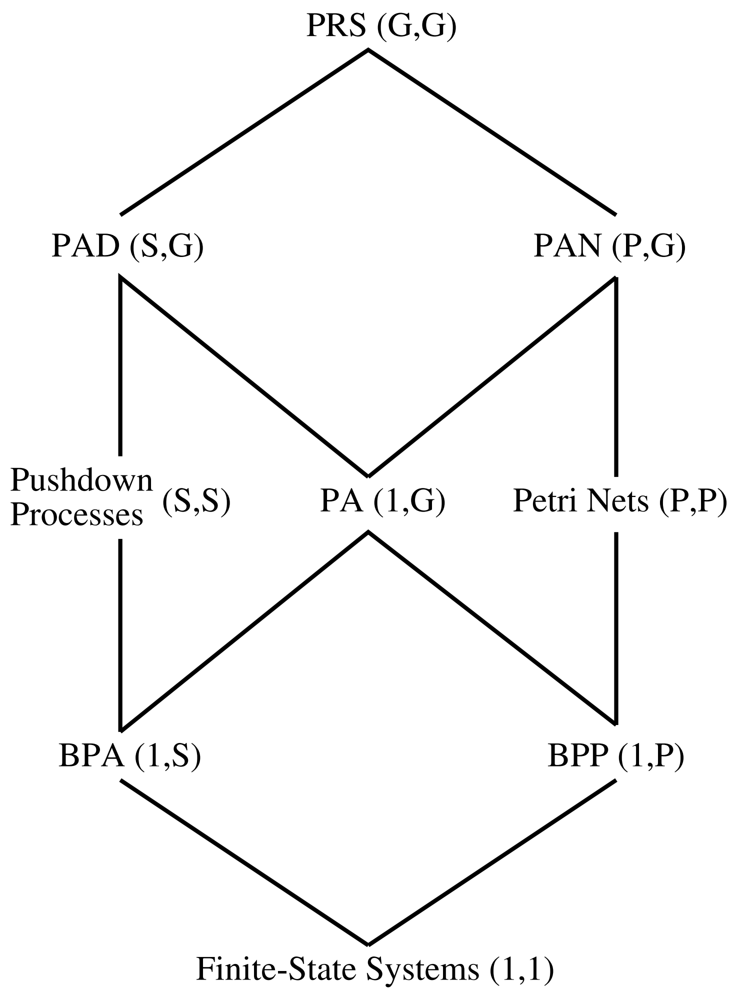
Decidability and Complexity of Model Checking Problems for Infinite-State Systems

Richard Mayr



Lehrstuhl für Theoretische Informatik und Grundlagen der KI
Institut für Informatik
der Technischen Universität München

Decidability and Complexity of Model Checking Problems for Infinite-State Systems

Richard Mayr

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. T. Nipkow, Ph.D.
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. J. Esparza
	2. Univ.-Prof. Dr. E. Mayr

Die Dissertation wurde am 22.12.1997 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.4.1998 angenommen.

Abstract

There are many different formal models for systems with infinite state spaces. Among them are context-free processes, Basic Parallel Processes, PA-processes, pushdown processes and Petri nets. They are used to build abstract models of programs and to verify their correctness with respect to a formal specification.

We present a unified view of all these models by describing them as subclasses of a very general term rewriting formalism, that we call *Process Rewrite Systems* (PRS). We define the *PRS-hierarchy* of these subclasses and show that it is strict with respect to bisimulation equivalence. Similar hierarchies of rewrite systems have already been defined by Stirling, Caucal and Moller, but only for systems with either only sequential composition or only parallel composition. The PRS-hierarchy subsumes these hierarchies and additionally contains systems that use both sequential and parallel composition. This unified view also yields natural generalizations of several models, i.e. a common generalization of Petri nets and pushdown processes.

We use temporal logics to specify properties of systems. We consider standard temporal logics like computation tree logic (CTL), the modal μ-calculus, linear-time temporal logic (LTL), the linear-time μ-calculus and several fragments of CTL. The model checking problem is, if a system described in a formal model satisfies a property encoded as a formula in a temporal logic. We study the decidability and the computational complexity of model checking problems for these temporal logics and the models in the PRS-hierarchy.

We prove several new results about the decidability of model checking problems. These results close the last gaps and so we can present a complete picture of the decidability of the model checking problem for all these logics and all models in the PRS-hierarchy.

We also solve some open problems about the computational complexity of model checking. These results can be combined with the results of other authors, and so we get an almost complete picture of the complexity of the model checking problem for all these logics and all models in the PRS-hierarchy. The only major open problems that remain are the exact complexity of the reachability problem for Petri nets and the complexity of model checking finite-state systems with the full modal μ-calculus.

Furthermore, we present a tableau system that solves the model checking problem for Petri nets and the interpretation of the linear-time μ-calculus on infinite runs. Tableau methods are particularly useful for semiautomatic verification.

Finally, we study several models in the PRS-hierarchy that arise as natural common generalizations of mutually incomparable models like Petri nets, pushdown processes and PA-processes. Some of these models (called PAN and PRS) are strictly more expressive that Petri nets. They can be interpreted as an extension of Petri nets with subroutines, since they can describe both parallelism and recursion. We show that the reachability problem is decidable for PAN and PRS. Thus they are more general than Petri nets, but not Turing-powerful.

Contents

Chapter 1

Introduction

An important problem in software engineering is to ensure the correctness of programs. Correctness means that the program fulfills the task for which it was designed. The correctness is defined with respect to a formal specification that characterizes the desired behavior of the program in an abstract way. The process of checking whether an implementation (a program) satisfies the requirements described by the specification is called *verification*.

The first and still the most common method of verification is testing. In testing, one observes the behavior of the program in different situations with different inputs and checks if it matches the specification. A difference between them indicates an error, which can then be localized and (hopefully) corrected. For the majority of programs nowadays, testing is the only verification technique applied.

The problem is, that it is incomplete. By testing one might find errors, but (except in very special and rare cases) one never gets a guarantee that a program is correct. Even if extensive testing reveals no errors, this doesn't mean that there are none. This is due to the fact that most programs have an extremely large state space. This means that there is a large number of different reachable states in each of which the program can behave differently. If this state space is finite, it would be theoretically possible to test the behavior of the program in every reachable state. In practice however, this often cannot be done, since even very small programs can well have a state space that is larger than the number of atoms in the universe. Thus it is impossible to test every situation that can possibly occur in the execution of such programs. So, except for programs with very small state spaces, testing can only reduce the probability of errors, but it can never give certainty.

Often the problem is even harder, because many programs have infinite state spaces. This can be due to the use of real numbers, arbitrarily large data structures, unbounded buffers or stacks, or the possibility to create arbitrarily many child-processes. In these cases complete testing is even theoretically impossible.

With parallel computer architectures and concurrent systems, the problem gets even worse. When several components of a program run in parallel the number of different possible execution sequences increases exponentially, even for finite-state systems. This problem is known as the *state explosion problem*. It is due to the fact that there are so many ways the components can influence each other, and many cases that depend on which event happens first in which component. For example if m copies of a process with n states run in parallel, then the whole system has n^m states.

Also concurrent systems more often have infinite state spaces, because the creation of arbitrarily many new parallel processes can be possible. Due to these additional problems, testing becomes even more unreliable for concurrent systems.

On the other hand, verification is even more important for concurrent systems, because, as experience has shown, it is more difficult for programmers to write error-free code for parallel programs and mistakes occur more often than in sequential systems. Therefore better verification techniques for both finite-state and infinite-state concurrent systems are needed. The approaches to develop such techniques can roughly be divided into two categories: *theorem provers* (for example [ORSv95, Pau94]) and *model checkers* (for example [Hol91, CGL94]).

With theorem provers one attempts to find a formal proof that the implementation satisfies some property expressed in the specification. This is essentially a semiautomatic technique where the theorem proving system and the user interact in the search for a proof. Normally, the user has to guide the system and do the crucial steps of the proof, while the theorem prover can do some simple subproblems automatically. While this technique is very powerful in principle, its applicability in practice is hindered by the fact that it is often cumbersome and requires a lot of knowledge, skill and practice from the user. Still it has been used successfully for the verification of safety-critical systems.

Model checkers are another approach to solve the verification problem. No full specification languages are used in model checking, but properties of systems are described in temporal logics (see Chapter 3). An instance of the *model checking problem* is then given by a description of a system (a program) and a temporal logic formula. The question is, if the system satisfies the property described by the formula. Unlike theorem provers, model checkers operate completely automatically and are thus very easy to use. Model checking has been used with considerable success in the verification of finite-state systems. To conquer the state explosion problem, techniques have been developed to reduce the size of the state space by finding equivalence classes of states that have the same (or a very similar) behavior, as far the verification is concerned. Verification can then be done by examining every equivalence class.

The problem with model checking is that such automatic techniques cannot be applied to arbitrary programs with infinite state spaces, because these are Turing-powerful and even simple questions like "Is a certain state reachable?" are undecidable.

The idea to extend model checking techniques to infinite-state systems was motivated by formal language theory [HU79]. In formal language theory infinite languages are finitely described and some problems about them (for example equivalence of regular languages) are decidable. So not every problem about infinite systems is undecidable. In analogy to formal language theory new formalisms to describe infinite-state systems were introduced.

A classical example for this are pushdown automata. In formal language theory they are used to describe Chomsky-2 languages, but they can also be seen as models for infinite-state systems. Every state of the finite control together with the content of the stack describes a state. Since the stack is unbounded, there can be infinitely many different states. The state changes when the automaton accepts a terminal symbol. However, this can also be interpreted as performing an action and changing the state by it.

Other models, like Petri nets [Pet81], can also describe concurrent systems. These *formal models* can describe the important aspects of the behaviors of programs by using a formalism that is less powerful than Turing-machines (or programming languages) and therefore easier to analyze. Being not Turing-powerful, these models cannot fully describe every aspect of the behavior of a program. However, they can often describe the aspects of the programs' behaviors that are important for the verification. Formal models are used for verification, because they are normally smaller and more easily handled than full programs. As they are not Turing-powerful some verification problems are decidable for them. Formal models should be simple enough to allow automated verification, or at least computer-assisted verification. On the other hand they should be as expressive as possible, so that most aspects of real programs can be modeled.

