \neq \emptyset$.

The function f is monotone in the sense that $U \subseteq U'$ implies $f(U) \subseteq f(U')$. As follows from Knaster-Tarski Theorem [58], the least fixed point of f can

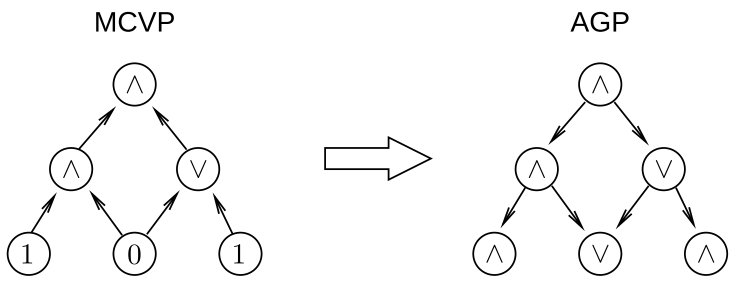


Figure 3.2: Reduction from MCVP to AGP

be computed as

$$W = \bigcup_{i=0}^{\infty} W_i$$

where $W_0 = \emptyset$ and $W_{i+1} = f(W_i)$ for $i \geq 0$. Note that $W_i \subseteq W_{i+1}$ for each i, and because V is finite there must be some $i \leq |V|$ such that $W_i = W_{i+1} = W_{i+2} = \cdots$. This implies that W can be computed in polynomial time since every W_i can be obviously computed in polynomial time and we need to compute at most $|V|$ of them.

Remark. In fact the algorithm can be implemented in such a way that its running time is $O(n)$ where n is the size of the instance.

The problem AGP is PTIME-hard, see for example [22], because the well-known PTIME-complete problem Monotone Circuit Value Problem (MCVP) [18] can be easily reduced to it using the construction illustrated in Figure 3.2. Recall that in MCVP we have given a boolean circuit with input values and the question is what value is on its output. Boolean circuit is an acyclic graph whose inner nodes represent gates computing boolean functions from their inputs. Only gates computing disjunction and conjunction are allowed in MCVP, in particular gates computing negation are not allowed. See [18] for more information about MCVP.

Because AGP is in PTIME, it follows that AGP is PTIME-complete.

3.3.2 Reduction

In the proof of PTIME-hardness of FS-EQ$_{\mathcal{R}}$ for any $\mathcal{R}$ between bisimilarity and trace preorder we show a logspace reduction from AGP that works for any such $\mathcal{R}$. In fact we show a logspace reduction from AGP to the complement of FS-EQ$_{\mathcal{R}}$, but this is not important since the class PTIME is closed under complement. For a given alternating graph $G = (V, E, t)$ with a distinguished node x the reduction constructs a corresponding finite-state system $\mathcal{T}_G = (S, Act, \longrightarrow)$ with two distinguished states $s, s' \in S$ such that if x is successful, then $s \not\sqsubseteq_{tr} s'$ which implies $(s, s') \notin \mathcal{R}$ for any $\mathcal{R}$ between bisimilarity and trace preorder, and if x is not successful, then $s \sim s'$ which implies $(s, s') \in \mathcal{R}$.

The set of states S is V. For each $v \in V$ we define a set of corresponding actions $Act(v)$. If $v \in V_\wedge$, then $Act(v) = \{\langle v \rangle\}$, and if $v \in V_\vee$, then $Act(v) = \{\langle v, i \rangle \mid 1 \leq i \leq |\text{SUCC}(v)|\}$. The set of actions Act is $\bigcup_{v \in V} Act(v)$. We can assume without loss of generality that successors of each node are ordered in some fixed order. The i-th successor of v where $1 \leq i \leq |\text{SUCC}(v)|$ is denoted $\text{SUCC}_i(v)$.

The transition relation contains transitions of three types:

1. $v \xrightarrow{\langle a \rangle} v$ for each $v \in V$ and $a \in Act$ such that $a \notin Act(v)$.
2. $v \xrightarrow{\langle v,i \rangle} v'$ for each $v \in V_\vee$ and $1 \leq i \leq |\text{SUCC}(v)|$ where $v' = \text{SUCC}_i(v)$.
3. $v \xrightarrow{\langle u \rangle} u'$ for each $v \in V$, $u \in V_\wedge$ and $u' \in V$ such that $u' \in \text{SUCC}(u)$.

We may assume without loss of generality that G contains at least one rejecting node z, because we can add such node (and no edges) without affecting the set of successful nodes, if this is not the case. The instance of FS-EQ$_{\mathcal{R}}$ then consists of $\mathcal{T}_G$ and states z and x, where x is the distinguished node from the instance of AGP.

Proposition 3.1 *If $v \in V$ is not successful then $z \sim v$.*

Proof. It is sufficient to show that $\{(z, v) \mid v \in (V - W)\} \cup Id$ is a bisimulation (Id denotes the identity relation $\{(v, v) \mid v \in V\}$). Let us consider some pair (z, v) where $v \in (V - W)$, and a transition $v \xrightarrow{a} v'$. This transitions is either of:

- type 1, and then it is matched by $z \xrightarrow{a} z$ of type 1, because $Act(z) = \emptyset$ and so $z \xrightarrow{a} z$ for every $a \in Act$,
- type 2, and then $v \in V_\vee$ and because v is unsuccessful, each $v' \in \text{SUCC}(v)$ is also unsuccessful, and so $v \xrightarrow{a} v'$ is matched by $z \xrightarrow{a} z$,

- type 3, and then it can be matched by $z \xrightarrow{a} v'$ of type 3.

Now consider a transition of the form $z \xrightarrow{a} z'$. It is of:

- type 1 and then $z' = z$ and either $a \notin Act(v)$, and $z \xrightarrow{a} z$ is matched by $v \xrightarrow{a} v$ of type 1, or $a \in Act(v)$ and then there are two possibilities:
  - if $v \in V_\vee$ then each $v' \in \textsc{succ}(v)$ is unsuccessful since v is unsuccessful, and so $z \xrightarrow{a} z$ can be matched by $v \xrightarrow{a} v'$ of type 2,
  - if $v \in V_\wedge$ then there is at least one unsuccessful $v' \in \textsc{succ}(v)$, and so $z \xrightarrow{a} z$ can be matched by $v \xrightarrow{a} v'$ of type 3,
- type 2, but this is not possible as $Act(z) = \emptyset$,
- type 3 and it can be matched by $v \xrightarrow{a} z'$ of type 3.

□

Proposition 3.2 *There is $w \in Act^*$ such that $w \notin Traces(v)$ for any successful $v \in V$.*

Proof. As W can be computed as the least fixed point of the function f defined in Subsection 3.3.1, we can define a sequence $W_0 \subseteq W_1 \subseteq W_2 \subseteq \cdots$ of subsets of W where $W_0 = \emptyset$ and $W_{i+1} = f(W_i)$ for $i > 0$. For each $v \in W$ there is some least i such that $v \in W_i$. This i is denoted $rank(v)$. Let $m = |W|$, and let $v_1, v_2, \ldots, v_m$ be the nodes in W ordered by their rank, i.e., if $i < j$ then $rank(v_i) \leq rank(v_j)$.

Let us consider a word $w_m = a_m a_{m-1} \cdots a_1$ where $a_i = \langle v_i \rangle$ if $v_i \in V_\wedge$, and if $v_i \in V_\vee$ then $a_i = \langle v_i, k \rangle$ where we choose k such that $v' = \textsc{succ}_k(v_i)$ is successful and $rank(v') < rank(v_i)$ (obviously there always exists at least one such v'). We show that $w_m \notin Traces(v)$ for any successful node v. In particular, for each $i \leq m$ we show that $w_i = a_i a_{i-1} \cdots a_1 \notin Traces(v_j)$ if $j \leq i$. We proceed by induction on i and in the proof we use the following simple observation: $w_i \notin Traces(v)$ iff for each v' such that $v \xrightarrow{a_i} v'$ is $w_{i-1} \notin Traces(v')$.

The base case ($i = 0$) is trivial. In the induction step we consider $i > 0$ and show that the proposition holds for every v_j where $1 \leq j \leq i$.

If $v_i \in V_\vee$ then $a_i = \langle v_i, k \rangle$. Any transition of the form $v_j \xrightarrow{a_i} v'$ is either of type 1, and then $v' = v_j$ and $j < i$, and by induction hypothesis $w_{i-1} \notin$

Traces(v'), or of type 2, and then $v' = \textsc{succ}_k(v_i)$, so v' is successful and *rank*$(v') <$ *rank*(v), and by induction hypothesis $w_{i-1} \notin$ *Traces*(v').

If $v_i \in V_\wedge$ then $a_i = \langle v_i \rangle$. Any transition of the form $v_j \xrightarrow{a_i} v'$ is either of type 1, and then $v' = v_j$ and $j < i$, and by induction hypothesis $w_{i-1} \notin$ *Traces*(v'), or of type 3, and then $v' \in \textsc{succ}(v_i)$ and so v' is successful and *rank*$(v') <$ *rank*(v_i), so by induction hypothesis $w_{i-1} \notin$ *Traces*(v'). □

Theorem 3.3 FS-EQ$_{\mathcal{R}}$ *is* PTIME*-hard for any* $\mathcal{R}$ *such that* $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$.

Proof. Notice that $z \xrightarrow{a} z$ for each $a \in Act$, because $Act(z) = \emptyset$, and so $Traces(z) = Act^*$. From this and Proposition 3.2 we have that $z \not\sqsubseteq_{tr} x$ if x is successful. On the other hand, from Proposition 3.1 we have that $z \sim x$ if x is not successful, and so the described reduction is correct.

The reduction can be obviously performed in a logarithmic space. Since the problem AGP is PTIME-complete and PTIME is closed under complement, we obtain the result. □

3.4 Composed Finite-State Systems

To simplify the proof, a new model called *reactive linear bounded automata* (RLBA) was introduced in [52]. It was shown there that deciding any relation between bisimulation equivalence and trace preorder is an EXPTIME-hard problem for RLBA. Because RLBA can be "modeled" by different types of composed systems, in particular by PCH and by 1-safe Petri nets, we obtain the similar EXPTIME-hardness result for all such systems.

To show EXPTIME-hardness for RLBA we describe a logspace reduction from a well known EXPTIME-complete problem called ALBA-ACCEPT. This is a problem whether a given alternating linear bounded automaton accepts a given word. Linear bounded automata (LBA) and alternating linear bounded automata (ALBA) are described in Subsection 3.4.1. Reactive linear bounded automata are introduced in Subsection 3.4.2 and it is shown there how that they can be modeled by different types of composed systems such as PCH and 1-safe Petri nets.

An RLBA is similar to a usual LBA, but is intended to generate a labelled transition system instead of accepting or rejecting an input. The equivalence checking problem where the instance is an RLBA and two of its configurations and the question is, whether they are in relation $\mathcal{R}$, is denoted RLBA-EQ$_{\mathcal{R}}$.

The main technical result of this section presented in Subsections 3.4.3, 3.4.4 and 3.4.5 shows that RLBA-EQ$_{\mathcal{R}}$ is EXPTIME-hard for any relation $\mathcal{R}$ satisfying $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$.

The construction in the reduction from ALBA-ACCEPT to RLBA-EQ$_{\mathcal{R}}$ is based on the construction that was used in Section 3.3 to show PTIME-hardness of FS-EQ$_{\mathcal{R}}$, but is more involved. The main idea is that a computation of an alternating linear bounded automaton can be viewed as an alternating graph, where successful nodes correspond to successful configurations, and this allows us to "shift" the previous result "higher" in the complexity hierarchy. We will construct an RLBA that will model the labelled transition system which we would obtain when we would apply the above mentioned reduction to the alternating graph corresponding to the computation of the ALBA. Moreover, logarithmic space will be sufficient for the construction of this RLBA from the instance of ALBA-ACCEPT.

3.4.1 Linear Bounded Automata

In the proof we use the reduction from the problem ALBA-ACCEPT, which is known to be EXPTIME-complete [12]. The instance of this problem is an alternating linear bounded automaton with its input. Alternating linear bounded automata are a generalization of linear bounded automata. Also reactive linear bounded automata defined in the following subsection are a generalization of linear bounded automata, so we start with definition of linear bounded automata.

A *linear bounded automaton* (LBA) is a tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$, where Q is a set of control states, Σ is an input alphabet, Γ is a tape alphabet, $\delta \subseteq (Q - \{q_{acc}, q_{rej}\}) \times \Gamma \times Q \times \Gamma \times \{-1, 0, +1\}$ is a set of transitions, $q_0, q_{acc}, q_{rej} \in Q$ are an initial, accepting and rejecting state, respectively. The alphabet Γ contains left and right endmarkers $\vdash$ and $\dashv$.

A *configuration* of A is a triple $\alpha = (q, w, i)$ where q is the current control state, $w = a_1 a_2 \cdots a_n$ is the tape content, and $1 \leq i \leq |w|$ is the head position. Only configurations where $w = \vdash w' \dashv$ and endmarkers do not occur in w' are allowed. The size $|\alpha|$ of α is $|w|$. A configuration $\alpha' = (q', w', i')$ is a *successor* of $\alpha = (q, w, i)$, written $\alpha \vdash_A \alpha'$ (or just $\alpha \vdash \alpha'$ when A is obvious), iff $(q, a, q', a', d) \in \delta$, w contains a on i-th position, $i' = i + d$, and w' is obtained from w by writing a' on position i. Endmarkers may not be overwritten, and the machine is constrained never to move left of the $\vdash$ nor right of the $\dashv$. Notice that when $\alpha \vdash \alpha'$, then $|\alpha| = |\alpha'|$. The initial configuration for an input $w \in \Sigma^*$ is $\alpha_{ini}(w) = (q_0, \vdash w \dashv, 1)$. A configuration is *accepting* iff $q = q_{acc}$, and *rejecting* iff $q = q_{rej}$.

An *alternating LBA* (ALBA) is an LBA extended with a function

$$l : Q \to \{\wedge, \vee\}$$

that labels each control state as either conjunctive or disjunctive. We extend l to configurations in an obvious manner and so also configurations are labeled as conjunctive and disjunctive. A configuration is *successful* iff it is either

- accepting, or
- disjunctive with at least one successful successor, or
- conjunctive with all successors successful.

An ALBA A accepts an input $w \in \Sigma^*$ iff $\alpha_{ini}(w)$ is successful.

The problem ALBA-ACCEPT is defined as:

INSTANCE: An ALBA A and a word $w \in \Sigma^*$.

QUESTION: Does A accept w?

Notice that there is a close relationship between AGP and ALBA-ACCEPT. A computation of an ALBA can be viewed as an alternating graph where successful nodes correspond to successful configurations. The size of this graph can be exponentially larger than the size of the corresponding instance of ALBA-ACCEPT.

3.4.2 Reactive Linear Bounded Automata

Reactive linearly bounded automata are introduced in this section. A *reactive linear bounded automaton* (RLBA) is like a usual LBA, but it has special control states, called *reactive* states, where it can perform actions from some given set of actions *Act*. Only the control state is changed after performing such actions, neither the tape content nor the head position is modified. The other control states are called *computational* and RLBA performs steps as a usual LBA in them. Each such step is represented as the invisible action τ.

Formally, an RLBA is a tuple $B = \{Q, \Gamma, \delta, Act, l, \longrightarrow\}$, where the meaning of Q, Γ and δ is the same as in a usual LBA, Act is the finite set of actions, the function $l : Q \to \{r, c\}$ partitions Q into sets Q_r and Q_c of reactive and computational states, and $\longrightarrow \subseteq Q_r \times (Act \cup \{\tau\}) \times Q$ is the transition relation (we write $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \longrightarrow$). It is also required that if $(q, b, q', b', d) \in \delta$ then $q \in Q_c$. The definition of a configuration and a successor relation is the same as for a usual LBA.

An RLBA B generates a labelled transition system $\mathcal{T}(B) = (S, Act, \longrightarrow)$, where S is the set of configurations of B, and where $\longrightarrow$ contains a transition $(q, w, i) \xrightarrow{a} (q', w', i')$ iff either

- $q \in Q_c$, $(q, w, i) \vdash (q', w', i')$ and $a = \tau$, or
- $q \in Q_r$, $q \xrightarrow{a} q'$, $w = w'$ and $i = i'$.

For each $\mathcal{R}$, such that $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$, we can define the problem RLBA-EQ$_{\mathcal{R}}$:

INSTANCE: An RLBA B and its two configurations α, α' of size n.

QUESTION: Is $\alpha \, \mathcal{R} \, \alpha'$?

An RLBA with configurations of size n can be easily modeled by various composed systems, as two following lemmas show.

