# Decidability and complexity of Petri net problems – an introduction

Javier Esparza⋆

Institut für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Germany

**Abstract.** A collection of 10 "rules of thumb" is presented that helps to determine the decidability and complexity of a large number of Petri net problems.

## 1 Introduction

The topic of this paper is the decidability and complexity of verification problems for Petri nets. I provide answers to questions like "is there an algorithm to decide if two Petri nets are bisimilar?", or "how much time is it needed (in the worst case) to decide if a 1-safe Petri net is deadlock-free?"

My intended audience are people who work on the development of algorithms and tools for the analysis of Petri net models and have some basic understanding of complexity theory. More precisely, I assume that the reader is familiar with the notion of undecidable problem, with the definitions of deterministic and nondeterministic complexity classes like NP or PSPACE, with the notion of hard and complete problems for a complexity class, and with the use of reductions to prove hardness and completeness results. Theoreticians acquainted with the topic of this paper are warned: They won't find much in it that they didn't know before.[1] On the other hand, they might be interested in the paper's unified view of complexity questions for 1-safe and general Petri nets, and in a few simplifications in the presentation of some proofs.

When I was invited to write this paper, I hesitated for a while. I remembered the statement of the Greek scepticist Gorgias:

> Nothing exists;
> if anything does exist, it is unknowable;
> if anything can be known, knowledge of it is incommunicable.

and imagined a Greek chorus advising me not to write the paper because, in their opinion:

> All results about decidability and complexity of Petri nets were already obtained in the early eighties;

⋆ Work partially supported by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen".

[1] Only one result has not been published before, namely a PSPACE algorithm for the model-checking problem of CTL and 1-safe Petri nets, presented in Section 4.

if there are new results, you have included them for sure in the paper "Decidability issues for Petri nets – a survey" you wrote with Mogens Nielsen in 1994 [10];

if you haven't included them in the survey, they are only of interest for specialists; moreover, these results just show that all interesting problems are intractable – finer classifications, like NP-, PSPACE- or EXPSPACE-hardness have no practical relevance.

Since, as you can see, I still decided to write the paper, I would like to anticipate my answer to these three possible criticisms.

• *There have been important recent developments about decidability and complexity questions, of interest for the whole Petri net community.*

During the late seventies and early eighties there was an outburst of theoretical work on decidability and complexity problems for (Place/Transition) Petri nets. Well-known computer scientists, like Rabin, Rackoff, Lipton, Mayr, Meyer, and Kosaraju, just to mention a few, obtained a very impressive collection of results. The decidability of most problems, like boundedness, liveness, reachability, language equivalence, etc. was settled, and in many cases tight complexity bounds were obtained.

However, while these results were being obtained, two developments in computer science opened new problems:

• In the late seventies, temporal logic was proposed as a query language for the specification of reactive and distributed systems; a few years later, model-checking was introduced as a technique for the verification of arbitrary temporal properties. Howell, Rosier, and Yen were the first to study the decidability and complexity of model-checking problems for Petri nets in the second half of the eighties [17, 19, 20]. Today most questions in this research field have been answered [9, 14].

• In the early eighties, process algebras were introduced for the formal description of concurrent and reactive systems. It was seen that language equivalence was not an adequate equivalence notion for this class of systems, since for instance it may consider deadlock-free systems as equivalent to systems with deadlocks. New equivalence relations were introduced, like bisimulation and failures equivalence. In the early nineties, the decidability of these equivalences for systems with infinite state spaces started to receive a lot of attention, and led to renewed interest in Petri nets. Jančar proved only a few years ago a fundamental result showing the undecidability for Petri nets of all equivalence notions described in the literature [22, 21].

These two developments still had another effect. During the eighties, many researchers started to study the relationship of process algebras to Petri nets. Net models in which a place can carry at most one token, like condition/event systems or elementary net systems, turned out to be particularly useful for these studies. These nets, which have by definition a finite number of states, became even more interesting after the introduction of automatic model-checkers, when it was realised that they could be used to model a large number of interesting systems which were within the reach of automatic verification. The questions that had been asked and mostly solved for Place/Transition nets were now asked again for these models. In the last years the complexity of classical properties (reachability, liveness ... ), model-checking problems for different temporal logics, and equivalence problems for different equivalence notions, has been completely determined [2, 23, 31].

• *This paper has a different approach than the '94 survey paper, and has been written to complement it.*

Research on the decidability and complexity of verification problems for Petri nets has produced well over 100 papers, maybe even 150. Many of them have been published in well-known journals, and are thus available in any good library. My survey paper with Mogens Nielsen [10] summarises many results, and provides a rather comprehensive list of references.

Petri net researchers often need information about the complexity of a particular problem (the Petri net mailing list receives now and then postings with this kind of requests). In most cases, a similar problem has already been studied in the literature, and pointers to relevant papers can be found in [10]. If one is familiar with a number of basic techniques, it is easy to apply these existing results to the new problem. However, acquiring this familiarity is at the moment a rather hard task, specially for Ph. D. students: one has to go through many papers and distill an understanding which is not explicitly contained in the papers themselves. The purpose of these pages is to make this task a bit easier. Instead of listing results and references, I concentrate on a few general results of broad applicability. I also provide "rules of thumb", which I think can be more useful than formal theorems.

• *All researchers interested in the development and implementation of analysis algorithms for Petri nets can greatly profit from some basic knowledge on the computational complexity of analysis problems.*

All researchers are regularly confronted with the problem of having to prove or disprove a conjecture. Should one first try to find a proof or a counterexample? The wrong choice can make one lose precious time. Complexity theory can often help by showing that the truth or falsity of the conjecture implies an unlikely fact, like P=NP or NP=PSPACE. I present here some examples in the form of three stories taken from my personal experience:

*Story I.* After graduating in Physics, I became a Ph. D. student of computer science. At that time I knew very little about theoretical computer science, and there were no theoreticians in my environment. I started to work on the analysis of free-choice Petri nets, a net class for which there was hope of finding efficient verification algorithms, and more precisely I began to investigate the liveness problem. My hope was to efficiently transform the problem into a set of linear inequations that could be solved using linear programming. 'Efficiently' meant for me that the number and size of the equations should grow quadratically, say, in the size of the net.

During the next four months I could not find any encoding, but I read some textbooks on theoretical computer science. I came across Garey and Johnson's book on the theory of NP-completeness [12], and I found the problem I was working on (more precisely, its complement) in the list of NP-complete problems at the end of the book. Since there exist polynomial algorithms for Linear Programming but the complement of the liveness problem for free-choice nets was NP-complete, the existence of an efficient encoding would imply P=NP, and so it was highly unlikely.

The NP-completeness of the non-liveness problem for free-choice Petri nets is proved in Section 10.

*Story II.* Some years ago I refereed a paper submitted to the Petri net conference. The paper contained a conjecture on the reachability problem for Petri nets that can be stated as follows. Let $\mathcal{N}$ be a net, and let $M_0$ and $M$ be markings of $\mathcal{N}$ such that $M$ is reachable from $M_0$. Conjecture: $M$ can be reached from $M_0$ through a sequence of transition firings which only visits intermediate markings of size $O(n + m_0 + m)$, where $n, m_0, m$ are the sizes of $\mathcal{N}$, $M_0$ and $M$, respectively. The author of the paper had constructed a random generator of nets and markings and had tested the conjecture in one thousand cases, always with a positive answer.

It is certainly possible to disprove the conjecture by exhibiting a counterexample, but it is faster to use a complexity argument. I show this argument in Section 7.

*Story III.* I have recently come across a paper containing a characterisation of the set of reachable markings of 1-safe Petri nets. A simple complexity analysis shows that the characterization is most probably wrong, although I haven't found a counterexample yet. In order to formulate the characterisation we need some definitions and notations. A *siphon* of a net is a subset of places $R$ satisfying ${}^\bullet R \subseteq R^\bullet$. A *trap* is a subset of places $R$ satisfying $R^\bullet \subseteq {}^\bullet R$. Given a net $\mathcal{N} = (S, T, F)$ and a set $U \subseteq T$, we define the net $\mathcal{N}_U$ as the result of first removing all transitions of $\mathcal{N}$ not belonging to $U$, and then removing all places that are not connected to any transition anymore.

Now, let $\mathcal{N} = (S, T, F)$ be a net, and let $M_0$ and $M$ be markings of $\mathcal{N}$ such that the Petri net $(\mathcal{N}, M_0)$ is 1-safe. The characterization states $M$ is reachable from $M_0$ if and only if there exists a mapping $X : T \to I\!N$ satisfying the following three properties:

(1) for every place $s$, $M(s) = M_0(s) + \sum_{t \in T}(F(t,s) - F(s,t)) \cdot X(t)$,
(2) every nonempty siphon of $\mathcal{N}_{TX}$ is marked at $M_0$, and
(3) every nonempty trap of $\mathcal{N}_{TX}$ is marked at $M$.

where $TX$ is the set of transitions $t$ such that $X(t) > 0$.

I strongly believe that the proof of this result contains a mistake, and that a counterexample exists. I show why in Section 3.[2]

• *The classification of a problem as NP-, PSPACE- or EXPSPACE-hard does have practical relevance*

The complexity of Petri nets was first studied in the seventies, when NP-complete problems were really intractable: computer scientists were unable to deal even with very small instances due to the lack of computing power and of good theoretical results. At that time it probably didn't make so much difference for a practitioner whether a problem was PSPACE-hard or only NP-complete. In my opinion, today's picture is very different:

[2] After I wrote this paper, but before its publication, Stephan Melzer found a counterexample with 5 places and 3 transitions.

– NP-complete problems are no longer "intractable". It is certainly true that all known algorithms that solve them have exponential worst-case complexity. However, today there exist commercial systems for standard NP-complete problems, like satisfiability of propositional logic formulas or integer linear programming problems, that routinely solve instances of large size.
– The last years have witnessed a proliferation of model-checking tools, like COSPAN, PEP, PROD, SMV, SPIN, and others (see [11] and [30] for comprehensive information). Although the problems they solve are PSPACE-complete, they have been successfully applied to the verification of many interesting finite state systems. Commercial versions are starting to appear.
– Experimental tools for the analysis of timed-systems are starting to emerge. Examples are Hy-Tech, KRONOS, UPPAAL [11]. Many of the problems solved by these tools are EXPSPACE-complete. The size of the instances they can handle is certainly much smaller than in the case of model-checkers, but the results are very promising.
– Theorem provers like HOL, Isabelle, PVS, and others are being applied with good success to the verification of systems with infinite state spaces. They use heuristics to try to solve particular instances of undecidable analysis problems.

My conclusion is that the old "tractable – intractable" classification has become too rough. A finer analysis provides very valuable information about the size of instances that can be handled by automatic tools, and about the possibility of applying existing tools to a particular problem.

### Organisation of the paper

The paper is divided into two parts. The first is devoted to 1-safe Petri nets, which are Place/Transition Petri nets having the property that no reachable marking puts more than one token in any place. Nearly all results hold for $n$-safe Petri nets (at most $n$ tokens on a place) too, assuming that the algorithms receive $n$ as part of the input, which implies in particular that $n$ must be known in advance. The second part is devoted to general Place/Transition nets. Both parts are divided into the same four sections. Each section contains one or more "rules of thumb". These are general informal statements which try to summarise a number of formal results in a concise, necessarily informal, but informative way. They could also be called "useful lies": statements which do not tell all the truth and nothing but the truth, but are more useful than a complicated formal theorem with many ifs and buts. There is a total of 10 rules of thumb in the paper; with their help I can solve most of the complexity questions I come across in my own research.

Rules of thumb are displayed in the text like this:

| Rule of thumb 0:<br>To find the rules of thumb, look for pieces of text within a box. |
|---|

This is only a rule of thumb, because other pieces of text are also surrounded by a box, in fact by a *double* box. They are fundamental formal results used to derive the rules of thumb.

> Fundamental results are displayed within a double box.

The first section contains a universal lower bound for "interesting" Petri net problems. The second section deals with upper bounds: for 1-safe Petri nets it is possible to give an almost universal upper bound, whereas the case of general Petri nets is more delicate. The third section deals with equivalence problems: are two given nets equivalent with respect to a given equivalence notion? Upper and lower bounds are considered simultaneously. Finally, the fourth section gives information about how far one can go with polynomial time algorithms.

Only some of the results mentioned in the paper are proved; for others the reader is referred to the literature. The results with a proof are those fulfilling two conditions: they are very general, applicable to a variety of problems, and admit relatively simple, non-technical proofs. I have devoted special effort to presenting proofs in the simplest possible way. My goal was to produce a paper that could be read straight through from beginning to end. I don't know if the goal has been achieved, but I tried my best.

# Table of Contents



## 2 Preliminaries

We assume that the reader is acquainted with the basic notions of net theory, like firing rule, reachable marking, liveness, boundedness, etc., and also with other basic computation models like Turing machines. This section just fixes some notations.

*Petri nets.* A *net* is a triple $\mathcal{N} = (S, T, F)$, where $S$ and $T$ are finite sets of *places* and *transitions*, and $F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*. We identify $F$ with its characteristic function $(S \times T) \cup (T \times S) \rightarrow \{0, 1\}$. The *preset* and *postset* of a place or transition $x$ are denoted by ${}^\bullet x$ and $x^\bullet$, respectively. Given a set $X \subseteq S \cup T$, we denote ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ and $X^\bullet = \bigcup_{x \in X} x^\bullet$. A *marking* is a mapping $M \colon S \rightarrow I\!N$. 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 $M(s) > 0$ for every $s \in {}^\bullet t$. If $t$ is enabled at $M$, then it can *fire* or *occur*, and its firing leads to the successor marking $M'$ which is defined for every place $s$ by

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

The expression $M \xrightarrow{t} M'$ denotes that $M$ enables transition $t$, and that the marking reached by the occurrence of $t$ is $M'$. A finite or infinite sequence $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \cdots$ is called a *firing sequence*. The maximal firing sequences of a Petri net (i.e., the infinite firing sequences plus the finite firing sequences which end with a marking that does not enable any transition) are called *runs*. Given a sequence $\sigma = t_1 t_2 \dots t_n$, $M \xrightarrow{\sigma} M'$ denotes that there exist markings $M_1, M_2, \dots, M_{n-1}$ such that $M \xrightarrow{t_1} M_1 \dots M_{n-1} \xrightarrow{t_n} M'$.

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

We encode a net $(S, T, F)$ as two $|S| \times |T|$ binary matrices *Pre* and *Post*. The entry *Pre*$(s, t)$ is 1 if there is an arc from $s$ to $t$, and 0 otherwise. The entry *Post*$(s, t)$ is 1 if there is an arc from $t$ to $s$, and 0 otherwise. The *size of a net* is the number of bits needed to write down these two matrices, and is therefore $O(|S| \cdot |T|)$. The *size of a Petri net* is the size of the net plus the size of its initial marking. Markings are encoded as vectors of natural numbers. The *size of a marking* is defined as the number of bits needed to write it down as a vector, where each component is written in binary. Observe that the size of a 1-safe Petri net is $O(|S| \cdot |T|)$, since the initial marking has size $O(|S|)$.

A *labelled net* is a fourtuple $(S, T, F, \lambda)$, where $(S, T, F)$ is a net and $\lambda$ is a mapping that associates to each transition $t$ a label $\lambda(t)$ taken from some given set of actions $Act$. Given $a \in Act$, we denote by $M \xrightarrow{a} M'$ that there is some transition $t$ such that $M \xrightarrow{t} M'$ and $\lambda(t) = a$. 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.

*Turing machines.* In the paper we use single tape Turing machines with one-way infinite tapes, i.e., the tape has a first but not a last cell. For our purposes it suffices to consider Turing machines starting on empty tape, i.e., on tape containing only blank symbols. So we define a *(nondeterministic) Turing machine* as a tuple $M = (Q, \Gamma, \delta, q_0, F)$, where $Q$ is the set of states, $\Gamma$ the set of tape symbols (containing a special *blank* symbol), $\delta\colon (Q \times \Gamma) \to \mathcal{P}(Q \times \Gamma \times \{R, L\})$ the transition function, $q_0$ the initial state, and $F$ the set of final states. The *size of a Turing machine* is the number of bits needed to encode its transition relation.

*Linearly and exponentially bounded automata.* We work several times with Turing machines that can only use a finite tape fragment, or equivalently, with Turing machines whose tape has both a first and a last cell. We call them *bounded automata*. If a bounded automaton tries to move to the right from the last tape cell it just stays in the last cell.

A function $f\colon \mathbb{N} \to \mathbb{N}$ induces the class of $f(n)$-*bounded automata*, which contains for all $k \geq 0$ the bounded automata of size $k$ that can use $f(k)$ tape cells. Notice that we deviate from the standard definition, which says that an automaton is $f(n)$-bounded if it can use at most $f(k)$ tape cells for an *input word* of length $k$. Since we only consider bounded automata working on empty tape, the standard definition is not appropriate for us. When $f(n) = n$ and $f(n) = 2^n$ we get the classes of *linearly bounded* and *exponentially bounded automata*, respectively.

*Complexity classes and reductions.* In the paper we use some of the most basic complexity classes, like P, NP, and PSPACE. We also use the class EXPSPACE, defined by[3]

$$\mathrm{EXPSPACE} = \bigcup_{k \geq 0} \mathrm{DSPACE}(2^{n^k})$$

We always work with polynomial reductions, i.e., given an instance $x$ of a problem A we construct in polynomial time an instance $y$ of a problem $B$. Many of the results also hold for logspace reductions, or even log-lin reductions, but we do not address this point.

[3] Notice that some books (for instance [1]) define $\mathrm{EXPSPACE} = \bigcup_{k \geq 0} \mathrm{DSPACE}(k \cdot 2^n)$.