Petri nets and process algebras are two popular kinds of formalisms used to build abstract models of concurrent systems. In this thesis we present a unified view of Petri nets and several simple process algebras by representing them as subclasses of a general rewriting formalism. We call this formalism *Process Rewrite Systems* (PRS). We also define a hierarchy of subclasses of this rewriting formalism. Such hierarchies have already been defined by Stirling, Caucal and Moller [Cau92, Mol96], but only for either purely sequential- or purely parallel systems. Here we generalize this to systems with both sequential and parallel composition. We call our hierarchy of rewrite systems the *PRS-hierarchy*.

We study the decidability and computational complexity of model checking problems for the models in the PRS-hierarchy and most standard temporal logics [MB96, Eme94, Bra92]. These logics include for example linear-time temporal logic (LTL), the linear-time μ-calculus, the modal μ-calculus, computation-tree logic (CTL) and several natural fragments of CTL. The aims of this thesis are the following:

1. The primary aim is to establish the decidability and computational complexity of all model checking problems for the models in the PRS-hierarchy.

   For the decidability we have achieved this and so we can present a complete picture of the decidability of model checking. For the complexity we solve several problems s.t. only few open problems remain. So we can give an almost complete picture of the complexity of model checking.

   The main new results are in three different areas:

   - The decidability and complexity of model checking with the branching-time temporal logic EF. EF is a simple but very natural fragment of computation-tree logic (CTL). We show that for almost all formal models, model checking with EF has a significantly lower complexity than model checking with any other branching-time logic.
   - We also show strict lower bounds for the model checking problems for linear-time temporal logic (LTL) and some simple process algebras like Basic Parallel Processes or context-free processes. These problems were already known to be decidable, but only much weaker lower bounds were known.
   - The unified view of formal models yields natural generalizations of Petri nets by subroutines and recursion. We show that these new models are strictly more expressive than Petri nets, but not Turing-powerful.

2. The secondary aim is to develop methods that make verification of infinite-state systems more feasible in practice.

   Since the complexity of model checking infinite-state systems is often very high, automatic methods cannot always be applied. Basically, there are two ways to counter this problem:

   - In practice it is not always necessary to use the full power of a temporal logic for the verification. Therefore it is important to find efficient algorithms for simple verification problems. In Section 8.3 we present polynomial algorithms for some simple verification problems in process algebras.

- In some cases automatic methods cannot be applied, because the model checking problem is undecidable or the complexity is too high. In these cases semiautomatic methods (i.e. theorem provers with human interaction) can be very useful. Tableau systems provide a theoretical basis for these semiautomatic methods. We develop such a tableau system in Chapter 9. This tableau system solves the model checking problem for Petri nets and the interpretation of the linear-time μ-calculus on infinite runs. Tableau systems are particularly useful for semiautomatic verification, because they give the user a better intuition and complete control over the verification process.

In order to give a complete picture of the decidability and complexity of model checking problems for infinite-state systems, many results by other authors are cited. Therefore some parts of this thesis look like a survey. However, results by other authors are only cited, but not proved. So every proof in this thesis is due to the author. Some chapters of this thesis are very short, since they only contain citations and no proofs.

We assume that the reader is familiar with formal languages [HU79] and the basics of complexity theory, like Turing-machines, counter machines, reductions, oracles and the complexity classes $\mathcal{P}$, $\mathcal{NP}$, the polynomial time hierarchy, *PSPACE*, *EXPTIME* and *EXPSPACE* [vL90]. The reader should also be familiar with Petri nets [Pet81]. Knowledge of temporal logics [Bra92, MB96, Eme94] and CCS [Mil89] is helpful, but not necessary.

Chapter 2 presents a unified view of many common- and several new formal models for infinite-state systems. Chapter 3 defines the temporal logics that are used to specify properties of systems. In Chapter 4 we give a brief introduction to tableau systems, an important method for model checking infinite-state systems. In Chapters 5 – 10 we study the decidability and complexity of model checking problems for the various process models. Chapter 11 summarizes the results and in Chapter 12 we draw some general conclusions.

Chapter 2

Formal Models

In this chapter we introduce the formal models used to describe infinite-state systems. We present a unified view of many widely known models like Basic Parallel Processes (BPP) [Chr93], context-free processes (BPA), pushdown processes, PA, Petri nets and others. We show that all these models can be seen as subclasses of a general term rewriting formalism.

Such unified representations have already been used by Stirling, Caucal and Moller [Cau92, Mol96], but only for either purely sequential- or purely parallel systems. Here we generalize this to systems with both sequential and parallel composition.

In Section 2.1 we present this formalism and define a hierarchy of subclasses with respect to their expressiveness. In Section 2.2 we explain the intuition behind the definition of the various subclasses. In Section 2.3 we show that popular models like Petri nets and process algebras correspond to certain subclasses in this hierarchy and in Section 2.4 we give an example. In Section 2.5 we show that this hierarchy is strict with respect to bisimulation equivalence.

2.1 The PRS-Hierarchy

Programs and their possible execution sequences can be formally described by *labeled transition systems (LTS)*.

Definition 2.1.1 (LTS)
A labeled transition system is a (possibly infinite) directed graph, whose nodes represent states and whose arcs are labeled with atomic actions from a predefined

set $Act = \{a, b, c, \ldots\}$. One special state is called the *initial state*. It is often denoted by s_0.

An arc leading from a node s_1 to a node s_2 that is labeled with an action a means that if the system is in state s_1, then it can do action a and will be in state s_2 afterwards. This is denoted by $s_1 \xrightarrow{a} s_2$.

Figure 2.1 shows an example of a LTS.

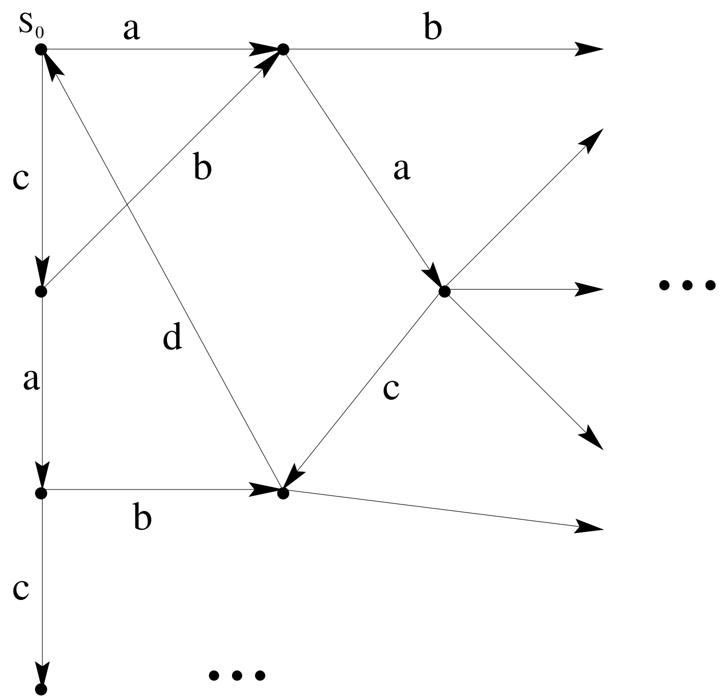


Figure 2.1: A labeled transition system

If a LTS is finite then it can be finitely described. However, as most programs have infinite state spaces, they yield infinite transition systems. Formal models like Petri nets, pushdown automata and process algebras are ways of finitely describing certain classes of infinite transition systems.

We present a unified view of many common formal models by showing that every single one can be seen as a special subclass of rewrite systems. Basically, the rewriting formalism is first-order prefix-rewrite systems on process terms without substitution and modulo commutativity and associativity of parallel composition and modulo associativity of sequential composition. The most general class of these systems is called *Process Rewrite Systems (PRS)*. In the following we describe this formalism.

Many classes of concurrent systems can be described by a (possibly infinite) set of process terms, representing the states, and a finite set of rewrite rules describing the dynamics of the system.

Definition 2.1.2 Let $Act = \{a, b, \dots\}$ be a countably infinite set of atomic actions and $Var = \{X, Y, Z, \dots\}$ a countably infinite set of process variables. The process terms that describe the states of the system have the following form:

$$P ::= \epsilon \mid X \mid P_1.P_2 \mid P_1 \| P_2$$

where ϵ is the empty term, X is a process variable (used as an atomic process in this context), "$\|$" means parallel composition and "." means sequential composition. Parallel composition is associative and commutative. Sequential composition is associative. Let $\mathcal{T}$ be the set of process terms.

Convention 1: We always work with equivalence classes of terms modulo commutativity and associativity of parallel composition and modulo associativity of sequential composition. Also we define that $\epsilon.P = P = P.\epsilon$ and $P \| \epsilon = P$.

Convention 2: We defined that sequential composition is associative. However, when we look at terms we think of it as left-associative. So when we say that a term t has the form $t_1.t_2$, then we mean that t_2 is either a single variable or a parallel composition of process terms.

The *size* of a process term is defined as the number of variables in it.

$$\begin{aligned}
size(\epsilon) &:= 0 \\
size(X) &:= 1 \\
size(P_1.P_2) &:= size(P_1) + size(P_2) \\
size(P_1 \| P_2) &:= size(P_1) + size(P_2)
\end{aligned}$$

For a term t the set $Var(t)$ is the set of variables that occur in t.

$$\begin{aligned}
Var(\epsilon) &:= \{\} \\
Var(X) &:= \{X\} \\
Var(P_1.P_2) &:= Var(P_1) \cup Var(P_2) \\
Var(P_1 \| P_2) &:= Var(P_1) \cup Var(P_2)
\end{aligned}$$

The dynamics of the system is described by a finite set of rules Δ of the form $(t_1 \xrightarrow{a} t_2)$ where t_1 and t_2 are process terms and $a \in Act$ is an atomic action. The finite set of rules Δ induces a (possibly infinite) labeled transition system with relations $\xrightarrow{a}$ with $a \in Act$. For every $a \in Act$, the transition relation $\xrightarrow{a}$ is the smallest relation that satisfies the following inference rules.

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Delta}{t_1 \xrightarrow{a} t_2} \quad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \| t_2 \xrightarrow{a} t_1' \| t_2} \quad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \| t_2 \xrightarrow{a} t_1 \| t_2'} \quad \frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2}$$

where t_1, t_2, t_1', t_2' are process terms.