Lemma 3.4 *There is a logspace reduction from* RLBA-EQ$_{\mathcal{R}}$ *to* PCH-EQ$_{\mathcal{R}}$*.*

Proof. Let us have an RLBA B and two its configurations of size n. We construct a PCH $\mathcal{T}$ of the form *hide* $\mathcal{B}$ *in* $(\mathcal{T}_c \parallel \mathcal{T}_1 \parallel \cdots \parallel \mathcal{T}_n)$ which models the LTS generated by B. In particular, $\mathcal{T}_c$ models the control unit, and $\mathcal{T}_1, \ldots, \mathcal{T}_n$ model the tape cells of B. A state of $\mathcal{T}_c$ represents the current control state and head position, and a state of $\mathcal{T}_i$ represents the symbol on the i-th position of the tape.

Let $I = \{1, \ldots, n\}$ be the set of all possible positions of the head. For each $i \in I$ is $\mathcal{T}_i = (S_i, Act_i, \longrightarrow_i)$ where $S_i = \Gamma$, $Act_i = \{\langle b, b', i\rangle \mid b, b' \in \Gamma\}$, and $\longrightarrow_i$ contains transitions $b \xrightarrow{\langle b,b',i\rangle} b'$ for each $b, b' \in \Gamma$.

In $\mathcal{T}_c = (S_c, Act_c, \longrightarrow_c)$ is $S_c = \{\langle q, i\rangle \mid q \in Q, i \in I\}$ and $Act_c = Act \cup Act_0 \cup \cdots \cup Act_n$ (without loss of generality we can assume that $Act \cap Act_i = \emptyset$ for each $i \in I$). To $\longrightarrow_c$ we add for each $(q, b, q', b', d) \in \delta$ and $i \in I$, such that $i + d \in I$, a transition $\langle q, i\rangle \xrightarrow{\langle b,b',i\rangle} \langle q', i + d\rangle$, and for each $q \in Q_r$, $q' \in Q$, $a \in (Act \cup \{\tau\})$ and $i \in I$ where $q \xrightarrow{a} q'$ we add a transition $\langle q, i\rangle \xrightarrow{a} \langle q', i\rangle$.

The set $\mathcal{B}$ in $\mathcal{T}$ is defined as $Act_1 \cup \cdots \cup Act_n$. Each configuration

$$\alpha = (q, a_1 a_2 \cdots a_n, i)$$

has a corresponding global state

$$g(\alpha) = (\langle q, i \rangle, a_1, a_2, \ldots, a_n).$$

As can be easily checked, $\alpha \xrightarrow{a} \alpha'$ in B iff $g(\alpha) \xrightarrow{a} g(\alpha')$ in $\mathcal{T}$, and so

$$\alpha \mathcal{R} \alpha' \text{ in } B \quad \text{iff} \quad g(\alpha)\, \mathcal{R}\, g(\alpha') \text{ in } \mathcal{T}$$

for any $\mathcal{R}$ such that $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$. It is obvious that $\mathcal{T}$ can be constructed from B in a logarithmic space. □

Lemma 3.5 *There is a logspace reduction from* RLBA-EQ$_{\mathcal{R}}$ *to* PN-EQ$_{\mathcal{R}}$.

Proof. Let us have an instance of RLBA-EQ$_{\mathcal{R}}$, i.e., an RLBA B and two of its configurations of size n. We construct corresponding labelled Petri net as follows. Let $I = \{1, \ldots, n\}$. The set of places will be $Q \cup \{\langle a, i \rangle \mid a \in \Gamma, i \in I\} \cup I$.

For each $(q, b, q', b', d) \in \delta$ and $i \in I$ where $q \in Q_c$ and $i + d \in I$ we add a transition $t = \langle q, b, q', b', i, i+d \rangle$ labelled with τ together with incoming arcs (q, t), $(\langle b, i \rangle, t)$, and (i, t), and outgoing arcs (t, q'), $(t, \langle s', i \rangle)$, and $(t, i + d)$.

For each $q, q' \in Q$ and $a \in (Act \cup \{\tau\})$ where $q \in Q_r$ and $q \xrightarrow{a} q'$ we add a new transition $t = \langle q, a, q' \rangle$ labelled with a together with an incoming arc $\langle q, t \rangle$ and an outgoing arc $\langle t, q' \rangle$.

For a configuration $\alpha = \{q, a_1 a_2 \cdots a_n, i\}$ we define a corresponding marking M_α where $M_\alpha(p) = 1$ if p is q, i, or $\langle a_j, j \rangle$ where $j \in I$, and $M_\alpha(p) = 0$ otherwise. It is easy to check that $\alpha \xrightarrow{a} \alpha'$ iff $M_\alpha \xrightarrow{a} M_{\alpha'}$. □

Similar proofs can be used for other types of composed systems as they are usually more general than PCH. However parallel composition of finite-state systems that synchronize on common actions and where hiding of actions is not possible (nor other similar mechanism) is not powerful enough to simulate RLBA. It is known that problems of deciding some equivalences, such as trace equivalence and other 'trace-like' equivalences, are in PSPACE for this kind of systems [54, 59].

3.4.3 Reduction

The description of the reduction from ALBA-ACCEPT to the complement of RLBA-EQ$_{\mathcal{R}}$ consists of several steps that are summarized in Figure 3.3 where $\overline{\text{FS-EQ}}_{\mathcal{R}}$ and $\overline{\text{RLBA-EQ}}_{\mathcal{R}}$ denote the complements of FS-EQ$_{\mathcal{R}}$ and RLBA-EQ$_{\mathcal{R}}$.

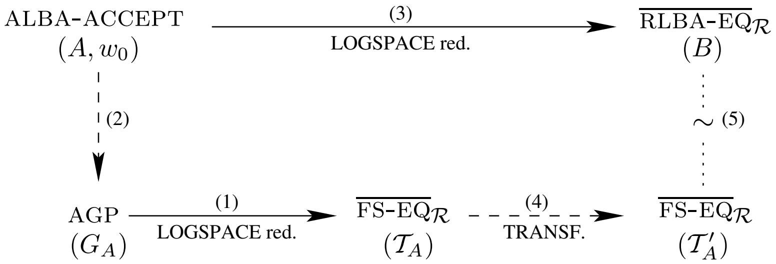


Figure 3.3: Outline of the reduction from ALBA-ACCEPT to $\overline{\text{FS-EQ}}_{\mathcal{R}}$

The reduction (1) from AGP to $\overline{\text{FS-EQ}}_{\mathcal{R}}$ can be applied to the alternating graph G_A that corresponds (2) to the ALBA A in the instance of ALBA-ACCEPT. We obtain an LTS $\mathcal{T}_A$. From the instance of ALBA-ACCEPT we construct (3) a RLBA B that models $\mathcal{T}_A$ in the sense, that after we apply a certain kind of transformation (4) to $\mathcal{T}_A$, we obtain an LTS $\mathcal{T}'_A$ bisimilar (5) with B. It will be proved that the transformation (4) preserves some important properties, in particular, states that were bisimilar are bisimilar after the transformation, and states that were not in trace preorder are not in trace preorder after the transformation. Bisimilarity (5) implies that the same is true for corresponding configurations of B, from which the correctness of the construction (3) follows. The EXPTIME-hardness of $\overline{\text{RLBA-EQ}}_{\mathcal{R}}$ implies the EXPTIME-hardness of RLBA-EQ$_{\mathcal{R}}$ since EXPTIME is closed under complement.

The rest of the section is devoted to the description of a logspace reduction from ALBA-ACCEPT to $\overline{\text{RLBA-EQ}}_{\mathcal{R}}$.

Let an ALBA $A = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ with a word $w_0 \in \Sigma^*$ be an instance of ALBA-ACCEPT. We can assume that transitions in δ are ordered and that this ordering determines the order of successors of a configuration. For simplicity we can assume without loss of generality that each configuration of A, which is not accepting nor rejecting, has exactly two successors, and that $l(q_{acc}) = \wedge$ and $l(q_{rej}) = \vee$. Let *Conf* be the set of all configurations of A of size $n = |w_0| + 2$, and let $Conf_{\wedge}$, $Conf_{\vee}$, and $Conf_{rej}$ be the sets of conjunctive, disjunctive, and rejecting configurations of size n,

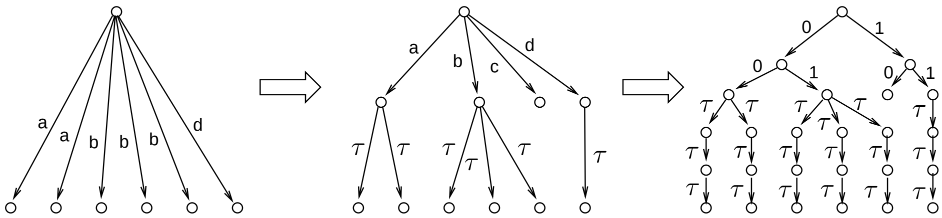


Figure 3.4: The transformation performed on $\mathcal{T}_A$

respectively. Notice that any configuration reachable from $\alpha_0 = \alpha_{ini}(w_0)$ is of size n.

The ALBA A has a corresponding alternating graph $G_A = (V, E, t)$, where $V = \mathit{Conf}$, $(\alpha, \alpha') \in E$ iff $\alpha \vdash \alpha'$, and $t(\alpha) = l(\alpha)$ for each $\alpha \in \mathit{Conf}$. Notice that a configuration α is successful in A iff the node α is successful in G_A, and that A accepts w iff the node α_0 is successful.

When we apply the logspace reduction described in Section 3.3 to G_A with a node α_0, we obtain the LTS $\mathcal{T}_A = (S_A, \mathit{Act}_A, \longrightarrow_A)$, where $S_A = \mathit{Conf}$, $\mathit{Act}(\alpha) = \{\langle\alpha\rangle\}$ for each $\alpha \in \mathit{Conf}_\wedge$, $\mathit{Act}(\alpha) = \{\langle\alpha, i\rangle \mid i \in \{1, 2\}\}$ for each $\alpha \in \mathit{Conf}_\vee - \mathit{Conf}_{rej}$, $\mathit{Act}(\alpha) = \emptyset$ for each $\alpha \in \mathit{Conf}_{rej}$, $\mathit{Act}_A = \bigcup_{\alpha \in \mathit{Conf}} \mathit{Act}(\alpha)$, and $\longrightarrow_A$ contains the following transitions for each $\alpha \in \mathit{Conf}$:

1. $\alpha \xrightarrow{x}_A \alpha$ for each $x \in (\mathit{Act}_A - \mathit{Act}(\alpha))$,
2. $\alpha \xrightarrow{\langle\alpha,i\rangle}_A \alpha'$ if $\alpha \in \mathit{Conf}_\vee$, and α' is the i-th successor of α,
3. $\alpha \xrightarrow{\langle\beta\rangle}_A \beta'$ for each $\beta \in \mathit{Conf}_\wedge$ and $\beta' \in \mathit{Conf}$ such that $\beta \vdash \beta'$.

Let $\alpha_{rej} \in \mathit{Conf}_{rej}$ be some rejecting configuration. The states α_{rej} and α_0 are the two distinguished states with the property, that if A accepts w_0, then $\alpha_{rej} \not\sqsubseteq_{tr} \alpha_0$, and $\alpha_{rej} \sim \alpha_0$ otherwise.

An RLBA $B = (Q_B, \Gamma_B, \delta_B, \mathit{Act}_B, l_B, \longrightarrow_B)$ that in some sense 'models' $\mathcal{T}_A$ will be constructed. The RLBA B will be described only informally, but it should be clear from this description how to construct it. In fact B models an LTS that we obtain from $\mathcal{T}_A$ by a transformation illustrated in Figure 3.4.

Figure 3.4 shows only transitions going from one state, but the same transformation is performed for all states and transitions. In this simplified example is $\mathit{Act}_A = \{a, b, c, d\}$. At first, the non-deterministic choice is postponed.

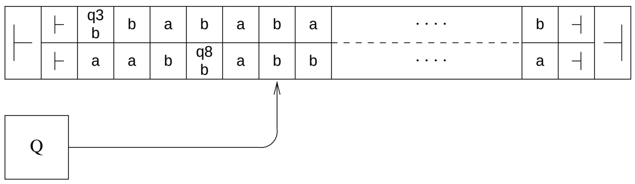


Figure 3.5: An example of a configuration of the RLBA B

Notice that that a new state is added for each action in Act_A. Next, each action from Act_A is replaced by sequence of actions from some 'small' alphabet Act_B. In our example is $Act_B = \{0, 1\}$ and a, b, c, d are replaced with 00, 01, 10 and 11, respectively. Invisible actions representing non-deterministic choice are replaced with sequences of τ actions of some fixed length m (in this example $m = 3$). This kind of transformation is described more formally in the next subsection.

Configurations of A can be written as words in an alphabet $\Delta = (Q \times \Gamma) \cup \Gamma$, where occurrence of the symbol from $Q \times \Gamma$ denotes the position of the head (there must be exactly one such symbol in the word). A word from Δ^* corresponding to a configuration α is denoted by $desc(\alpha)$. Actions from $Act(\alpha)$ are replaced with sequences of actions corresponding to $desc(\alpha)$ in B. In particular, $Act_B = \Delta_{Act} \cup \{1, 2\}$ where $\Delta_{Act} = \Delta - \{(q_{rej}, a) \mid a \in \Gamma\}$. Actions from $\{1, 2\}$ are used to identify a successor of a disjunctive configuration.

B has a tape with two tracks, denoted track 1 and track 2, respectively. A current state α of $\mathcal{T}_A$ is is stored as a word $desc(\alpha)$ on track 1. B also needs to store information about the label of a transition that $\mathcal{T}_A$ performs. The configuration from the label of the transition is stored on track 2. Formally this means that $\Gamma_B = (\Delta \times \Delta) \cup \{\vdash, \dashv\}$. See Figure 3.5 for an example.

As mentioned above, a transition of $\mathcal{T}_A$ labelled with an action from $Act(\beta)$ is represented in B as a sequence of transitions. Each such sequence starts and ends in a configuration where track 1 contains the current state α of $\mathcal{T}_A$ and where the head of B points to the first symbol of $desc(\alpha)$, i.e., it is on position 2. The contents of track 2 is not important, since it will be overwritten. The sequence of transitions of B corresponding to one transition of $\mathcal{T}_A$ has two phases (denoted as phase 1 and phase 2):

1. Actions representing symbols of $desc(\beta)$ are performed one by one and the corresponding symbols are stored on track 2. The head of B goes from left to right.

2. Depending on some some properties of α and β that will be described below, there can be some number of possible transitions. (Information about the properties of α and β needed in this step can be kept in the control unit of B.) One of the possibilities is chosen non-deterministically. The possibilities are always of one of the following types:
   (a) The head of B moves back to the left endmarker without changing anything.
   (b) A chosen successor of β is stored on track 1 while the head returns back to the left endmarker. This involves copying of track 2 to track 1 with the necessary modifications on positions where β and its successor differ.

The three following steps are performed for each symbol a of $desc(\beta)$ during phase 1:

- the symbol from track 1 is read into the control unit,
- an action a is performed, and remembered in the control unit,
- a is written on track 2 and the head moves to the next cell.

This means that actions $\tau a \tau$ are performed for each symbol a. Phase 1 ends when the right endmarker $\dashv$ is reached. If $\beta \in Conf_{\vee}$, then phase 1 includes also an action $a \in \{1, 2\}$ identifying a successor of β. This number is stored in the control unit of B.

The possible choices at the start of phase 2 depend only on whether $\alpha = \beta$, and on the type of β (if it is accepting, conjunctive or disjunctive). This information can be stored in the control unit of B. To find out if $\alpha = \beta$, notice that we can compare symbols on tracks 1 and 2 during phase 1. The possible non-deterministic choices are the following: if β is disjunctive, the successor of β that was chosen at the end of phase 1 can be stored on track 1, and if β is conjunctive, the non-deterministically chosen successor of β can be stored on track 1. The choice (a), i.e., to keep track 1 intact, is possible only when $\alpha \neq \beta$. Notice that when β is accepting, there are no successors of β and so there are no transitions possible when $\alpha = \beta$.

B can be constructed in such a way that only valid configurations can be written on track 2 during phase 1, and that the number of steps performed during phase 2 is some fixed value m such that $m \in O(n)$. In particular, we can put $m = 2n + 4$, because two steps are needed to copy one symbol from track 2 to track 1, and we need two additional steps to modify track 1 to reflect one step of A. We also need additional steps at the beginning and at the end of phase 2.

3.4.4 Decomposition of Transitions

In this subsection we describe the transformation performed on $\mathcal{T}_A$ more formally and we show that it preserves some important properties of the original LTS.

Let us have an LTS $\mathcal{T} = (S, Act, \longrightarrow)$, a set of actions Act', some positive integer m and a mapping $h : Act \to Act'^*$ such that $h(a)$ is not a prefix of $h(a')$ if $a \neq a'$. Let $H = \{h(a) \mid a \in Act\}$ and let $\mathit{Pref}(H)$ be the set of all prefixes of words from H.