# Part I

# 1-safe Petri nets

We study the complexity of analysis problems for 1-safe Petri nets. Given a 1-safe Petri net $(\mathcal{N}, M_0)$, where $\mathcal{N} = (S, T, F)$, we say that the *possible markings of* $\mathcal{N}$ or just the *markings of* $\mathcal{N}$ are the set of markings that put at most one token in a place. Clearly, there are $2^{|S|}$ possible markings. Each of the markings can be identified with the set of places marked at it. Observe that the size of a marking is linear in the size of the net.

## 3 A universal lower bound

In this section we obtain a universal lower bound for the complexity of deciding whether a 1-safe Petri net satisfies an interesting behavioural property:

> Rule of thumb 1:
> All interesting questions about the behaviour of 1-safe Petri nets are PSPACE-hard.

Notice that a rule of thumb is not a theorem. There are behavioural properties of 1-safe Petri nets that can be solved in polynomial time. For instance, the question "Is the initial marking a deadlock?" can be answered very efficiently; however, it is so trivial that hardly anybody would consider it really interesting. So a more careful formulation of the rule of thumb would be that all questions described in the literature as interesting are at least PSPACE-hard. Here are 14 examples:

- Is the Petri net live?
- Is some reachable marking a deadlock?
- Is a given marking reachable from the initial marking?
- Is there a reachable marking that puts a token in a given place?
- Is there a reachable marking that does not put a token in a given place?
- Is there a reachable marking that enables a given transition?
- Is there a reachable marking that enables more than one transition?
- Is the initial marking reachable from every reachable marking?
- Is there an infinite run?
- Is there exactly one run?
- Is there a run containing a given transition?
- Is there a run that does not contain a given transition?
- Is there a run containing a given transition infinitely often?
- Is there a run which enables a transition infinitely often but contains it only finitely often?

The PSPACE-hardness of all these problems is a consequence of one single fundamental fact, first observed by Jones, Landweber and Lien in 1977 [24]:

> A linearly bounded automaton of size $n$ can be simulated by a 1-safe Petri net of size $O(n^2)$. Moreover, there is a polynomial time procedure which constructs this net.

The notion of simulation used here is very strong: a 1-safe Petri net simulates a Turing machine if there is bijection $f$ between configurations of the machine and markings of the net such that the machine can move from a configuration $c_1$ to a configuration $c_2$ in one step if and only if the Petri net can move from the marking $f(c_1)$ to the marking $f(c_2)$ through the firing of exactly one transition.

Let $A = (Q, \Gamma, \Sigma, \delta, q_0, F)$ be a linearly bounded automaton of size $n$. The computations of $M$ visit at most the cells $c_1, \ldots, c_n$. Let $C$ be this set of cells. The simulating Petri net $N(A)$ contains a place $s(q)$ for each state $q \in Q$, a place $s(c)$ for each cell $c \in C$, and a place $s(a, c)$ for each symbol $a \in \Gamma$ and for each cell $c \in C$. A token on $s(q)$ signals that the machine is in state $q$. A token on $s(c)$ signals that the machine reads the cell $c$. A token on $s(a, c)$ signals that the cell $c$ contains the symbol $a$. The total number of places is $|Q| + n \cdot (1 + |\Sigma|)$.

The transitions of $N(A)$ are determined by the state transition relation of $A$. If $(q', a', R) \in \delta(q, a)$, then we have for each cell $c$ a transition $t(q, a, c)$ whose input places are $s(q)$, $s(c)$, and $s(a, c)$ and whose output places are $s(q')$, $s(a', c)$ and $s(c')$, where $c'$ is the cell to the right of $c$ (this signals that the tape head has moved to the right) unless $c$ is the last cell, in which case $c' = c$. The last cell is an exception, because by assumption the machine cannot move to the right from there. If $(q', a', L) \in \delta(q, a)$ then we add a similar set of transitions; this time the first cell is the exception. The total number of transitions is at most $2 \cdot |Q|^2 \cdot |\Gamma|^2 \cdot n$, and so $O(n^2)$, because the size of $A$ is $O(|Q|^2 \cdot |\Gamma|^2)$.

The initial marking of $N(A)$ puts one token on $s(q_0)$, on $s(c_1)$, and on the place $s(B, c_i)$ for $1 \leq i \leq n$, where $B$ denotes the blank symbol. The total size of the Petri net is $O(n^2)$.

It follows immediately from this definition that each move of $A$ corresponds to the firing of one transition. The configurations reached by $A$ along a computation correspond to the markings reached along its corresponding run. These markings put one token in exactly one of the places $\{s(q) \mid q \in Q\}$, in exactly one of the places $\{s(c) \mid c \in C\}$, and in exactly one of the places $\{s(a, c) \mid a \in \Sigma\}$ for each cell $c \in C$. So $N(A)$ is 1-safe.

In order to answer a question about a linearly bounded automaton $A$ we can construct the net $N(A)$, which is only polynomially larger than $A$, and solve the corresponding question about the runs of $A$. For instance, the question "does any of the computations of $A$ terminate?" corresponds to "has the Petri net $N(A)$ a deadlock?"

It turns out that most questions about the computations of linearly bounded automata are PSPACE-hard. To begin with, the *(empty tape) acceptance problem* is PSPACE-complete:

Given: a linearly bounded automaton $A$.
To decide: if $A$ accepts the empty input.

Moreover, the PSPACE-hardness of this problem is very robust: it remains PSPACE-complete if we restrict it to

- deterministic bounded automata,
- bounded automata having one single accepting state,

– bounded automata having one single accepting configuration.

Many other problems can be easily reduced to the acceptance problem in polynomial time, and so are PSPACE-hard too. Examples are:

– does $A$ halt?,
– does $A$ visit a given state?,
– does $A$ visit a given configuration?
– does $A$ visit a given configuration infinitely often?

We obtain in this way a large variety of PSPACE-hard problems. Since $N(A)$ is only polynomially larger than $A$, all the corresponding Petri net problems are PSPACE-hard as well. For instance, a reduction from the problem "does $A$ ever visit a given configuration?" proves PSPACE-hardness of the reachability problem for 1-safe Petri nets. Furthermore, once we have some PSPACE-hard problems for 1-safe Petri nets we can use them to obtain new ones by reduction. For instance, the following problems can be easily reduced to the problem of deciding if there is a reachable marking that puts a token on a given place:

– is there a reachable marking that concurrently enables two given transitions $t_1$ and $t_2$?
– can a given transition $t$ ever occur?
– is there a run containing a given transition $t$ infinitely often?

13 out of the 14 problems at the beginning of the section (and many others) can be easily proved PSPACE-hard using these techniques. The liveness problem, the first in our list, is a bit more complicated. The interested reader can find the reduction in [2].

**The solution to Story III**

Recall the conjecture of Story III: Let $\mathcal{N} = (S, T, F)$ be a net, and let $M_0$ and $M$ be markings of $\mathcal{N}$ such that the Petri net$(\mathcal{N}, M_0)$ is 1-safe. $M$ is reachable from $M_0$ in $\mathcal{N}$ if and only if there exists a mapping $X : T \to I\!N$ satisfying the following three properties:

(1) for every place $s$, $M(s) = M_0(s) + \sum_{t \in T}(F(t,s) - F(s,t)) \cdot X(t)$,
(2) every nonempty siphon of $\mathcal{N}_{TX}$ is marked at $M_0$, and
(3) every nonempty trap of $\mathcal{N}_{TX}$ is marked at $M$.

where $TX$ is the set of transitions $t$ such that $X(t) > 0$.

We show that if the conjecture is true, then the reachability problem for 1-safe Petri nets belongs to NP. Since we know that this problem is PSPACE-hard, the truth of the conjecture implies NP=PSPACE, which is highly unlikely. So, very probably, the conjecture is false; one should look for a counterexample instead of trying to prove it.

We need a well-known result (see for instance [16]):

| |
|---|
| There is a polynomial time nondeterministic algorithm Feasible($S$) for the problem of deciding if a system of linear equations $S$ with integer coefficients has a solution in the natural numbers. |

It is easy to decide if every siphon of a net $\mathcal{N}$ is marked at a given marking $M$. The following (deterministic) algorithm, due to Starke [33, 5], does it for you. It first computes the largest siphon $R$ contained in the set of places not marked at $M$. Clearly, all nonempty siphons are marked at $M$ if and only if $R$ is empty.

Algorithm All_Siphons_Marked($\mathcal{N}$, $M$):

**variable:** $R$ of type set of places;

**begin**
  $R :=$ set of places of $N$ unmarked under $M$;
  **while** there is $s \in R$ and $t \in {}^\bullet s$ such that $t \notin R^\bullet$ **do**
    $R\colon = R \setminus \{s\}$
  **od**;
  **if** $R = \emptyset$ **then return true**
  **else return false**
**end**

The algorithm All_Traps_Marked is very similar: just change the loop condition to: there is $s \in R$ and $t \in s^\bullet$ such that $t \notin {}^\bullet R$. Clearly, these two algorithms run in polynomial time.

The following nondeterministic algorithm checks conditions (1), (2) and (3). It first guesses the set $TX$ of transitions, and checks that (2) and (3) hold. Then, it checks if condition (1) holds for a vector $X$ such that $TX = \{t \in T \mid X(t) > 0\}$. For that, it checks if the system of equations $\mathcal{S}$ containing the equations of condition (1) plus the equation $X(t) \geq 1$ for every $t \in TX$, and the equation $X(t) = 0$ for every $t \in T \setminus TX$ has a solution.

Algorithm Check_Conditions($\mathcal{N}$, $M_0$, $M$):

**begin**
  guess a subset of transitions $TX$ of $\mathcal{N}$;
  **if** All_Siphons_Marked($\mathcal{N}_{TX}$, $M_0$)
    **and** All_Traps_Marked($\mathcal{N}_{TX}$, $M$)
    **and** Feasible($\mathcal{S}$)
  **then return true fi**
**end**

Since the system of equations $\mathcal{S}$ has linear size in the net $N$, Feasible($\mathcal{S}$) runs in polynomial time in the size of the net. So Check_Conditions runs in polynomial time, and the problem of checking if conditions (1), (2), and (3) hold belongs to NP.

*Remark* Even if we didn't know about the All_Siphons_Marked algorithm, we could still conclude that the conjecture is probably false. Only from the existence of the procedure Feasible($\mathcal{S}$) we can already conclude that the reachability problem for 1-safe nets belongs to $\Sigma_2^P$, the second level of the polynomial-time hierarchy (see for instance [1]). The general opinion of complexity theorists is that $\Sigma_2^P$ = PSPACE is almost as unlikely as NP=PSPACE.

## 4 A nearly universal upper bound

In this section we obtain a nearly universal upper bound matching the PSPACE-hard lower bound of the last section:

> Rule of thumb 2:
> Nearly all interesting questions about the behaviour of 1-safe Petri nets can be decided in polynomial space.

Observe that the rule of thumb says "nearly all" and no longer "all". The reason is that the literature contains at least one interesting question requiring more than polynomial space. This exception to the rule is described at the end of the section.

We substantiate the rule of thumb with the help of temporal logics. Since their first application to computer science in the late seventies by Pnueli and others, temporal logics have become the standard query languages used to express properties of reactive and distributed systems. A good introduction to the application of temporal logics to computer science can be found in [6].

Temporal logics can be *linear-time* and *branching-time*: linear-time logics are interpreted on the single computations of a system, while branching-time logics are interpreted on the tree of all its possible computations. The most popular linear and branching-time temporal logics are LTL (*linear-time propositional temporal logic*) and CTL (*computation tree logic*). Most of the safety and liveness properties of interest for practitioners, like deadlock-freedom, reachability, liveness (in the Petri net sense), starvation-freedom, strong and weak fairness, etc. can be expressed in LTL or in CTL (often in both).

We show that all the properties expressible in LTL and CTL can be decided in polynomial space. Actually, we even show that they can be *uniformly* decided in polynomial space, i.e., we prove that the degree of the polynomial does not depend on the property we consider. More precisely, let $|N|$ denote the size of a Petri net $N$, and let $|\phi|$ denote the length of a formula $\phi$ (its number of symbols). For each of LTL and CTL we give an algorithm that accepts as input a Petri net $N$ and a formula $\phi$, and answers "yes" or "no" according to whether the net satisfies the formula or not; the algorithm uses $O(p(|N| + |\phi|))$ space, where $p$ is a polynomial independent of $N$ and $\phi$.

### 4.1 Linear-time propositional temporal logic

The formulas of LTL are built from a set *Prop* of atomic propositions, and have the following syntax:

$$
\begin{array}{rll}
\phi ::= & p \in Prop & \\
& \neg\phi & \\
& \phi_1 \wedge \phi_2 & \\
& X\phi & (\phi \text{ holds at the next state}) \\
& \phi_1 U \phi_2 & (\phi_1 \text{ holds until } \phi_2 \text{ holds})
\end{array}
$$

Usual abbreviations are $true = p \vee \neg p$, $F\phi = trueU\phi$ (eventually $\phi$), and $G\phi = \neg F \neg \phi$ (always $\phi$).

LTL formulas are interpreted on *computations*. A computation is a finite or infinite sequence $\pi = P(0)P(1)P(2)\ldots$ of sets of atomic propositions. Intuitively, $P(i)$ is the set of propositions that hold in the computation after $i$ steps. For a computation $\pi$ and a point $i$ in the computation, we have that:

$$
\begin{array}{lll}
\pi, i \models p & \text{iff} & p \in P(i) \\
\pi, i \models \neg\phi & \text{iff} & \text{not}(\pi, i \models \phi) \\
\pi, i \models \phi_1 \wedge \phi_2 & \text{iff} & \pi, i \models \phi_1 \text{ and } \pi, i \models \phi_2 \\
\pi, i \models X\phi & \text{iff} & \text{there exists a point } i+1 \text{ in the computation, and} \\
& & \pi, i+1 \models \phi \\
\pi, i \models \phi_1 U \phi_2 & \text{iff} & \text{for some } j \geq i, \text{ we have } \pi, j \models \phi_2 \text{ and} \\
& & \text{for all } k, i \leq k < j, \text{ we have } \pi, k \models \phi_1
\end{array}
$$

We say that a computation $\pi$ satisfies a formula $\phi$, denoted $\pi \models \phi$, if $\pi, 0 \models \phi$.

The atomic propositions are intended to be propositions on the states of a system. They can only be chosen after the class of systems on which the logic is to be applied has been fixed. In the case of 1-safe Petri nets the states of the system are the markings, and so the atomic propositions are predicates on the possible markings of the net. It is then natural to have one atomic proposition per place. The markings satisfying the atomic proposition $s$ are those that put a token in $s$. Observe that a computation is now a sequence of sets of places, and so a sequence of markings. In particular, the sequences of markings obtained from the runs of $N$ by removing the intermediate transitions are computations. Abusing language, we also call these particular computations runs. We now define that a Petri net $N$ satisfies $\phi$ if *all* its runs satisfy $\phi$. Here are some LTL formulas that can be interpreted on the Petri net of Figure 1, which models a variation of Lamport's 1-bit mutual exclusion algorithm for two processes [26]:

(1) All runs are infinite (true for the net of Figure 1): $GXtrue$.
(2) All runs mark place $cs_1$ infinitely often (false): $GFcs_1$.
(3) In all runs, if place $req_1$ becomes marked then place $cs_1$ will eventually become marked (true): $G(req_1 \Rightarrow Fcs_1)$.

Formula (1) expresses deadlock-freedom; formula (3) expresses that the requests of the first process to the critical section are eventually granted.

The model-checking problem for LTL and 1-safe Petri nets consists of, given a 1-safe Petri net $N$ and a formula $\phi$, deciding whether $N$ satisfies $\phi$ or not.

The solution to the model-checking problem we give here makes use of automata theory. We have to introduce automata on infinite words. Let $A = (\Sigma, Q, q_0, \delta, F)$ be a

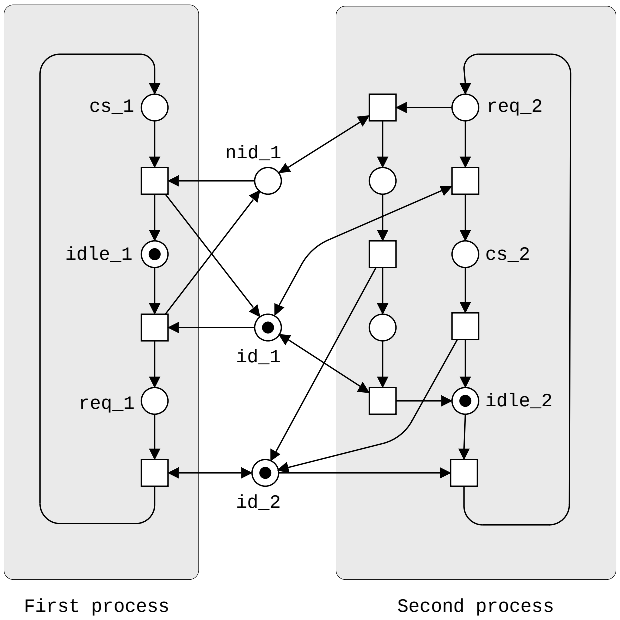


**Fig. 1.** A Petri net model of Lamport's 1-bit mutex algorithm

nondeterministic automaton, where $\Sigma$ is a finite alphabet, $Q$ is a finite set of states, $q_0$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F$ is a set of finite states. The language of $A$, denoted by $L(A)$, is defined as the set of finite words accepted by $A$. We define now the language of *infinite words* accepted by $A$, which we denote by $L_\omega(A)$. A word $w = a_0 a_1 a_2 \dots$ belongs to $L_\omega(A)$ if there is an infinite sequence of states $q_0 q_1 q_2 \dots$ such that $(q_i a_i q_{i+1}) \in \delta$ for every $i \geq 0$.