Since Δ is finite, the generated LTS is finitely branching [1]. Also every single Δ uses only a finite subset

$$Var(\Delta) := \bigcup_{(t_1 \xrightarrow{a} t_2) \in \Delta} (Var(t_1) \cup Var(t_2))$$

of variables and only a finite subset

$$Act(\Delta) := \bigcup_{(t_1 \xrightarrow{a} t_2) \in \Delta} \{a\}$$

of atomic actions. Thus for every Δ only finitely many of the generated transition relations $\xrightarrow{a_i}$ for $a_i \in Act$ are nonempty. (Those for which $a_i \in Act(\Delta)$). Still the generated transition system can be infinite. (Consider the analogy: Every labeled Petri net has only finitely many transitions and uses only finitely many different atomic actions, but the state space can be infinite.)

The relation $\xrightarrow{a}$ is generalized to sequences of actions in the standard way. Sequences are denoted by σ. Normally, σ stands for a sequence of actions, but sometimes we also consider sequences of applications of rules. In these cases σ stands for a sequence of rules and not only for the actions associated to these rules. (Note that different rules can be marked with the same action.)

Without restriction we can assume that the initial state of a system is described by a term consisting of a single variable.

Remark 2.1.3 *There is no operator "+" for nondeterministic choice in the process terms, because this is encoded in the set of rules Δ! There can be several rules with the same term on the left hand side. It is also possible that several rules are applicable at different places in a term. The rule that is applied and the position where it is applied are chosen nondeterministically.*

Also there is no such thing as action prefixes in the process terms. The atomic actions are introduced by the rules.

[1] For some classes of systems (e.g. Petri nets) the branching-degree is bounded by a constant that depends on Δ. For other classes (e.g. PA) the branching-degree is finite at every state, but it can get arbitrarily high.

Many common models of systems fit into this scheme. In the following we characterize subclasses of rewrite systems. The expressiveness of a class depends on what kind of terms are allowed on the left hand side and right hand side of the rewrite rules in Δ.

Definition 2.1.4 (Classes of process terms)
We distinguish four classes of process terms:

1 Terms consisting of a single process variable like X.

S Terms consisting of a single variable or a sequential composition of process variables like $X.Y.Z$.

P Terms consisting of a single variable or a parallel composition of process variables like $X\|Y\|Z$.

G General process terms with arbitrary sequential and parallel composition like $(X.(Y\|Z))\|W$.

Also let $\epsilon \in S, P, G$, but $\epsilon \notin 1$. It is easy to see the relations between these classes of process terms: $1 \subset S$, $1 \subset P$, $S \subset G$ and $P \subset G$. S and P are incomparable and $S \cap P = 1 \cup \{\epsilon\}$.

We characterize classes of process rewrite systems (PRS) by the classes of terms allowed on the left hand sides and the right hand sides of rewrite rules.

Definition 2.1.5 (PRS)
Let $\alpha, \beta \in \{1, S, P, G\}$. A (α, β)-PRS is a finite set of rules Δ where for every rewrite rule $(l \xrightarrow{a} r) \in \Delta$ the term l is in the class α and $l \neq \epsilon$ and the term r is in the class β (and can be ϵ). The initial state is given as a term $t_0 \in \alpha$. A (G, G)-PRS is simply called PRS.

Remark 2.1.6 *W.l.o.g. it can be assumed that the initial state t_0 of a PRS is a single variable. There are only finitely many terms $t_1, \ldots, t_n$ s.t. $t_0 \xrightarrow{a_i} t_i$. If t_0 is not a single variable then we can achieve this by introducing a new variable X_0 and new rules $X_0 \xrightarrow{a_i} t_i$ and declaring X_0 to be the initial state.*

(α, β)-PRS where α is more general than β or incomparable to β (for example $\alpha = G$ and $\beta = S$) do not make any sense. This is because the terms that are introduced by the right side of rules must later be matched by the left sides of other rules. So in a (G, S)-PRS the rules that contain parallel composition on the left hand side will never be used (assuming that the initial state is a single variable). Thus one may as well use a (S, S)-PRS. So we restrict our attention to (α, β)-PRS with $\alpha \subseteq \beta$.

Many of these (α, β)-PRS correspond to widely known models like Petri nets, pushdown processes, context-free processes and others. In Section 2.3 this will be explained in detail. Figure 2.2 shows a graphical description of the hierarchy of (α, β)-PRS models. This figure contains all (α, β)-PRS with $\alpha \subseteq \beta$. We give both the common name and the specific (α, β) for every model. A line from a lower model to a higher one means that the higher one is more general than the lower one.

2.2 The Intuition

In Section 2.3 we look at each (α, β)-PRS in detail and examples are given.

However, in this section we explain the general intuition behind the definition of (α, β)-PRS. What does it mean that parallel/sequential/arbitrary composition is allowed in terms on the left/right hand sides of rules?

If parallel composition is allowed on the right hand side of rules, then there can be rules of the form $t \xrightarrow{a} t_1 \| t_2$. This means that it is possible to create processes that run in parallel. The rule can be interpreted that, by action a, the process t becomes the process t_1 and spawns off the process t_2 or vice versa.

If sequential composition is allowed on the right hand side of rules, then there are rules of the form $t \xrightarrow{a} t_1.t_2$. The interpretation is that process t calls a subroutine t_1 and becomes process t_2. It resumes its execution when the subroutine t_1 terminates.

If arbitrary sequential and parallel composition is allowed on the right hand side of rules then both parallelism and subroutines are possible.

If parallel composition is allowed on the left hand side of rules, then there are rules of the form $t_1 \| t_2 \xrightarrow{a} t$. This can be interpreted as synchronization/communication of the parallel processes t_1 and t_2. This is because this action can only occur if both t_1 and t_2 change in a certain defined way.

If sequential composition is allowed on the left hand side of rules, then there can be rules of the form $t_1'.t_2 \xrightarrow{a} t'$ and $t_1''.t_2 \xrightarrow{a} t''$. The intuition is that a process t called a subroutine t_1 and became process t_2 by a rule $t \xrightarrow{a} t_1.t_2$. The subroutine may in its computation reach a state t_1' or t_1''. Now one of these rules is applicable. This means that the result of the computation of the subroutine affects the behavior of the caller when it becomes active again, since the caller

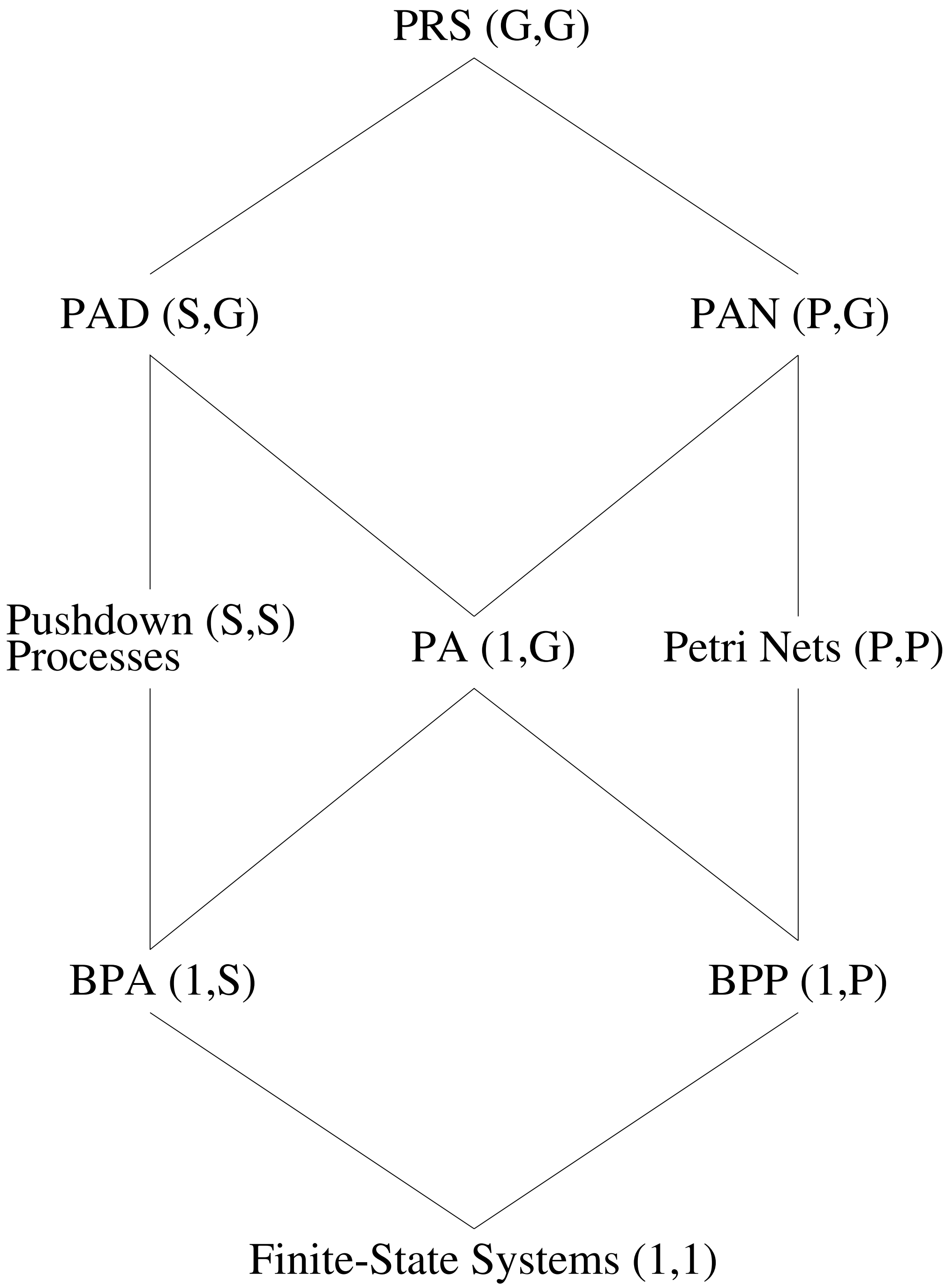


Figure 2.2: The PRS-hierarchy.

can become t' or t''. The interpretation is that the subroutine returns a value to the caller when it terminates. (See the example in Section 2.4.)

If arbitrary sequential and parallel composition is allowed on the left hand side of rules then both synchronization and returning of values by subroutines are possible.