We can construct a new LTS $\mathcal{T}' = (S', Act', \longrightarrow')$ where

$$S' = \{\langle s, w\rangle \mid s \in S,\, w \in \mathit{Pref}(H)\} \cup \{\langle s, i\rangle \mid s \in S,\, 0 \leq i < m\}$$

where we identify the the states $\langle s, 0\rangle$ and $\langle s, \varepsilon\rangle$ (i.e., $\langle s, 0\rangle$ and $\langle s, \varepsilon\rangle$ are the same state), and where $\longrightarrow'$ contains transitions:

- $\langle s, w\rangle \xrightarrow{a} \langle s, wa\rangle$ for each $s \in S$, $w \in Act'^*$ and $a \in Act'$ such that $wa \in \mathit{Pref}(H)$,
- $\langle s, w\rangle \xrightarrow{\tau} \langle s', m-1\rangle$ for each $s, s' \in S$ and $a \in Act$ such that $s \xrightarrow{a} s'$ and $h(a) = w$,
- $\langle s, i\rangle \xrightarrow{\tau} \langle s, i-1\rangle$ for each $s \in S$ and $0 < i < m$.

For each state $s \in S$ in $\mathcal{T}$ there is a corresponding state $\langle s, \varepsilon\rangle \in S'$ in $\mathcal{T}'$.

Lemma 3.6 *For each $s, s' \in S$: if $s \sim s'$, then $\langle s, \varepsilon\rangle \sim \langle s', \varepsilon\rangle$, and if $s \not\sqsubseteq_{tr} s'$, then $\langle s, \varepsilon\rangle \not\sqsubseteq_{tr} \langle s', \varepsilon\rangle$.*

Proof (sketch). To prove the first part of the lemma, it is sufficient to show that

$$\begin{aligned} R \;=\; & \{(\langle s, w\rangle, \langle t, w\rangle) \mid s \sim t, w \in \mathit{Pref}(H)\} \cup \\ & \{(\langle s, i\rangle, \langle t, i\rangle) \mid s \sim t, 0 < i < m\} \end{aligned}$$

is a bisimulation.

To prove the second part, let us define a mapping $\hat{h} : Act^* \to Act'^*$ such that $\hat{h}(\varepsilon) = \varepsilon$, and $\hat{h}(aw) = h(a)\tau^m\hat{h}(w)$. By induction on $|w|$ we can show that for every $s \in S$ and $w \in Act^*$ is $w \in \mathit{Traces}(s)$ iff $\hat{h}(w) \in \mathit{Traces}(\langle s, \varepsilon\rangle)$.

If $s \not\sqsubseteq_{tr} s'$ then there is some $w \in Act^*$ such that $w \in \mathit{Traces}(s)$ and $w \notin \mathit{Traces}(s')$. This implies that

$$\hat{h}(w) \in \mathit{Traces}(\langle s, \varepsilon\rangle) \qquad \text{and} \qquad \hat{h}(w) \notin \mathit{Traces}(\langle s', \varepsilon\rangle).$$

□

3.4.5 Correctness of the Construction of the RLBA

Theorem 3.7 *The problem* RLBA-EQ$_{\mathcal{R}}$ *is* EXPTIME*-hard for any* $\mathcal{R}$ *such that* $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$.

Proof. Let us return to the construction of B and consider the corresponding $\mathcal{T}_A$. We define a mapping $h : Act_A \to Act_B^*$ such that $h(\langle\alpha\rangle) = \tau a_1 \tau\tau a_2 \tau \cdots \tau a_n \tau$ for $\alpha \in \mathit{Conf}_\wedge$, and $h(\langle\alpha, i\rangle) = \tau a_1 \tau\tau a_2 \tau \ldots \tau a_n \tau i$ for $\alpha \in \mathit{Conf}_\vee - \mathit{Conf}_{rej}$, where $\mathit{desc}(\alpha) = a_1 a_2 \cdots a_n$. We apply the transformation described in the previous subsection with h and $m = 2n + 4$ to $\mathcal{T}_A$, and we obtain $\mathcal{T}'_A$. It is straightforward to construct a bisimulation that relates configurations of B and states of $\mathcal{T}'_A$.

States α_{rej} and α_0 from $\mathcal{T}_A$ correspond to a rejecting, respectively initial, configuration of A. If A accepts w, then $\alpha_{rej} \not\sqsubseteq_{tr} \alpha_0$ in $\mathcal{T}_A$, and so $\langle\alpha_{rej}, \varepsilon\rangle \not\sqsubseteq_{tr} \langle\alpha_0, \varepsilon\rangle$ in $\mathcal{T}'_A$, and if A does not accept w, then $\alpha_{rej} \sim \alpha_0$ in $\mathcal{T}_A$ and $\langle\alpha_{rej}, \varepsilon\rangle \sim \langle\alpha_0, \varepsilon\rangle$ in $\mathcal{T}'_A$ by Lemma 3.6. The same holds for the corresponding configurations of B. This shows that the described construction is correct.

RLBA B with two configurations can be constructed from an instance of ALBA-ACCEPT in a logarithmic space, since it is obvious that some fixed number of pointers pointing to symbols in the instance would be sufficient for the construction. The problem ALBA-ACCEPT is EXPTIME-complete and EXPTIME is closed under complement. □

So from Theorem 3.7 and Lemmas 3.4 and 3.5 we obtain the following result:

Theorem 3.8 *The problems* PCH-EQ$_{\mathcal{R}}$ *and* PN-EQ$_{\mathcal{R}}$ *are* EXPTIME*-hard for any* $\mathcal{R}$ *such that* $\sim \subseteq \mathcal{R} \subseteq \sqsubseteq_{tr}$.

3.5 Summary of the Results

Summary of known results for equivalence-checking problems is given for explicit finite-state systems, for parallel composition of finite-state systems that synchronize on common action but do not use hiding, and for other types of composed systems (PCH, 1-safe Petri nets, ...).

Explicit finite-state systems:

- 'Simulation-like' equivalences are PTIME-complete.
- 'Trace-like' equivalences are PSPACE-complete.
- Any relation between bisimulation equivalence and trace preorder is (at least) PTIME-hard.

Parallel composition where systems synchronize on common actions but without hiding:

- 'Simulation-like' equivalences are EXPTIME-complete.
- 'Trace-like' equivalences are PSPACE-complete.
- Any relation between bisimulation equivalence and simulation preorder is EXPTIME-hard.

Parallel composition with hiding, 1-safe Petri nets, and other composed systems:

- 'Simulation-like' equivalences are EXPTIME-complete.
- 'Trace-like' equivalences are in EXPSPACE.
- Any relation between bisimulation equivalence and trace preorder is (at least) EXPTIME-hard.

Chapter 4

One-Counter Automata

One-Counter Automata (OCA) and One-Counter Nets (OCN) were defined in Section 2.6. This chapter presents after a short overview of known results in Section 4.1 two results. Section 4.2 contains a proof of undecidability of simulation equivalence (and simulation preorder) for OCA. This result was published in [32]. The Sections 4.3 and 4.4 then present a general method for proving DP-hardness of equivalence-checking and model-checking problems concerning one-counter automata and one-counter nets, published in [31].

(Recall that the class DP [48] consists of those languages which are expressible as a difference of two languages from NP, and is generally conjectured to be larger than the union of NP and coNP. Section 4.3.2 contains further comments on DP.)

The 'generic part' of the method is presented in Section 4.3, where we define a simple fragment of Presburger arithmetic, denoted OCL ("One-Counter Logic") which has two important properties:

- It is sufficiently powerful so that satisfiability and unsatisfiability of boolean formulas are both polynomially reducible to the problem of deciding the truth of formulas of OCL, which implies that this latter problem is DP-hard (Theorem 4.5).

- It is sufficiently simple so that the problem of deciding the truth of OCL formulas is polynomially reducible to various equivalence-checking and model-checking problems (thus providing the "application part" of the proposed method). The reduction is typically constructed inductively on the structure of OCL formulas, thus making the proofs readable and easily verified.

The method is applied in Section 4.4 to several problems. Subsection 4.4.1 describes application to the $\mathsf{OCN} \leftrightarrow \mathsf{OCN}$ problem where $\leftrightarrow$ is any relation which subsumes bisimilarity and is subsumed by simulation preorder (thus, besides bisimilarity and simulation equivalence also, e.g., ready simulation equivalence or 2-nested simulation equivalence), showing DP-hardness of these problems (Theorem 4.8). This results improves the coNP lower bound for the $\mathsf{OCN} \sim \mathsf{OCN}$ problem established in [36]. In Subsection 4.4.2 we concentrate on simulation problems between one-counter and finite-state automata, and prove that $\mathsf{OCA} \sqsubseteq \mathsf{FS}$, $\mathsf{FS} \sqsubseteq \mathsf{OCA}$, and $\mathsf{OCA} \simeq \mathsf{FS}$ are all DP-hard (Theorem 4.12). Finally, in Section 4.5 we draw some conclusions and present a detailed summary of known results.

4.1 State of the Art

The $\mathsf{OCN} \sqsubseteq \mathsf{OCN}$ problem was first considered in [1], where it was shown that if two one-counter net processes are related by *some* simulation, then they are also related by a semilinear simulation (i.e. a simulation definable in Presburger arithmetic), which suffices for semidecidability (and thus decidability) of the positive subcase. (The negative subcase is semidecidable by standard arguments.) A simpler proof was given later in [32] by employing certain 'geometric' techniques which allow us to conclude that the simulation preorder over a given one-counter net is itself semilinear. Moreover, it was shown there that the $\mathsf{OCA} \sqsubseteq \mathsf{OCA}$ problem is undecidable. The proof of this undecidability is presented in Section 4.2.

The decidability of the $\mathsf{OCA} \sim \mathsf{OCA}$ problem was demonstrated in [23] by showing that the greatest bisimulation relation over the states of a given one-counter automaton is also semilinear. The relationship between simulation and bisimulation problems for processes of one-counter automata has been studied in [29] where it was shown that one can effectively reduce certain simulation problems to their bisimulation counterparts by applying a technique proposed in [38]. The complexity of bisimilarity-checking with one-counter automata was studied in [36], where the problem $\mathsf{OCN} \sim \mathsf{OCN}$ is shown to be coNP-hard and the problem of *weak* bisimilarity between OCN and FS processes even DP-hard; moreover, the problem $\mathsf{OCA} \sim \mathsf{FS}$ was shown to be solvable in polynomial time. Complexity bounds for simulation-checking were given in [37], where it was shown that the problems $\mathsf{OCN} \sqsubseteq \mathsf{FS}$

and $\mathsf{FS} \sqsubseteq \mathsf{OCN}$ (and thus also $\mathsf{OCN} \simeq \mathsf{FS}$) are in PTIME, while $\mathsf{OCA} \sqsubseteq \mathsf{FS}$ and $\mathsf{OCA} \simeq \mathsf{FS}$ are coNP-hard (and solvable in exponential time).

4.2 Undecidability Result

To show undecidability of $\mathsf{OCA} \sqsubseteq \mathsf{OCA}$ we use a reduction from the halting problem for Minsky machines with 2 counters, which is well-known to be undecidable [45]. We use the following definition:

Definition 4.1 *A* Minsky machine C *with two nonnegative counters* c_1, c_2 *is a program*

$$1 : COMM_1;\ 2 : COMM_2;\ \ldots;\ n : COMM_n$$

where $COMM_n$ *is a halt-command and* $COMM_i$ $(i = 1, 2, \ldots, n-1)$ *are commands of the following two types (assuming* $1 \leq k, k_1, k_2 \leq n, 1 \leq j \leq 2$*)*

(1) $c_j := c_j + 1;$ *goto* k

(2) if $c_j = 0$ *then goto* k_1 *else* $(c_j := c_j - 1;$ *goto* $k_2)$

Note that the computation of the machine C (starting with $COMM_1$, the counters initialized to 0) is deterministic.

Theorem 4.2 *The problem* $\mathsf{OCA} \sqsubseteq \mathsf{OCA}$ *is undecidable even for deterministic one-counter automata. (This holds even in restricted cases, where one of these one-counter automata is fixed.)*

Proof. Given a Minsky machine C with 2 counters c_1, c_2, we describe the construction of two deterministic one-counter automata A_1 and A_2, with specified control states p, q respectively, such that $p(0) \sqsubseteq q(0)$ iff C does not halt. We define the set of actions Act of A_1 and A_2 as $\{\mathbf{i}_1, \mathbf{i}_2, \mathbf{z}_1, \mathbf{z}_2, \mathbf{d}_1, \mathbf{d}_2, \mathbf{h}\}$. The action $\mathbf{i}_j$, where $j \in \{1, 2\}$, represents the execution of a command of type (1) on the counter c_j. Similarly the actions $\mathbf{z}_j$ and $\mathbf{d}_j$ represent the execution of a command of type (2) on the counter c_j when c_j is zero (action $\mathbf{z}_j$) or non-zero (action $\mathbf{d}_j$), respectively. The action $\mathbf{h}$ represents the execution of the *halt*-command. A computation of C can be described with a sequence of symbols from Act, where different symbols correspond to the different actions of C.

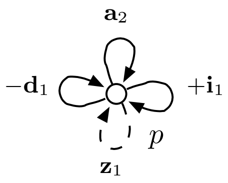


Figure 4.1: Construction of A_1 when A_1 is fixed

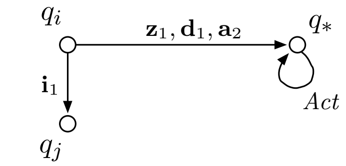
In an obvious way, C can be transformed to a deterministic one-counter automaton A_1 with n control states (n being the number of commands of C), with the set of actions Act, and such that its counter 'behaves' like c_1 (actions $\mathbf{i}_1$, $\mathbf{z}_1$, $\mathbf{d}_1$ depend on c_1 and change it) while the actions $\mathbf{i}_2$, $\mathbf{z}_2$, $\mathbf{d}_2$ ignore the counter (c_2 is 'missing'). Thus a computation of A_1 can digress from that of C by performing an action $\mathbf{d}_2$ instead of $\mathbf{z}_2$ or vice versa. The one-counter automaton A_2 can be constructed similarly, now with the counter corresponding to c_2 while c_1 is ignored. Moreover, for each control state of A_2 with 'outgoing arcs' labelled with $\mathbf{z}_2$ (enabled when c_2 is zero) and $\mathbf{d}_2$ (enabled when c_2 is positive) we add new 'complementary' arcs labelled $\mathbf{z}_2$ (for positive) and $\mathbf{d}_2$ (for zero) which lead to a special control state q_* which has a loop for any action (ignoring the counter). Note that A_2 remains deterministic. Finally we add a new outgoing arc to the halting control state of A_1 labelled with the action $\mathbf{h}$ and we do not add any outgoing arcs to the halting control state of A_2. Let p, q be the initial states of A_1 and A_2 respectively. Obviously we have $p(0) \sqsubseteq q(0)$ iff C does not halt.

Below we briefly describe two modifications of this construction which show that A_1 or A_2 can be fixed (not depending on C), even with just one or two control states respectively.

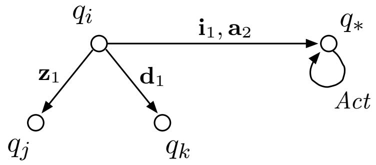
A_1 **fixed:** A_1 has only one state, $Q_1 = \{p\}$, and A_2 has control states $Q_2 = \{q_1, q_2, \ldots, q_n, q_*\}$. The common set of actions is $Act = \{\mathbf{i}_1, \mathbf{z}_1, \mathbf{d}_1, \mathbf{a_2}\}$ ($\mathbf{i}_2$, $\mathbf{z}_2$, and $\mathbf{d}_2$ are merged into one symbol $\mathbf{a}_2$). See Figure 4.1 for a construction of A_1. The outgoing transitions in q_i in A_2 depend on the type of the i-th command of C, see Figure 4.2.

A_2 **fixed:** A_1 has control states $Q_1 = \{p_1, p_2, \ldots, p_n\}$, and$A_2$ has control states $Q_2 = \{q_0, q_*\}$. The common set of actions is $Act = \{\mathbf{a}_1, \mathbf{i}_2, \mathbf{z}_2, \mathbf{d}_2, \mathbf{h}\}$ ($\mathbf{i}_1$, $\mathbf{z}_1$, and $\mathbf{d}_1$ are merged into one symbol $\mathbf{a}_1$). See Figure 4.3 for a con-

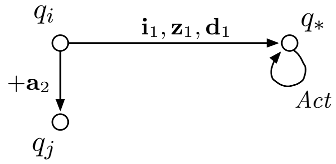
$i : c_1 := c_1 + 1;\ \mathit{goto}\ j$



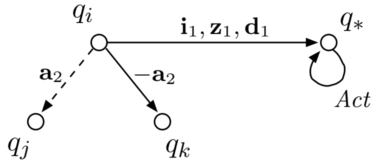
$i : \mathit{if}\ c_1 = 0\ \mathit{then}\ \mathit{goto}\ j$
$\quad \mathit{else}\ (c_1 := c_1 - 1;\ \mathit{goto}\ k)$



$i : c_2 := c_2 + 1;\ \mathit{goto}\ j$



$i : \mathit{if}\ c_2 = 0\ \mathit{then}\ \mathit{goto}\ j$
$\quad \mathit{else}\ (c_2 := c_2 - 1;\ \mathit{goto}\ k)$



$i : \mathit{halt}$

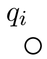


Figure 4.2: Construction of A_2 when A_1 is fixed

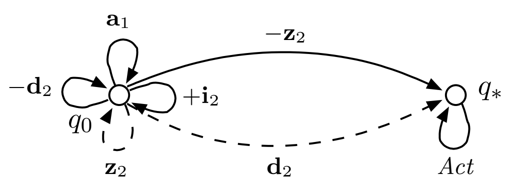


Figure 4.3: Construction of A_2 when A_2 is fixed

struction of A_2. Outgoing transitions in p_i depend again on the type of i-th command in C, see Figure 4.4. $\square$

Corollary 4.3 *The problem* $\mathsf{OCA} \simeq \mathsf{OCA}$ *is undecidable for nondeterministic one-counter automata.*

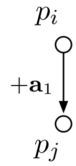
Proof. For initial control states p and q, taken from A_1 and A_2 respectively, we give a construction of A'_1 with initial control state r_1 and A'_2 with initial control state r_2 so that $p(0) \sqsubseteq q(0)$ iff $r_1(0) \simeq r_2(0)$. The result then follows directly from Theorem 4.2.