When we are interested in the language of infinite words of an automaton, then we call it *Büchi automaton*.

We have the following important result:

Given an LTL formula $\phi$, one can build a finite automaton $A_\phi$ and a Büchi automaton $B_\phi$ such that $L(A_\phi) \cup L_\omega(B_\phi)$ is exactly the set of computations satisfying the formula $\phi$.

Since computations are sequences of sets of atomic propositions, the alphabet of the automata $A_\phi$ and $B_\phi$ is the set $2^{Prop}$. In our case *Prop* is the set of places of the net, and so the alphabet of the automata is the set of all markings.

The construction of $A_\phi$ and $B_\phi$ exceeds the scope of this paper (see for instance [37]). For our purposes, it suffices to know the following facts:

- The states of $A_\phi$ are sets of subformulas of $\phi$; the states of $B_\phi$ are pairs of sets of subformulas of $\phi$. Since there are exponentially many sets of subformulas, $A_\phi$ and $B_\phi$ may have exponentially many states in $|\phi|$.
- Given two states $q_1, q_2$ of $A_\phi$ or $B_\phi$ and a marking $M$, there is an algorithm which decides using polynomial space whether $(q_1, M, q_2) \in \delta_\phi$.

We also need two automata $A_N = (2^S, Q_N, q_{0N}, \delta_N, F_N^A)$ and $B_N = (2^S, Q_N, q_{0N}, \delta_N, F_N^B)$ obtained from the Petri net $N$, as follows:

- $Q_N$ is the set of reachable markings of $N$;
- $q_{0N}$ is the initial marking $M_0$;
- $\delta_N$ contains the triples of markings $(M_1, M_1, M_2)$ such that $M_1 \xrightarrow{t} M_2$ for some transition $t$;
- $F_N^A$ is the set of deadlocked reachable markings of $N$;
- $F_N^B = Q$, i.e., $F_N^B$ is the set of reachable markings of $N$.

Loosely speaking, both automata correspond to the reachability graph of $N$, with the peculiarity that edges are labelled with the marking they come from. $A_N$ and $B_N$ differ only in their final states. Clearly, $L(A_N)$ is the set of all finite runs of $N$, and $L_\omega(B_N)$ the set of all infinite runs.

In order to solve the model-checking problem for input $N$, $\phi$, let $A$ be the product of the automata $A_{\neg\phi}$ and $A_N$, and let $B$ be the product of the automata $B_{\neg\phi}$ and $B_N$, where the product $(\Sigma, Q, q_0, \delta, F)$ of two automata $(\Sigma, Q_1, q_{01}, \delta_1, F_1)$ and $(\Sigma, Q_2, q_{02}, \delta_2, F_2)$ is defined in the usual way:

$$\begin{aligned}
Q &= Q_1 \times Q_2 \\
q_0 &= (q_{01}, q_{02}) \\
\delta &= \{((q_1, q_2), a, (q_1', q_2')) \mid (q_1, a, q_1') \in \delta_1 \text{ and } (q_2, a, q_2') \in \delta_2\} \\
F &= F_1 \times F_2
\end{aligned}$$

Clearly, we have $L(A) = L(A_{\neg\phi}) \cap L(A_N)$ and $L_\omega(B) = L_\omega(B_{\neg\phi}) \cap L_\omega(B_N)$.[4] So the union of $L(A)$ and $L_\omega(B)$ is the set of runs of $N$ that do *not* satisfy $\phi$; in other words, $N$ satisfies $\phi$ if and only if $L(A) = \emptyset$ and $L_\omega(B) = \emptyset$.

We have reduced the model checking problem to the following one: Given $N$ and $\phi$, decide if $L(A)$ and $L_\omega(B)$ are empty. We have to solve this problem using only polynomial storage space in the size of $N$ and $\phi$. The first natural idea is to construct $A$ and $B$, and then use the standard algorithms for emptiness of automata for finite and infinite words. Unfortunately, both $A$ and $B$ may have exponentially many states in $|N|$ and $|\phi|$.

At this point, complexity theory helps us by means of Savitch's construction. Recall that a nondeterministic decision procedure for a problem is an algorithm which can return "yes" or fail, and satisfies the following property: the answer to the problem is "yes" if and only if *some* (not necessarily all) execution of the algorithm returns "yes". A deterministic decision procedure always answers "yes" or "no".

[4] The product of two Büchi automata doesn't always accept the intersection of the languages, but this is so in our case.

| |
|---|
| Savitch's construction:<br>Given a nondeterministic decision procedure for a given problem using $f(n)$ space, Savitch's construction yields a deterministic procedure for the same problem using $f^2(n)$ space. |

This construction makes our life easier: it suffices to give a *nondeterministic* algorithm for the emptiness problem of $A$ and $B$ running in polynomial space. Actually, it also suffices to give a nondeterministic algorithm for the *nonemptiness* problem: by Savitch's construction there exists a deterministic algorithm for the nonemptiness problem, and by reversing the answer of this algorithm we obtain another one for the emptiness problem.

The nondeterministic algorithm for the nonemptiness problem constructs $A$ and $B$ "on the fly". The algorithm keeps track of a current state of $A$ or $B$, which is initially set to the initial state. The algorithm repeatedly guesses a next state, checks that there is a transition leading from the current state to the next state, and updates the current state. In the case of $A$, the algorithm returns "true" when (and if) it reaches a final state:

Algorithm Nonempty_A($N$, $\phi$)

**variables:** $q$ of type state of $A_{\neg\phi}$;
$M$ of type state of $A_N$ (i.e., of type marking);

**begin**
$(q, M) := (q_{0\neg\phi}, M_0)$;
**while** $(q, M)$ is not a final state of $A$ **do**
choose a state $q'$ of $A_{\neg\phi}$ such that $(q, M, q') \in \delta_{\neg\phi}$
and a marking $M'$ such that $M \xrightarrow{t} M'$ for some transition $t$;
$(q, M) := (q', M')$;
**od**;
**return true**
**end**

In order to estimate the space used by Nonempty_A, observe that all the operations and tests can be performed in polynomial space. For that, recall that given two states $q_1, q_2 \in Q_{\neg\phi}$ and $M \in 2^S$, there is an algorithm which decides using polynomial space whether $(q_1, M, q_2) \in \delta_{\neg\phi}$. The algorithm needs to store one state $q$ of $A_{\neg\phi}$ and a marking $M$ of $N$. Since the states of $A_{\neg\phi}$ are sets of subformulas of $\phi$, $q$ has quadratic size in $|\phi|$. Since $M$ has linear size in $|N|$, polynomial space suffices.

The case of $B$ is a bit more complicated. Since $B$ has finitely many states, $L_\omega(B)$ is nonempty if and only if there exists a reachable final state $q$ such that there is a loop from $q$ to itself. So the algorithm proceeds as in the case of $A$, but, at some point, it guesses that the current final state will be revisited; it then stores the current state to be able to check if the guess is true. The rest of the algorithm checks the guess nondeterministically.

Algorithm Nonempty_B($N$, $\phi$):

**variables:** $M, M_r$ of type state of $B_N$ (i.e., of type marking);
$q, q_r$ of type state of $B_{\neg\phi}$;
*flag* of type boolean;

```
begin
  (q, M) := (q0¬φ, M0); flag := false;
  while flag = false do
    choose a state q' of A¬φ such that (q, M, q') ∈ δ¬φ
    and a marking M' such that M --t--> M' for some t;
    (q, M) := (q', M');
    if (q, M) is a final state then
      choose between flag := false and flag := true
    fi
  od;
  (qr, Mr) := (q, M);
  repeat
    choose a state q' of A¬φ such that (q, M, q') ∈ δ¬φ
    and a marking M' such that M --t--> M' for some t;
    (q, M) := (q', M')
  until (q, M) = (qr, Mr);
  return true
end
```

Again, Nonempty_B($N$, $\phi$) uses only polynomial space. Since the deterministic algorithm obtained after the application of Savitch's construction to Nonempty_A and Nonempty_B also needs polynomial space, the model-checking problem for LTL belongs to PSPACE.

Observe that the only properties of 1-safe nets we have used in order to obtain this result are:

- a state has polynomial size (actually, even linear) in $|N|$, and
- given two markings $M, M'$, it can be decided in polynomial space if $M \xrightarrow{t} M'$ for some transition $t$.

These conditions are very weak, and so the PSPACE result can be extended to a number of other models. As observed in [35], conditions (1) and (2) hold for other Petri net classes, like *condition/event systems*, *elementary net systems*, but also for process algebras with certain limitations to recursion, and for several other models based on a finite number of state machines communicating by finite means. The conditions also hold for bounded Petri nets, assuming that the bound is also given to Nonempty_A and Nonempty_B as part of the input. This assumption is necessary, because the bound of a bounded Petri net (the maximal number of tokens a place can contain under a reachable marking) can be much bigger than the size of the net, and so we may need more than polynomial space in order to just write down a reachable marking.

The PSPACE result can also be extended to more general logics, like the linear-time mu-calculus, for which the translation into automata still works (see for instance [4]).

### 4.2 Computation Tree Logic

Some interesting properties of Petri nets cannot be expressed in LTL. An example is liveness (in the Petri net sense). Recall that a transition is live if it can always occur again. One possibility to express this to allow existential or universal quantification on the set of computations starting at a marking. CTL introduces this quantification on top of LTL's syntax The syntax of CTL is

$$
\begin{array}{lll}
\phi ::= & p \in Prop & \\
 & \neg\phi & \\
 & \phi_1 \wedge \phi_2 & \\
 & EX\phi & \text{existential next operator} \\
 & AX\phi & \text{universal next operator} \\
 & E[\phi_1 U \phi_2] & \text{existential until operator} \\
 & A[\phi_1 U \phi_2] & \text{universal until operator}
\end{array}
$$

Disjunction and implication are defined as usual. Other abbreviations are $true = p \vee \neg p$, $EF\phi = E[trueU\phi]$ (possibly $\phi$), $AG\phi = \neg EF\neg\phi$ (always $\phi$), $AF\phi = A[trueU\phi]$ (eventually $\phi$) and $EG\phi = \neg AF\neg\phi$ ($\phi$ holds at every state of some computation).

CTL formulas are interpreted on *computation trees*, which are possibly infinite trees where each node $n$ is labelled with a set of atomic propositions $P(n)$. A path of a computation tree that cannot be extended to a larger path is called a computation; notice that it is a computation in the LTL sense. The intuition is that the nodes of the tree correspond to the states of a system; a state may have an arbitrary number of successors, corresponding to different computations. $P(n)$ is the set of atomic propositions that hold at node (state) $n$. For a tree $\tau$ and a node $n$ we have that:

$$
\begin{array}{lll}
\tau, n \models p & \text{iff} & p \in P(n) \\
\tau, n \models \neg\phi & \text{iff} & \text{not}(\tau, n \models \phi) \\
\tau, n \models \phi_1 \wedge \phi_2 & \text{iff} & \tau, n \models \phi_1 \text{ and } \tau, n \models \phi_2 \\
\tau, n \models AX\phi & \text{iff} & \text{for every child } n' \text{ of } n,\ \tau, n' \models \phi \\
\tau, n \models EX\phi & \text{iff} & \text{for some child } n' \text{ of } n,\ \tau, n' \models \phi \\
 & & (n \text{ must have at least one child}) \\
\tau, n \models A[\phi_1 U \phi_2] & \text{iff} & \text{for all computations } n = n_0 n_1 n_2 \ldots \\
 & & \text{there exists } i \geq 0 \text{ such that } n_i \models \phi_2 \\
 & & \text{and for every } j,\ 0 \leq j < i,\ n_j \models \phi_1 \\
\tau, n \models E[\phi_1 U \phi_2] & \text{iff} & \text{for some computation } n = n_0 n_1 n_2 \ldots \\
 & & \text{there exists } i \geq 0 \text{ such that } n_i \models \phi_2 \\
 & & \text{and for every } j,\ 0 \leq j < i,\ n_j \models \phi_1
\end{array}
$$

If the tree $\tau$ is clear from the context we shorten $\tau, n \models \phi$ to $n \models \phi$. We say that a tree $\tau$ satisfies a formula $\phi$ if $root(\tau) \models \phi$.

Observe that $AX\phi$ is equivalent to $\neg EX\neg\phi$, i.e., $EX$ and $AX$ are dual operators. So actually we could remove $AX$ from the syntax without losing expressive power. It might seem that the existential and universal until operators are also dual of each other, but this is not true. The dual operator of the universal until is the existential *weak* until, with syntax $E[\phi_1\, WU \phi_2]$, and the following semantics:

$$\tau, n \models E[\phi_1\, WU \phi_2] \ \text{ iff } \ \tau, n \models E[\phi_1 U \phi_2] \vee EG(\phi_1)$$

It holds that

$$A[\phi_1 U \phi_2] = \neg E[\neg\phi_2\, WU \neg\phi_1]$$

In order to use CTL to specify properties of a 1-safe Petri net $N$, we choose again the places of $N$ as atomic propositions. With this choice a computation tree is a tree of sets of places, and so a set of markings. We can associate to $N$ a computation tree $\tau_N$ as follows: the root is labelled with the initial marking $M_0$; the children of a node labelled by $M$ are labelled with the markings $M'$ such that $M \xrightarrow{t} M'$ for some transition $t$. We say that $N$ satisfies $\phi$ if the tree $\tau_N$ satisfies $\phi$.

The computation tree corresponding to the the net of Figure 1 is shown in Figure 2. Essentially, the tree is just the unfolding into a tree of the reachability graph of the net. Different nodes in the tree can be labelled with the same marking, but all subtrees

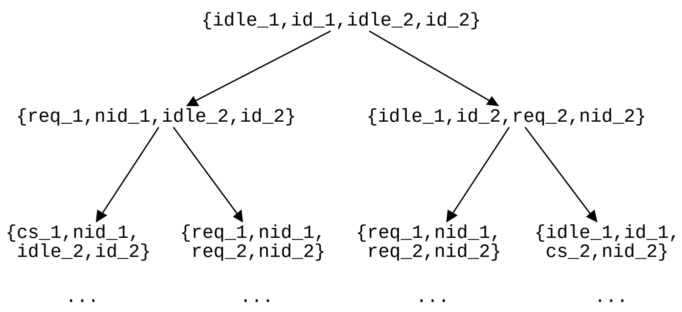


**Fig. 2.** Computation tree of the Petri net of Figure 1

whose roots are labelled with the same marking are isomorphic. Given a formula $\phi$ and a marking $M$, either all or none of the nodes labelled by $M$ satisfy $\phi$. So it makes sense to say that $M$ satisfies $\phi$, meaning that all nodes labelled by $M$ satisfy $\phi$.

Here are some CTL queries on the Petri net of Figure 1:

– No reachable marking puts tokens in $cs_1$ and $cs_2$ (true): $AG(\neg cs_1 \vee \neg cs_2)$.
– The output transition of the place $req_1$ is live (true): $AGEF(req_1 \wedge id_2)$.
– The initial marking is reachable from every reachable marking (true): $AGEF(idle_1 \wedge id_1 \wedge id_2 \wedge idle_2))$
– Eventually place $cs_1$ becomes marked (false): $AFcs_1$
– There is a run that never marks $cs_2$ (true): $EG\neg cs_2$
– If $req_2$ becomes marked, then eventually $cs_2$ becomes marked (false): $AG(req_2 \Rightarrow AFcs_2)$

We show that the model checking problem for CTL is in PSPACE. It follows from the discussion above that it suffices to give a polynomial space algorithm for the syntax

$$\phi ::= s \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid EX\phi \mid E[\phi_1 U \phi_2] \mid E[\phi_1 \, WU \, \phi_2]$$

We give a (deterministic) algorithm Check($M$, $\phi$) with a marking $M$ and a formula $\phi$ as parameters which answers "true" if $M$ satisfies $\phi$, and "false" otherwise. The model-checking problem is then solved by Check($M_0$, $\phi$).

Check($M, \phi$) is a recursive procedure on the structure of $\phi$, i.e., Check($M, Op(\phi_1, \ldots, \phi_n)$), where $Op$ is some operator of the logic, calls Check($M$, $\phi_1$), ... , Check($M$, $\phi_n$).

Algorithm Check($M$, $\phi$):

**begin**
  **if** $\phi = s$ **then**
    **if** $M(s) = 1$ **then return true else return false fi**
  **elseif** $\phi = \neg\phi_1$ **then return not** Check($M$, $\phi_1$)
  **elseif** $\phi = \phi_1 \wedge \phi_2$ **then return** Check($M$, $\phi_1$) **and** Check($M$, $\phi_2$))
  **elseif** $\phi = EX\phi_1$ **then**
    **for** every $M'$ such that $M \xrightarrow{t} M'$ for some transition $t$ **do**
      **if** Check($M'$, $\phi_1$) **then return true fi**
    **od**
  **elseif** $\phi = E[\phi_1 U \phi_2]$ **then return** EU($M$, $\phi_1$, $\phi_2$)
  **elseif** $\phi = E[\phi_1 \, WU \phi_2]$ **then return** EWU($M$, $\phi_1$, $\phi_2$)
  **fi**
**end**

It remains to define the procedures EU($M$, $\phi_1$, $\phi_2$) and EWU($M$, $\phi_1$, $\phi_2$). We start with EU($M$, $\phi_1$, $\phi_2$).