2.3 The Models in Detail

(α, β)-PRS were introduced to provide a unified view of many models of infinite-state systems. Most of these models have been used for a long time and are known under other names (e.g. Petri nets, pushdown processes, Basic Parallel Processes, …). Here we look at these models in detail. We start at the bottom of the hierarchy and proceed upwards.

2.3.1 $(1,1)$-PRS = Finite-State Systems

Every PRS has only a finite number of rules, so it uses only finitely many different variables. In a $(1,1)$-PRS all rewrite rules only have single variables on both sides. So all reachable states are single variables and the state space is finite, since it is bounded by $|Var(\Delta)| \leq 2|\Delta|$. Every variable in $Var(\Delta)$ corresponds to a state and every rule in Δ corresponds to an arc in the LTS. In the same way every finite LTS can be represented by a $(1,1)$-PRS. So there is a one-to-one correspondence between finite LTS and $(1,1)$-PRS.

Chapter 5 shows some results about model checking finite-state systems.

2.3.2 $(1,P)$-PRS = Basic Parallel Processes (BPP)

In a $(1,P)$-PRS all rules have single variables on the left hand side and arbitrary parallel composition on the right hand side. $(1,P)$-PRS are equivalent to "Basic Parallel Processes" (BPP), a simple process algebra that was introduced by Christensen [Chr93].

The choice of this name is unfortunate, since the complexity class $\mathcal{BPP}$ of the problems that can be solved in polynomial time with high probability [vL90], was defined much earlier. However, the name BPP for this process algebra has become very common in the concurrency community. The complexity class $\mathcal{BPP}$ does not play a role here, and thus in this thesis BPP always means "Basic Parallel Processes".

There are several different representations of BPP: As a subclass of CCS, as $(1, P)$-PRS and as communication-free nets (a subclass of Petri nets). In the following pages we show that all these representations are equivalent.

Originally (in [Chr93]) BPP where defined as a natural subclass of Milner's Calculus of Communicating Systems (CCS) [Mil89]; they are CCS without restriction and relabeling and without the rule for communication. Of course CCS is a much more expressive model since it is Turing-powerful [Mil89] (while BPP is not).

Assume a countably infinite set of atomic actions $Act = \{a, b, c, \ldots\}$ and a countably infinite set of process variables $Var = \{X, Y, Z, \ldots\}$. The class of BPP expressions is defined by the following abstract syntax [Chr93, CHM93a]:

$$E \stackrel{\text{def.}}{=} \epsilon \mid X \mid aE \mid E_1 + E_2 \mid E_1 \| E_2$$

where ϵ is the empty process, X is a process variable, aE is an action prefix (meaning: first do action a and then process E), "+" is nondeterministic choice and "$\|$" is parallel composition.

A BPP is defined by a finite family of recursive equations $\{X_i := E_i \mid 1 \leq i \leq n\}$ where the X_i are distinct and the E_i are BPP expressions at most containing the variables $\{X_1, \ldots, X_n\}$. It is assumed that every variable occurrence in the E_i is *guarded*, i.e. appears within the scope of an action prefix. The variable X_1 is singled out as the *leading variable* and $X_1 := E_1$ is called the *leading equation*. Any finite family of BPP equations determines a labeled transition system. For every $a \in Act$ the transition relation $\xrightarrow{a}$ is the least relation satisfying the following inference rules:

$$aE \xrightarrow{a} E \qquad \frac{E \xrightarrow{a} E'}{E+F \xrightarrow{a} E'} \qquad \frac{F \xrightarrow{a} F'}{E+F \xrightarrow{a} F'}$$

$$\frac{E \xrightarrow{a} E'}{E\|F \xrightarrow{a} E'\|F} \qquad \frac{F \xrightarrow{a} F'}{E\|F \xrightarrow{a} E\|F'} \qquad \frac{E \xrightarrow{a} E'}{X \xrightarrow{a} E'}(X{:=}E)$$

BPP processes generate finitely branching transition graphs, i.e. $\{F \mid E \xrightarrow{a} F\}$ is finite for each E and a. This would not be true if unguarded expressions were allowed. For example, the process $X := a + a\|X$ generates an infinitely branching transition graph. Let P_i, $i \in \mathbb{N}$ be terms consisting of an arbitrary parallel composition of variables. A BPP is in *normal form*, if every expression E_i at the right hand side of an equation is of the form $a_1P_1 + \cdots + a_nP_n$. It is shown in [Chr93] that every BPP is semantically equivalent (strongly bisimilar) to a BPP in normal form.

Any BPP in normal form can be represented by a $(1, P)$-PRS: For every recursive equation $X := a_1 P_1 + \cdots + a_n P_n$ introduce n new rules $X \xrightarrow{a_1} P_1, \ldots, X \xrightarrow{a_n} P_n$. The reverse translation is analogous.

Remark 2.3.1 *Note that in PRS-rules there is no operator for nondeterministic choice anymore. The nondeterminism has been encoded in the set of rules in Δ. Nondeterministic choice now occurs when rules are applied, because more than one rule may be applicable at a time.*

Another view of BPP (and $(1, P)$-PRS) is to see them as a special class of Petri nets.

Definition 2.3.2 (Petri nets)
A *labeled Petri net* N is described by a fourtuple (S, T, W, l) where S is a finite set of places, T is a finite set of transitions, $W : (S \times T) \cup (T \times S) \to \mathbb{N}$ is a weight function and $l : T \to Act$ is a function that assigns actions to the transitions.

A marking of N is a mapping $M : S \to \mathbb{N}$. A marking M enables a transition t if $M(s) \geq W(s, t)$ for every place s. If t is enabled at M then it can occur. If this happens, then the action $l(t)$ occurs and the resulting successor marking is M', which is defined for every place s by

$$M'(s) := M(s) + W(t, s) - W(s, t)$$

See [Pet81] for more details about Petri nets.

Definition 2.3.3 A *communication-free net* is a labeled Petri net where every transition has exactly one place in its preset. Formally, this is defined by

$$\forall t \in T.\ (\exists_1 s \in S.\ W(s, t) = 1 \wedge \forall s' \neq s.\, W(s', t) = 0)$$

A $(1, P)$-PRS can be translated into a communication-free net and vice versa.

The translation of a $(1, P)$-PRS Δ into a communication-free net goes as follows: We work with equivalence classes of terms modulo commutativity and associativity of parallel composition, so the order of variables in a term doesn't matter. Every term stands for a marking of the net and every variable in $Var(\Delta)$ stands for a place in the net. The number of occurrences of a variable in the term corresponds to the number of tokens on this place. Every rule in Δ corresponds to a transition in the net. For a rule $X \xrightarrow{a} Y_1^{m_1} \cdots Y_n^{m_n}$ introduce a transition t labeled by a, an arc labeled by 1 leading from place X to the transition t and arcs labeled by m_i leading from t to places Y_i. It is important to note that in these nets every transition has exactly one input place with an arc labeled by 1. The reverse translation is analogous.

We now give an example of a BPP and describe it in the three different formalisms: first as a system of recursive equations, then as a $(1, P)$-PRS and finally as a communication-free net.

The system is defined by the following recursive equations:

$$\begin{aligned} X &:= a.(Y \| Y) + b.(Y \| Z) \\ Y &:= c + d.X \\ Z &:= a.(X \| Y) \end{aligned}$$

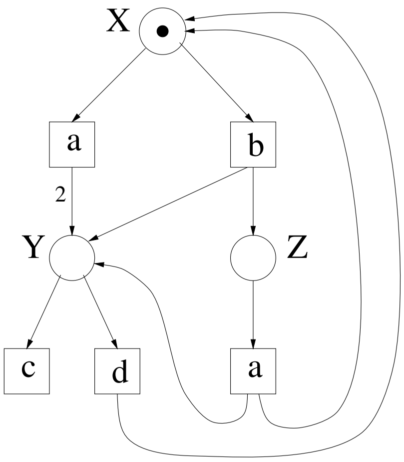
The initial state (the leading variable) is X.

An equivalent representation can be given as a $(1, P)$-PRS.

$$\begin{aligned} X &\xrightarrow{a} Y \| Y \\ X &\xrightarrow{b} Y \| Z \\ Y &\xrightarrow{c} \epsilon \\ Y &\xrightarrow{d} X \\ Z &\xrightarrow{a} X \| Y \end{aligned}$$

The initial state is X.

Finally, this system can also be described by a communication-free net. The variables X, Y, Z now correspond to places and the transitions are labeled with atomic actions.



The initial state is the marking that contains only one token on place X.

BPP has received considerable attention, because it is one of the few models for which semantic equivalences like bisimulation [Jan94, Mil89, HJM94, CHS92, CHM93a, Jan95, CHM93b, HJM96, JE96, BCS95, May96a, May96c, May97d] are decidable.

Model checking BPP has also been intensively studied [Esp97, Esp96, May96c, EK95, Hab97]. Chapter 6 shows results about the complexity of model checking BPP.

2.3.3 (P, P)-PRS = Petri Nets

In (P, P)-PRS all rewrite rules have arbitrary parallel composition on both sides, but no sequential composition. So all rules have the form $X_1 \| X_2 \| \ldots \| X_i \xrightarrow{a} Y_1 \| Y_2 \| \ldots \| Y_k$. There is a 1-to-1 correspondence between (P, P)-PRS and Petri nets. Remember that in transition systems described by PRS every state is represented by a process term. Every process variable used in a (P, P)-PRS corresponds to a place in the Petri net, and every process term corresponds to a marking. The number of occurrences of a variable in a term corresponds to the number of tokens on the place in the net. This is because we work with classes of terms modulo commutativity and associativity of parallel composition. Every rewrite rule in Δ corresponds to a transition in the net. It can only be applied if there are enough variables in the term (tokens on places in the net) and replaces a multiset of variables (tokens) by another one.

In Definition 2.1.5 we defined that the left hand sides of rules of a PRS cannot be ϵ. Thus we have the condition that every transition in the Petri net has at least one place in its preset. This is no restriction, since every Petri net can be transformed into an equivalent one that satisfies this condition. In this transformation we just add an extra place to the preset of each transition and arcs from this place to the transition and back.

Petri nets are more general than BPP (see Subsection 2.3.2), because Petri nets are more general than the subclass of communication-free nets. See [Pet81] for a general book on Petri nets.