We take A'_1 to be the disjoint union of A_1 and A_2, adding a new control state r_1 and putting $\delta'_1(r, a) = \{(p, 0),\ (q, 0)\}$ (a is an arbitrary action; here is the only use of nondeterminism when A_1 and A_2 are deterministic). A'_2 arises from A_2 by adding a new control state r_2 and putting $\delta'_2(r_2, a) = \{(q, 0)\}$. Figure 4.5 illustrates this construction. It is easily seen that $r_2(0) \sqsubseteq r_1(0)$, and that $r_1(0) \sqsubseteq r_2(0)$ iff $p(0) \sqsubseteq q(0)$. $\square$

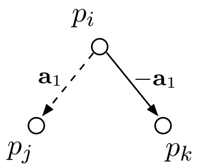
4.3 The OCL Fragment of Arithmetic

In this section, we introduce a fragment of (Presburger) arithmetic, denoted OCL ("One-Counter Logic"). We then show how to encode the problems of satisfiability and unsatisfiability of boolean formulas in OCL, and thus deduce DP-hardness of the truth problem for (closed formulas of) OCL. The name of the language is motivated by a relationship to one-counter automata which will be explored in the next section.

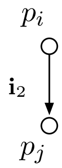
$i : c_1 := c_1 + 1;\ \textit{goto } j$



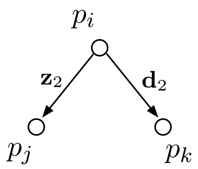
$i : \textit{if } c_1 = 0 \textit{ then goto } j$
$\quad \textit{else } (c_1 := c_1 - 1;\ \textit{goto } k)$



$i : c_2 := c_2 + 1;\ \textit{goto } j$



$i : \textit{if } c_2 = 0 \textit{ then goto } j$
$\quad \textit{else } (c_2 := c_2 - 1;\ \textit{goto } k)$



$i : \textit{halt}$

Figure 4.4: Construction of A_1 when A_2 is fixed

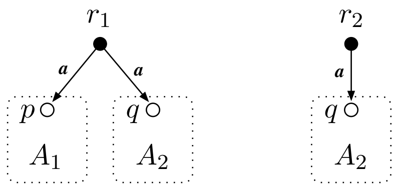


Figure 4.5: Reduction from $\mathsf{OCA} \sqsubseteq \mathsf{OCA}$ to $\mathsf{OCA} \simeq \mathsf{OCA}$

4.3.1 Definition of OCL

OCL can be viewed as a certain set of first-order arithmetic formulas. We shall briefly give the syntax of these formulas; the semantics will be obvious. Since we only consider the interpretation of OCL formulas in the standard structure of natural numbers $\mathbb{N}$, the problem of deciding the truth of a closed OCL formula is well defined:

Problem: TruthOCL

Instance: A closed formula $Q \in$ OCL.

Question: Is Q true ?

Let x and y range over (first-order) *variables*. A formula $Q \in$ OCL can have at most one free variable x (i.e., outside the scope of quantifiers). We shall write $Q(x)$ to indicate the free variable (if there is one) of Q; that is, $Q(x)$ either has the one free variable x, or no free variables at all.

For a number $k \in \mathbb{N}$, $\lceil k \rceil$ stands for a special term denoting k. We can think of $\lceil k \rceil$ as

$$SS \ldots S0$$

i.e., the successor function S applied k times to 0.

We use *size*(Q) to denote *size* of the formula Q. Recall that constants are represented in unary.

The formulas Q of OCL and their sizes are defined inductively as follows:

	Q	$size(Q)$
(a)	$x = 0$	1
(b)	$\lceil k \rceil \mid x$ (‘k divides x’; $k>0$)	$k+1$
(c)	$\lceil k \rceil \nmid x$ (‘k does not divide x’; $k>0$)	$k+1$
(d)	$Q_1(x) \wedge Q_2(x)$	$size(Q_1) + size(Q_2) + 1$
(e)	$Q_1(x) \vee Q_2(x)$	$size(Q_1) + size(Q_2) + 1$
(f)	$\exists y \leq x : Q'(y)$ (x and y distinct)	$size(Q') + 1$
(g)	$\forall x : Q'(x)$	$size(Q') + 1$

The size of an instance Q of TruthOCL is stipulated to be *size*(Q).

We shall need to consider the truth value of a formula $Q(x)$ in a valuation assigning a number $n \in \mathbb{N}$ to the (possibly) free variable x. This is given by the formula $Q[n/x]$ obtained by replacing each free occurrence of the variable x in Q by n. Slightly abusing notation, we shall denote this by $Q(n)$. (Symbols like i, j, k, n range over natural numbers, not variables.) For example, if $Q(x)$ is the formula $\exists y \leq x : ((3 \mid y) \wedge (2 \nmid y))$, then $Q(5)$ is true while $Q(2)$ is false; and if $Q(x)$ is a closed formula, then the truth value of $Q(n)$ is independent of n.

4.3.2 DP-hardness of TruthOCL

Recall the following problem:

Problem: SAT-UNSAT

INSTANCE: A pair (φ, ψ) of boolean formulas in conjunctive normal form (CNF).

QUESTION: Is it the case that φ is satisfiable while ψ is unsatisfiable ?

This problem is DP-complete, which corresponds to an intermediate level in the polynomial hierarchy, harder than both Σ_1^p and Π_1^p but still contained in Σ_2^p and Π_2^P (cf., e.g., [48]). Our aim here is to show that SAT-UNSAT is polynomial-time reducible to TRUTHOCL. In particular, we show how, given a boolean formula φ in CNF, we can in polynomial time construct a (closed) formula of OCL which claims that φ is satisfiable, and also a formula of OCL which claims that φ is unsatisfiable (Theorem 4.5).

First we introduce some notation. Let $Var(\varphi) = \{x_1, x_2, \ldots, x_m\}$ denote the set of (boolean) variables in φ. Furthermore, let π_j (for $j \geq 1$) denote the j^{th} prime number. For every $n \in \mathbb{N}$ we define the assignment

$$\nu_n : Var(\varphi) \to \{true, false\}$$

by

$$\nu_n(x_j) = \begin{cases} true, & \text{if } \pi_j \mid n, \\ false, & \text{otherwise.} \end{cases}$$

Note that for an arbitrary assignment ν there exists an $n \in \mathbb{N}$ such that $\nu_n = \nu$; it suffices to take

$$n = \prod_{j=1}^{m} h_j$$

where $h_j = \pi_j$ if $\nu(x_j) = true$, and $h_j = 1$ otherwise.

By $\|\varphi\|_\nu$ we denote the truth value of φ under the assignment ν.

Lemma 4.4 *There is a polynomial-time algorithm which, given a boolean formula φ in CNF, constructs* OCL-*formulas $Q_\varphi(x)$ and $\overline{Q}_\varphi(x)$ such that both size(Q_φ) and size$(\overline{Q}_\varphi)$ are in $O(|\varphi|^3)$, and such that for every $n \in \mathbb{N}$*

$$Q_\varphi(n) \textit{ is true} \quad \textit{iff} \quad \overline{Q}_\varphi(n) \textit{ is false} \quad \textit{iff} \quad \|\varphi\|_{\nu_n} = \textit{true}.$$

Proof. Let $Var(\varphi) = \{x_1, \ldots, x_m\}$. Given a literal ℓ (that is, a variable x_j or its negation $\overline{x}_j$), define the OCL-formula $Q_\ell(x)$ as follows:

$$Q_{x_j}(x) \;=\; \lceil \pi_j \rceil \mid x \quad \text{and} \quad Q_{\overline{x}_j}(x) \;=\; \lceil \pi_j \rceil \nmid x.$$

Clearly, $Q_\ell(n)$ is true iff $Q_{\overline{\ell}}(n)$ is false iff $\|\ell\|_{\nu_n} = \textit{true}$.

- Formula $Q_\varphi(x)$ is obtained from φ by replacing each literal ℓ with $Q_\ell(x)$. It is clear that $Q_\varphi(n)$ is true iff $\|\varphi\|_{\nu_n} = \textit{true}$.
- Formula $\overline{Q}_\varphi(x)$ is obtained from φ by replacing each $\wedge$, $\vee$, and ℓ with $\vee$, $\wedge$, and $Q_{\overline{\ell}}(x)$, respectively. It is readily seen that $\overline{Q}_\varphi(n)$ is true iff $\|\varphi\|_{\nu_n} = \textit{false}$.

It remains to evaluate the size of Q_φ and $\overline{Q}_\varphi$. Here we use a well-known fact from number theory (cf., e.g., [3]) which says that π_m is in $O(m^2)$. Hence *size*(Q_ℓ) is in $O(|\varphi|^2)$ for every literal ℓ of φ. As there are $O(|\varphi|)$ literal occurrences and $O(|\varphi|)$ boolean connectives in φ, we can see that *size*(Q_φ) and *size*$(\overline{Q}_\varphi)$ are indeed in $O(|\varphi|^3)$. □

We now come to the main result of the section.

Theorem 4.5 *Problem* SAT-UNSAT *is reducible in polynomial time to problem* TRUTHOCL. *Therefore,* TRUTHOCL *is* DP*-hard.*

Proof. We give a polynomial-time algorithm which, given an instance (φ, ψ) of SAT-UNSAT, constructs a closed OCL-formula Q, with *size*(Q) in $O(|\varphi|^3 + |\psi|^3)$, such that Q is true iff φ is satisfiable and ψ is unsatisfiable.

The formula Q will be of the form $Q_1 \wedge Q_2$ where Q_1 is true iff φ is satisfiable and Q_2 is true iff ψ is unsatisfiable.

Expressing the unsatisfiability of ψ is straightforward: by Lemma 4.4, ψ is unsatisfiable iff the OCL-formula

$$\forall x : \overline{Q}_\psi(x)$$

is true. Thus, let Q_2 be this formula.

Expressing the satisfiability of φ is more involved. Let $g = \pi_1\pi_2\ldots\pi_m$, where $Var(\varphi) = \{x_1,\ldots,x_m\}$. Clearly φ is satisfiable iff there is some $n \leq g$ such that $\|\varphi\|_{\nu_n} = \mathit{true}$. Hence φ is satisfiable iff the OCL-formula $\exists y \leq x : Q_\varphi(y)$ is true for any valuation assigning some $i \geq g$ to x.

As it stands, it is unclear how this might be expressed. However, note that the equivalence still holds if we replace the condition "$i \geq g$" with "i is a non-zero multiple of g". In other words, φ is satisfiable iff for every $i \in \mathbb{N}$ we have that either $i = 0$, or $g \nmid i$, or there is some $n \leq i$ such that $Q_\varphi(n)$ is true. This can be written as

$$\forall x \ : \ x = 0 \ \vee \ (\lceil \pi_1 \rceil \nmid x \vee \cdots \vee \lceil \pi_m \rceil \nmid x) \ \vee \ \exists y \leq x : Q_\varphi(y)$$

We thus let Q_1 be this formula.

Hence, (φ, ψ) is a positive instance of the SAT-UNSAT problem iff the formula

$$Q = Q_1 \wedge Q_2$$

is true. To finish the proof, we observe that $\mathit{size}(Q)$ is indeed in $O(|\varphi|^3 + |\psi|^3)$. □

4.3.3 TruthOCL is in Π_2^p

The conclusions we draw for our verification problems are that they are DP-hard, as we reduce the DP-hard problem TRUTHOCL to them. We cannot improve this lower bound by much using the reduction from TRUTHOCL, as TRUTHOCL is in Π_2^p. In this section we sketch the ideas of a proof of this fact.

Theorem 4.6 TRUTHOCL *is in* Π_2^p

Proof. We start by first proving that for every formula $Q(x)$ of OCL there is a d such that $0 < d \leq 2^{\mathit{size}(Q)}$ and $Q(i) = Q(i-d)$ for every $i > 2^{\mathit{size}(Q)}$. Hence, $\forall x : Q(x)$ holds iff $\forall x \leq 2^{\mathit{size}(Q)} : Q(x)$ holds. (Note that $\forall x \leq 2^{\mathit{size}(Q)} : Q(x)$ is not a formula of OCL.)

We prove the existence of d for every formula $Q(x)$ by induction on the structure of $Q(x)$. If $Q(x)$ is $x = 0$ then we can take $d = 1$, and if $Q(x)$ is $\lceil k \rceil \mid x$ or $\lceil k \rceil \nmid x$ then we can take $d = k$.

If $Q(x)$ is $Q_1(x) \wedge Q_2(x)$ or $Q_1(x) \vee Q_2(x)$, then we may assume by the induction hypothesis the existence of the relevant d_1 for Q_1 and d_2 for Q_2. We can then take $d = d_1 d_2$ to give the desired property that $Q(i) = Q(i-d)$ for every $i > 2^{size(Q)}$.

If $Q(x)$ is $\exists y \leq x : Q'(y)$ where x and y distinct then by the induction hypothesis there is a d' such that $0 < d' \leq 2^{size(Q')}$ and $Q'(i) = Q'(i-d')$ for every $i > 2^{size(Q')}$. It follows that if $Q'(i)$ is true for some i, then it is true for some $i \leq 2^{size(Q')} < 2^{size(Q)}$. Furthermore, if $Q'(i)$ is true for some i then $Q(j)$ is true for every $j \geq i$; on the other hand, if $Q'(i)$ is false for every i, then $Q(j)$ is false for every j. Thus we can take $d = 1$.

If $Q(x)$ is $\forall y : Q'(y)$, then x is not free in $Q'(y)$, so the truth value of $Q(i)$ does not depend on i and we can take $d = 1$.

Next we note that every OCL-formula $Q(x)$ can be transformed into a formula $\widehat{Q}(x)$ (which need not be in OCL) in (pseudo-)prenex form

$$\begin{aligned}&(\forall x_1 \leq 2^{size(Q_1)}) \cdots (\forall x_k \leq 2^{size(Q_k)}) \\ &\qquad\qquad (\exists y_1 \leq z_1) \cdots (\exists y_\ell \leq z_\ell)\, \mathcal{F}(x_1, \ldots, x_k, y_1, \ldots, y_\ell)\end{aligned}$$

where

- $\forall x_i : Q_i(x_i)$ is a subformula of $Q(x)$;
- each $z_i \in \{x_1, \ldots, x_k,\, y_1, \ldots, y_{i-1}\}$; and
- $\mathcal{F}(x_1, \ldots, x_k,\, y_1, \ldots, y_\ell)$ is a $\wedge, \vee$-combination of atomic subformulas of $Q(x)$.

This can be proved by induction on the structure of $Q(x)$. The only case requiring some care is the case when $Q(x)$ is of the form $\exists y \leq x : Q'(y)$, because $\exists y \forall z : P(y,z)$ and $\forall z \exists y : P(y,z)$ are not equivalent in general, but they are in our case, as z never depends on y due to restrictions in OCL. Note that the size of $\widehat{Q}(x)$ is polynomial in $size(Q)$ (assuming that $2^{size(Q_1)}, \ldots, 2^{size(Q_k)}$ are encoded in binary).

We can construct an alternating Turing machine which first uses its universal states to assign all possible values (bounded as mentioned above) to $x_1, \ldots, x_k$, then uses its existential states to assign all possible values to $y_1, \ldots, y_\ell$, and finally evaluates (deterministically) the formula

$$\mathcal{F}(x_1, \ldots, x_k,\, y_1, \ldots, y_\ell)$$

It is clear that this alternating Turing machine can be constructed so that it works in time which is polynomial in $size(Q)$. This implies the membership of TRUTHOCL in Π_2^p. □

4.4 Application to One-Counter Automata Problems

As we mentioned above, the language OCL was designed with one-counter automata in mind. The problem TRUTHOCL can be relatively smoothly reduced to various verification problems for such automata, by providing relevant constructions ("implementations") for the various cases (a)-(g) of the OCL definition, and thus it constitutes a useful tool for proving lower complexity bounds (DP-hardness) for these problems. We shall demonstrate this for the OCN $\leftrightarrow$ OCN problem, where $\leftrightarrow$ is any relation satisfying that $\sim \subseteq \leftrightarrow \subseteq \sqsubseteq$, and then also for the OCA $\sqsubseteq$ FS, FS $\sqsubseteq$ OCA, and OCA $\simeq$ FS problems.