It is not possible to deterministically explore the infinitely many computations starting at $M$, and check directly if one of them satisfies $\phi_1 U \phi_2$. The reader might feel tempted to give a nondeterministic algorithm which explores one of the computations, and then apply Savitch's technique. This seems to be a good idea, but in fact doesn't work! There is a rather subtle problem. Consider the formulas

$$\phi_n = E[E \ldots E[s_0 U s_1] \ldots ]U s_{n-1}]U s_n]$$

where $s_1, \ldots, s_n$ are places. We obtain a checking algorithm $\phi_n$ through $n$ applications of Savitch's technique. It is easy to give a $\Omega(|N|)$-space nondeterministic algorithm for $E[s_0 U s_1]$. Unfortunately, the deterministic algorithm obtained by Savitch's technique requires $\Omega(|N|^2)$ space, the algorithm for $E[E[s_0 U s_1]U s_2]$ $\Omega(|N|^4)$ space, and the algorithm for $\phi_n$ no less than $\Omega(|N|^{2n})$ space. So the degree of the polynomial in $|N|$ depends on the formula we are considering.

We proceed in a different way. In a fist step we reduce the problem to the exploration of a finite number of finite paths. We extend the syntax of CTL with new operators $E[\phi_1 U_b \phi_2]$, one for each natural number $b$. Loosely speaking, a node satisfies $E[\phi_1 U_b \phi_2]$ if in at least one of the computations starting at it we find a node satisfying $\phi_2$ *after at most $b$ steps*, and all nodes before it satisfy $\phi_1$. Formally:

$\tau, n \models E[\phi_1 U_b \phi_2]$ iff for some computation $n = n_0 n_1 n_2 \ldots$
there exists $i$, $0 \leq i \leq b-1$ such that
$n_i \models \phi_2$ and $n_j \models \phi_1$ for every $j$, $0 \leq j < i$

It follows immediately from this definition that if $\tau, n$ satisfies $E[\phi_1 U_b \phi_2]$ for some number $b$ then it also satisfies $E[\phi_1 U \phi_2]$.

Now, let $n$ be an arbitrary node of $\tau_N$, and let $k$ be the number of places of $N$. We prove

$$n \models E[\phi_1 U \phi_2] \iff E[\phi_1 U_{2^k} \phi_2]$$

It suffices to prove that $n \models E[\phi_1 U \phi_2]$ implies $n \models E[\phi_1 U_{2^k} \phi_2]$. Assume that $n$ satisfies $E[\phi_1 U \phi_2]$. Then, $\tau_N$ contains a computation $n = n_0 n_1 n_2 \ldots$ satisfying $\phi_1 U \phi_2$: $n_i \models \phi_1$ for some $i \geq 0$ and $n_j \models \phi_1$ for every $j$, $0 \leq j < i$. If $i \leq 2^k - 1$, then this computation satisfies $\phi_1 U_{2^k} \phi_2$, and so $n \models \phi_1 U_{2^k} \phi_2$. Let us now consider the case $i \geq 2^k$. Let $M_0 M_1 M_2 \ldots$ be the sequence of markings corresponding to $n_0 n_1 n_2 \ldots$. Since $N$ is 1-safe and has $k$ places, it has at most $2^k$ reachable markings. So there are indices $j_1$ and $j_2$, $0 \leq j_1 < j_2 \leq i$, such that $M_{j_1} = M_{j_2}$. Since the markings labelling the successors of a node are completely determined but the marking labelling the node itself, $\tau_N$ contains another computation starting at $n_0$ and labelled by

$$M_0 \ldots M_{j_1} M_{j_2+1} M_{j_2+2} \ldots$$

Loosely speaking, the sequence of markings of the new computation is obtained from the old sequence by "cutting out" the piece $M_{j_1+1} \ldots M_{j_2}$ and "glueing" the two ends $M_{j_1}$ and $M_{j_2+1}$. In this new sequence the marking $M_i$ appears at the position $i - (j_2 - j_1)$, and so closer to $M_0$ than in the original computation. We now iterate the "cutting and glueing" procedure until $M_i$ appears before the $2^k$ -th position. The computation so obtained satisfies $\phi_1 U_{2^k} \phi_2$, and so $n \models \phi_1 U_{2^k} \phi_2$.

So we have solved our first problem: instead of a potentially infinite number of computations, it suffices to explore finitely many paths containing at most $2^k$ nodes, and check that at least one of them satisfies $\phi_1 U_{2^k} \phi_2$ (more precisely, that at least one of them can be extended to a computation satisfying $\phi_1 U_{2^k} \phi_2$).

We construct EU($M$, $\phi_1$, $\phi_2$) with the help of another algorithm Path($M$, $M'$, $\phi$, $\psi$, $l$), still to be designed, with the following specification:

> Path($M$, $M'$, $\phi$, $\psi$, $l$) returns "true" if and only if $\tau_N$ has a path $n_0 \ldots n_l$ such that
> - $n_0$ is labelled by $M$ and $n_l$ is labelled by $M'$,
> - $n_i \models \phi$ for every $i$, $0 \leq i < l$, and
> - $n_l \models \psi$.

We can take:

Algorithm EU($M$, $\phi_1$, $\phi_2$)

**constant:** $k =$ number of places of $N$;

**begin**
    **for** every marking $M'$ of $N$ and every $0 \leq l < 2^k$ **do**
        **if** Path($M$, $M'$, $\phi_1$, $\phi_2$, $l$) **then return true**
    **od**;
    **return false**
**end**

Since each iteration of the **for** loop can reuse the same space, the space used by EU($M$, $\phi_1$, $\phi_2$) is the space used by Path($M$, $M'$, $\phi_1$, $l$) plus the space needed to store $M'$ and $l$. So Path($M$, $M'$, $\phi_1$, $l$) should use at most polynomial space for every $l < 2^k$. A backtracking algorithm, which would be the obvious choice, does not meet this requirement, because it stores all the nodes of the computation being currently explored having still unexplored branches, and there can be exponentially many of those.

A trick frequently applied in complexity theory[5] helps us out of the problem. Loosely speaking, for each reachable marking $M''$, we explore all paths leading from $M$ to $M''$ and containing $\lceil \frac{l}{2} \rceil + 1$ nodes, and then, *reusing the same space*, all paths leading from $M''$ to $M'$ and containing $\lfloor \frac{l}{2} \rfloor + 1$ nodes. This trick of splitting the paths into two parts is applied recursively until paths having at most 2 nodes are reached.

Algorithm Path($M$, $M'$, $\phi$, $\psi$, $l$)

**constant:** $k =$ number of places of $N$;

**begin**
    **if** $l = 0$ **then**
        **if** $M = M'$ **and** Check($M, \psi$)
        **then return true fi**
    **fi**;
    **if** $l = 1$ **then**
        **if** $M \xrightarrow{t} M'$ for some transition $t$
            **and** Check($M, \phi$) **and** Check($M', \psi$)
        **then return true fi**
    **fi**;
    **for** every marking $M''$ of $N$ **do**
        **if** Path($M$, $M''$, $\phi$, *true*, $\lceil \frac{l}{2} \rceil$) **and** Path($M''$, $M$, $\phi$, $\psi$, $\lfloor \frac{l}{2} \rfloor$)
        **then return true fi**
    **od**;
    **return false**
**end**

In order to estimate the space complexity of Path($M$, $M'$, $\phi$, $l$), let $c(\phi)$ be the maximum over all markings $M$ of the space needed by Check($M$, $\phi$), and let $p(\phi, \psi, l)$ be the maximum over all pairs of markings $M$, $M'$ of the space needed by Path($M$, $M'$, $\phi$, $\psi$, $l$). Then we have

$$\begin{aligned} p(\phi, \psi, 0) &= O(c(\psi)) \\ p(\phi, \psi, 1) &= O(\max\{c(\phi), c(\psi)\}|N|) \\ p(\phi, \psi, l) &= O(\max\{p(\phi, \psi, \lceil \tfrac{l}{2} \rceil), p(\phi, \psi, \lfloor \tfrac{l}{2} \rfloor)\}|N|) \end{aligned}$$

and so, in particular

$$p(\phi, \psi, 2^k) = O(\max\{c(\phi), c(\psi)\} + k \cdot |N|) = O(\max\{c(\phi), c(\psi)\} + |N|^2)$$

It remains to construct EWU($M, \phi_1, \phi_2$). The interested reader can easily prove that for every node $n$ of $\tau_N$

$$n \models E[\phi_1\, WU\, \phi_2] \iff E[\phi_1\, WU_{2^k}\, \phi_2]$$

[5] In fact, this trick lies at the heart of Savitch's technique.

where the semantics of $E[\phi_1 \, WU_b \phi_2]$ is given by

$$\tau, n \models E[\phi_1 \, WU_b \phi_2] \text{ iff } \begin{array}{l} n \models E[\phi_1 U_b \phi_2] \text{ or} \\ \text{there exists a path } n = n_0 n_1 n_2 \ldots n_b \\ \text{such that } n_i \models \phi_1 \text{ for every } 0 \leq i \leq b \end{array}$$

So we can take

Algorithm EWU($M$, $\phi_1$, $\phi_2$)

**constant:** $k$ = number of places of $N$;

**begin**
  **if** EU($M$, $\phi_1$, $\phi_2$) **then return true**
  **else**
    **for** every marking $M'$ of $N$ **do**
      **if** Path($M$, $M'$, $\phi_1$, $true$, $2^k$) **then return true**
    **od**;
  **return false**
**end**

This completes the definition of Check($M, \phi$). It is easy to see that it runs in polynomial space in $|N|$ and $|\phi|$, but let us determine the space complexity a bit more precisely. We have:

$$\begin{aligned} c(s) &= O(|N|) \\ c(\phi_1 \wedge \phi_2) &= O(\max\{c(\phi_1), c(\phi_2)\} + |N|) \\ c(\neg\phi) &= O(c(\phi)) \\ c(E[\phi_1 U \phi_2]) &= O(p(\phi_1, \phi_2, 2^k) + |N|) \\ &= O(\max\{c(\phi_1), c(\phi_2)\} + |N|^2) \\ c(E[\phi_1 U_w \phi_2]) &= O(\max\{c(E[\phi_1 U \phi_2]), p(\phi_1, true, 2^k)\} + |N|) \\ &= O(\max\{c(\phi_1), c(\phi_2)\} + |N|^2) \end{aligned}$$

and so we finally get $c(\phi) = O(|\phi| \cdot |N|^2)$.

### 4.3 An exception

The most interesting exception to Rule of Thumb 2 is the *controllability* property. Let $T_0$ be a subset of transitions of a 1-safe Petri net $N = (S, T, F, M_0)$, and let $t \in T \setminus T_0$. We say that $T_0$ *controls* $t$ *by a sequence* $\sigma \in T_0^*$ if for every occurrence sequence $M_0 \xrightarrow{\tau} M$ such that the projection of $\tau$ onto $T_0$ is $\sigma$, the transition $t$ cannot occur at $M$. The intuition is that $T_0$ can control $t$ in the sense that once the sequence $\sigma$ has occurred, possibly interleaved with transitions of $T \setminus T_0$, $t$ cannot occur until transitions of $T_0$ occur again. We say that $T_0$ *can control* $t$ if $T_0$ can control $t$ by at least one sequence $\sigma$.

The controllability problem is defined as follows:

Given: a 1-safe Petri net with a set $T$ of transitions, $T_0 \subseteq T$, $t \in T \setminus T_0$
To decide: if $T_0$ can control $t$.

Jones, Landweber and Lien show in [24] that controllability is EXPSPACE-complete.

### 4.4 A remark on action-based temporal logics

We have defined LTL and CTL as *state-based* logics, because in order to know if a run satisfies a property one only needs information about the states – the markings – visited during its execution, and not about which transitions lead from a marking to the next. It is possible to define *action-based* versions of these logics, in which the identities of the markings visited during the execution of a run is irrelevant, while the information is carried by the sequence of transitions that occur. These action-based versions are particularly useful for labelled Petri nets.

The action-based version of LTL – tailored for labelled Petri nets – looks as follows: the set of basic propositions contains only one element, namely the proposition *true*. The operators $X$ and $U$ are replaced by a set of *relativised* operators $X_K$, $U_K$, where $k$ is a subset of a certain finite set of actions *Act*. A *computation* is now a finite or infinite sequence $\pi = a_0 a_1 a_2 \ldots$ of actions. Let $\pi^{(i)} = a_i a_{i+1} \ldots$. We have:

| | | |
|---|---|---|
| $\pi \models true$ | | always |
| $\pi \models X_K \phi$ | iff | $\pi \neq \epsilon$, $a_0 \in k$, and $\pi^{(1)} \models \phi$ |
| $\pi \models \phi_1 U_K \phi_2$ | iff | for some $j \geq 0$ we have $\pi^{(j)} \models \phi_2$ and |
| | | for all $k$, $0 \leq k \leq j$, we have $a_i \in k$ and $\pi^{(k)} \models \phi_1$ |

In order to interpret the logic on a 1-safe labelled Petri net $N$, we choose *Act* as the set of labels carried by the transitions of $N$. We say that $N$ satisfies a formula $\phi$ if *all* the sequences of transition labels obtained from the runs of $N$ by removing the markings satisfy $\phi$.

Similarly, in the action-based version of CTL the operators of the logic $EX$, $AX$, $E[\ldots U \ldots]$, and $A[\ldots U \ldots]$ are replaced by sets of relativised operators $EX_K$, $AX_K$, $E[\ldots U_K \ldots]$, and $A[\ldots U_K \ldots]$. Computation trees are now trees whose edges are labelled with actions. The semantics is exactly what one expects.

It is easy to prove that the model-checking problem for these two new logics can be reduced to the model-checking problem for their state-based versions. More precisely: given a labelled 1-safe Petri net $N$ and a formula $\phi$ of action-based LTL (CTL), one can construct in polynomial time an unlabelled 1-safe Petri net $N'$ and a formula $\phi'$ of state-based LTL (CTL) such that $N$ satisfies $\phi$ if and only if $N'$ satisfies $\phi'$. It follows that the model-checking problem for the action-based LTL and CTL is also in PSPACE.

In Section 8 we study the model checking problems for temporal logics and arbitrary Petri nets. There, the distinction between state-based and action-based logics plays a much more important rôle.

## 5 Deciding equivalences

In this section we investigate the complexity of deciding if two labelled 1-safe Petri nets are equivalent with respect to a given equivalence notion.

Since the early eighties many different equivalence notions have been presented in the literature. Van Glabbeek has classified them in several papers, e.g. [36]. Most of these equivalences fit between the so-called *trace* equivalence, which is a process theory counterpart of the classical language equivalence used in formal language theory, and *bisimulation* equivalence. An equivalence notion X fits between trace and bisimulation equivalence if bisimilar systems are X-equivalent, and X-equivalent systems are trace equivalent.

Trace and bisimulation equivalences are defined as follows. Let $N$ be a labelled Petri net, where transitions are labelled with the elements of a set of actions *Act*. The set of *traces* of $N$, denoted by $\mathcal{T}(N)$ is the set of words $a_1 \ldots a_n \in Act^*$ such that there exist markings $M_1, \ldots M_n$ satisfying $M_0 \xrightarrow{a_1} M_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} M_n$[6]. Two Petri nets $N_1$ and $N_2$ are *trace equivalent* if $\mathcal{T}(N_1) = \mathcal{T}(N_2)$.

A relation $\mathcal{R}$ between the sets of markings of two nets is a *(strong) bisimulation* if for every pair $(M_1, M_2) \in \mathcal{R}$ and for every action $a \in Act$,

– if $M_1 \xrightarrow{a} M_1'$, then $M_2 \xrightarrow{a} M_2'$ for some marking $M_2'$ such that $(M_1', M_2') \in \mathcal{R}$, and
– if $M_2 \xrightarrow{a} M_2'$, then $M_1 \xrightarrow{a} M_1'$ for some marking $M_1'$ such that $(M_1', M_2') \in \mathcal{R}$.

Two Petri nets $N_1$ and $N_2$ are *(strongly) bisimilar* if there exists a (strong) bisimulation $\mathcal{R}$ containing the pair $(M_{01}, M_{02})$ of initial markings of $N_1$ and $N_2$.

We have the following

| |
|---|
| Rule of thumb 3:<br>Equivalence problems for 1-safe Petri nets are harder to solve than model-checking problems, but they need at most exponential space. |

We provide a first piece of evidence for this rule of thumb by showing that the equivalence problem for 1-safe Petri nets and any equivalence notion fitting between trace and bisimulation equivalence is PSPACE-hard. It turns out that all the concrete equivalences mentioned in the literature have at least DEXPTIME-hard equivalence problems, and so this general PSPACE-hardness lower bound can possibly be improved.

We proceed by reduction from the following PSPACE-hard problem

Given: a 1-safe Petri net $N$, a place $s$ of $N$
To decide: if some reachable marking of $N$ puts a token on $s$.

We start by labelling each transition of $N$ with the same label, say $a$. $N$ is now a labelled net. We put $N$ side by side with the labelled net $N'$ consisting of a loop containing one single place marked with one token and one single transition labelled by $a$. We denote the resulting Petri net by $N \parallel N'$.

Now, we consider two labelled nets. The first one is $N \parallel N'$; the second is a small modification of it obtained by adding a new output transition to the place $s$ of $N$. The new transition has $s$ as unique input place, no output places, and carries a label different from $a$, say $b$.

The following holds:

– If some reachable marking puts a token on $s$, then the two nets are not trace equivalent: the second one can do a $b$, while the first one can't.

[6] Recall: $M \xrightarrow{a} M'$ denotes that there is a transition $t$ labelled by $a$ such that $M \xrightarrow{t} M'$.

– If no reachable marking puts a token on $s$, then the two nets are bisimilar: the relation containing all pairs $(M_1, M_2)$, where $M_1$ is a reachable marking of the first net and $M_2$ a reachable marking of the second net, is clearly a bisimulation.

Therefore, given any equivalence notion X fitting between trace and bisimulation equivalence, we can solve the PSPACE-hard problem above by constructing the two nets and deciding if they are X-equivalent. So the equivalence problem for any such notion is PSPACE-hard.