The following definitions also apply to BPP. They are used in Chapters 6 and 9.

Definition 2.3.4 Every Petri net N with n places and m transitions can be described by a (n, m)-matrix C of integers. The entry $h_{i,j}$ at row i and column j defines how many tokens a firing of transition t_j adds to place s_i. $h_{i,j}$ can be negative. Every marking M is described by a vector of natural numbers of dimension n. The i-th component of this vector is $M(s_i)$.

Let σ be a sequence of transitions. The *Parikh-vector* $P(\sigma)$ of σ is an m-dimensional vector of natural numbers. The j-th component of $P(\sigma)$ is the number of times that the transition t_j occurs in σ.

$E(\sigma) := C \cdot P(\sigma)$ is called the *effect-vector* of σ. Unlike for the Parikh-vector, the elements of the effect-vector can be negative.

It follows that if $M \xrightarrow{\sigma} M'$ then $M' = M + C \cdot P(\sigma) = M + E(\sigma)$. Sometimes the effect-vector $E(\sigma)$ is seen as a function on the set of places. Then $E(\sigma)(s_i)$ is the i-th component of $E(\sigma)$.

Chapter 9 is about model checking Petri nets.

2.3.4 $(1, S)$-PRS = Basic Process Algebra (BPA)

In a $(1, S)$-PRS all rules have single variables on the left hand side and arbitrary sequential composition on the right hand side, like for example $X \xrightarrow{a} Y_1.Y_2.Y_3$. $(1, S)$-PRS are equivalent to the class of Basic Process Algebra (BPA) processes of Bergstra and Klop [BK85]. They are transition systems associated with Greibach normal form (GNF) context-free grammars in which only left-most derivations are permitted. BPA-processes are also called *context-free processes*.

In analogy to BPP, BPA-processes can be represented by recursive equations and process terms that contain action prefixes and the operator "+" for nondeterministic choice. Just as for BPP, nondeterminism can be encoded in the set of rewrite rules and thus the operator "+" is no longer necessary.

Chapter 7 contains results about model checking BPA.

2.3.5 (S, S)-PRS = Pushdown Processes

Pushdown automata are a very common concept in formal language theory. They are used to describe context-free languages (Chomsky-2 languages) [HU79]. However, they can also be used as a process model. The state of the finite control and the stack content then describe a state of the process. The rules that define if a terminal symbol is accepted now describe which atomic actions the process can perform and how the state is changed by these actions. If pushdown automata are used as a process model, then they are called *pushdown processes* or *pushdown systems*.

To present such a pushdown process as a restricted form of a (S, S)-PRS Δ, we partition the set of variables $Var(\Delta)$ into disjoint sets Q (finite control states) and Γ (stack symbols). The rewrite rules are then of the form $p.A \xrightarrow{a} q.\beta$ with $p, q \in Q$, $A \in \Gamma$ and β in Γ^* (β is a sequential composition of variables in Γ). This represents the usual transition of pushdown automata which says that while in control state p with the symbol A at the top of the stack, one can read the input symbol a, move into control state q, and replace the stack element A with the sequence β. Caucal [Cau92] showed that any unrestricted (S,S)-PRS can be presented as a pushdown process, in the sense that the transition systems are isomorphic up to the labeling of states. Thus pushdown processes and (S,S)-PRS are equivalent. The general idea of the proof is to introduce finitely many new variables that are used as abbreviations for sequences (sequential compositions) of normal variables that occur in the rules of the (S,S)-PRS. Only finitely many are needed, because the set of rules is finite. Then the rules are modified by replacing sequences of variables in the rules by the shortest possible abbreviations.

Consider the following (S,S)-PRS.

$$\begin{array}{rcl} X.Y & \xrightarrow{a} & W \\ X.Y.Z & \xrightarrow{b} & X.Y.W \\ X.Y.W & \xrightarrow{c} & Z \\ X & \xrightarrow{d} & \epsilon \end{array}$$

If we introduce an abbreviation K for $X.Y$ then the system becomes

$$\begin{array}{rcl} K & \xrightarrow{a} & W \\ K.Z & \xrightarrow{b} & K.W \\ K.W & \xrightarrow{c} & Z \\ X & \xrightarrow{d} & \epsilon \\ K & \xrightarrow{d} & Y \end{array}$$

This is a pushdown process and it is equivalent to the (S,S)-PRS above.

Pushdown processes are used as a model for sequential systems with subroutines, especially for dataflow analysis in recursive systems [BS95, Ste93]. There has also been work towards a generalization of this to parallel systems by regarding parallel compositions of pushdown processes [BS94]. Process rewrite systems are a more general and more flexible approach to model concurrent systems with recursion (see Subsection 2.3.9, Section 2.4 and Chapter 10).

Chapter 7 is about model checking pushdown processes.

2.3.6 $(1, G)$-PRS = PA-Processes

In $(1, G)$-PRS every rule only has one variable on the left hand side and an arbitrary process term on the right hand side. This class of processes is equivalent to the so-called "PA-processes" (PA stands for "Process Algebra") that were introduced in [BW90] as a natural subset of ACP processes. Nowadays this name hardly fits any more, because the term "Process Algebra" now has a much wider meaning and includes much more general formalisms like CCS [Mil89]. These processes have nondeterminism, sequential composition and parallel composition, but no synchronization.

PA is not a syntactical subset of Milner's Calculus of Communicating Systems (CCS) [Mil89], because CCS does not have an explicit operator for sequential composition. However, CCS is still much more expressive, since it is Turing powerful and can simulate sequential composition by parallel composition and synchronization.

Originally, PA-processes were presented in a different form by recursive equations. However, it can be shown that the two formalisms are equivalent.

Assume a countably infinite set of atomic actions $Act = \{a, b, c, \dots\}$ and a countably infinite set of process variables $Var = \{X, Y, Z, \dots\}$. The class of PA expressions is defined by the following abstract syntax

$$E ::= \epsilon \mid X \mid aE \mid E + E \mid E \| E \mid E.E$$

Convention: We always work with equivalence classes of terms modulo commutativity and associativity of parallel composition and modulo associativity of sequential composition. Also we define that $\epsilon.E = E = E.\epsilon$ and $E \| \epsilon = E$.

A PA is defined by a family of recursive equations $\{X_i := E_i \mid 1 \leq i \leq n\}$, where the X_i are distinct and the E_i are PA expressions at most containing the variables $\{X_1, \dots, X_n\}$. We assume that every variable occurrence in the E_i is *guarded*, i.e. appears within the scope of an action prefix, which ensures that PA-processes generate finitely branching transition graphs. This would not be true if unguarded expressions were allowed. For example, the process $X := a + a \| X$ generates an infinitely branching transition graph.

For every $a \in Act$ the transition relation $\xrightarrow{a}$ is the least relation satisfying the following inference rules:

$$aE \xrightarrow{a} E \qquad \frac{E \xrightarrow{a} E'}{E + F \xrightarrow{a} E'} \qquad \frac{F \xrightarrow{a} F'}{E + F \xrightarrow{a} F'} \qquad \frac{E \xrightarrow{a} E'}{X \xrightarrow{a} E'}(X := E)$$

$$\frac{E \xrightarrow{a} E'}{E \| F \xrightarrow{a} E' \| F} \qquad \frac{F \xrightarrow{a} F'}{E \| F \xrightarrow{a} E \| F'} \qquad \frac{E \xrightarrow{a} E'}{E.F \xrightarrow{a} E'.F}$$

Definition 2.3.5 A PA-process is in *normal form* if all its equations are of the form

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

where $1 \leq i \leq n$, $n_i \in \mathbb{N}$, $a_{ij} \in Act$ and E_i are process terms as used in PRS (that means without "+" (choice) and action prefix).

It has been shown in [BEH95] that any PA-process is semantically equivalent (up to bisimulation (see also Def. 2.5.1)) to a PA-process in normal form. This PA-process in normal form can be effectively constructed.

A PA-process in normal form can be represented by a $(1, G)$-PRS by transforming each recursive equation

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

into n_i new rules

$$X_i \xrightarrow{a_{i1}} E_{i1} \quad \ldots \quad X_i \xrightarrow{a_{in_i}} E_{in_i}$$

The reverse transformation is analogous.

Chapter 8 and especially Section 8.3 shows results on verification problems for PA-processes.

2.3.7 (S, G)-PRS = PAD

PA-processes subsume context-free processes (BPA), but they do not subsume pushdown processes. This observation has led to the definition of a more general model that subsumes both PA-processes and pushdown processes. In the framework of PRS the generalization is obvious: PA-processes are $(1, G)$-PRS and pushdown processes are (S, S)-PRS, so the 'smallest' common generalization is (S, G)-PRS. They have also been called PAD in [May97c]. This name is an artificial construct; PAD = PA + PD.

Like PA-processes, PAD-processes do not allow synchronization between parallel components, but, unlike PA, they can model a limited communication between a subroutine and its caller. Consider the process $t.X$, where t is a process term and X is a variable. In PA the process t may or may not terminate, but it can never affect process X. In PAD the process t may develop into a process described by a single variable Y. There may be a rule $Y.X \xrightarrow{a} t'$ that is now applicable. Thus the behavior of process t can affect process X. In Section 2.5 we show that PAD is strictly more general than PA w.r.t. bisimulation equivalence.

Now we show a small example of a PAD-process. It is described by the following set of rules Δ and has the initial state X.

$$\begin{array}{rcl} X & \xrightarrow{a} & (Y \| X).Z \\ Y & \xrightarrow{b} & \epsilon \\ X.Z & \xrightarrow{c} & X \end{array}$$

Chapter 8 is about model checking PAD.

2.3.8 (P, G)-PRS = PAN

(P, G)-PRS arise naturally as the 'smallest' common generalization of Petri nets ((P, P)-PRS) and PA-processes ($(1, G)$-PRS). They extend Petri nets with sequential composition, which can be seen as the possibility to invoke subroutines. The name PAN has been introduced in [May97a] as a combination of PN (Petri nets) and PA. Although PAN is more general than Petri nets (see Section 2.5), it is not Turing-powerful. Chapter 10 describes this and other results about PAN.