The method was also used in [31] by A. Kučera for showing DP-hardness of model-checking with the logic EF and one-counter nets. However this result is not presented in this thesis.

4.4.1 Results for One-Counter Nets

In this section we show that, for any relation $\leftrightarrow$ satisfying $\sim \subseteq \leftrightarrow \subseteq \sqsubseteq$, the problem of deciding whether two (states of) one-counter nets are in $\leftrightarrow$ is DP-hard. We first state an important technical result, but defer its proof until after we derive the desired theorem as a corollary.

Proposition 4.7 *There is an algorithm which, given a formula $Q = Q(x) \in$ OCL as input, halts after $O(size(Q))$ steps and outputs a one-counter net with two distinguished control states p and p' such that for every $k \in \mathbb{N}$ we have:*

- *if $Q(k)$ is true then $p(k) \sim p'(k)$;*
- *if $Q(k)$ is false then $p(k) \not\sqsubseteq p'(k)$.*

Note that if Q is a closed formula, then this implies that $p(0) \sim p'(0)$ if Q is true, and $p(0) \not\sqsubseteq p'(0)$ if Q is false.

Theorem 4.8 *For any relation $\leftrightarrow$ such that $\sim \subseteq \leftrightarrow \subseteq \sqsubseteq$, the following problem is* DP*-hard:*

INSTANCE*: A one-counter net with two distinguished control states p and p'.*

QUESTION*: Is $p(0) \leftrightarrow p'(0)$?*

Proof. We recall that problem TRUTHOCL is DP-hard (Theorem 4.5), and we shall reduce it to our problem. Given an instance of TRUTHOCL, i.e., a *closed* formula $Q \in$ OCL, we use the (polynomial) algorithm of Proposition 4.7 to construct a one-counter net with the two distinguished control states p and p'. If Q is true, then $p(0) \sim p'(0)$, and hence $p(0) \leftrightarrow p'(0)$, and if Q is false, then $p(0) \not\sqsubseteq p'(0)$, and hence $p(0) \not\leftrightarrow p'(0)$. $\square$

Proof of Proposition 4.7: We proceed by induction on the structure of Q. For each case, we show an *implementation*, i.e., the corresponding one-counter net N_Q with two distinguished control states p and p'. Constructions are sketched by figures which use our notational conventions; the distinguished control states are denoted by black dots (the left one p, the right one p'). It is worth noting that we only use two actions, a and b.

(a) $Q(x) = (x = 0)$: A suitable (and easily verifiable) implementation looks as depicted in Figure 4.6.

Figure 4.6: Construction for $Q(x) = (x = 0)$

(b,c) $Q(x) = \lceil k \rceil \mid x$ or $Q(x) = \lceil k \rceil \nmid x$, where $k>0$: Given $J \subseteq \{0, 1, 2, \ldots, k-1\}$, let $R_J(x) = ((x \bmod k) \in J)$. We shall show that the formula $R_J(x)$ can be implemented in our sense; taking $J = \{0\}$ then gives us the construction for case (b), and taking $J = \{1, \ldots, k-1\}$ gives us the construction for case (c).

An implementation of $R_J(x)$, where for the point of illustration we have $1, 2 \in J$ but $0, 3, k-1 \notin J$, looks as shown in Figure 4.7. In this picture, each node q_i has an outgoing edge going to a "dead" state; this edge is labelled b if $i \in J$ and labelled $-b$ if $i \notin J$. It is straightforward to check that the proposed implementation of $R_J(x)$ is indeed correct.

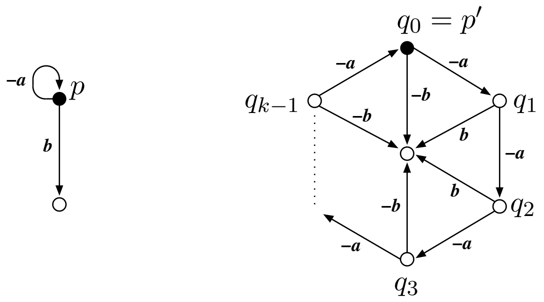


Figure 4.7: Construction for $R_J(x)$

(d) $Q(x) = Q_1(x) \wedge Q_2(x)$: We can assume (by induction) that implementations N_{Q_1} of $Q_1(x)$ and N_{Q_2} of $Q_2(x)$ have been constructed. N_Q is constructed, using N_{Q_1} and N_{Q_2}, as shown in Figure 4.8. The dotted

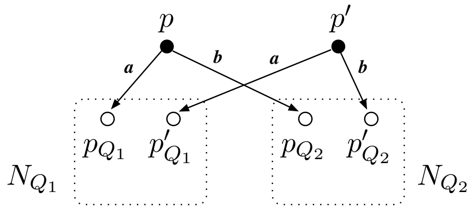


Figure 4.8: Construction for $Q(x) = Q_1(x) \wedge Q_2(x)$

rectangles represent the graphs associated to N_{Q_1} and N_{Q_2} (only the distinguished control states are depicted). Verifying the correctness of this construction is straightforward.

(e) $Q(x) = Q_1(x) \vee Q_2(x)$: As in case (d), the construction uses the implementations of $Q_1(x)$ and $Q_2(x)$; but the situation is slightly more involved in this case, see Figure 4.9. To verify correctness, we first consider the case when $Q(k)$ is true. By induction, either $p_{Q_1}(k) \sim p'_{Q_1}(k)$ or $p_{Q_2}(k) \sim p'_{Q_2}(k)$. In the first case, $p_{Q_1}(k) \sim p'_{Q_1}(k)$ implies that $p_1(k) \sim p_2(k)$, which in turn implies that $p(k) \sim p'(k)$; similarly, in the second case, $p_{Q_2}(k) \sim p'_{Q_2}(k)$ implies that $p_1(k) \sim p_3(k)$, which also implies that $p(k) \sim p'(k)$. Hence in either case $p(k) \sim p'(k)$.

Now consider the case when $Q(k)$ is false. By induction, $p_{Q_1}(k) \not\sqsubseteq p'_{Q_1}(k)$ and $p_{Q_2}(k) \not\sqsubseteq p'_{Q_2}(k)$. Obviously, $p_{Q_1}(k) \not\sqsubseteq p'_{Q_1}(k)$ implies that

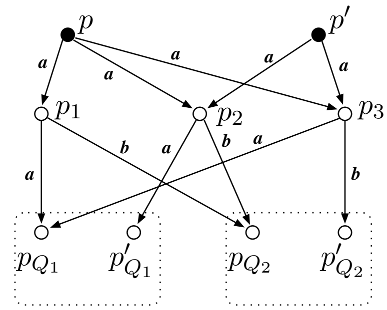


Figure 4.9: Construction for $Q(x) = Q_1(x) \vee Q_2(x)$

$p_1(k) \not\sqsubseteq p_2(k)$, and $p_{Q_2}(k) \not\sqsubseteq p'_{Q_2}(k)$ implies that $p_1(k) \not\sqsubseteq p_3(k)$. From this we have $p(k) \not\sqsubseteq p'(k)$.

(f) $Q(x) = \exists y \leq x : Q_1(y)$ (where x, y are distinct): We use the construction from Figure 4.10. To verify correctness, we first consider the case

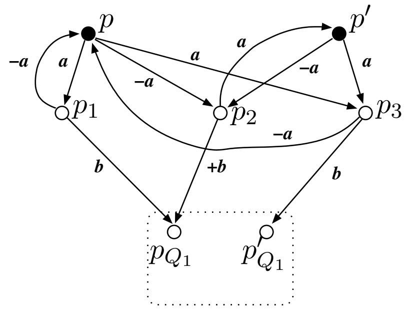


Figure 4.10: Construction for $Q(x) = \exists y \leq x : Q_1(y)$

when $Q(k)$ is true. This means that $Q_1(i)$ is true for some $i{\leq}k$, which by induction implies that $p_{Q_1}(i) \sim p'_{Q_1}(i)$ for this $i{\leq}k$. Our result, that $p(k) \sim p'(k)$, follows immediately from the following:

Claim 4.9 *For all k, if $p_{Q_1}(i) \sim p'_{Q_1}(i)$ for some $i{\leq}k$, then $p(k) \sim p'(k)$.*

Proof. By induction on k. For the base case ($k{=}0$), if $p_{Q_1}(i) \sim p'_{Q_1}(i)$ for some $i{\leq}0$, then $p_{Q_1}(0) \sim p'_{Q_1}(0)$, which implies that $p_1(0) \sim p_3(0)$, and hence that $p(0) \sim p'(0)$. For the induction step ($k{>}0$), if

$p_{Q_1}(i) \sim p'_{Q_1}(i)$ for some $i \le k$, then either $p_{Q_1}(k) \sim p'_{Q_1}(k)$, which implies that $p_1(k) \sim p_3(k)$ which in turn implies that $p(k) \sim p'(k)$; or $p_{Q_1}(i) \sim p'_{Q_1}(i)$ for some $i \le k-1$, which by induction implies that $p(k-1) \sim p'(k-1)$, which implies that $p_1(k) \sim p_2(k-1)$, which in turn implies that $p(k) \sim p'(k)$. $\square$

Next, we consider that case when $Q(k)$ is false. This means that $Q_1(i)$ is false for all $i \le k$, which by induction implies that $p_{Q_1}(i) \not\sqsubseteq p'_{Q_1}(i)$ for all $i \le k$. Our result, that $p(k) \not\sqsubseteq p'(k)$, follows immediately from the following:

Claim 4.10 *For all k, if $p(k) \sqsubseteq p'(k)$ then $p_{Q_1}(i) \sqsubseteq p'_{Q_1}(i)$ for some $i \le k$.*

Proof. By induction on k. For the base case (k=0), if $p(0) \sqsubseteq p'(0)$ then $p_1(0) \sqsubseteq p_3(0)$, which in turn implies that $p_{Q_1}(0) \sqsubseteq p'_{Q_1}(0)$. For the induction step (k>0), if $p(k) \sqsubseteq p'(k)$ then either $p_1(k) \sqsubseteq p_2(k-1)$ or $p_1(k) \sqsubseteq p_3(k)$. In the first case, $p_1(k) \sqsubseteq p_2(k-1)$ implies that $p(k-1) \sqsubseteq p'(k-1)$, which by induction implies that $p_{Q_1}(i) \sqsubseteq p'_{Q_1}(i)$ for some $i \le k-1$ and hence for some $i \le k$; and in the second case, $p_1(k) \sqsubseteq p_3(k)$ implies that $p_{Q_1}(k) \sqsubseteq p'_{Q_1}(k)$. $\square$

(g) $Q = \forall x : Q_1(x)$: The implementation in Figure 4.11 can be easily verified.

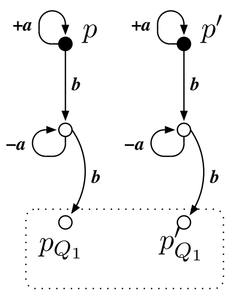


Figure 4.11: Construction for $Q = \forall x : Q_1(x)$

For any $Q \in$ OCL, the described construction terminates after $O(size(Q))$ steps, because we add only a constant number of new nodes in each subcase except for (b) and (c), where we add $O(k)$ new nodes (recall that the size of $\lceil k \rceil$ is $k+1$). $\square$

4.4.2 Simulation Problems for One-Counter Automata and Finite-State Systems

Now we establish DP-hardness of the $\mathsf{OCA} \sqsubseteq \mathsf{FS}$, $\mathsf{FS} \sqsubseteq \mathsf{OCA}$, and $\mathsf{OCA} \simeq \mathsf{FS}$ problems. Again, we use the inductively defined reduction from the problem TruthOCL, only the particular constructions are now slightly different.

By an *implementation* we now mean a tuple

$$(A, F, F', A')$$

where A, A' are one-counter automata, and F, F' are finite-state systems; the role of distinguished states is now played by the initial states, denoted q for A, f for F, f' for F', and q' for A'. We again first state an important technical result, and again defer its proof until after we derive the desired theorem as a corollary.

Proposition 4.11 *There is an algorithm which, given $Q = Q(x) \in \mathrm{OCL}$ as input, halts after $O(size(Q))$ steps and outputs an implementation*

$$(A, F, F', A')$$

where q, f, f' and q' are the initial control states of A, F, F' and A', respectively, such that for every $k \in \mathbb{N}$ we have:

$$Q(k) \text{ is true} \quad \text{iff} \quad q(k) \sqsubseteq f \quad \text{iff} \quad f' \sqsubseteq q'(k).$$

Note that if Q is a closed formula, then this implies that

$$Q \text{ is true} \quad \text{iff} \quad q(0) \sqsubseteq f \quad \text{iff} \quad f' \sqsubseteq q'(0).$$

Theorem 4.12 *Problems* $\mathsf{OCA} \sqsubseteq \mathsf{FS}$*,* $\mathsf{FS} \sqsubseteq \mathsf{OCA}$*, and* $\mathsf{OCA} \simeq \mathsf{FS}$ *are* DP*-hard.*

Proof. Recalling that TruthOCL is DP-hard, DP-hardness of the first two problems readily follows from Proposition 4.11.

DP-hardness of the third problem follows from a simple (general) reduction of $\mathsf{OCA} \sqsubseteq \mathsf{FS}$ to $\mathsf{OCA} \simeq \mathsf{FS}$: given a one-counter automaton A with initial state q, and a finite-state system F with initial state f, we first transform F to F_1 by adding a new state f_1 and transition $f_1 \xrightarrow{a} f$, and then create A_1 by taking (disjoint) union of A, F_1 and adding $\overline{f_1} \xrightarrow{a} q$, where $\overline{f_1}$ is the copy of f_1 in A_1. Clearly $q(k) \sqsubseteq f$ iff $\overline{f_1}(k) \simeq f_1$. $\square$

Proof of Proposition 4.11: We proceed by induction on the structure of Q. In the constructions we use only two actions, a and b; this also means that a state with non-decreasing a and b loops is *universal*, i.e, it can simulate "everything".

(a) $Q(x) = (x = 0)$: A straightforward implementation is depicted in Figure 4.12.

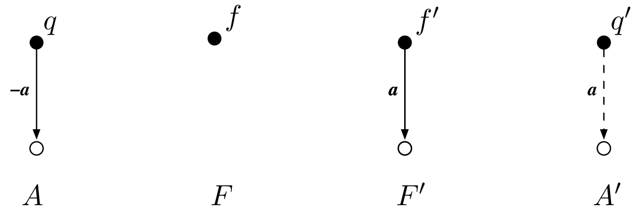


Figure 4.12: Construction for $Q(x) = (x = 0)$

(b,c) $Q = \lceil k \rceil \mid x$ or $Q = \lceil k \rceil \nmid x$, where $k > 0$: Given $J \subseteq [0, k-1]$, let $R_J(x) = ((x \bmod k) \in J)$. We shall show that the formula $R_J(x)$ can be implemented in our sense; taking $J = \{0\}$ then gives us the construction for case (b), and taking $J = \{1, \ldots, k{-}1\}$ gives us the construction for case (c).

An implementation of $R_J(x)$, where $1, 2 \in J$ but $0, 3, k{-}1 \notin J$, looks as depicted in Figures 4.13 and 4.14. In this pictures, node f_i has

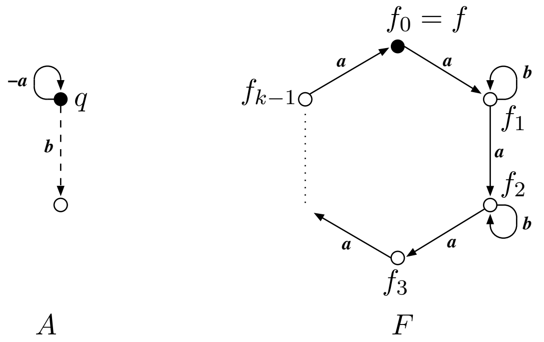
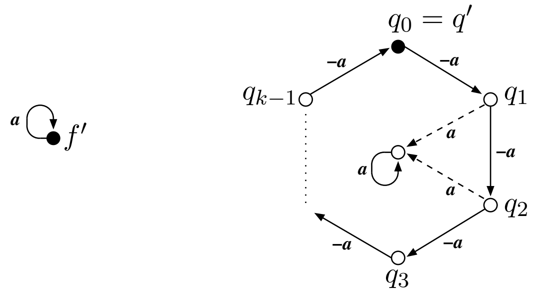


Figure 4.13: Construction of A and F for $R_J(x)$

a b-loop in F, and node q_i has an outgoing dashed a-edge in A', iff



F' A'

Figure 4.14: Construction of F' and A' for $R_J(x)$

$i \in J$. It is straightforward to check that the proposed implementation of $R_J(x)$ is indeed correct.