Apart from this little result, the real evidence supporting the rule of thumb above is the work of Rabinovich [31] and Jategaonkar and Meyer [23]. This last paper contains a table with the complexity of 18 equivalence notions. Bisimilarity and many variants of it are DEXPTIME-complete, while trace equivalence, failures equivalence, and several variants of them are EXPSPACE-complete. They also consider so-called *partial order equivalences*, for which the concurrent execution of two actions is not equivalent to their interleaved execution (i.e., a system that executes $a$ and $b$ in parallel is not considered to be equivalent to a system which chooses between executing $a$ and then $b$, or $b$ and then $a$). The complexity results (up to some open problems) are similar.

## 6 Can anything be done in polynomial time?

We have seen that all interesting problems for arbitrary 1-safe Petri nets are at least PSPACE-hard, and so that there is very little hope of finding polynomial algorithms for them. The natural question to ask is if there are important subclasses of 1-safe Petri nets for which one could solve at least some problems in polynomial time. In this section we get some general answers in the form of rules of thumb.

A first rule, which tends to be surprising for many people is

| Rule of thumb 4:<br>Most interesting questions about the behaviour of *acyclic* 1-safe Petri nets are NP-hard. |
|---|

Here, as in Section 3, a word of warning is required about the meaning of "interesting". Liveness is certainly an interesting question for arbitrary 1-safe nets, but not for the acyclic ones: 1-safe acyclic Petri nets are always non-live, because no transition can fire more than once. Interesting questions for 1-safe acyclic Petri nets, all of them NP-hard, are

– Is a given marking reachable from the initial marking?
– Is there a reachable marking which marks a given place?
– Is there a reachable marking which does not mark a given place?
– Is there a reachable marking which enables a given transition?
– Is the initial marking reachable from every reachable marking?
– Is there a run containing a given transition?
– Is there a run that does not contain a given transition?

Let us prove NP-hardness of the second problem: Is there a reachable marking which marks a given place? We present a polynomial time construction which associates to a boolean formula in conjunctive normal form an acyclic 1-safe Petri net. The net nondeterministically selects a truth assignment for the variables of the formula, and then checks if the formula is true under the assignment. The construction is illustrated in Figure 3 by means of an example.

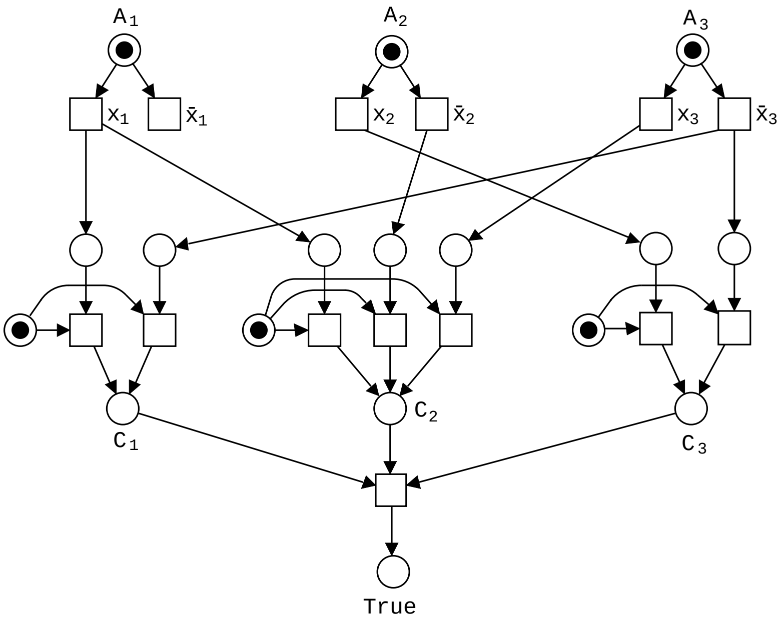


**Fig. 3.** Acyclic net corresponding to the formula $(x_1 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3)$

It seems[7] that in order to obtain classes with polynomial decision algorithms one has to impose *local* constraints on the net's structure. Here "local constraint" means a constraint which can be shown not to hold by looking at only a small part of the net. For instance, "every transition has exactly one input place" is a local constraint; if the constraint does not hold, then one can always point at a particular transition in the net, together with its input places, and show that the constraint is not satisfied because of this transition. A constraint like "the net is acyclic" is not local, because the smallest circuit of the net may be the net itself.

The two following local constraints have been very intensely studied in the literature:

– the *conflict-freeness* constraint: $s^\bullet \subseteq {}^\bullet s$ for every place $s$ with more than one output transition; in the case of 1-safe Petri nets this constraint is equivalent to "every place has at most one output transition" for nearly all purposes;

[7] Although I don't know of any formal proof.

– the *free-choice* constraint: if $(s,t)$ is an arc from a place to a transition, then so is $(s',t')$ for every place $s' \in {}^\bullet t$ and for every transition $t' \in s^\bullet$.

Unfortunately, it is not possible to summarise the results of the research on conflict-free and free-choice Petri nets in a concise and general rule of thumb. But we can still say:

| |
|---|
| Rule of thumb 5:<br>Many interesting questions about 1-safe conflict-free Petri nets are solvable in polynomial time.<br>Some interesting questions about *live* 1-safe free-choice Petri nets are solvable in polynomial time (and liveness of 1-safe free-choice Petri nets is decidable in polynomial time too).<br>Almost no interesting questions for 1-safe net classes substantially larger than free-choice Petri nets are solvable in polynomial time. |

Among the "many" interesting polynomial questions for conflict-free nets are all those that can be expressed in the fragment of CTL with syntax

$$\phi ::= s \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EX\phi \mid EF\phi$$

(see [7]). Among the "some" interesting polynomial questions for live free-choice nets are the following [5]:

– Is there a reachable marking which marks a given place?
– Is there a reachable marking which does not mark a given place?
– Is there a reachable marking which enables a given transition?
– Is the initial marking reachable from every reachable marking?
– Is there a run that does not contain a given transition?

Interestingly, the reachability problem for 1-safe live free-choice nets is NP-complete [8], and so it is unlikely that it will ever be added to this list.

# Part II

# General Petri nets

In this second part of the paper we consider arbitrary (finite) Place/Transition Petri nets. The *possible markings of a net* $\mathcal{N}$ or just the *markings of* $\mathcal{N}$ are now the set of all mappings $S \to I\!N$, where $S$ is the set of places of $\mathcal{N}$. Observe that, contrary to the 1-safe case, there is no a priori relation between the size of a net and the size of its markings. Notice also that the set of reachable markings may be infinite.

## 7 A universal lower bound

This section is the counterpart of Section 3 for Place/Transition Petri nets. The rule of thumb is now:

> Rule of thumb 6:
> All interesting questions about the behaviour of (Place/Transition) Petri nets are EXPSPACE-hard. More precisely, they require at least $2^{O(\sqrt{n})}$-space.

In particular, all the questions we asked about 1-safe Petri nets can be reformulated for Petri nets, and turn out to have at least this space complexity. As in the case of 1-safe Petri nets, this is a consequence of one single fundamental fact:

> A deterministic, exponentially bounded automaton of size $n$ can be simulated by a Petri net of size $O(n^2)$. Moreover, there is a polynomial time procedure which constructs this net.

In order to answer a question about the computation of an exponentially space bounded automaton $A$, we can construct the net that simulates $A$, which has size $O(n^2)$, and solve the corresponding question. If the original question requires $2^n$ space, as is the case for many properties, then the corresponding question about nets requires at least $2^{O(\sqrt{n})}$-space.

The fundamental fact above was first proved by Lipton [27]. Mayr and Meyer proved in [29] that it is possible to make the simulating net *reversible* (a net is reversible if for each transition $t$ there is a reverse transition $\overline{t}$ which "undoes" the effect of $t$). Since reversible nets are equivalent to commutative semigroups, the construction by Mayr and Meyer has important applications in mathematics.

Since Mayr and Meyer's construction is more involved than Lipton's, and since reversibility is not a main concern for this paper, we consider Lipton's construction in detail. It would have been easier to refer to Lipton's paper, but unfortunately it only exists as an old Yale report, quite difficult to find.

Bounded automata and general Place/Transition Petri nets do not "fit" well. It is not appropriate to model a cell of a bounded automaton as a place, as we did in the 1-safe case, because the cell contains one out of a *finite* number of possible symbols, while the place can contain infinitely many tokens, and so the same information as a nonnegative integer variable. So we use an intermediate model, namely *counter programs*. It is well-known that so-called bounded counter programs can simulate bounded automata (see below), and we show that Petri nets can simulate bounded counter programs.

A counter program is a sequence of *labelled commands* separated by semicolons. Basic commands have the following form, where l, $\mathrm{l}_1$, $\mathrm{l}_2$ are *labels* or *addresses* taken from some arbitrary set, for instance the natural numbers, and $x$ is a variable over the natural numbers, also called a *counter*:

| | |
|---|---|
| l: $x := x + 1$ | |
| l: $x := x - 1$ | |
| l: **goto** $\mathrm{l}_1$ | unconditional jump |
| l: **if** $x = 0$ **then goto** $\mathrm{l}_1$ **else goto** $\mathrm{l}_2$ | conditional jump |
| l: **halt** | |

A program is syntactically correct if the labels of commands are pairwise different, and if the destinations of jumps correspond to existing labels. For convenience we can also require the last command to be a **halt** command.

A program can only be executed once its variables have received initial values. In this paper we assume that the initial values are always 0. The semantics of programs is that suggested by the syntax. The only point to be remarked is that the command $l : x := x - 1$ *fails* if $x = 0$, and causes abortion of the program. Abortion must be distinguished from proper termination, which corresponds to the execution of a **halt** command. Observe in particular that counter programs are deterministic.

A counter program $C$ is $k$-bounded if after any step in its unique execution the contents of all counters are smaller than or equal to $k$. We make use of a well known construction of computability theory:

> There is a polynomial time procedure which accepts a deterministic bounded automaton $A$ of size $n$ and returns a counter program $C$ with $O(n)$ commands simulating the computation of $A$ on empty tape; in particular, $A$ halts if and only if $C$ halts. Moreover, if $A$ is exponentially bounded, then $C$ is $2^{2^n}$-bounded.

Now, it suffices to show that a $2^{2^n}$-bounded counter program of size $O(n)$ can be simulated by a Petri net of size $O(n^2)$. This is the goal of the rest of this section.

Since a direct description of the sets of places and transitions of the simulating net would be very confusing, we introduce a net programming notation with a very simple net semantics. It is very easy to obtain the net corresponding to a program, and execution of a command corresponds exactly to the firing of a transition. So we can and will look at the programming notation as a compact description language for Petri nets.

A *net program* is rather similar to a counter program, but does not have the possibility to branch on zero; it can only branch nondeterministically. However, it has the possibility of transferring control to a subroutine. The basic commands are as follows:

| | |
|---|---|
| l: $x := x + 1$ | |
| l: $x := x - 1$ | |
| l: **goto** $l_1$ | unconditional jump |
| l: **goto** $l_1$ **or goto** $l_2$ | nondeterministic jump |
| l: **gosub** $l_1$ | subroutine call |
| l: **return** | end of subroutine |
| l: **halt** | |

Syntactical correctness is defined as for counter programs. We also assume that programs are well-structured. Loosely speaking, a program is *well-structured* if it can be decomposed into a main program that only calls first-level subroutines, which in turn only call second-level subroutines, etc., and the jump commands in a subroutine can only have commands of the same subroutine as destinations.[8] We do not formally define well-structured programs, it suffices to know that all the programs of this section are well-structured.

We sketch a (Place/Transition) Petri net semantics of well-structured net programs. The Petri net corresponding to a program has a place for each label, a place for each variable, a distinguished *halt* place, and some additional places used to store the calling address of a subroutine call. There is a transition for each assignment and for each unconditional jump, and two transitions for each nondeterministic jump, as shown in Figure 4. We illustrate the semantics of the subroutine command by means of the program

1: **gosub** 4;
2: **gosub** 4;
3: **halt**;
4: **goto** 5 **or goto** 6;
5: **return**;
6: **return**

The corresponding Petri net is shown in Figure 5. Observe that the places *1_calls_4* and *2_calls_4* are used to remember the address from which the subroutine was called.

Clearly, the Petri net corresponding to a net program with $k$ commands has $O(k)$ places and $O(k)$ transitions, and its initial marking has size $O(k)$. So it is of size $O(k^2)$.

Let $C$ be a $2^{2^n}$-bounded counter program with $O(n)$ commands. We show that $C$ can be simulated by a net program $N(C)$ with $O(n)$ commands, which corresponds to a Petri net of size $O(n^2)$. Unfortunately, the construction of $N(C)$ requires quite a bit of low-level programming. But the reward is worth the hacking effort.

The notion of simulation is not as strong as in the case of 1-safe Petri nets. In particular, net programs are nondeterministic, while counter programs are deterministic. A net program $N$ simulates a counter program $C$ if the following property holds: $C$ halts (executes the command **halt**) if and only if *some* computation of $N$ halts (other computations may fail).

Each variable $x$ of $N$ (be it a variable from $C$ or an auxiliary variable) has an auxiliary *complement variable* $\overline{x}$. $N$ takes care of setting $\overline{x} = 2^{2^n}$ at the beginning

[8] Here we consider the main program as a zero-level subroutine, i.e., jump commands in the main program can only have commands of the main program as destinations.

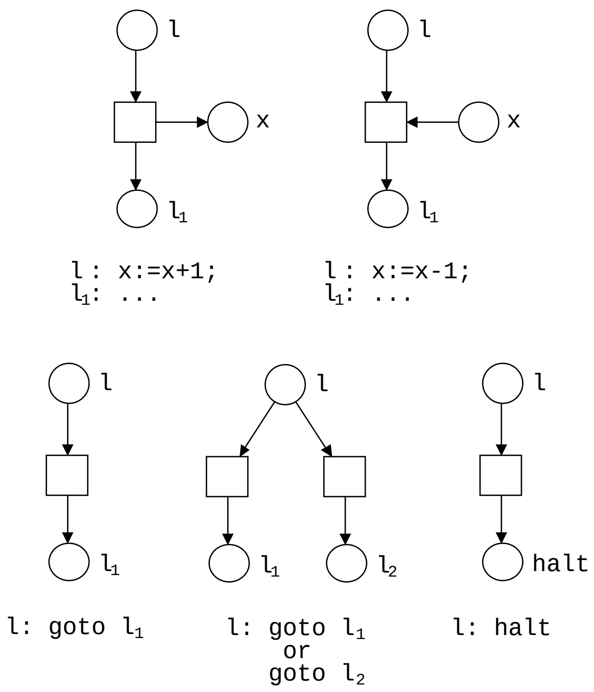


**Fig. 4.** Net semantics of assignments and jumps

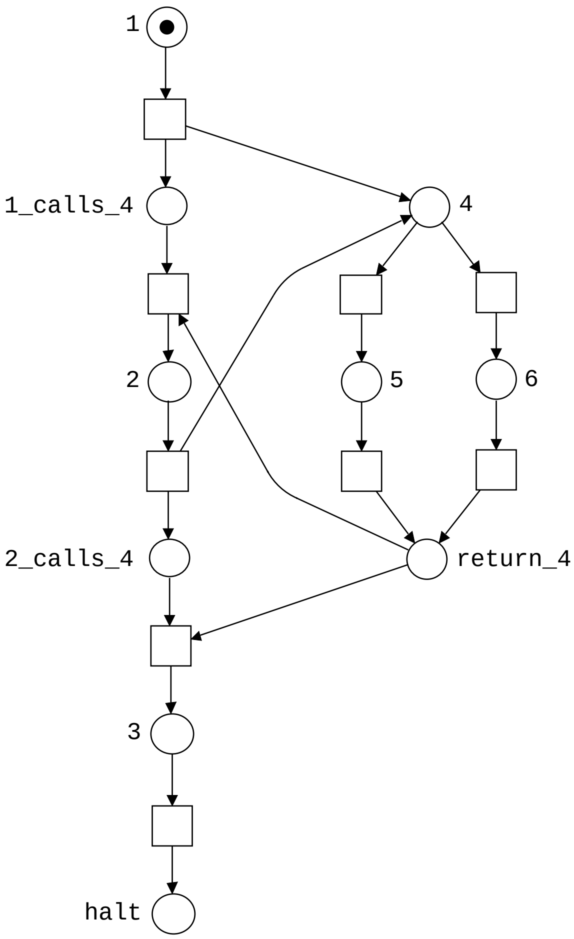


**Fig. 5.** Net semantics of subroutines

of the program. We call the code that takes care of this $N_{init}(C)$.[9] The rest of $N(C)$, called $N_{sim}(C)$, simulates $C$ and takes care of keeping the invariant $\overline{x} = 2^{2^n} - x$.

We design $N_{sim}(C)$ first. This program is obtained through replacement of each command of $C$ by an adequate net program. Commands of the form $x := x + 1$ ($x := x - 1$) are replaced by the net program $x := x+1; \overline{x} := \overline{x} - 1$ ($x := x-1; \overline{x} := \overline{x}+1$). Unconditional jumps are replaced by themselves. Let us now design a program

$$\text{Test}_n(x,\texttt{ZERO}, \texttt{NONZERO})$$

to replace a conditional jump of the form

l: **if** $x = 0$ **then goto** `ZERO`
**else goto** `NONZERO`

The specification of $\text{Test}_n$ is as follows:

> If $x = 0$ ($1 \leq x \leq 2^{2^n}$), then some execution of the program leads to `ZERO` (`NONZERO`), and no computation leads to `NONZERO` (`ZERO`); moreover the program has no side-effects: after any execution leading to `ZERO` or `NONZERO` no variable has changed its value.