2.3.9 (G, G)-PRS = Process Rewrite Systems

(G, G)-PRS are the most general class of process rewrite systems. They were introduced in [May97c]. By definition they subsume all previously mentioned models. Intuitively, they can be seen as an extension of Petri nets with subroutines. Just like in PAD, a subroutine can return a value to the caller when it terminates. PRS can be seen as a generalized approach to model concurrent systems with recursion, because they are more flexible than parallel compositions of pushdown processes (see Subsection 2.3.5). One possible application is dataflow analysis and the analysis of dependencies of subroutines on each other.

An interesting point about PRS is that they are strictly more expressive than the other models (see Section 2.5), but not Turing-powerful. This is shown in Chapter 10.

2.4 Intended Applications

Process Rewrite Systems are a formalism that can be used to model parallel processes with recursion. We describe a small example of a system that can be modeled with PRS. The system is a parallel program that recursively computes a boolean value. First we write the program in a PASCAL-like pseudo-code.

```
function f(x : data) : boolean;
var x1, x2, x3 : data;
var b1, b2 : boolean;
begin
  if size(x) ≤ 2 then return(Q(x)) fi;   /* Q is some predicate */
  x1 := P1(x, 1);   /* Splitting into subproblems */
  x2 := P1(x, 2);   /* P1 somehow modifies x */
  x3 := P1(x, 3);
  b1 := h(x1) || b2 := h(x2);   /* Parallel call */
  if (b1 or b2)   /* if at least one was successful */
  then
    return(f(x3));   /* apply f to the new instance */
  else
    return(false);
  fi;
end;

function h(x : data) : boolean;
var x1, x2, x3 : data;
var b1, b2, b3 : boolean;
begin
  x1 := P2(x, 1);   /* Splitting into subproblems */
  x2 := P2(x, 2);   /* P2 somehow modifies x */
  x3 := P2(x, 3);
  /* parallel call with different instances */
  b1 := f(x1) || b2 := f(x2) || b3 := f(x3);
  if (b1 and b2 and b3)   /* if all are successful */
  then
    print("Now processing ",x1, x2, x3);
    return(true);
  else
    return(false);
  fi;
end;
```

Of course we cannot model the whole program in PRS, because PRS is not Turing-powerful. However, we can accurately model the basic control structure. An instance of problem $f(x)$ (function f, data x) will be described by the process variable X. An instance of problem $h(x)$ (function h, data x) will be described by the process variable Z. We also have to describe how to handle booleans. Let variable T stand for *true* and F for *false*. The rules for conjunction are

$$T\|T \xrightarrow{and} T \quad T\|F \xrightarrow{and} F \quad F\|T \xrightarrow{and} F \quad F\|F \xrightarrow{and} F$$

In this context the variables T, F are always interpreted conjunctively. In order to be able to enforce a disjunctive interpretation we define new variables to stand for the same boolean values. Let variable R (right) stand for *true* and W (wrong) stand for *false*. The rules for disjunction are

$$R\|R \xrightarrow{or} R \quad R\|W \xrightarrow{or} R \quad W\|R \xrightarrow{or} R \quad W\|W \xrightarrow{or} W$$

Now we describe the rules for the program:

$$\begin{array}{lcll}
X & \xrightarrow{\quad true \quad} & T & (1) \\
X & \xrightarrow{\quad false \quad} & F & (2) \\
P_1 & \xrightarrow{\quad prepare_1 \quad} & \epsilon & (3) \\
X & \xrightarrow{\quad decomp_1 \quad} & P_1.(Z\|Z).X & (4) \\
W.X & \xrightarrow{\quad stop \quad} & F & (5) \\
R.X & \xrightarrow{\quad nextstep \quad} & X & (6) \\
P_2 & \xrightarrow{\quad prepare_2 \quad} & \epsilon & (7) \\
Z & \xrightarrow{\quad decomp_2 \quad} & P_2.(X\|X\|X).Y & (8) \\
F.Y & \xrightarrow{\quad result\ no \quad} & W & (9) \\
T.Y & \xrightarrow{\quad result\ ok \quad} & G.R & (10) \\
G & \xrightarrow{\quad actions \quad} & \epsilon & (11)
\end{array}$$

These rules have the following meanings:

(1) X describes the main program that solves an instance of the problem. If the instance is small enough then the result is clear. In this case it is *true*.

(2) In this case it is *false*.

(3) P_1 stands for some computations that are necessary to decompose the problem X.

(4) In this case the problem is decomposed into smaller problems. First we do some preparation P_1. Then we solve two independent instances of a problem $(h(x_1), h(x_2))$ described by Z. This can be done in parallel. The two results are interpreted disjunctively. If one of them is true, then we solve a (smaller) instance of the main problem X. Otherwise we return *false*.

(5) If the previous result was W (*wrong*), then there is no reason to go on. The result is F (*false*).

(6) If the previous result was R (*right*), then the result only depends on the smaller instance of the main problem X ($f(x_3)$ in the example).

(7) P_2 stands for some computations that are necessary to decompose the problem Z.

(8) The problem Z is also decomposed into three independent (parallel) instances of the problem X. The results are interpreted conjunctively.

(9) If the result was F (*false*), then we terminate immediately and return the value W (*wrong*).

(10) If the result was true, then we first do some other actions G, before returning the value R (*right*).

(11) G stands for some actions that are necessary if an instance of the problem Z was successful. It could be updating a lookup table (for dynamic programming) or outputting a progress message (as it is done here in the program).

Let Δ be the set of rules defined here. It is clear that Δ is a PRS, but no PAN, PAD, PA or Petri net. This is because here the subroutines return values to their callers when they terminate and there is synchronization between parallel components.

In Chapter 10 we describe algorithms that can be used to verify this system. There we show that the reachability problem is decidable for PRS. It is even decidable if there is a reachable state that satisfies certain properties that can be encoded in a simple logic.

2.5 The PRS-Hierarchy is Strict

The question arises if this hierarchy of (α, β)-PRS is strict. In other words, are the higher models in the PRS-hierarchy strictly more expressive than the lower ones?

In order to define what it means to be more expressive, we first define what it means to have the same expressiveness as another model. We use a semantic equivalence called *bisimulation* [Mil89, Mol96]. Bisimulation equivalence has become a very popular semantic equivalence in the formal verification community.

Definition 2.5.1 A binary relation R over the states of a labeled transition system is a *bisimulation* iff

$$\begin{array}{ll} \forall (s_1, s_2) \in R\ \forall a \in Act. & (s_1 \xrightarrow{a} s_1' \ \Rightarrow \exists s_2 \xrightarrow{a} s_2'.\ s_1' R s_2') \wedge \\ & (s_2 \xrightarrow{a} s_2' \ \Rightarrow \exists s_1 \xrightarrow{a} s_1'.\ s_1' R s_2') \end{array}$$

Two states s_1 and s_2 are *bisimilar* iff there is a bisimulation R such that $s_1 R s_2$. This definition can be extended to states in different transition systems by putting them 'side by side' and considering them as a single transition system. It is easy to see that there always exists a largest bisimulation which is an equivalence relation. It is called *bisimulation equivalence* or *bisimilarity* and it is denoted by $\sim$.

The main reason why we use bisimulation is that bisimilar processes satisfy exactly the same set of temporal logic formulae. The converse also holds: Two processes are bisimilar if they cannot be distinguished by Hennessy-Milner Logic (see Subsection 3.1.1). See [Mol96] for a survey on decidability and complexity of bisimilarity for most process models in the PRS-hierarchy. More results on bisimulation can be found in [Jan94, Mil89, HJM94, CHS92, CHM93a, Jan95, CHM93b, HJM96, JE96, BCS95, May96a, May96c, May97d].

Definition 2.5.2 A class of processes A is more general than a class of processes B with respect to bisimulation iff the following two conditions are satisfied:

1. For every B-process there is a semantically equivalent A-process.

$$\forall t \in B.\, \exists t' \in A.\ t' \sim t$$

2. There is an A-process that is not bisimilar to any B-process.

$$\exists t \in A.\, \forall t' \in B.\ t \not\sim t'$$

It has already been established in [BCS96, Mol96] that the classes of finite-state systems, BPP, BPA, pushdown systems, PA and Petri nets are all different with respect to bisimulation. For PAD, PAN and PRS this remains to be shown.

The proof has two parts: First we show that there is a pushdown process that is not bisimilar to any PAN-process. Then we show that there is a Petri net that is not bisimilar to any PAD-process.

Definition 2.5.3 Consider the following pushdown system:

$$\begin{array}{lllllllll}
U.X & \stackrel{a}{\rightarrow} & U.A.X & U.A & \stackrel{a}{\rightarrow} & U.A.A & U.A & \stackrel{b}{\rightarrow} & U.B.A \\
U.X & \stackrel{b}{\rightarrow} & U.B.X & U.B & \stackrel{b}{\rightarrow} & U.B.B & U.B & \stackrel{a}{\rightarrow} & U.A.B \\
U.X & \stackrel{c}{\rightarrow} & V.X & U.A & \stackrel{c}{\rightarrow} & V.A & U.B & \stackrel{c}{\rightarrow} & V.B \\
U.X & \stackrel{d}{\rightarrow} & W.X & U.A & \stackrel{d}{\rightarrow} & W.A & U.B & \stackrel{d}{\rightarrow} & W.B \\
V.A & \stackrel{a}{\rightarrow} & V & V.B & \stackrel{b}{\rightarrow} & V & V.X & \stackrel{e}{\rightarrow} & V \\
W.A & \stackrel{a}{\rightarrow} & W & W.B & \stackrel{b}{\rightarrow} & W & W.X & \stackrel{f}{\rightarrow} & W
\end{array}$$

with the initial state $U.X$. The execution sequences of this system are as follows: First it does a sequence of actions in $\{a, b\}^*$ and then one of two things:

1. A "c", the sequence in reverse and finally a "e".
2. A "d", the sequence in reverse and finally a "f".

Now we show that this pushdown system is not bisimilar to any PAN-process. First we need several definitions and lemmas.

Definition 2.5.4 Let t be an arbitrary process and σ a sequence of actions. The runs of t are its computations of maximal length (see Def. 3.0.15). We define that $only(t, \sigma)$ is true iff the following conditions are satisfied:

- All runs of t are finite.
- All these runs do the sequence of actions σ.

The following general lemma was proved by Dickson in [Dic13].