(d) $Q(x) = Q_1(x) \wedge Q_2(x)$: The elements of the implementation

$$(A_Q, F_Q, F'_Q, A'_Q)$$

for Q can be constructed from the respective elements of the implementations for Q_1, Q_2 (assumed by induction): A_Q from A_{Q_1} and A_{Q_2}; F_Q from F_{Q_1} and F_{Q_2}; F'_Q from F'_{Q_1} and F'_{Q_2}; and A'_Q from A'_{Q_1} and A'_{Q_2}. All these cases follow the schema depicted in Figure 4.15. Correctness is easily verifiable.

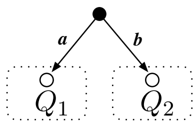


Figure 4.15: Construction for $Q(x) = Q_1(x) \wedge Q_2(x)$

(e) $Q(x) = Q_1(x) \vee Q_2(x)$: We give constructions just for A and F (the constructions for F' and A' are almost identical) in Figure 4.16. For any k, $Q(k)$ is true iff $Q_1(k)$ is true or $Q_2(k)$ is true, which by induction is true iff $q_{Q_1}(k) \sqsubseteq f_{Q_1}$ or $q_{Q_2}(k) \sqsubseteq f_{Q_2}$, which is true iff $q_1(k) \sqsubseteq f_1$ or $q_1(k) \sqsubseteq f_2$, which in turn is true iff $q(k) \sqsubseteq f$.

(f) $Q(x) = \exists y \leq x : Q_1(y)$ (where x, y are distinct): We use the constructions depicted in Figures 4.17 and 4.18. We prove that the

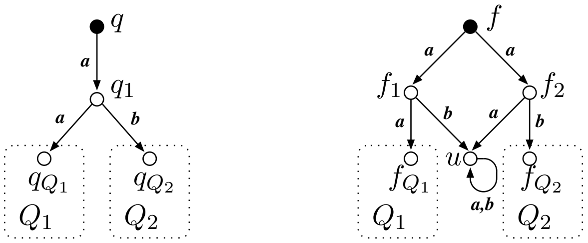


Figure 4.16: Construction for $Q(x) = Q_1(x) \vee Q_2(x)$

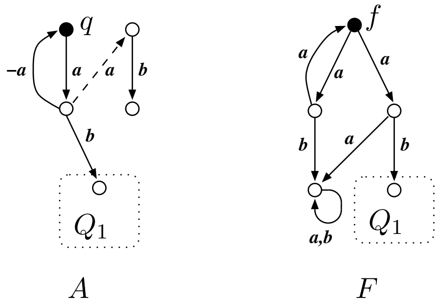


Figure 4.17: Construction of A and F when $Q(x) = \exists y \leq x : Q_1(y)$

construction is correct for F' and A' (the other case being similar). $Q(k)$ is true iff $Q_1(i)$ is true for some $i{\leq}k$, which by induction is true iff $f'_{Q_1} \sqsubseteq q'_{Q_1}(i)$ for some $i{\leq}k$, which in turn is true iff $f'_1 \sqsubseteq q'_2(i)$ for some $i{\leq}k$. Our result, that this is true iff $f' \sqsubseteq q'(k)$, follows immediately from the following:

Claim 4.13 *For all k, $f' \sqsubseteq q'(k)$ iff $f'_1 \sqsubseteq q'_2(i)$ for some $i{\leq}k$.*

Proof. By induction on k. For the base case ($k{=}0$), the result is immediate. For the induction step ($k{>}0$), first note that $f'_1 \sqsubseteq q'_1(k{-}1)$ iff $f' \sqsubseteq q'(k{-}1)$, which by induction is true iff $f'_1 \sqsubseteq q'_2(i)$ for some $i{\leq}k{-}1$. Thus $f' \sqsubseteq q'(k)$ iff $f'_1 \sqsubseteq q'_2(k)$ or $f'_1 \sqsubseteq q'_1(k{-}1)$, which is true iff $f'_1 \sqsubseteq q'_2(k)$ or $f'_1 \sqsubseteq q'_2(i)$ for some $i{\leq}k{-}1$, which in turn is true iff $f'_1 \sqsubseteq q'_2(i)$ for some $i{\leq}k$. □

(g) $Q = \forall x : Q_1(x)$: It is easy to show the correctness of the implementation in Figure 4.19.

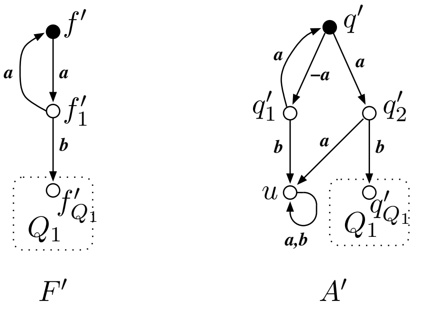


Figure 4.18: Construction of F' and A' when $Q(x) = \exists y \leq x : Q_1(y)$

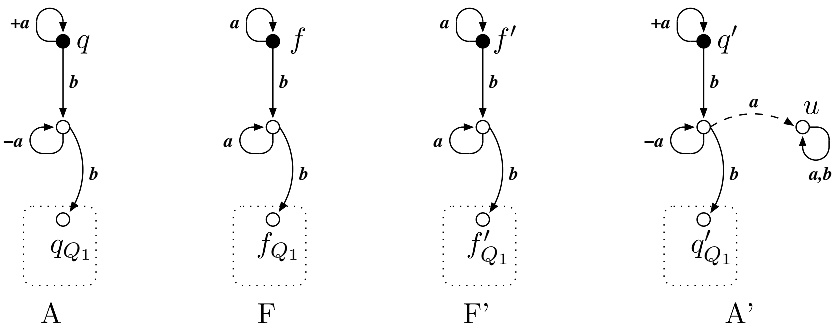


Figure 4.19: Construction for $Q = \forall x : Q_1(x)$

For any $Q \in$ OCL, the described construction terminates after $O(size(Q))$ steps, because we add only a constant number of new nodes in each subcase except for (b) and (c), where we add $O(k)$ new nodes. $\square$

4.5 Summary of the Results

Intuitively, the reason why we could not lift the DP lower bound to some higher complexity class (e.g., PSPACE) is that there is no apparent way to implement a "step-wise guessing" of assignments which would allow us to encode, e.g., the QBF problem. The difficulty is that if we modify the counter value, we were not able to find a way to check that the old and new values encode "compatible" assignments which agree on a certain subset of propositional constants. Each such attempt resulted in an exponential blow-up in the number of control states.

A summary of known results about equivalence-checking with one-counter automata is given below (where $\approx$ denotes weak bisimilarity).

- $\mathsf{OCN} \approx \mathsf{OCN}$ and $\mathsf{OCA} \approx \mathsf{OCA}$ remain open.
- $\mathsf{OCA} \sqsubseteq \mathsf{OCA}$ and $\mathsf{OCA} \simeq \mathsf{OCA}$ are undecidable.
- $\mathsf{OCA} \sim \mathsf{OCA}$, $\mathsf{OCN} \sim \mathsf{OCN}$, $\mathsf{OCN} \sqsubseteq \mathsf{OCN}$ and $\mathsf{OCN} \simeq \mathsf{OCN}$ are decidable and DP-hard, but without any known upper bound.
- $\mathsf{OCA} \approx \mathsf{FS}$, $\mathsf{OCN} \approx \mathsf{FS}$, $\mathsf{OCA} \sqsubseteq \mathsf{FS}$, $\mathsf{FS} \sqsubseteq \mathsf{OCA}$ and $\mathsf{OCA} \simeq \mathsf{FS}$ are decidable, DP-hard, and in $\mathsf{EXPTIME}$. The $\mathsf{EXPTIME}$ upper bound is due to the fact that all of the mentioned problems can be easily reduced to the model-checking problem with pushdown systems (see, e.g., [28, 38]) and the modal μ-calculus which is $\mathsf{EXPTIME}$-complete [62].
- $\mathsf{OCA} \sim \mathsf{FS}$, $\mathsf{OCN} \sim \mathsf{FS}$, $\mathsf{OCN} \sqsubseteq \mathsf{FS}$, $\mathsf{FS} \sqsubseteq \mathsf{OCN}$ and $\mathsf{OCN} \simeq \mathsf{FS}$ are in PTIME.

Chapter 5

Basic Parallel Processes

In this chapter we show two applications of the technique of Jančar that was used in [25] to show that the problem of deciding bisimilarity for BPP systems, BPP-BISIM, is in PSPACE.

The problem BPP-BISIM is formulated as follows:

INSTANCE: Two BPPs Δ_1 and Δ_2 with distinguished initial states.

QUESTION: Is $\Delta_1 \sim \Delta_2$?

The special case when one of Δ_1 and Δ_2 is a finite-state system is denoted BPP-FS-BISIM. It is shown in this chapter that BPP-FS-BISIM can be decided in polynomial time, the result that was published in [35].

The other application of the technique of Jančar presented in this chapter is a polynomial time algorithm for deciding distributed bisimilarity on BPP. This algorithm was not published yet. A polynomial time algorithm for the problem was already published in [40], however the algorithm presented here is more efficient and simpler, and provides an explicit degree of the polynomial bounding the complexity of the algorithm, something that was missing in [40].

The chapter starts with an overview of known results in Section 5.1. The polynomial time algorithm for BPP-FS-BISIM is presented in Section 5.2, and the polynomial time algorithm for deciding distributed bisimilarity in Section 5.3.

5.1 State of the Art

The problem BPP-BISIM was shown to be decidable in [14], but no complexity bounds were presented there. It was proven in [56] that the problem is PSPACE-hard and P. Jančar has shown in [25] that the problem is in PSPACE, and hence the problem is PSPACE-complete. The problem can be decided in polynomial time for *normed* BPP [20, 27].

It was shown in [35] that BPP-FS-BISIM can be solved in polynomial time.

It was shown in [40] that distributed bisimilarity can be decided in polynomial time, however the algorithm in this paper uses the algorithm for normed BPP from [20] as a subroutine, and no explicit degree of the polynomial bounding the complexity of the algorithm is provided there.

5.2 Bisimilarity with a Finite-State System

This section contains description of the algorithm presented in [35]. The running time of the algorithm is $O(n^4)$ where n is the size of the instance. The result 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.

5.2.1 Basic Definitions

In this section we assume that BPPs are represented as (communication-free) Petri nets, and for technical convenience we suppose that also a finite-state systems are represented as Petri nets 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*.

Let Δ_1 and Δ_2 be the BPP and the finite-state system from the instance of BPP-FS-BISIM. We can define their disjoint union Δ in an obvious manner. Let

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

be this disjoint union. Markings of Δ_1 and Δ_2 can be extended to markings of Δ by setting all remaining elements to 0. Let $\mathcal{M}$ be the set of markings of Δ. All markings considered in this section are from $\mathcal{M}$.

We assume that Δ_1 and Δ_2 in the instance of BPP-FS-BISIM are represented succinctly, i.e., in such a way that values of $F(t,p)$ are stored in binary. Let n be the size of the instance in this compact representation. We show that the problem BPP-FS-BISIM can be solved in time $O(n^4)$.

In the rest of this section P_{FS} and Tr_{FS} denote the sets of places and transitions of the finite-state system 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 $x \geq 1$, $\omega \cdot x = x \cdot \omega = \omega$.

5.2.2 The Algorithm

The algorithm is based on ideas that were used by Jančar in [25] to show that BPP-BISIM is in PSPACE. The algorithm constructs 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 [25]. Let $(S, Act, \longrightarrow)$ 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 a *bisimulation invariant* if for each $s, s' \in S$, $s \sim s'$ implies $\mathcal{C}(s) = \mathcal{C}(s')$. If we have a set of functions $\{\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_l\}$ where $\mathcal{C}_i : S \to \mathcal{D}_i$, we say the set is a *bisimulation invariant* iff every $\mathcal{C}_i$ is a bisimulation invariant. A predicate $\mathcal{P}$ on S can be viewed as a mapping $\mathcal{P} : S \to \{0,1\}$, and so $\mathcal{P}$ is a bisimulation invariant iff for each $s, s' \in S$, $s \sim s'$ implies ($\mathcal{P}(s)$ iff $\mathcal{P}(s')$). Note that if $\mathcal{P}$ is a bisimulation invariant, then $\neg\mathcal{P}$ is also a bisimulation invariant.

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

Claim 5.1 *If $\mathcal{P}$ is a bisimulation invariant then* $\text{DIST}(\mathcal{P})$ *is a bisimulation invariant.*

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 $w \in Act^*$ 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 a bisimulation invariant. $\square$

Let us now consider the BPP Δ from the instance of BPP-FS-BISIM. Let $T \subseteq Tr$. We say T is *disabled* in M iff 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 $a \in Act$, then 'T is disabled' is a bisimulation invariant. 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

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

where $c_i \in \mathbb{N}_\omega$ and k is the number of places, see [25] for details. Coefficients $c_1, c_2, \ldots, c_k$ of L for the given Q can be computed by the algorithm in Figure 5.1. Intuitively, c_i is the minimal number of transitions that remove one token in 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 $p = \textsc{pre}(t)$, and chooses between these candidates the one with the minimal value.

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

$$\forall t : \textsc{pre}(t) \in R \Rightarrow (R \cap \textsc{succ}(t) \neq \emptyset)$$

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 5.2 *If $L = \textsc{norm}(Q)$ for some $Q \subseteq P$ and $L(M) = \omega$, then $L(M') = \omega$ for every M' such that $\exists w \in Act^* : M \xrightarrow{w} M'$.*

for each $p \in P$ **do**
  if $p \in Q$ **then** $c_p := \omega$ **else** $c_p := 0$
$Q' := Q$
$T := \{t \in Tr \mid \textsc{pre}(t) \in Q'\}$
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 $\textsc{succ}(t) \cap Q' = \emptyset$ **do**
    remove t from T
    $p := \textsc{pre}(t)$; $R := \textsc{succ}(t)$
    $d_t := 1 + \sum_{q \in R} c_q \cdot F(t, q)$
    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 $\textsc{pre}(t) = p_{min}$
end while

Figure 5.1: Computing coefficients of $\textsc{norm}(Q)$ function

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

$$\delta_t^L = -c_i + \sum_{j=1}^{k} c_j \cdot F(t, p_j) \tag{5.1}$$

where $\textsc{pre}(t) = p_i$. The value δ_t^L represents the "change" on the value of L when the transition t is performed.

Claim 5.3 *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 a bisimulation invariant. 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 a bisimulation invariant, $\mathcal{L}$ is also a bisimulation invariant, 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 the procedure STEP described below.
3. If $\mathcal{L}$ has changed in the previous step, go to 2.

The procedure 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 $\forall L \in \mathcal{F} : \delta_t^L = \delta_{t'}^L$. 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 $\mathcal{T}_1 \cup \{T_0\}$ (or just as $\mathcal{T}_1$ when T_0 is empty). Note that $\mathcal{T}$ 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 show now that the algorithm is correct.

Lemma 5.4 *Every L added to $\mathcal{L}$ by the algorithm is a bisimulation invariant.*

Proof. We proceed by induction on the number of steps. The proposition is trivially true at the start. Assume now the algorithm performs the procedure 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 5.1 it is sufficient to show that $\textsc{zero}(Q)$ is a bisimulation invariant. 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 $\delta_t^L \neq \delta_{t'}^L$. By Claim 5.3 this implies $L(M_1') \neq L(M_2')$, a contradiction. $\square$

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 5.5 *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 the procedure $\textsc{Step}$ for p. Notice that $\textsc{zero}(R)(M_1)$ holds 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 the procedure $\textsc{Step}$.

Let us consider a transition $M_1 \xrightarrow{t} M_1'$ first. 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 5.3 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 5.2, $L(M_1') = L(M_2') = \omega$. 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. $\square$

5.2.3 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 section n denotes the size of the input instance.

The running time of the algorithm depends on implementation details of the procedure Step. In Subsection 5.2.2 we described how to, for the given $p \in P_{FS}$, compute in the procedure Step sets $\mathcal{F}$, Ω-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 for each $p \in P_{FS}$ the algorithm maintains the corresponding values of Ω-carr$(\mathcal{F})$ and $\mathcal{T}$. Note that $\mathcal{T}$ always contains at most $|\text{succ}(p)| + 1$ equivalence classes. The algorithm also maintains for each $T \in \mathcal{T}$ and for Ω-carr$(\mathcal{F})$ a boolean flag indicating whether it has changed since the last invocation of the procedure Step and adds a new function $L = \text{norm}(\Omega\text{-carr}(\mathcal{F}) \cup T)$ to $\mathcal{L}$ only when Ω-carr$(\mathcal{F})$ or T 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 5.6 *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 5.7 *The number of functions added to $\mathcal{L}$ is in $O(n^2)$.*

Proof. Let us consider all invocations of the procedure 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.