Actually, it is easier to design a program $\text{Test}'_n(x,\texttt{ZERO}, \texttt{NONZERO})$ with the same specification but a *side-effect*: after an execution leading to `ZERO`, the values of $x$ and $\overline{x}$ are swapped.[10] Once $\text{Test}'_n$ has been designed, we can take:

Program $\text{Test}_n(x, \texttt{ZERO}, \texttt{NONZERO})$:

$\text{Test}'_n(x, \texttt{continue}, \texttt{NONZERO})$;
`continue`: $\text{Test}'_n(\overline{x}, \texttt{ZERO}, \texttt{NONZERO})$

because the values of $x$ and $\overline{x}$ are swapped $0$ times if $x > 0$ or twice if $x = 0$, and so $\text{Test}_n$ has no side effects.

The key to the design of $\text{Test}'_n$ lies in the following observation: Since $x$ never exceeds $2^{2^n}$, testing $x = 0$ can be replaced by nondeterministically choosing

- to decrease $x$ by $1$, and if we succeed then we know that $x > 0$, or
- to decrease $\overline{x}$ by $2^{2^n}$, and if we succeed then we know that $\overline{x} = 2^{2^n}$, and so $x = 0$.

If we choose wrongly, that is, if for instance $x = 0$ holds and we try to decrease $x$ by $1$, then the program fails; this is not a problem, because we only have to guarantee that the program *may* (not must!) terminate, and that if it terminates then it provides the right answer.

Decreasing $x$ by $1$ is easy. Decreasing $\overline{x}$ by $2^{2^n}$ is the difficult part. We leave it for a routine $\text{Dec}_n$ to be designed, which must satisfy the following specification:

---

[9] Recall that by definition all variables of $N$ have initial value $0$. Therefore, if we need $\overline{x} = 2^{2^n}$ initially, then we have to design preprocessing code for it.

[10] Executions leading to `NONZERO` must still be free of side-effects.

If the initial value of $s$ is smaller than $2^{2^n}$, then every execution of $\mathrm{Dec}_n$ fails. If the value of $s$ is greater than or equal to $2^{2^n}$, then all executions terminating with a **return** command have the same effect as $s := s - 2^{2^n}; \overline{s} := \overline{s} + 2^{2^n}$; in particular, there are no side-effects. All other executions fail.

$\mathrm{Test}'_n$ proceeds by transferring the value of $x$ to a special variable $s_n$, and then calling the routine $\mathrm{Dec}_n$, which decreases $s_n$ by $2^{2^n}$. In this way we need one single routine $\mathrm{Dec}_n$, instead of one for each different variable to be decreased, which leads to a smaller net program.

Program $\mathrm{Test}'_n(x$, `ZERO`, `NONZERO`):

** initially $s_n = 0$ and $\overline{s}_n = 2^{2^n}$ **
          **goto** `nonzero` **or goto** `loop`;
`nonzero`: $x := x - 1; x := x + 1$; goto `NONZERO`;
    `loop`: $\overline{x} := \overline{x} - 1; x := x + 1; s_n := s_n + 1; \overline{s}_n := \overline{s}_n - 1$;
          **goto** `exit` **or goto** `loop`
    `exit`: **gosub** `dec`$_\mathtt{n}$; **goto** `ZERO`
** the routine called at `dec`$_\mathtt{n}$ is $\mathrm{Dec}_n(s_n)$ **

It is easy to see that $\mathrm{Test}'_n$ meets its specification: if $x > 0$, then we may choose the `nonzero` branch and reach `NONZERO`. If $x = 0$, then $\overline{x} = 2^{2^n}$. After looping $2^{2^n}$ times on `loop` the values of $x$, $\overline{x}$ and $s_n$, $\overline{s}_n$ have been swapped. The values of $s_n$ and $\overline{s}_n$ are swapped again by the subroutine $\mathrm{Dec}_n$, and then the program moves to `ZERO`. Moreover, if $x = 0$ then no execution reaches the `NONZERO` branch, because the program fails at $x := x - 1$. If $x > 0$, then no execution reaches the `ZERO` branch, because $s_n$ cannot reach the value $2^{2^n}$, and so $\mathrm{Dec}_n$ fails.

The next step is to design $\mathrm{Dec}_n$. We proceed by induction on $n$, starting with $\mathrm{Dec}_0$. This is easy, because it suffices to decrease $s$ by $2^{2^0} = 2$. So we can take

Subroutine $\mathrm{Dec}_0(s)$:

$s := s - 1; \overline{s} := \overline{s} + 1$;
$s := s - 1; \overline{s} := \overline{s} + 1$;
**return**

Now we design $\mathrm{Dec}_{i+1}$ under the assumption that $\mathrm{Dec}_i$ is already known. The definition of $\mathrm{Dec}_{i+1}$ contains two copies of a program $\mathrm{Test}'_i$, called with different parameters. We define this program by substituting $i$ for $n$ everywhere in $\mathrm{Test}'_n$. $\mathrm{Test}'_i$ calls the routine $\mathrm{Dec}_i$ at the address $dec_i$. Notice that this is correct, because we are assuming that the routine $\mathrm{Dec}_i$ has already been defined.

The key to the design of $\mathrm{Dec}_{i+1}$ is that decreasing by $2^{2^{i+1}}$ amounts to decreasing $2^{2^i}$ times by $2^{2^i}$, because

$$2^{2^{i+1}} = (2^{2^i})^2 = 2^{2^i} \cdot 2^{2^i}$$

So decreasing by $2^{2^{i+1}}$ can be implemented by two nested loops, each of which is executed $2^{2^i}$ times, such that the body of the inner loop decreases $s$ by 1. The loop variables have initial values $2^{2^i}$, and termination of the loops is detected by testing the loop variables for $0$. This is done by the $\text{Test}'_i$ programs.

Subroutine $\text{Dec}_{i+1}(s)$:

** Initially $y_i = 2^{2^i} = z_i, \overline{y}_i = 0 = \overline{z}_i$ **
** The initialisation is carried out by $\text{N}_{init}$ **

```
outer_loop: y_i := y_i - 1; ȳ_i := ȳ_i + 1;
inner_loop: z_i := z_i - 1; z̄_i := z̄_i + 1;
            s := s - 1; s̄ := s̄ + 1;
            Test'_i(z_i, inner_exit , inner_loop);
inner_exit: Test'_i(y_i, outer_exit, outer_loop);
outer_exit: return
```

Observe also that both instances of $\text{Test}'_i$ call the same routine at the same label.

It could seem that $\text{Dec}_{i+1}$ swaps the values of $y_i$, $\overline{y}_i$ and $z_i$, $\overline{z}_i$, which would be a side-effect contrary to the specification. But this is not the case. These swaps are compensated by the side-effects of the `ZERO` branches of the $\text{Test}'_i$ programs! Notice that these branches are now the `inner_exit` and `outer_exit` branches. When the program leaves the inner loop, $\text{Test}'_i$ swaps the values of $z_i$ and $\overline{z}_i$. When the program leaves the outer loop, $\text{Test}'_i$ swaps the values of $y_i$ and $\overline{y}_i$.

This concludes the description of the program $\text{Test}_n$, and so the description of the program $N_{sim}(C)$. It remains to design $N_{init}(C)$. Let us first make a list of the initialisations that have to be carried out. $N_{sim}(C)$ contains

- the variables $x_1, \ldots, x_l$ of $C$ with initial value $0$; their complementary variables $\overline{x}_1, \ldots, \overline{x}_l$ with initial value $2^{2^n}$;
- a variable $s$ with initial value $0$; its complementary variable $\overline{s}$ with initial value $2^{2^n}$;
- two variables $y_i, z_i$ for each $i$, $0 \leq i \leq n-1$, with initial value $2^{2^i}$; their complementary variables $\overline{y}_i, \overline{z}_i$ for each $i$, $0 \leq i \leq n-1$, with initial value $0$.

Now, the specification of $N_{init}(C)$ is simple

> $N_{init}(C)$ uses only the variables in the list above; every successful execution leads to a state in which the variables have the correct initial values.

$N_{init}(C)$ calls programs $\text{Inc}_i(v_1, \ldots, v_m)$ with the following specification:

> All successful executions have the same effect as
>
> $$\begin{gathered} v_1 := v_1 + 2^{2^i}; \\ \ldots; \\ v_m := v_m + 2^{2^i} \end{gathered}$$
>
> In particular, there are no side-effects.

These programs are defined by induction on $i$, and are very similar to the family of $\text{Dec}_i$ programs. We start with $\text{Inc}_0$:

Program $\mathrm{Inc}_0(v_1, \ldots, v_m)$:

$v_1 := v_1 + 1; v_1 := v_1 + 1;$
$\ldots$
$v_m := v_m + 1; v_m := v_m + 1$

and now give the inductive definition of $\mathrm{Inc}_{i+1}$:

Program $\mathrm{Inc}_{i+1}(v_1, \ldots, v_m)$:

```
    ** Initially y_i = 2^{2^i} = z_i, ȳ_i = 0 = z̄_i **
outer_loop:  y_i := y_i − 1; ȳ_i := ȳ_i + 1;
inner_loop:  z_i := z_i − 1; z̄_i := z̄_i + 1;
             v_1 := v_1 + 1;
             ...
             v_m := v_m + 1;
             Test'_i(z_i, inner_exit , inner_loop);
inner_exit:  Test'_i(y_i, outer_exit, outer_loop);
outer_exit:  ...
```

It is easy to see that these programs satisfy their specifications. Now, let us consider $\mathrm{N}_{init}(C)$. Apparently, we face a problem: in order to initialise the variables $v_1, \ldots, v_m$ to $2^{2^{i+1}}$ the variables $y_i$ and $z_i$ must have already been initialised to $2^{2^i}$! Fortunately, we find a solution by just carrying out the initialisations in the right order:

Program $\mathrm{N}_{init}(C)$:

$\mathrm{Inc}_0(y_0, z_0);$
$\mathrm{Inc}_1(y_1, z_1);$
$\ldots$
$\mathrm{Inc}_{n-1}(y_{n-1}, z_{n-1});$
$\mathrm{Inc}_n(\overline{s}, \overline{x}_1, \ldots, \overline{x}_l)$

This concludes the description of $N(C)$, and it is now time to analyse its size. Consider $N_{sim}(C)$ first. It contains two assignments for each assignment of $C$, an unconditional jump for each unconditional jump in $C$, and a different instance of $\mathrm{Test}_k$ for each conditional jump. Moreover, it contains (one single instance of) the routines $\mathrm{Dec}_n, \mathrm{Dec}_{n-1}, \ldots, \mathrm{Dec}_0$ (notice that $\mathrm{Test}_n$ calls $\mathrm{Dec}_n$, which calls $\mathrm{Dec}_{n-1}$, etc.). Both $\mathrm{Test}_n$ and the routines have constant length. So the number of commands of $N_{sim}(C)$ is $O(n)$.

$N_{init}(C)$ contains (one single instance of) the programs $\mathrm{Inc}_i$ $1 \leq i \leq n$. The programs $\mathrm{Inc}_1, \ldots, \mathrm{Inc}_{n-1}$ have constant size, since they initialise a constant number of variables. The number of commands of $\mathrm{Inc}_n$ is $O(n)$, since it initialises $O(n)$ variables.

So we have proved that $N(C)$ contains $O(n)$ commands. It follows that its corresponding Petri net has size $O(n^2)$, which concludes our presentation of Lipton's result.

### The solution to Story II

Recall the conjecture of Story II: given a net $\mathcal{N}$ and two markings $M_1$ and $M_2$, if $M_2$ is reachable from $M_1$ then it is reachable from $M_1$ through a sequence $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1}$

$\cdots \xrightarrow{t_n} M_n = M$ such that all the markings $M_1, \ldots, M_n$ have size $O(n + m_0 + m)$, where $n, m_0, m$ are the sizes of $\mathcal{N}$, $M_0$ and $M$ respectively.

Let $c$ be the constant such that $M_0, \ldots, M_n$ have size at most $c \cdot (n+m_0+m)$. If the conjecture is true, then the following nondeterministic algorithm solves the reachability problem, since it may always answer "true" when $M$ is reachable:

Algorithm Reachable($\mathcal{N}$, $M_0$, $M$):

**variable:** $M'$ of type marking;

**begin**
  $M' := M_0$;
  **while** $M' \neq M$ **do**
    choose a marking $M''$ of size at most $c \cdot (n + m_0 + m)$
    such that $M' \xrightarrow{t} M''$ for some transition $t$;
    if there is no such marking then stop;
    $M' := M''$;
  **od**;
  **return true**
**end**

Since the algorithm only visits markings of size $c \cdot (n + m_0 + m)$, it runs in linear space. By Savitch's construction there is a deterministic algorithm which uses quadratic space. Since the reachability problem requires exponential space, the conjecture is false.

## 8 Upper bounds

The general exponential space lower bound of the last section is almost the best we can hope for, because Rackoff gave in [32] an almost matching exponential space upper bound for the covering and boundedness problems for Petri nets. More precisely, the upper bound is $2^{O(n \log n)}$ space, very close to the $2^{O(\sqrt{n})}$ lower bound. The covering problem consists of deciding if there exists a reachable marking $M$ such that $M \geq M'$ for a given marking $M'$, i.e., if there exists a reachable marking $M$ *covering* $M'$; the boundedness problem consists of deciding if the number of reachable markings is finite.

Yen showed some years later in [38] that the same upper bound holds for the problem of deciding if there exists a firing sequence

$$M_0 \xrightarrow{\sigma_1} M_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} M_k$$

satisfying a given predicate $F(M_1, \ldots, M_k, \sigma_1, \ldots, \sigma_k)$ constructed using the following syntax:[11]

$$\begin{aligned} F ::= {} & M_i(s) \geq c \mid M_i(s) > c \\ & M_i(s) > M_j(s) \mid M_i(s) < M_j(s) \mid M_i(s) = M_j(s) \\ & \#_{\sigma_i}(t) > c \mid \#_{\sigma_i}(t) < c \mid \#_{\sigma_i}(t) = c \\ & \#_{\sigma_i}(t) > \#_{\sigma_j}(t') \mid \#_{\sigma_i}(t) < \#_{\sigma_j}(t') \mid \#_{\sigma_i}(t) = \#_{\sigma_j}(t') \\ & F_1 \wedge F_2 \mid F_1 \vee F_2 \end{aligned}$$

[11] The syntax is actually more general, see [38] for the details.

where $s$ is a place, $t$ and $t'$ are transitions, $c$ is a constant, and $\#_\sigma(t)$ denotes the number of times that $t$ occurs in $\sigma$. Both the covering and the boundedness problem can be reduced to Yen's problem. The covering problem for a marking $M = (m_1, \ldots, m_n)$ corresponds to deciding if there exists a firing sequence $M_0 \xrightarrow{\sigma_1} M_1$ such that $M_1(s_1) \geq m_1 \wedge \ldots \wedge M_1(s_n) \geq m_n$. The boundedness problem can be easily shown to be equivalent to the problem of deciding if there exists a sequence $M_0 \xrightarrow{\sigma_1} M_1 \xrightarrow{\sigma_2} M_2$ such that $M_1(s_1) \geq M_2(s_1) \wedge \ldots \wedge M_1(s_n) \geq M_2(s_n)$ and $M_1(s_1) > M_2(s_1) \vee \ldots \vee M_1(s_n) > M_2(s_n)$. Observe however that the reachability problem *cannot* be reduced to Yen's problem, because the predicate $M(s) = c$ does not belong to the syntax. The reachability problem was shown to be decidable by Mayr [28] and shortly after with a simpler proof by Kosaraju [25], but all known algorithms are non-primitive recursive. Closing the gap between the exponential space lower bound and the non-primitive recursive upper bound is one of the most relevant open problems of net theory.

Is it possible to give more general results about the properties that are decidable, and the properties that are decidable in exponential space? In particular, we would like to show that all the properties of a certain temporal logic are decidable, or decidable in exponential space. As we are going to see, there is a very significant difference between state-based logics and action-based logics, and so we consider them separately.

### 8.1 The state-based case

We have the following very general rule of thumb:

| |
|---|
| Rule of thumb 7:<br>The model-checking problems of all interesting state-based logics are undecidable. |

As in the 1-safe case, we first have to choose a set of atomic propositions. We take again $Prop = S$, i.e., the atomic propositions are the places of $N$. We say that a marking $M$ satisfies the proposition $s$ if $M$ is marked at $s$. Observe that a computation is *no longer* a sequence of markings; a computation is a sequence of sets of places, as in the 1-safe case, but the markings of general Place/transition nets are not sets of places anymore.

With this choice of atomic propositions we can only express that a place is marked or not; we can say nothing about the number of tokens it contains. Unfortunately, even with this restricted expressive power the model checking problems for LTL and CTL turn out to be undecidable.

The proof is in both cases by reduction from the following problem, which is known to be undecidable:

Given: a counter program $C$ with counters initialised to $0$.
To decide: if $C$ halts.

We simulate once again counter programs by net programs. Given a counter program $C$, we obtain a net program $N'(C)$ through replacement of each counter command

$$\text{l: } \textbf{if } \ x = 0 \ \textbf{ then goto } \ \text{l}_1 \textbf{ else goto } \ \text{l}_2$$

by the net program

$$\begin{array}{r l} \text{l: } & \textbf{goto}\ \texttt{test\_l}_1 \ \textbf{ or goto } \ \texttt{test\_l}_2; \\ \texttt{test\_l}_1 & \textbf{goto } \ \text{l}_1; \\ \texttt{test\_l}_2 & \textbf{goto } \ \text{l}_2 \end{array}$$

while other commands are replaced by themselves.