Lemma 2.5.5 (Dickson's Lemma)
Given an infinite sequence of vectors $M_1, M_2, M_3, \ldots$ in $\mathbb{N}^k$ there are $i < j$ s.t. $M_i \leq M_j$ ($\leq$ taken componentwise).

Remember this: P is the class of process terms that contain only parallel composition; see Def. 2.1.4.

Lemma 2.5.6 *For every PAN Δ there is a sequence $\sigma \in \{a,b\}^*$ s.t. no $\alpha \in P$ satisfies any of the following two conditions:*

Cond1 $\exists \alpha_c.\ \alpha \xrightarrow{c} \alpha_c \wedge only(\alpha_c, \sigma e)$

Cond2 $\exists \alpha_d.\ \alpha \xrightarrow{d} \alpha_d \wedge only(\alpha_d, \sigma f)$

Proof We assume the contrary and derive a contradiction. Let Δ be the PAN. Consider an infinite sequence of sequences $\sigma_1, \sigma_2, \dots \in \{a,b\}^*$ s.t. for all $i < j$ σ_i is not a prefix of σ_j, for example $\sigma_i := a^i.b$ for $i \in \mathbb{N}$. Let $\alpha^i \in P$ be the term that belongs to σ_i and satisfies Cond1 or Cond2. There must be an infinite subsequence of $\alpha_1, \alpha_2, \dots$ where Cond1 is always satisfied or an infinite subsequence of $\alpha_1, \alpha_2, \dots$ where Cond2 is always satisfied. W.r. we assume that there is an infinite subsequence where Cond1 is always satisfied. Now we only regard this infinite subsequence. Since Δ is finite, there are only finitely many different rules in Δ that are marked with the action c. Let $(t_1 \xrightarrow{c} t'_1), \dots, (t_n \xrightarrow{c} t'_n)$ be those rules. (Note that $t_i \in P$ for every i, because Δ is a PAN. However, t'_i need not be in P.) It follows that one of these rules must be used infinitely often to obtain α^i_c from α^i. Let this rule be $(t_k \xrightarrow{c} t'_k)$ for some $k \in \{1, \dots, n\}$. Thus there is an infinite subsequence of the sequence $\alpha_1, \alpha_2, \dots$ where only this rule is used to obtain α^i_c from α^i. Now we consider only this infinite subsequence.

We regard the sequence α^i of the α that satisfy Cond1. $Var(\Delta)$ is finite and $\alpha^i \in P$. Moreover, all α_i only contain variables from the finite set $Var(\Delta)$. Thus we can apply Dickson's Lemma. By Dickson's Lemma there are $j, j' \in \mathbb{N}$ s.t. $j' > j$ and $\alpha^{j'} \geq \alpha^j$ (this means $\alpha^{j'} = \alpha^j \| \beta$ for some $\beta \in P$.)

For both α^j and $\alpha^{j'}$ the rule $(t_k \xrightarrow{c} t'_k)$ is used to obtain α^j_c, $\alpha^{j'}_c$. Thus $\alpha^j = t_k \| \gamma$ for some $\gamma \in P$ and $\alpha^j_c = t'_k \| \gamma$. Also we have $\alpha^{j'} = \alpha^j \| \beta = t_k \| \gamma \| \beta$ and $\alpha^{j'}_c = t'_k \| \gamma \| \beta = \alpha^j_c \| \beta$. By Cond1 we have $only(\alpha^j_c, \sigma_j e)$. However, $\alpha^{j'}_c$ also enables the sequence $\sigma_j e$. This is a contradiction, because $only(\alpha^{j'}_c, \sigma_{j'} e)$ and σ_j is not a prefix of $\sigma_{j'}$. ∎

Lemma 2.5.7 *For every PAN Δ there is a sequence $\Sigma \in \{A,B\}^*$ s.t. no process term t (w.r.t. Δ) is bisimilar to the pushdown system $U.\Sigma.X$ of Def. 2.5.3.*

Proof We assume the contrary and derive a contradiction. Assume that there is a PAN Δ s.t. for every sequence $\Sigma \in \{A,B\}^*$ there is a term $t(\Sigma)$ s.t.

$$t(\Sigma) \sim U.\Sigma.X$$

For every Σ let $t(\Sigma)$ be the smallest term that has this property.

For any sequence $\Sigma \in \{A, B\}^*$ let $\sigma(\Sigma)$ be the sequence of actions a and b that is obtained by converting Σ to lowercase letters.

It follows from the definition of bisimulation that no process that has only finite computations can be bisimilar to a process that has an infinite computation. Thus by Def. 2.5.3 it follows that for every sequence $\Sigma \in \{A, B\}^*$ and every state $t(\Sigma)$ the following properties hold:

C There is a state $t_c(\Sigma)$ s.t. $t(\Sigma) \xrightarrow{c} t_c(\Sigma)$ and $t_c(\Sigma) \sim V.\Sigma.X$ and thus $only(t_c(\Sigma), \sigma(\Sigma)e)$.

D There is a state $t_d(\Sigma)$ s.t. $t(\Sigma) \xrightarrow{d} t_d(\Sigma)$ and $t_d(\Sigma) \sim W.\Sigma.X$ and thus $only(t_d(\Sigma), \sigma(\Sigma)f)$.

For every $t(\Sigma)$ the action c disables the action d and vice versa. Thus the actions c and d must both occur in the same subterm α of $t(\Sigma)$ and $\alpha \in P$ must be a parallel composition of process variables in $Var(\Delta)$. This is because for a PAN and a term of the form $t_1.t_2\|t_3$ (where t_1, t_2 and t_3 are not ϵ) no single action can change both t_1 and t_3.

Now we show that $t(\Sigma)$ can not have the form $t(\Sigma) = (t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha$ where α is a parallel composition of variables and t_i, t_i', $1 \le i \le n$ are process terms. We assume the contrary and derive a contradiction.

Let $t(\Sigma) = (t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha$ where α is a parallel composition of variables and t_i, t_i', $1 \le i \le n$ are process terms. W.r. we can assume that $t_1, \ldots, t_n$ are not deadlocked. We have $\alpha \xrightarrow{c} \alpha_c$ and $\alpha \xrightarrow{d} \alpha_d$ s.t. $t_c(\Sigma) = (t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha_c$ and $t_d(\Sigma) = (t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha_d$. It follows from the conditions C and D that $only(t_c(\Sigma), \sigma(\Sigma)e)$ and $only(t_d(\Sigma), \sigma(\Sigma)f)$. Now there are two cases:

1. Assume that Σ starts with A. Thus $V.\Sigma.X$ can do action a, but not action b. $(t_1.t_1')\|\ldots\|(t_n.t_n')$ is not deadlocked and cannot synchronize with α_c. Thus it must be able to do action a, but not action b. As $t(\Sigma) \sim U.\Sigma.X$ and $U.\Sigma.X \xrightarrow{b} U.B.\Sigma.X$ there must be a t' s.t. $t(\Sigma) \xrightarrow{b} t'$ and $t' \sim U.B.\Sigma.X$. The action b must occur in α, because $(t_1.t_1')\|\ldots\|(t_n.t_n')$ cannot do b. Thus $\alpha \xrightarrow{b} \alpha'$ and $(t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha' \sim U.B.\Sigma.X$. Now $U.B.\Sigma.X \xrightarrow{c} V.B.\Sigma.X$ and thus $\alpha' \xrightarrow{c} \alpha''$ s.t. $(t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha'' \sim V.B.\Sigma.X$, because $(t_1.t_1')\|\ldots\|(t_n.t_n')$ cannot do action c. But now $(t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha''$ can do action a, because $(t_1.t_1')\|\ldots\|(t_n.t_n')$ can do action a. This is a contradiction, because $V.B.\Sigma.X$ cannot do action a.

2. Now assume that Σ starts with B. The proof is similar to the previous case. Just exchange A and B and a and b. Again we get a contradiction.

It follows that $t(\Sigma)$ can never have the form $(t_1.t_1')\|\ldots\|(t_n.t_n')\|\alpha$.

Therefore $t(\Sigma) = (\alpha.t)\|t'$ where $\alpha \in P$ is a parallel composition of variables and t, t' are process terms. (It is possible that t, t' are ϵ.) We have $\alpha \xrightarrow{c} \alpha_c$ and $\alpha \xrightarrow{d} \alpha_d$ s.t. $t_c(\Sigma) = (\alpha_c.t)\|t'$ and $t_d(\Sigma) = (\alpha_d.t)\|t'$. By conditions C and D we have $only(t_c(\Sigma), \sigma(\Sigma)e)$ and $only(t_d(\Sigma), \sigma(\Sigma)f)$. By Lemma 2.5.6 there exists a sequence Σ s.t. no $\alpha \in P$ that satisfies any of the following two conditions:

- $\exists \alpha_c.\ \alpha \xrightarrow{c} \alpha_c \wedge only(\alpha_c, \sigma(\Sigma)e)$
- $\exists \alpha_d.\ \alpha \xrightarrow{d} \alpha_d \wedge only(\alpha_d, \sigma(\Sigma)f)$

Thus α must always terminate by a sequence of actions that is a prefix of $\sigma(\Sigma)$. Thus there must be a suffix Σ' of Σ s.t. $only(t\|t', \sigma(\Sigma')e)$ and another suffix Σ'' of Σ s.t. $only(t\|t', \sigma(\Sigma'')f)$. This is a contradiction. ■

Lemma 2.5.8 *The pushdown system $U.X$ of Def. 2.5.3 is not bisimilar to any PAN Δ with initial state t_0.*

Proof We assume the contrary and derive a contradiction. Assume that there is a PAN Δ with initial state t_0 s.t. $t_0 \sim U.X$. Let Σ be the sequence from Lemma 2.5.7. (Note that Σ depends on Δ.) The process $U.X$ can reach the state $U.\Sigma.X$. Thus t_0 must be able to reach a state t s.t. $t \sim U.\Sigma.X$. However, by Lemma 2.5.7 such a term t does not exist. Thus we have a contradiction. ■

It follows directly that the pushdown system from Def. 2.5.3 is not bisimilar to any PA-process either. However, as PAD and PRS subsume pushdown processes, it is a PAD and PRS-process. Thus PAD is strictly more general then PA and PRS is strictly more general than PAN. PAD subsumes BPP and BPP is incomparable to pushdown systems. Thus PAD is also more general than pushdown processes.

Now we show that there is a Petri net that is not bisimilar to any PAD-process.