Notice that Ω-CARR$(\mathcal{F})$ can only grow, so the number of invocations of the procedure STEP when Ω-CARR$(\mathcal{F})$ has changed is $O(|P|)$. Because $|\mathcal{T}|$ is at most $h+1$ where $h = \textsc{succ}(p)$, 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 $O(|\mathit{Tr}|)$ times. It remains to estimate the total number of classes from $\mathcal{T}_1$. It is in $O(|\mathit{Tr}|)$ as follows from Fact 5.6, since sequence of values of $\mathcal{T}_1$ in subsequent invocations of the procedure STEP can be extended to a sequence of refined partitions by adding some classes to each $\mathcal{T}_1$.

Let us now sum 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

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

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

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

Proposition 5.8 *For each $p \in P$, $\mathit{size}(c_p)$ 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 $\mathit{size}(t)$ be the number of bits of representation of $t \in T$, i.e.,

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

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 $t_1, \ldots, t_i$. □

Proposition 5.9 *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 $p_j \in \textsc{succ}(t_i)$. From Proposition 5.8 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

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

. □

Proposition 5.10 *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 5.1 from Subsection 5.2.2. 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 5.8 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 5.9 for coefficients deduce that aggregated complexity of all multiplications is in $O(n^2)$, from which the result follows. □

Lemma 5.11 *The algorithm adds one L to $\mathcal{L}$ in time $O(n^2)$.*

Proof. As follows from Propositions 5.9 and 5.10, 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 [2, 47]), 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 5.12 *There is an algorithm solving* BPP-FS-BISIM *with running time $O(n^4)$.*

Proof. The algorithm was described above. Lemmas 5.4 and 5.5 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 5.7 and 5.11 that the running time of the algorithm is $O(n^4)$. □

Remark. In fact, the algorithm can be improved so that its time complexity is $O(n^3 \log n)$ using the fact that if the finite-state system in the instance of BPP-FS-BISIM has m states, then if a norm of some state is finite, then its value is at most $m - 1$. This implies that we do not need to consider finite norms greater or equal to m, and we can replace each such value with some special symbol requiring only $O(1)$ space. Values smaller than m can be represented in $O(\log n)$ bits.

5.3 Distributed Bisimilarity

In this section we show a polynomial time algorithm for deciding distributed bisimilarity on BPP, i.e., the algorithm that solves the following problem called BPP-DBISIM:

INSTANCE: A BPP process definition Δ in normal form and two variables $X, Y \in \mathit{Var}(\Delta)$.

QUESTION: Is $X \sim Y$?

This algorithm is a joint work with Petr Jančar and was not published yet.

Remark. In this section the symbol $\sim$ always denotes *distributed* bisimilarity.

In BPP-DBISIM we ask whether processes consisting of single variable are distributed bisimilar. A more general problem where we consider distributed bisimilarity of arbitrary expressions can be easily reduced to BPP-DBISIM.

Let us have an input instance of BPP-DBISIM consisting of some BPP process description Δ in normal form and two variables $X, Y \in \mathit{Var}(\Delta)$. We assume that Δ is given a set of rules of the form

$$X \xrightarrow{a} (P, Q)$$

where numbers of occurrences of variables in P and Q are encoded in binary. Without loss of generality we can assume that $\mathit{Var}(\Delta) = \{X_1, X_2, \ldots, X_k\}$ and that there is a linear order $\prec$ such that $X_1 \prec X_2 \prec \cdots \prec X_k$.

The main idea of the algorithm is to construct a sequence of approximations of distributed bisimilarity $\sim$ from above.

Before going into details we need some technical definitions. Let $\mathcal{D}$ be some (non-empty) set and let $f : \mathit{Var}^{\oplus} \to \mathcal{D}$ be a function. We say that f is a *bisimulation invariant* iff for every $P, Q \in \mathit{Var}^{\oplus}$ $P \sim Q$ implies $f(P) = f(Q)$. The notion of bisimulation invariant can be used also for predicates over processes from $\mathit{Var}^{\oplus}$, because a predicate can be viewed as a function $f : \mathit{Var}^{\oplus} \to \mathcal{D}$ where $\mathcal{D} = \{0, 1\}$.

Rule $t \in \Delta$ is *disabled* in $P \in \mathit{Var}^{\oplus}$ iff $P(\textsc{pre}(t)) = 0$, and it is *enabled* in P iff $P(\textsc{pre}(t)) > 0$. For $T \subseteq \Delta$ we define predicates $\textsc{disabled}(T)$ and $\textsc{enabled}(T)$ on elements of $\mathit{Var}^{\oplus}$ such that $\textsc{disabled}(T)(P)$ holds iff every $t \in T$ is disabled in P, and $\textsc{enabled}(T)(P)$ iff $\neg\textsc{disabled}(T)(P)$.

Let $a \in \mathit{Act}$ be an action and let $T_a = \{t \in \Delta \mid \lambda(t) = a\}$. Note that $\textsc{disabled}(T_a)$ is a bisimulation invariant for every $a \in \mathit{Act}$.

Given a set $T \subseteq \Delta$, the *norm* of T, denoted $\textsc{norm}(T)$, is a function $\textsc{norm}(T) : \mathit{Var}^{\oplus} \to \mathbb{N}$ defined for $P \in \mathit{Var}^{\oplus}$ such that the value of $\textsc{norm}(T)(P)$ is the length of the shortest sequence $Q_0, Q_1, \ldots, Q_m$ of basic processes (the length of the sequence is m) where $Q_0 = P$, and for $1 \leq i \leq m$ there are some a_i and P_i such that $Q_{i-1} \xrightarrow{a_i} [P_i, Q_i]$, and where $\textsc{disabled}(T)(Q_m)$ holds.

It will be shown that if $\textsc{disabled}(T)$ is a bisimulation invariant then also $\textsc{norm}(T)$ is a bisimulation invariant. Moreover, it will be shown that for any $T \subseteq \Delta$, the function $\textsc{norm}(T)$ can be expressed as a *linear function*, i.e., as a function $L : \mathit{Var}^{\oplus} \to \mathbb{N}$ of the form

$$L(P) = \sum_{i=1}^{k} c_i \cdot P(X_i)$$

for $P \in \mathit{Var}^{\oplus}$, where each *coefficient* $c_i \in \mathbb{N}$ can be computed efficiently.

The algorithm creates a set of linear functions $\mathcal{L}$ such that each linear function L from $\mathcal{L}$ will be a norm of some set of rules $T \subseteq \Delta$ such that $\textsc{disabled}(T)$ is a bisimulation invariant. Because of this, each L from $\mathcal{L}$ will be also a bisimulation invariant, and if there will be some L such that $L(P) \neq L(Q)$ this would immediately imply that $P \not\sim Q$. Moreover, $\mathcal{L}$ will be constructed in such a way that if $L(P) = L(Q)$ for every $L \in \mathcal{L}$, this would imply $P \sim Q$. More formally we define for $\mathcal{L}$ an equivalence $\equiv_{\mathcal{L}}$ on $\mathit{Var}^{\oplus}$ such that $P \equiv_{\mathcal{L}} Q$ iff $\forall L \in \mathcal{L} : L(P) = L(Q)$, and we show that $P \sim Q$ iff $P \equiv_{\mathcal{L}} Q$.

The algorithm starts with $\mathcal{L} = \emptyset$ and successively adds functions to $\mathcal{L}$ until no more function can be added.

For each linear function $L = \sum_{i=1}^{k} c_i \cdot x_i$ and rule $t \in \Delta$ we can compute the value δ_t^L representing the ‘change’ on value of L caused by t as

$$\delta_t^L = -c_\ell + \sum_{j=1}^{k} c_j \cdot G(t, X_j)$$

where $X_\ell = \textsc{pre}(t)$. Note that δ_t^L can be easily computed if we know values of coefficients c_i, and that $L(P'') = L(P) + \delta_t^L$ when $P \xrightarrow{t} [P', P'']$.

For each L we can define the equivalence $=_L$ on rules from Δ. Let us have rules $t = (X \xrightarrow{a} (P, P'))$ and $t' = (Y \xrightarrow{a'} (Q, Q'))$. We define $t =_L t'$ iff $a = a'$, $L(P) = L(Q)$, and $\delta_t^L = \delta_{t'}^L$. For a set of linear functions $\mathcal{L}$ we define relation $=_{\mathcal{L}}$ such that $t =_{\mathcal{L}} t'$ iff $t =_L t'$ for each $L \in \mathcal{L}$.

The algorithm maintains a partition of Δ denoted $\mathcal{T}$, and successively refines it. For each class T of $\mathcal{T}$ the algorithm adds to $\mathcal{L}$ a function $L = \textsc{norm}(T)$. The algorithm also maintains a 'queue' $\mathcal{Q}$ of unprocessed classes of $\mathcal{T}$.

The algorithm starts with $\mathcal{Q} = \mathcal{T} = \{T_a \mid a \in Act\}$ where $T_a = \{t \in \Delta \mid \lambda(t) = a\}$, and proceeds as follows.

While $\mathcal{Q} \neq \emptyset$:

1. Take some $T \in \mathcal{Q}$ and remove T from $\mathcal{Q}$.
2. Compute coefficients of $L = \textsc{norm}(T)$ and add L to $\mathcal{L}$.
3. For each $t \in \Delta$ where t is of the form $(X \xrightarrow{a} (P, Q))$ compute values $L(P)$, and δ_t^L, and refine $\mathcal{T}$ according to the relation $=_L$. (Put transitions t, t' such that $t \neq_L t'$ to different classes of $\mathcal{T}$.)
4. Add to $\mathcal{Q}$ each new class of $\mathcal{T}$ created in the previous step.

Now we prove the correctness of the algorithm.

Claim 5.13 *Let $T \subseteq \Delta$ be a set of rules such that* $\textsc{disabled}(T)$ *is a bisimulation invariant. Then* $\textsc{norm}(T)$ *is a bisimulation invariant.*

Proof. Let $L = \textsc{norm}(T)$. We show that if $P \sim Q$ and $L(P) = m$, then $L(Q) = m$, which proves the result. We proceed by induction on m. If $m = 0$, the $L(Q) = 0$ follows from the assumption that T is a bisimulation invariant. Consider $m > 0$ and let us assume $L(Q) = m'$ where $m' \neq m$. Without loss of generality we can assume that $m < m'$. There must be a transition $P \xrightarrow{a} [P', P'']$ such that $L(P'') = m-1$. Since $P \sim Q$, there must be a matching transition $Q \xrightarrow{a} [Q', Q'']$ such that $P' \sim Q'$ and $P'' \sim Q''$, but obviously $L(Q'') \geq m' - 1 > m - 1$, but on the other hand $P'' \sim Q''$ and $L(P'') = m - 1$ imply $L(Q'') = m - 1$ by induction hypothesis, a contradiction. □

Lemma 5.14 *If $P \sim Q$ then $P \equiv_{\mathcal{L}} Q$.*

Proof. It is sufficient to show that each L added to $\mathcal{L}$ in step 2 of the algorithm is a bisimulation invariant. Because L is computed as $\textsc{norm}(T)$ from some $T \subseteq \Delta$, due to Claim 5.13 it is sufficient to show that $\textsc{disabled}(T)$ is a bisimulation invariant. We show that that following invariant holds in every step of the algorithm: For every class T of $\mathcal{T}$, $\textsc{disabled}(T)$ is a bisimulation invariant. To show it, we proceed by induction of the number of steps of the algorithm. The invariant obviously holds at the start of the algorithm when $\mathcal{T}$ contains classes T_a for each $a \in Act$.

Now consider T created in step 3 of the algorithm. Let us assume $P \sim Q$ where $\text{ENABLED}(T)(P)$ and $\text{DISABLED}(T)(Q)$, so there is some $t \in T$ such that $P \xrightarrow{t} [P', P'']$ and there must be some t' such that $Q \xrightarrow{t'} [Q', Q'']$ such that $\lambda(t) = \lambda(t')$ and $P' \sim Q'$ and $P'' \sim Q''$. Obviously $t' \notin T$, and so $t \neq_L t'$ for some $L \in \mathcal{L}$ which is a bisimulation invariant by induction hypothesis and Claim 5.13. From $P \sim Q$ we have $L(P) = L(Q)$, $L(P') = L(Q')$ and $L(P'') = L(Q'')$, but $t \neq_L t'$ implies that $L(P') \neq L(Q')$ or $\delta_t^L \neq \delta_{t'}^L$. Because $L(P'') = L(P) + \delta_t^L$ and $L(Q'') = L(Q) + \delta_{t'}^L$, we obtain that either $L(P') \neq L(Q')$ or $L(P'') \neq L(Q)''$, so $P' \not\sim Q'$ or $P'' \not\sim Q''$, a contradiction. $\square$

Lemma 5.15 *If $P \equiv_{\mathcal{L}} Q$ then $P \sim Q$.*

Proof. We just need to show that $\equiv_{\mathcal{L}}$ is distributed bisimulation. Let us have $P, Q \in Var^{\oplus}$ such that $P \equiv_{\mathcal{L}} Q$ and a rule t such that $P \xrightarrow{t} [P', P'']$. Let T be the set of all rules t' such that $t' =_{\mathcal{L}} t$. Obviously $\text{ENABLED}(T)(P)$ and $\text{ENABLED}(T)(Q)$. So let t' be a rule from T enabled in Q, such that $Q \xrightarrow{t'} [Q', Q'']$ From $t =_{\mathcal{L}} t'$ we obtain $L(P') = L(Q')$ for every $L \in \mathcal{L}$, and so $P' \equiv_{\mathcal{L}} Q'$. $t =_{\mathcal{L}} t'$ also implies $\delta_t^L = \delta_{t'}^L$ for every $L \in \mathcal{L}$, and because $L(P) = L(Q)$ for every L, and $L(P'') = L(P) + \delta_t^L$ and $L(Q'') = L(Q) + \delta_{t'}^L$, we obtain $L(P'') = L(Q'')$ for every L, and so $P'' \equiv_{\mathcal{L}} Q''$. $\square$

Now we show that $\text{NORM}(T)$ for $T \subseteq \Delta$ can be expressed as a linear function

$$L(P) = \sum_{i=1}^{k} c_i \cdot P(X_i)$$

and that coefficients c_i in L can be computed efficiently.

Recall we assume $X_1 \prec X_2 \prec \cdots \prec X_k$. The subroutine that computes coefficients $c_1, c_2, \ldots, c_k$, computes them in this order. Each coefficient c_i can be computed as $\text{NORM}(T)(\{X_i\})$. If $\text{DISABLED}(T)(\{X_i\})$ then $c_i = 0$. Otherwise consider the set T_i of all rules t from T such that $\text{PRE}(t) = X_i$. For each $t \in T_i$ where $t = (X_i \xrightarrow{a} (P, Q))$ we can compute the value $d_t = \text{NORM}(T)(Q)$ as

$$d_t = \sum_{j=1}^{i-1} c_j \cdot Q(X_j)$$

since variables X_j where $j \geq i$ do not occur in Q, and coefficients c_j where $j < i$ were already computed. We compute c_i as $1 + \min\{d_t \mid t \in T_i\}$. Obviously $c_i = \text{NORM}(T)(\{X_i\})$.

To analyze the complexity we need the following definitions. For $x \in \mathbb{N}$, *size*(x) denotes the number of bits of x when encoded in binary. We suppose that $\mathit{size}(x+y) = 1+\max\{\mathit{size}(x), \mathit{size}(y)\}$, and $\mathit{size}(x \cdot y) = \mathit{size}(x)+\mathit{size}(y)$. For $t \in \Delta$, *size*(t) denotes the number of bits in the representation of t where numbers of occurrences of variables on right-hand sides are encoded in binary.

We use n to denote the size of Δ, i.e., $n = \sum_{t\in\Delta} \mathit{size}(t)$.