The net program $N'(C)$ simulates $C$ in a much weaker sense than that of Section 7. $N'(C)$ has a *honest* run that exactly mimics the (unique) execution of $C$: whenever $C$ executes the command $l$, $N'(C)$ chooses the same branch as $C$. However, it also has many other runs that "cheat", i.e., runs that at some point choose the wrong branch. The labels $\texttt{test\_l}_1$ and $\texttt{test\_l}_2$ correspond to two places of $N'(C)$ which can be used to test if the program has cheated or not when executing the conditional jump.

Suppose that there exists a temporal logic formula *Halt* with the following property:

$N'(C)$ satisfies *Halt* if and only if the honest execution of $N'(C)$ halts.[12]

Since the honest run exactly mimics the execution of the counter program $C$, $N'(C)$ satisfies *Halt* if and only if $C$ halts. Therefore, the problem of deciding if *Halt* is satisfied by a given Petri net $N$ is undecidable. It follows that the model-checking problem of those logics in which *Halt* was expressed is undecidable as well.

We construct in CTL and LTL very simple formulas *LTL-Halt* and *CTL-Halt*. We first define a formula *Cheat* without temporal operators. *Cheat* is the conjunction over all conditional jumps l: **if** $x = 0$ **then goto** $\text{l}_1$ **else goto** $\text{l}_2$ of the formulas:

$$(\texttt{test\_l}_1 \wedge x) \vee (\texttt{test\_l}_2 \wedge \neg x)$$

If a run visits a marking satisfying *Cheat*, then we know that it is dishonest: if the marking satisfies $(\texttt{test\_l}_1 \wedge x)$, then at some conditional jump the run has taken the $\text{l}_1$ branch even though $x > 0$; if $(\texttt{test\_l}_2 \wedge \neg x)$, then the run has taken the $\text{l}_2$ branch even though $x = 0$. Now, we define

$$LTL - Halt = F(Cheat \vee halt)$$

where *halt* is the place in the net semantics corresponding to all the **halt** commands. A run satisfies *LTL-Halt* if at some point it cheats or it halts. $N'(C)$ satisfies *LTL-Halt* iff every run satisfies *LTL-Halt*. Since the honest run is the only one that doesn't cheat, $N'(C)$ satisfies *LTL-Halt* iff the honest run halts.

The formula *CTL-Halt* is :

$$CTL - Halt = AF(Cheat \vee halt)$$

[12] Since $N'(C)$ is just a shorthand description of a Petri net, it makes sense to ask if $N'(C)$ satisfies a property formalised as a temporal formula.

It follows immediately from the semantics of formulae that $N'(C)$ satisfies *CTL-Halt* if and only if it satisfies *LTL-Halt*.

Since the formula *CTL-Halt* only contains the operator $AF$, the fragment of CTL that extends propositional logic with the operators $EF$ and its dual $AG$ could still be decidable. Unfortunately, a different proof [9] shows that this is not the case.

### 8.2 The action-based case

As mentioned above, the action-based case is very different from the state-based case:

> Rule of thumb 8:
> The model-checking problems of all interesting branching-time, action-based logics are undecidable. The model-checking problems of all interesting linear-time, action-based logics are decidable.

The undecidability of branching-time logics in the action-based case is an immediate consequence of the following fact: given an unlabelled Petri net $N$ and a formula $\phi$ of state-based CTL there is a labelled net $N'$ and a formula $\phi'$ of action-based CTL such that $N$ satisfies $\phi$ if and only if $N'$ satisfies $\phi'$.

The net $N'$ is obtained by labelling the transitions of $N$ with some label, say $a$, and then adding for each place $s$ a new transition $t_s$ having $s$ as only input place, no output place at all, and labelled by $s$. The formula $\phi'$ is obtained through replacement of each atomic proposition $s$ by $EX_s true$, and of each temporal operator $EX$, $AX$, $E[\ldots U \ldots]$, $A[\ldots U \ldots]$ by $EX_{\{a\}}$, $AX_{\{a\}}$, $E[\ldots U_{\{a\}\ldots]}$, and $A[\ldots U_{\{a\}} \ldots]$, respectively. Observe that $s$ holds iff the transition $t_s$ can occur, i.e., iff $EX_s true$ holds.

We cannot use the same technique to prove the undecidability of the model-checking problem for LTL, because the problem is decidable! As in the 1-safe case, the model-checking algorithm is based on automata theory. Given an LTL formula $\phi$, one can build a finite automaton $A_\phi$ and a Büchi automaton $B_\phi$ such that $L(A_\phi) \cup L_\omega(B_\phi)$ is exactly the set of computations satisfying the formula $\phi$. In the action-based case both $A_\phi$ and $B_\phi$ are automata over the alphabet *Act*.

In the 1-safe case, given a net $N$ and a formula $\phi$, we first constructed two automata $A_{\neg\phi}$ and $B_{\neg\phi}$ such that $L(A_{\neg\phi}) \cup L_\omega(B_{\neg\phi})$ is exactly the set of computations *violating* the formula $\phi$. In the general case we proceed exactly in the same way. The second step was to construct two finite automata $A_N$ and $B_N$ from the Petri net $N$, which were both essentially equal to the reachability graph of the net. Here we have a problem: the automata $A_N$ and $B_N$ can be defined just as in the 1-safe case, but since $N$ may now have infinitely many reachable markings, they are not guaranteed to be finite.

The solution to this problem is easy: instead of constructing two automata $A_N$ and $B_N$ out of the Petri net $N$, we construct two labelled Petri nets $NA_{\neg\phi}$ and $NB_{\neg\phi}$ out of the automata $A_{\neg\phi}$ and $B_{\neg\phi}$ in the following obvious way:

- the places of $NA_\phi$ are the states of $A_\phi$;
- for each transition $q \xrightarrow{a} q'$ in $A_\phi$ add a transition to $NA_\phi$, labelled by $a$, with $q$ and $q'$ as input and output place.

$NB_\phi$ is constructed analogously. Now we construct the products $N \times NA_\phi$ and $N \times NB_\phi$, where the product $N_1 \times N_2$ of two Petri nets $N_1$ and $N_2$ is another Petri net defined in the following way:

- the set of places of $N$ is the union of the sets of places of $N_1$ and $N_2$;
- for each pair of transitions $t_1$ of $N_1$ and $t_2$ of $N_2$ labelled by a same action $a$, the product $N$ contains a transition $(t_1, t_2)$ also labelled by $a$; the input (output) places of $(t_1, t_2)$ are the union of the input (output) places of $t_1$ and $t_2$.

The two following results are easy to prove:

- $L_\omega(B_N) \cap L(B_\phi) \neq \emptyset$ holds if and only if the Petri net $N \times NB_\phi$ has a run which marks some place corresponding to a final state of $B_\phi$ infinitely often.
- $L(A_N) \cap L(A_\phi) \neq \emptyset$ holds if and only if the Petri net $N \times NA_\phi$ has a reachable dead marking which marks some place corresponding to a final state of $A_\phi$.

Finding a run of $N \times NB_\phi$ that marks some place from a given set $FS$ of final places infinitely often is equivalent to deciding if there exists a firing sequence $M_0 \xrightarrow{\sigma_1} M_1 \xrightarrow{\sigma_2} M_2 \xrightarrow{\sigma_3} M_3$ in the net $N \times NB_\phi$ such that

$$(\bigwedge_{s \in S} M_3(s) \geq M_1(s)) \ \wedge \ (\bigvee_{s \in FS} M_2(s) \geq 1)$$

where $S$ denotes the set of all places. By Yen's result, introduced at the beginning of this section, the problem can be solved in exponential space in the size of $N \times NB_\phi$. In a more detailed analysis [14], Habermehl shows that this problem is EXPSPACE-complete in the size of $N$ and PSPACE-complete in the length of $\phi$.

Finding a dead reachable marking of $N \times NA_\phi$ that marks some place from a given set $FS$ of final places can be reduced to and is at least as hard as the reachability problem. Therefore, there exist so far no primitive recursive algorithms for it.

As in the 1-safe case, these results can be generalised to any logic for which the translation into automata theory holds [9].

## 9 All equivalence problems are undecidable

This section's rule of thumb has a rather negative flavour:

| |
|---|
| Rule of thumb 9:<br>All equivalence problems for Petri nets are undecidable. |

This rule is supported by a recent and very nice result due to Jančar, showing that every equivalence notion between trace and bisimulation equivalence is undecidable for Petri nets.[13] Jančar himself has presented his result very clearly in [22]; here we do it in a slightly different way. We proceed by reduction from the problem

[13] Actually, the result is a bit stronger, since bisimulation can be replaced by an even finer equivalence.

Given: a counter program $C$,
To decide: if $C$ halts (recall that all counters are initialised to 0).

which is known to be undecidable.

Although the result can be presented directly by constructing two Petri nets out of $C$ (and this is the way the proof in [22] goes), we prefer to use again a net programming language with a very simple net semantics, this time a language of *guarded commands*. A program is a sequence of instructions, and instructions are expressions of the form

$$
\begin{array}{ll}
1 : [ & \Box\ \mathit{guard}_1 \xrightarrow{action_1} \mathit{command}_1 \\
 & \Box\ \mathit{guard}_2 \xrightarrow{action_2} \mathit{command}_2 \\
 & \ldots \\
 & \Box\ \mathit{guard}_n \xrightarrow{action_n} \mathit{command}_n\ ]
\end{array}
$$

where l is a label, *action*$_1$, ... , *action*$_n$ are actions, a *guard* is either the special string **true** or a conjunction of expressions of the form $x > 0$ (no guards of the form $x = 0$ are allowed), and the possible *commands* are **skip**, **goto** l , **halt**, or a sequence of assignments $x_1 := x_1 \pm 1;\ \ldots ;\ x_n := x_n \pm 1$, where the $x_i$ are pairwise different.

Operationally, an instruction is executed as follows: one of the guards that evaluate to true at the current state is nondeterministically selected (if no guard evaluates to true, the program aborts). Then, two things happen: the action of the selected guard is sent to the environment, and its command is executed (if the command contains the assignment $x := x - 1$ and $x = 0$ holds, then the program aborts). If the command is a jump **goto** l, then execution continues at the instruction with label l. If the command is **skip** or an assignment, then execution continues with the next instruction. An observer can only see the actions executed by the program, but not the values of its variables, or the label of the instruction being currently executed.

Guarded command programs can be easily translated into labelled Petri nets. Figure 6 shows the labelled net corresponding to the instruction

$$
\begin{array}{ll}
1 : [ & \Box\ x > 0 \xrightarrow{a} x := x - 1 \\
 & \Box\ \mathit{true} \xrightarrow{b} x := x + 1 \\
 & \Box\ x > 0 \wedge y > 0 \xrightarrow{a} \textbf{goto}\ 3 \\
 & \Box\ \mathit{true} \xrightarrow{c} \textbf{halt}\ ]
\end{array}
$$

(where we assume that the instruction following 1 in the program is labelled by 2). There is a place for each variable and each label, plus a special place *halt*. There is a transition for each alternative, labelled by the alternative's action. The semantics of a program is obtained by merging places of the nets corresponding to its instructions carrying the same label. We identify a program with its corresponding labelled Petri net. In particular, two programs are trace or bisimulation equivalent if their corresponding labelled nets are.

Given a counter program $C$, we construct two net programs $N_1(C)$ and $N_2(C)$ satisfying the following two properties:

(1) if $C$ halts, then $N_1(C)$ and $N_2(C)$ are not trace equivalent, and

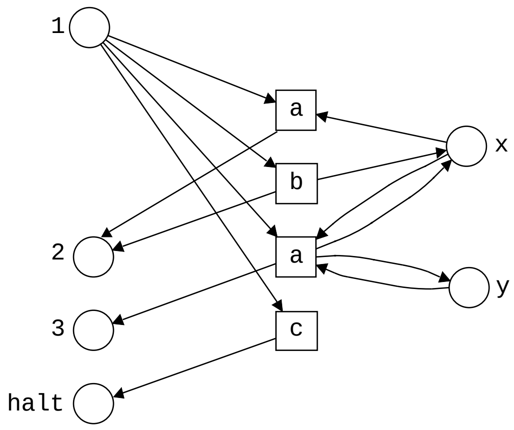


**Fig. 6.** Net corresponding to an instruction

(2) if $C$ does not halt, then $N_1(C)$ and $N_2(C)$ are bisimilar.

For the proof of these properties it is very useful to characterise trace and bisimulation equivalences in terms of *two-person games*. We describe first the features common to both trace and the bisimulation games. The board of the games are the two programs $N_1(C)$ and $N_2(C)$ in their initial states. The games are played by two players, Alice and Bob, who alternate moves. Alice makes the first move. A move is the execution of (one of the alternatives of) an instruction in either $N_1(C)$ or $N_2(C)$, and is named after the action corresponding to the executed alternative. That is, an $a$-move is the execution of an alternative of the form *guard* $\xrightarrow{a}$ *command*. If Alice makes an $a$-move in one of the programs, then Bob can only answer with an $a$-move in the other program. It may help your intuition to imagine that Alice wishes the programs to be non-equivalent, while Bob wishes them to be equivalent. The winner of a game is decided as follows:

- if Alice has no move available, then Bob wins;
- if Bob cannot answer to Alice's move, then Alice wins;
- if the game does not terminate, then Bob wins.

If you find the idea of a non-terminating game awkward, think of chess without the 50-move rule. If a position with only the two kings on the board is reached, then the game goes on forever. In the trace and bisimulation games a situation like this is not a draw, but a win for Bob. Bob only wins after infinite time, which can make the game rather tedious, but that's his problem: the winning condition is well defined, and every game has a winner.

We describe now the differences between the trace and bisimulation games, which are surprisingly small. In a trace game, Alice chooses one of the programs *at the beginning of the game*, and makes *all her moves* in this program; Bob must make all his moves in the other program. In a bisimulation game, Alice chooses one of the programs *before each move*, and makes *her next move* in this program. For instance, in the bisimulation game Alice can make her first move in the first program (Bob must answer in the second), and her second move in the second program (Bob must answer in the first).

A *strategy* for a player is a function which gets the list of moves played so far and yields the player's next move. A strategy is *winning* if a player that sticks to it wins *all* games. We have the following nice result (see for instance [34]), which at least in the case of the trace game is intuitively very plausible:

> In the trace and bisimulation games for $N_1(C)$ and $N_2(C)$:
> if Alice has a winning strategy, then the two programs are not equivalent; if Bob has a winning strategy, then the two programs are equivalent.

So the properties (1) and (2) that $N_1(C)$ and $N_2(C)$ – both to be constructed – have to satisfy can be reformulated as follows:

(1) if $C$ halts, then Alice has a winning strategy in the trace game, and
(2) if $C$ does not halt, then Bob has a winning strategy in the bisimulation game.

It is time to start with the definition of $N_1(C)$ and $N_2(C)$. To make things a bit simpler, assume without loss of generality that the counter program $C$ contains one single **halt** instruction, and that this instruction is the last one.[14] The programs $N_1(C)$ and $N_2(C)$ look as follows:

Program $N_1(C)$:

`start`:[ **true** $\xrightarrow{start} y := y + 1$ ];
$\quad N'(C)$;
`halt`:[ $y > 0 \xrightarrow{halt}$ **halt** ]

Program $N_2(C)$:

`start`:[ **true** $\xrightarrow{start} z := z + 1$ ];
$\quad N'(C)$;
`halt`:[ $z > 0 \xrightarrow{halt}$ **halt** ]

where the program $N'(C)$ still has to be defined. Observe that the two programs differ only in the first instruction. After this instruction is executed, $y = 1, z = 0$ in $N_1(C)$, and $y = 0, z = 1$ in $N_2(C)$.

The program $N'(C)$ is obtained by replacing each command of $C$ but the unique **halt** command through an instruction of the new language. The instructions corresponding to assignments and jumps are:

l: $x := x + 1$ is replaced by l: [`true` $\xrightarrow{inc} x := x + 1$]
l: $x := x - 1$ is replaced by l: [`true` $\xrightarrow{dec} x := x - 1$]
l: **goto** $l_1$ is replaced by l: [`true` $\xrightarrow{jump}$ **goto** $l_1$]

Conditional jumps are the delicate part. A command of the form

l: **if** $x = 0$ **then goto** `ZERO`
**else goto** `NONZERO`

is replaced by the following sequence of two instructions:

[14] If there are several **halt** instructions, we can replace them by jumps to a new label at the end of the program, and place there a unique **halt** command.

$$
\begin{aligned}
\mathsf{l}: [\ \Box\ & x > 0 \xrightarrow{nonzero} \textbf{goto}\ \mathtt{NONZERO} \\
\Box\ & \textbf{true} \xrightarrow{zero} \textbf{skip} \\
\Box\ & x > 0 \wedge y > 0 \xrightarrow{zero} y := y - 1; z := z + 1\ ] \\
\Box\ & x > 0 \wedge z > 0 \xrightarrow{zero} y := y + 1; z := z - 1\ ];
\end{aligned}
$$

$$\mathsf{l}' \text{: [ } \textbf{true} \xrightarrow{zero} \textbf{goto}\ \mathtt{ZERO}\ ]$$

This completes the description of $N_1(C)$ and $N_2(C)$. Before going on, we observe that the program $N'(C)$ has an *honest* run that mimics the execution of $C$, and looks as follows: whenever $C$ executes a command, $N'(C)$ executes its corresponding instruction. If the command is a conditional jump and $C$ takes the `NONZERO`-branch, then $N'(C)$ chooses the *nonzero* alternative of the corresponding instruction; if $C$ takes the *ZERO* branch, then $N'(C)$ chooses the *first* of the two *zero* alternatives, namely **true** $\xrightarrow{zero}$ **skip**, and then it executes the **goto** `ZERO` instruction.

There is an important difference between $N_1(C)$ and $N_2(C)$. Assume that in both $N_1(C)$ and $N_2(C)$ we execute the *start* action, followed by the honest execution of $N'(C)$. If and when the honest execution terminates, we can execute the *halt* action in $N_1(C)$, because $y$ has been set to 1 by the *start* action, but we *cannot* execute it in $N_2(C)$, because $y$ still has the value 0 there.

We are now ready to describe the winning strategies for Alice and Bob in the different games.

*Assume that $C$ halts.* Here is the strategy for Alice in the trace game. Alice chooses to play on $N_1(C)$, and so Bob is forced to play on $N_2(C)$. Alice sticks to the following sequence of moves, completely disregarding Bob's answers: she plays the *start*-move, continues with the moves of the honest execution of $N'(C)$, and – if the honest run terminates – finishes with a *halt*-move.

We show in the first place that, if Alice follows this strategy, then from the second move on Bob is forced to play *exactly the same moves* as Alice (i.e., exactly the same alternatives in the same commands). When Alice plays a *nonzero* move, Bob can only answer with a unique *nonzero* move, so this case is easy. When Alice plays a *zero* move, it seems that Bob can choose between three *zero*-answers, namely

$$
\begin{aligned}
& \textbf{true} \xrightarrow{zero} \textbf{skip} \\
& x > 0 \wedge y > 0 \xrightarrow{zero} y := y - 1; z := z + 1 \\
& x > 0 \wedge z > 0 \xrightarrow{zero} y := y + 1; z := z - 1
\end{aligned}
$$

But remember: Alice is playing the honest run, and so she only plays a *zero*-move when $x = 0$. So, whenever Alice plays a *zero* move, Bob observes that the guard $x > 0$ evaluates to false, and so that his only move is **true** $\xrightarrow{zero}$ **skip**.

Let us now see that Alice's strategy is winning. Since $C$ halts, the honest run terminates, and so eventually Alice plays a *halt* move.[15] All along the game Bob has patiently

[15] Incidentally, observe that Alice can indeed play *halt*, because she set $y$ to 1 with her *start* move, and she never touched $y$ during the honest execution.

repeated Alice's moves, waiting for a chance, but his efforts are in vain: he cannot reply to Alice's *halt* move, because in his program $N_2(C)$ the variable $y$ has the value $0$, and so the guard $y > 0$ of the *halt* move evaluates to false. So Bob loses.

*Assume that $C$ does not halt.* Here is the strategy for Bob in the bisimulation game. Alice has to play the *start* move in one of the two programs, and Bob just replies with the *start* move in the other program. Then, as long as Alice plays the honest run of $N'(C)$ (possibly switching between the two programs), Bob patiently repeats her moves in the other program.[16] If Alice deviates from the honest run by playing one of

$$x > 0 \wedge y > 0 \xrightarrow{zero} y := y - 1; z := z + 1$$
$$x > 0 \wedge z > 0 \xrightarrow{zero} y := y + 1; z := z - 1$$

in one of the programs, Bob replies with

$$\textbf{true} \xrightarrow{zero} \textbf{skip}$$

in the other program. If Alice deviates from the honest run by playing

$$\textbf{true} \xrightarrow{zero} \textbf{skip}$$

in one of the programs at a point in which $x > 0$, Bob replies with one of

$$x > 0 \wedge y > 0 \xrightarrow{zero} y := y - 1; z := z + 1$$
$$x > 0 \wedge z > 0 \xrightarrow{zero} y := y + 1; z := z - 1$$

in the other program, depending on which guard is enabled. [17] After this move, Bob goes on playing exactly the same moves as Alice.

Let us see that Bob wins all games. If Alice sticks to the honest execution, then, since $C$ does not halt, she never plays a *halt*-move, and since all other moves can be mimicked by Bob without problems, the game never terminates: a win for Bob. So Alice's only chance to win is to deviate from the honest run at some point. Observe that just before deviating we have $y = 1, z = 0$ in $N_1(C)$ and $y = 0, z = 1$ in $N_2(C)$. We show that by deviating Alice digs her own grave: she allows Bob to reply in such a way that after his move *all* variables have exactly the same value in $N_1(C)$ and $N_2(C)$! Bob then wins easily by playing the same moves as Alice.

Alice can deviate from the honest run in three different ways. She can play

$$x > 0 \wedge y > 0 \xrightarrow{zero} y := y - 1; z := z + 1$$

to which Bob replays **true** $\xrightarrow{zero}$ **skip**, and then we have $y = 0, z = 1$ in both programs. She can also play

$$x > 0 \wedge z > 0 \xrightarrow{zero} y := y + 1; z := z - 1$$

[16] He has no choice anyway!

[17] Observe that exactly one of the two guards is enabled, because the *start* action makes the assertion $y + z = 1$ true, and the other actions keep this assertion invariant.

and after Bob's reply we have $y = 1, z = 0$ in both programs. Finally, she can play

$$\textbf{true} \xrightarrow{zero} \textbf{skip}$$

at a state in which $x > 0$, and after Bob's reply we have either $y = 0, z = 1$ or $y = 1, z = 0$ in both programs, depending on his answer.

### 9.1 Partial-order equivalences are also undecidable

As we mentioned in Section 5, the literature contains many so-called *partial-order* equivalence notions which do not fit between trace and bisimulation equivalence. So Jančar's result might seem not to apply for them. But it does. Say that two transitions $t_1$ and $t_2$ are *concurrently enabled* at a marking $M$ if $M(s) \geq F(s, t_1) + F(s, t_2)$ for every place $s$, and say that a Petri net is *sequential* if no reachable marking enables two transitions concurrently. It is easy to see that the Petri nets $N_1(C)$ and $N_2(C)$ we have constructed above are sequential. So, actually, we have just proved that any equivalence relation which fits between trace and bisimulation equivalence *for the class of sequential Petri nets* is undecidable. Partial-order equivalences turn out to fit between trace and bisimulation equivalence for sequential nets. Actually, this is what one would expect: partial-order equivalences should distinguish concurrency from interleaving, but if there is no concurrency at all then there is also nothing to distinguish.

## 10 Can anything be done in polynomial time?

The general EXSPACE-hardness bound of Section 7 raises the question if there are better results (PSPACE, NP, polynomial problems) for classes of Place/Transition Petri nets. Since a complete treatment of this question is out of the scope of this paper, we concentrate on how far can one go with polynomial algorithms. Obviously, we cannot expect to go further than for 1-safe Petri nets. So the first question is if at least some problems for conflict-free nets and free-choice nets that are not necessarily 1-safe can still be solved in polynomial time. The answer is a qualified "no". Even though [18, 39] contain some polynomial algorithms for conflict-free Petri nets, most of the important problems for these two classes become at least NP-hard. For instance, the reachability problem for conflict-free Petri nets is NP-complete [8], and the liveness problem for free-choice Petri nets is co-NP-complete (i.e., it is the complement of an NP-complete problem) [24, 5] (the proof is sketched below as the solution to Story I). Notice that the liveness and reachability problems for arbitrary Petri nets are much harder, and so these NP-completeness results can also be seen as positive results.

Is there any interesting constraint leading to polynomial algorithms for many problems? There seems to be essentially a single non-trivial one: every place has exactly one input transition and exactly one output transition ("exactly" can also be generalised to "at most") The Petri nets satisfying this constraint have been called *marked graphs*, *synchronisation graphs*, and *T-systems*. Two of the oldest papers in net theory show that many problems for these nets can be solved using simple graph algorithms or linear programming [3, 13]. So let us formulate our last rule of thumb:

Rule of thumb 10:
Many interesting problems about marked graphs are solvable in polynomial time. Almost no interesting problems about Petri net classes substantially larger than marked graphs are solvable in polynomial time.

**The solution to Story I**

The non-liveness problem for free-choice Petri nets can be formulated as follows:

Given: a free-choice Petri net $N$,
To decide: if $N$ is non-live.

Membership in NP is non-trivial; it follows from Commoner's theorem [15, 5]. NP-hardness, on the contrary, is very easy to prove by a reduction, first presented in [24], from the satisfiability problem for boolean formulas in conjunctive normal form.[18] Figure 7 shows the Petri net corresponding to the formula

$$(x_1 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3)$$

and we explain the construction on this example. Loosely speaking, the Petri net works as follows: first, the variables are nondeterministically assigned truth values by firing either the transition $x_i$ or $\overline{x}_i$ for each variable $x_i$. Once all variables have been assigned a value, a transition $C_j$ is enabled if and only if the assignment makes the clause $C_j$ false. For instance, $C_2$ is enabled if and only if the transitions $\overline{x}_1, x_2, \overline{x}_3$ have fired; this corresponds to the assignment $x_1 := \mathit{false}$, $x_2 := \mathit{true}$, $x_3 := \mathit{false}$, which is the only assignment making $C_2$ false. So we have that the place *False* gets tokens if and only if the formula is false under the assignment. If the formula is satisfiable, then there is an assignment making the formula true, and for this assignment the place *False* never gets marked. So the Petri net is not live. On the contrary, if the formula is unsatisfiable, then the place *False* can always get marked again, and the net is live.
Since the formula is satisfiable, the Petri net of Figure 7 is non-live.

## 11 Conclusions

I'd like to conclude by listing the 10 rules of thumb of the paper. You can find them in Table 11. I've allowed myself to suppress the word "interesting" from all the rules, since it should no longer lead to confusion.

## Acknowledgments

Many thanks to Eike Best, Peter Habermehl, Ernst Mayr, Richard Mayr, Peter Rossmanith, P.S. Thiagarajan, Antti Valmari and Frank Wallner for helpful suggestions, discussions, and informations. Many thanks also to Serge Haddad and Zdenek Sawa found a mistake in the version of this paper which appeared in LNCS 1491: the programs $N_1(C)$ and $N_2(C)$ given in Section 9 were incorrect. They suggested the correct programs of this version. Zdenek Sawa also found some more mistakes. The PSPACE-algorithm for CTL of Section 4 is joint work with Peter Rossmanith.

[18] It is interesting to compare this reduction with the one of Section 6.

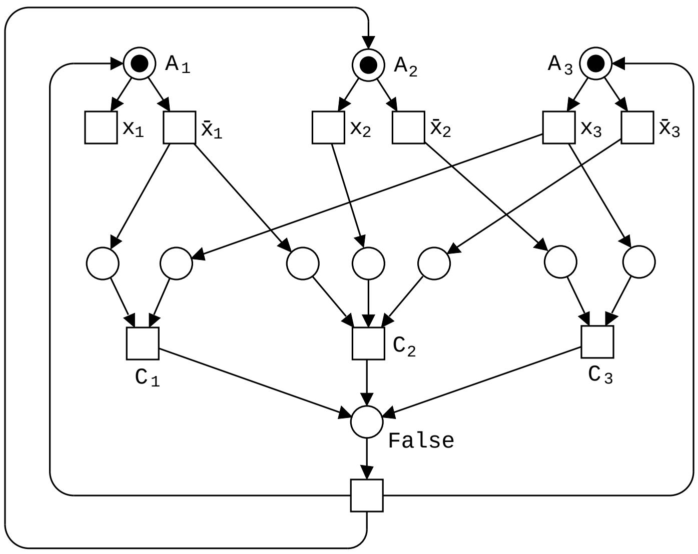


**Fig. 7.** Petri net corresponding to the formula $(x_1 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3)$

The 10 Rules of Thumb

1. All questions about the behaviour of 1-safe Petri nets are PSPACE-hard.
2. Nearly all questions about the behaviour of 1-safe Petri nets can be solved in polynomial space.
3. Equivalence problems for 1-safe Petri nets are harder to solve than model-checking problems. They need at most exponential space.
4. Most questions about the behaviour of acyclic 1-safe Petri nets are NP-hard.
5. Many questions about 1-safe conflict-free Petri nets are solvable in polynomial time.
   Some questions about live 1-safe free-choice Petri nets are solvable in polynomial time (and liveness of 1-safe free-choice Petri nets is decidable in polynomial time too).
   Almost no questions for 1-safe net classes substantially larger than free-choice Petri nets are solvable in polynomial time.
6. All questions about the behaviour of Petri nets are EXPSPACE-hard.
7. The model-checking problems for Petri nets and all state-based logics are undecidable.
8. The model-checking problems for Petri nets and all branching-time, action-based logics are undecidable.
   The model-checking problems for Petri nets and all linear-time, action-based logics are decidable.
9. All equivalence problems for Petri nets are undecidable.
10. Many questions about marked graphs are solvable in polynomial time.
    Almost no questions about Petri net classes substantially larger than marked graphs are solvable in polynomial time.

**Table 1.**

## References


1. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *Monographs in theoretical Computer Science*. Springer-Verlag, 1988.
2. A. Cheng, J. Esparza, and J. Palsberg. Complexity Results for 1-safe Nets. *Theoretical Computer Science*, 147:117–136, 1995.
3. F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked Directed Graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.
4. M. Dam. Fixpoints of Büchi automata. In *Proceedings of the 12th International Conference of Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 39–50, 1992,
   Also: LFCS Report, ECS-LFCS-92-224, University of Edinburgh.
5. J. Desel and J. Esparza. *Free-choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.
6. E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science Volume B*, pages 995–1027, 1990.
7. J. Esparza. Model Checking Using Net Unfoldings. *Science of Computer Programming*, 23:151–195, 1994.
8. J. Esparza. Reachability in Live and Safe Free-Choice Petri Nets is NP-Complete. Technical Report SFB-Bericht Nr. 342/12/96 A, Technische Universität München, 1996. To appear in Theoretical Computer Science.
9. J. Esparza. Decidability of Model-Checking for Infinite-State Concurrent Systems. *Acta Informatica*, 34:85–107, 1997.
10. J. Esparza and M. Nielsen. Decidability Issues for Petri Nets – a Survey. In *Bulletin of the EATCS*, volume 52, pages 245–262, 1994
    Also: Journal of Information Processing and Cybernetics 30(3):143–160, 1995.
11. Formal methods page of the WWW Virtual Library at http://www.comlab.ox.ac.uk/archive/formal-methods.html#notations.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1979.
13. H. J. Genrich and K. Lautenbach. Synchronisationsgraphen. *Acta Informatica*, 2:143–161, 1973.
14. P. Habermehl. On the Complexity of the Linear-Time Mu-Calculus for Petri Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, 1997.
15. M. H. T. Hack. *Analysis of Production Schemata by Petri Nets*. M.s. thesis, Cambridge, Mass.: MIT, Dept. Electronical Engineering, 1972.
16. J. E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
17. R. R. Howell and L. Rosier. On Questions of Fairness and Temporal Logic for Conflict-free Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 340 of *Lecture Notes in Computer Science*, pages 200–220, 1988.
18. R. R. Howell and L. E. Rosier. An $O(n^{1.5})$ Algorithm to Decide Boundedness for Conflict-free Vector Replacement Systems. *Information Processing Letters*, 25(1):27–33, 1987.
19. R. R. Howell and L. E. Rosier. Problems Concerning Fairness and Temporal Logic for Conflict-free Petri Nets. *Theoretical Computer Science*, 64:305–329, 1989.
20. R. R. Howell, L. E. Rosier, and H. Yen. A Taxonomy of Fairness and Temporal Logic Problems for Petri Nets. *Theoretical Computer Science*, 82:341–372, 1991.
21. P. Jančar. All Action-based Behavioural Equivalences are Undecidable for Labelled Petri Nets. *Bulletin of EATCS*, 56:86–88, 1995.

22. P. Jančar. Undecidability of Bisimilarity for Petri Nets and Some Related Problems. *Theoretical Computer Science*, 148:281–301, 1995.
23. L. Jategaonkar and A. Meyer. Deciding True Concurrency Equivalences on Safe, Finite Nets. *Theoretical Computer Science*, 154(1):107–143, 1996.
24. N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of Some Problems in Petri Nets. *Theoretical Computer Science*, 4:277–299, 1977.
25. S.R. Kosaraju. Decidability of Reachability in Vector Addition Systems. In *14th Annual ACM Symposium on Theory of Computing*, pages 267–281, San Francisco, 1982.
26. L. Lamport. The Mutual Exclusion Problem. Part II – Statement and Solutions. *Journal of the ACM*, 33(2), 1986.
27. R. Lipton. The Reachability Problem Requires Exponential Space. Technical Report 62, Yale University, 1976.
28. E. W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal on Computing*, 13:441–460, 1984.
29. E.W. Mayr and A.R. Meyer. The Complexity of the Word Problems for Commutative Semigroups and Polynomial Ideals. *Advances in Mathematics*, 46:305–329, 1982.
30. WWW page on Petri net tools at http://www.daimi.aau.dk/petrinets/tools/.
31. A. Rabinovich. Complexity of Equivalence Problems for Concurrent Systems of Finite Agents. *Information and Computation*, 127(2):164–185, 1997.
32. C. Rackoff. The Covering and Boundedness Problem for Vector Addition Systems. *Theoretical Computer Science*, 6:223–231, 1978.
33. P. Starke. *Analyse von Petri-Netz-Modellen*. Teubner, 1990.
34. C. Stirling. Bisimulation, Model Checking and Other Games. Notes for Mathfit instructional meeting on games and computation, Edinburgh, June 1977. Available at http://www.dcs.ed.ac.uk/home/cps/.
35. A. Valmari. *State Space Generation: Efficiency and Practicality*. Phd thesis, Tampere University of Technology, 1988.
36. R. J. van Glabbeek. The Linear Time – Branching Time Spectrum. In *Proceedings of CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, 1990.
37. M. Vardi. An Automata-Theoretic Approach to Linear temporal Logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–265, 1996.
38. H. C. Yen. A Unified Approach for Deciding the Existence of Certain Petri Nets Paths. *Information and Computation*, 96(1):119–137, 1992.
39. H. C. Yen. A Polynomial Time Algorithm to Decide Pairwise Concurrency of Transitions for 1-Bounded Conflict Free Petri Nets. *Information Processing Letters*, 38:71–76, 1991.