Proposition 5.16 *$\mathit{size}(c_i) \in O(n)$ for every coefficient c_i of L.*

Proof. Let $X'_1, X'_2, \ldots, X'_{k'}$ be a subset of variables where $c_i > 0$, and let t_i be the rule (with X'_i is on the left-hand side) that was used in the computation of c_i. We show by induction on i the following proposition from which the result directly follows:

$$\mathit{size}(c_i) \leq \sum_{j=1}^{i} \mathit{size}(t_i)$$

This holds trivially for $i = 1$ because c_1 is always 1, so suppose $i > 1$. Let $t_i = (X \xrightarrow{a} (P, Q))$. Note that

$$c_i = 1 + \sum_{j=1}^{i-1} c_j \cdot G(t_i, X_j)$$

and $\mathit{size}(c_j \cdot G(t_i, X_j)) = \mathit{size}(c_j) + \mathit{size}(G(t_i, X_j))$. 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

$$\mathit{size}(\max\{c_j \mid 1 \leq j < i\}) + \sum_{j=1}^{i-1} \mathit{size}(G(t_i, X_j)).$$

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 $t_1, t_2, \ldots, t_{i-1}$. Therefore c_i can be written in the space needed for representations of t_1, t_2, $\ldots$, t_i. $\square$

Proposition 5.17 *Coefficients of a linear function L can be computed in time $O(n^2)$.*

Proof. The most time-consuming step is computation of all d_t. 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_t is computed only once. During computation of d_t we need to determine all products $c_j \cdot G(t, X_j)$ where $G(t, X_j) > 0$. From Proposition 5.16 we know that $size(c_j)$ is in $O(n)$ for every c_j. Hence one product is computed in $O(n \cdot size(G(t, X_j)))$. If we sum complexities of such products for all rules and variables, we get the aggregated complexity of all multiplications

$$O(\sum_{i,j}(n \cdot size(G(t_i, X_j)))) = O(n \cdot \sum_{i,j} size(G(t_i, X_j))) = O(n^2).$$

□

Proposition 5.18 *Values of δ_t^L for a linear function L and all rules $t \in \Delta$ can be computed in time $O(n^2)$.*

Proof. Again the complexity of additions is dominated by the complexity of multiplications $c_j \cdot G(t, X_j)$. Each such product is computed only once. From Proposition 5.16 we know that each c_j is in $O(n)$. Each $G(t, X_j)$ is used only once and is part of Δ. Hence we can similarly as the previous case for coefficients deduce that aggregated complexity of all multiplications is in $O(n^2)$, from which the result follows. □

Theorem 5.19 *There is an algorithm deciding distributed bisimilarity on BPP with running time $O(n^3)$.*

Proof. We can use the algorithm described above to compute $\mathcal{L}$ and then check whether $\{X\} \equiv_{\mathcal{L}} \{Y\}$ where X and Y are variables from the instance of BPP-DBISIM. The correctness of the algorithm follows from Lemmas 5.14 and 5.15. The number of subsets of Δ inserted into $\mathcal{Q}$ (and so the the number of functions in $\mathcal{L}$) is $O(n)$ as follows from Fact 5.6. As follows from Propositions 5.17 and 5.18, each such subset can be processed in time $O(n^2)$, and hence the overall time complexity of the algorithm is $O(n^3)$. □

5.4 Summary of the Results

The algorithm for deciding BPP-FS-BISIM with time complexity $O(n^4)$ was presented. However, the algorithm is not optimal and can be further improved. We plan to do it in the future.

Other problem, where the technique from [25] 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 [26] and PSPACE-hard [56], but no upper bound is known for the problem.

Distributed bisimilarity can be decided in polynomial time ($O(n^3)$) on BPP. As distributed bisimilarity coincides with other non-interleaving equivalences on BPP, that result holds also for all such equivalences.

Chapter 6

Conclusion

This thesis concentrates on complexity and decidability of some equivalence-checking problems, i.e., the problems where for given (descriptions of) transition systems we ask whether they are equivalent with respect to some notion of equivalence, or, more generally, whether they are in some given relation.

Several specific types of systems were considered in the thesis – finite-state systems, parallel compositions of finite-state systems, 1-safe Petri nets, one-counter automata, one-counter nets, and Basic Parallel Processes. Also several different types of relations were studied on these systems, especially bisimulation equivalence, simulation equivalence, simulation preorder, trace equivalence, and trace preorder. Some results hold for whole families of relations, for example for all relations between bisimulation equivalence and trace preorder.

The rest of this chapter contains an overview of the results presented in the thesis and an overview of open problems connected with these problems.

6.1 Summary of the Results

Finite-state systems were considered in Chapter 3. It was shown that deciding any relation between bisimulation equivalence and trace preorder is PTIME-hard for finite-state systems that are explicitly given (they are called explicit finite-state systems in the thesis). It was also shown that the problem of deciding any relation between bisimulation equivalence and trace preorder becomes EXPTIME-hard when we consider different types of composed systems such as parallel composition of finite-state systems that synchronize of common actions where actions can be hidden (i.e., replaced with τ actions), called parallel composition with hiding (PCH) in the thesis, or 1-safe Petri nets. This result approves a conjecture by Rabinovich [51]. To simplify the proof, a new model called *reactive linear bounded automata* (RLBA) was introduced. The EXPTIME-hardness result was proved at first for RLBA. Because RLBA can be 'simulated' by other different types of composed systems such PCH or 1-safe Petri nets, EXPTIME-hardness result applies also to them.

One-counter machines were considered in Chapter 4. It was shown that deciding simulation equivalence and simulation preorder is undecidable for one-counter automata. A new general method for proving DP-hardness of different types of problems concerning one-counter automata was devised. This method was used for proving DP-hardness of deciding any relation between bisimulation-equivalence and simulation preorder for one-counter nets (one-counter automata that cannot test for zero), and DP-hardness of deciding simulation equivalence and simulation preorder between a one-counter automaton and a finite-state system (in both directions). This general method uses a fragment of Presburger arithmetic called One-Counter Logic (OCL) chosen is such a way that deciding the truth of formulas in OCL can be easily reduced to problems concerning one-counter automata. Because, as was shown, deciding the truth of formulas in OCL is DP-hard, DP-hardness of these problems follows.

Basic Parallel Processes (BPP) were studied in Chapter 5. Using a technique of Jančar [25], polynomial time algorithms for deciding bisimulation equivalence between a BPP and a finite-state system (with time complexity $O(n^4)$), and for deciding distributed bisimilarity of two BPP systems (with time complexity $O(n^3)$), were shown. A polynomial time algorithm for the latter problem was already known [40], but the algorithm presented in this thesis has lower time complexity and gives an explicit degree of the polynomial, that was missing in the algorithm presented in [40].

6.2 Open Problems

The problems of deciding bisimulation equivalence of one-counter automata and one-counter nets and problems of deciding simulation equivalence and simulation preorder of one-counter nets are known to be decidable and DP-hard, however no upper bound on the complexity is known for these problems.

Decidability of *weak* bisimulation equivalence on one-counter automata or one-counter nets remains open.

Decidability of *weak* bisimulation equivalence on BPP remains open.

The algorithm for deciding bisimulation equivalence of a BPP and a finite-state system is probably not the most optimal, and the complexity of the problem can be further improved. We conjecture that an algorithm with time complexity $O(n^3)$ exists for this problem.

Appendix A

List of Publications

The following list contains publications of the author in the chronological order:

- Jančar, P., F. Moller and Z. Sawa, *Simulation problems for one-counter machines*, in: *Proceedings of SOFSEM'99*, Lecture Notes in Computer Science **1725** (1999), pp. 404–413. [32]
- Sawa, Z. and P. Jančar, *P-hardness of equivalence testing on finite-state processes*, in: *Proceedings of SOFSEM 2001*, Lecture Notes in Computer Science **2234** (2001), p. 326. [53]
- Jančar, P., A. Kučera, F. Moller and Z. Sawa, *Equivalence-checking with one-counter automata: A generic method for proving lower bounds*, in: *Proceedings of FoSSaCS 2002*, Lecture Notes in Computer Science **2303** (2002), pp. 172–186. [30]
- Sawa, Z., *Equivalence checking of non-flat systems is EXPTIME-hard*, in: *Proceedings of CONCUR 2003*, Lecture Notes in Computer Science **2761** (2003), pp. 237–250. [52]
- Jančar, P., A. Kučera, F. Moller and Z. Sawa, *DP lower bounds for equivalence-checking and model-checking of one-counter automata*, Information and Computation **188** (2004), pp. 1–19. [31]
- Kot, M. and Z. Sawa, *Bisimulation equivalence of a BPP and a finite-state system can be decided in polynomial time*, in: *Proceedings of Infinity 2004 (A Satellite Workshop of CONCUR 2004)*, 2004, pp. 73–81. [35]

Bibliography

[1] Abdulla, P. and K. Čerāns, *Simulation is decidable for one-counter nets*, in: *Proceedings of CONCUR'98*, Lecture Notes in Computer Science **1466** (1998), pp. 253–268.

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

[3] Bach, E. and J. Shallit, "Algorithmic Number Theory. Vol. 1, Efficient Algorithms," The MIT Press, 1996.

[4] Baeten, J. C. M. and W. P. Weijland, *Process algebra*, Cambridge Tracts in Theoretical Computer Science **18** (1990).

[5] Balcázar, J., J. Gabarró and M. Sántha, *Deciding bisimilarity is P-complete*, Formal Aspects of Computing **4** (1992), pp. 638–648.

[6] Bérard, B., M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen and P. McKenzie, "Systems and Software Verification: Model-Checking Techniques and Tools," Springer, 2001.

[7] Bergstra, J. A. and J. W. Klop, *Algebra of communicating processes with abstraction*, Theoretical Computer Science **37** (1985), pp. 77–121.

[8] Bouajjani, A., R. Echahed and P. Habermehl, *Verifying infinite state processes with sequential and parallel composition*, in: *Proceedings of POPL'95* (1995), pp. 95–106.

[9] Burkart, O., D. Caucal, F. Moller and B. Steffen, *Verification on infinite structures*, in: *Handbook of Process Algebra* (2001), pp. 545–623.

[10] Castellani, I., "Bisimulations for Concurrency," Ph.D. thesis, University of Edinburg (1988).

[11] Castellani, I., *Process algebras with localities, chapter 15*, Handbook of Process Algebra (2001), pp. 945–1046.

[12] Chandra, A. K., D. C. Kozen and L. J. Stockmeyer, *Alternation*, Journal of the ACM **28** (1981), pp. 114–133.

[13] Christensen, S., "Decidability and Decomposition in Process Algebras," Ph.D. thesis, The University of Edinburgh (1993).

[14] Christensen, S., Y. Hirsfeld and F. Moller, *Bisimulation is decidable for all basic parallel processes*, in: *Proc. CONCUR'93*, Lecture Notes in Computer Science **715** (1993), pp. 143–157.

[15] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 1999.

[16] Darondeau, P. and P. Degano, *Causal trees*, in: *Automata, Languages and Programming (ICALP '89)* (1989), pp. 234–248.

[17] Emerson, E. A., *Temporal and modal logic*, Handbook of Theoretical Computer Science **B** (1991), pp. 995–1072.

[18] Gibbons, A. and W. Rytter, "Efficient Parallel Algorithms," Cambridge University Press, 1988.

[19] Gorrieri, R., M. Roccetti and S. Stancampiano, *A theory of processes with durational actions*, Theoretical Computer Science **140** (1995), pp. 73–94.

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

[21] Hüttel, H. and S. Shukla, *The complexity of deciding behavioural equivalences and preorders*, Technical Report SUNYA-CS-96-03, State University of New York at Albany (1996).

[22] Immerman, N., "Descriptive Complexity," Springer-Verlag, 1998, 53–54 pp.

[23] Jančar, P., *Decidability of bisimilarity for one-counter processes*, Information and Computation **158** (2000), pp. 1–17.

[24] Jančar, P., *Nonprimitive recursive complexity and undecidability for Petri net equivalences*, Theoretical Computer Science **256** (2001), pp. 23–30.

[25] Jančar, P., *Strong bisimilarity on basic parallel processes is PSPACE-complete*, in: *Proceedings of the eightteenth Annual IEEE Symposium on Logic in Computer Science (LICS-03)9* (2003), pp. 218–227.

[26] Jančar, P. and J. Esparza, *Deciding finiteness of Petri nets up to bisimulation*, in: *Proc. of ICALP'96*, Lecture Notes in Computer Science **1099** (1996), pp. 478–489.

[27] 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.

[28] Jančar, P., A. Kučera and R. Mayr, *Deciding bisimulation-like equivalences with finite-state processes*, Theoretical Computer Science **258** (2001), pp. 409–433.

[29] Jančar, P., A. Kučera and F. Moller, *Simulation and bisimulation over one-counter processes*, in: *Proceedings of STACS 2000*, Lecture Notes in Computer Science **1770** (2000), pp. 334–345.

[30] Jančar, P., A. Kučera, F. Moller and Z. Sawa, *Equivalence-checking with one-counter automata: A generic method for proving lower bounds*, in: *Proceedings of FoSSaCS 2002*, Lecture Notes in Computer Science **2303** (2002), pp. 172–186.

[31] Jančar, P., A. Kučera, F. Moller and Z. Sawa, *DP lower bounds for equivalence-checking and model-checking of one-counter automata*, Information and Computation **188** (2004), pp. 1–19.

[32] Jančar, P., F. Moller and Z. Sawa, *Simulation problems for one-counter machines*, in: *Proceedings of SOFSEM'99*, Lecture Notes in Computer Science **1725** (1999), pp. 404–413.

[33] Jategaonkar, L. and A. R. Meyer, *Deciding true concurrency equivalences on safe, finite nets*, Theoretical Computer Science **154** (1996), pp. 107–143.

[34] Kanellakis, P. C. and S. A. Smolka, *CCS expressions, finite state processes, and three problems of equivalence*, Information and Computation **86** (1990), pp. 43–68.

[35] Kot, M. and Z. Sawa, *Bisimulation equivalence of a BPP and a finite-state system can be decided in polynomial time*, in: *Proceedings of Infinity 2004 (A Satellite Workshop of CONCUR 2004)*, 2004, pp. 73–81.

[36] Kučera, A., *Efficient verification algorithms for one-counter processes*, in: *Proceedings of ICALP 2000*, Lecture Notes in Computer Science **1853** (2000), pp. 317–328.

[37] Kučera, A., *On simulation-checking with sequential systems*, in: *Proceedings of ASIAN 2000*, Lecture Notes in Computer Science **1961** (2000), pp. 133–148.

[38] Kučera, A. and R. Mayr, *Simulation preorder over simple process algebras*, Information and Computation **173** (2002), pp. 184–198.

[39] Laroussinie, F. and P. Schnoebelen, *The state explosion problem from trace to bisimulation equivalence*, in: *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000), Berlin, Germany, Mar.-Apr. 2000*, Lecture Notes in Computer Science **1784** (2000), pp. 192–207.

[40] Lasota, S., *A polynomial-time algorithm for deciding true concurrency equivalences of Basic Parallel Processes*, in: *MFCS: Symposium on Mathematical Foundations of Computer Science*, 2003.

[41] Mayr, R., *Combining Petrin nets and PA-processes*, in: *Proceedings of TACS'97*, Lecture Notes in Computer Science **1281** (1997).

[42] Mayr, R., *Process rewrite systems*, Electronic Notes in Theoretical Computer Science **7** (1997), proceedings of Expressivness in Concurrency (EXPRESS'97).

[43] Mayr, R., *Process rewrite systems*, Information and Computation **156** (2000), pp. 264–286.

[44] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

[45] Minsky, M., "Computation: Finite and Infinite Machines," Prentice-Hall, 1967.

[46] Moller, F., *Infinite results*, in: *Proceedings of CONCUR'96*, Lecture Notes in Computer Science **1119** (1996), pp. 195–216.

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

[48] Papadimitriou, C. H., "Computational Complexity," Addison-Wesley, 1994.

[49] Park, D., *Concurrency and automata on infinite sequences*, in: *Proceedings* 5^{th} *GI Conference*, Lecture Notes in Computer Science **104** (1981), pp. 167–183.

[50] Peterson, J., "Petri Net Theory and the Modelling of Systems," Prentice-Hall, 1981.

[51] Rabinovich, A., *Complexity of equivalence problems for concurrent systems of finite agents*, Information and Computation **139** (1997), pp. 111–129.

[52] Sawa, Z., *Equivalence checking of non-flat systems is EXPTIME-hard*, in: *Proceedings of CONCUR 2003*, Lecture Notes in Computer Science **2761** (2003), pp. 237–250.

[53] Sawa, Z. and P. Jančar, *P-hardness of equivalence testing on finite-state processes*, in: *Proceedings of SOFSEM 2001*, Lecture Notes in Computer Science **2234** (2001), p. 326.

[54] Shukla, S. K., H. B. Hunt, D. J. Rosenkrantz and R. E. Stearns, *On the complexity of relational problems for finite state processes*, Lecture Notes in Computer Science **1099** (1996), p. 466.

[55] Srba, J., *Roadmap of infinite results*, Bulletin of the European Association for Theoretical Computer Science **78** (2002), pp. 163–, columns: Concurrency.

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

[57] Stirling, C., *Modal and temporal logics*, Handbook of Logic in Computer Science **2** (1992), pp. 477–563.

[58] Tarski, A., *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics **5** (1955), pp. 285–309.

[59] Valmari, A. and A. Kervinen, *Alphabet-based synchronisation is exponentially cheaper*, Lecture Notes in Computer Science **2421** (2002), p. 161.

[60] van Glabbeek, R., *The linear time—branching time spectrum*, Handbook of Process Algebra (1999), pp. 3–99.

[61] van Glabbeek, R. J. and U. Goltz, *Equivalence notions for concurrent systems and refinement of actions*, in: A. Kreczmar and G. Mirkowska, editors, *Proc. Conf. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **379** (1989), pp. 237–248.

[62] Walukiewicz, I., *Pushdown processes: Games and model-checking*, Information and Computation **164** (2001), pp. 234–263.