Definition 2.5.9 Consider the following Petri net, which is given as a (P, P)-PRS.

$$\begin{array}{llll}
X \xrightarrow{g} X\|A\|B & X \xrightarrow{c} Y & Y\|A \xrightarrow{a} Y & Y\|B \xrightarrow{b} Y \\
X\|A \xrightarrow{d} Z & X\|B \xrightarrow{d} Z & Y\|A \xrightarrow{d} Z & Y\|B \xrightarrow{d} Z
\end{array}$$

The initial state is $X\|A\|B$.

Lemma 2.5.10 *If there is a PAD-process that is bisimilar to the state* $X\|A\|B$ *of the Petri net of Def. 2.5.9, then there is also a pushdown process that is bisimilar to* $X\|A\|B$*.*

Proof Let Δ be a PAD and Q the initial state s.t. $Q \sim X\|A\|B$. W.r. we can assume that Q is a single variable (see Def. 2.1.5). Now we construct a pushdown process (an (S,S)-PRS) Δ' that is also bisimilar to $X\|A\|B$.

Now we show that in every state of Δ the form $(t_1\|t_2).t_3$ (t_3 can be ϵ) t_1 or t_2 must be deadlocked.

Assume that there is a state $(t_1\|t_2).t_3$ that is reachable from Q. Then a state M must be reachable from $X\|A\|B$ s.t. $t_1\|t_2 \sim M$. There are two cases:

1. If M is deadlocked then t_1 and t_2 must be deadlocked.
2. If M is not deadlocked then there is an M' s.t. $M \xrightarrow{d} M'$ and M' is deadlocked. By the definition of PAD a single action d can only change t_1 or t_2, but not both. Thus either t_1 or t_2 must be deadlocked.

Thus in every state $t_1\|t_2$ that is reachable from Q at least one t_i is deadlocked. Since Q is a single variable, parallel composition can only be introduced by PAD-rules. If such a rule has the form $(u \xrightarrow{x} u_1\|u_2) \in \Delta$ for some action x, then u_1 or u_2 must be deadlocked. W.r. let u_1 be deadlocked. However, the term $u_1.t$ for some term t is not necessarily deadlocked. Thus in Δ' we replace the rule $(u \xrightarrow{x} u_1\|u_2)$ by the rule rule $u \xrightarrow{x} u_2.u_1$. (We assume w.r. that u_2 cannot influence u_1. This means that there is no rule in Δ' whose left hand side is $v_2.v_1$ where v_2 is a nonempty suffix of u_2 and v_1 is a nonempty prefix of u_1. This can be achieved by renaming of variables in u_2 and Δ' if necessary.)

The other case where parallel composition occurs in a rule in Δ is when a rule has the form $u \xrightarrow{x} u_1.(u_2\|u_3).u_4$, where u_1 or u_4 can be ϵ. There are two cases:

1. If u_1 can terminate then the term $(u_2\|u_3)$ can become active. Therefore u_2 or u_3 must be deadlocked. W.r. let u_2 be deadlocked. Then in Δ' we replace this rule by the rule $u \xrightarrow{x} u_1.u_3.u_2.u_4$. Note that u_2 is deadlocked, but $u_2.u_4$ is not necessarily deadlocked. (We assume w.r. that u_1 cannot influence u_3 and u_3 cannot influence u_2. This can be achieved by renaming of variables in u_1 and u_3 and Δ' if necessary.)
2. If u_1 cannot terminate then in Δ' we replace this rule by the equivalent rule $u \xrightarrow{x} u_1$.

It follows that no parallel composition occurs in the rules in Δ'. Thus, if the preconditions are correct, the (S,S)-PRS Δ' with initial state Q is bisimilar to $X\|A\|B$. This is the pushdown process that we are looking for. ■

Definition 2.5.11 Let Δ be a (α,β)-PRS for $\alpha,\beta \in \{1,S,P,G\}$ and t_0 the initial state. The language generated by this system is the set of all sequences σ s.t. $\exists t.\ t_0 \xrightarrow{\sigma} t$ and t is deadlocked.

Lemma 2.5.12 *If a process P is bisimilar to a pushdown process then the language generated by P is a context-free language.*

Proof Directly from Def. 2.5.1 and the definition of pushdown processes. ■

Lemma 2.5.13 *The Petri net of Def. 2.5.9 is not bisimilar to any PAD-process.*

Proof We assume the contrary and derive a contradiction. If there is a PAD-process that is bisimilar to the Petri net of Def. 2.5.9, then by Lemma 2.5.10 there is a pushdown process that is bisimilar to this Petri net. Then by Lemma 2.5.12 the Petri net of Def. 2.5.9 generates a context-free language L. By the definition of this Petri net L is

$$\begin{array}{l}
\{g^m c\sigma \mid m \geq 0 \wedge \sigma \in \{a,b\}^* \wedge \#_a\sigma = m+1 \wedge \#_b\sigma = m+1\} \cup \\
\{g^m d \mid m \geq 0\} \cup \\
\left\{ g^m c\sigma d \;\middle|\; \begin{array}{l} m \geq 0 \wedge \sigma \in \{a,b\}^* \wedge \\ \#_a\sigma \leq m+1 \wedge \#_b\sigma \leq m+1 \wedge \#_a\sigma + \#_b\sigma \leq 2m+1 \end{array} \right\}
\end{array}$$

For every $m \in \mathbb{N}$ the word $g^m c a^{m+1} b^{m+1}$ is in L. Now we apply the Pumping-Lemma for context-free languages [HU79]. There is a constant n s.t for every $m \geq n$ the word $g^m c a^{m+1} b^{m+1}$ can be written as $uvwxy$ s.t. $|vwx| \leq n$ and $|vx| \geq 1$ and for every $i > 0$ $uv^iwx^iy \in L$. There are several cases:

1. v contains c or x contains c. This is a contradiction, because no word in L contains c more than once.
2. vwx is part of g^m. This is a contradiction, because $g^{m+k} c a^{m+1} b^{m+1} \notin L$ for any $k > 0$.
3. v is part of g^m and x is part of a^{m+1}. This is a contradiction, because there are no $k, k' > 0$ s.t. $g^{(m+k)} c a^{(m+1+k')} b^{(m+1)}$ is in L.

4. Neither v nor x contains the symbol g. This is a contradiction, because there are no $k, k' > 0$ s.t. $g^m ca^{(m+1+k)} b^{(m+1+k')}$ is in L. ∎

It follows that PAD and PAN are incomparable and PRS is strictly more general than PAD. By combining these results with the other results above we get the following theorem.

Theorem 2.5.14 *The PRS-hierarchy is strict with respect to bisimulation.*

Chapter 3

Temporal Logics and Model Checking

Temporal logics play an important role in formal verification. They are used to specify properties of processes. Such properties are for example "The process is deadlock-free. (A deadlocked state is not reachable)" or "In every infinite execution of a process action b must occur infinitely often".

The *Model Checking Problem* is the problem if a process, which is described by a (possibly infinite, but finitely described) labeled transition system, satisfies a property encoded in a formula in a temporal logic.

<u>MODEL CHECKING</u>

Instance: A finite description of a (possibly infinite) labeled transition system, a state s in this system and a temporal logic formula Φ.
Question: Does the state s satisfy the formula Φ (denoted $s \models \Phi$) ?

The size of an instance of a model checking problem depends on two parameters:

1. The size of the description of the labeled transition system and the state s. Let x be this size.
2. The size of the formula Φ. Let y be this size.

We study the decidability and the computational complexity of model checking problems. Since the size of an instance is described by two parameters x and y there are three different complexities to analyze.

- In the most general case we consider the complexity of the model checking problem in the size of the whole instance $(x + y)$.

- In practice, the formula is normally very small, while the description of the transition system is often very large. Thus y is often very small, while x can be large. Therefore the complexity of the problem in the parameter x is very important. Thus we also consider the complexity of the problem in the parameter x for a fixed formula (and thus fixed y). However, we assume the worst case for this fixed formula.

- It is also possible to consider the complexity of the problem in the parameter y. For this one assumes the worst case of a fixed system (fixed x) and studies the complexity in the size of the formula (parameter y). In practice however, x is often large while y is almost always small. Therefore the complexity of the problem in y is not relevant in practice. Thus this question is not considered in this thesis.

The property that a state in a transition system satisfies a temporal logic formula is defined separately for every temporal logic in the rest of this chapter. In this thesis (α, β)-PRS are used to describe labeled transition systems, but of course this is not the only possibility.

Temporal logic formulae are described by an abstract syntax and interpreted over the computations (runs) of processes. The following definitions apply to all model checking problems.

Definition 3.0.15 (Paths and Runs)
Let $s_0, s_1, s_2, \ldots$ be states of a labeled transition system. A *path* of a labeled transition system is a finite or infinite sequence $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} s_2 \ldots$ such that every triple $s_i \stackrel{a_i}{\rightarrow} s_{i+1}$ belongs to the set of transitions. A *run* is a maximal path, i.e., a path that is either infinite, or terminates in a state without successors.

For a given transition system let $paths(s)$ be the set of paths starting at state s and $runs(s)$ the set of runs starting at state s. For any path $\pi \in paths(s)$ let $prefs(\pi)$ be the set of finite prefixes of π.

For any path $\pi = s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} s_2 \ldots$ we write $\pi(i)$ for s_i and π^i for the path $s_i \stackrel{a_i}{\rightarrow} s_{i+1} \stackrel{a_{i+1}}{\rightarrow} \cdots$.

Let $firstact(\pi) := a_0$. Thus $a_i = firstact(\pi^i)$.

There are two main classes of temporal logics which differ at how their interpretations are defined. *Branching-time logics* are interpreted over the *computation-tree* of the process being analyzed.

Definition 3.0.16 (Computation-tree)
The *computation-tree* of a process is a (possibly infinite) tree whose nodes are labeled with states and whose arcs are labeled with atomic actions. The root node is marked with the initial state s_0. If a node is marked with a state s and $s \xrightarrow{a_i} s_i$, $i = 1, \ldots, n$ then the node has n child-nodes $s_1, \ldots, s_n$. The arc from s to s_i is labeled with a_i.

Note that (α, β)-PRS generate finitely branching transition systems. Thus their computation-trees are finitely branching. The truth of a branching-time logic formula at a state of a process depends on the state and the subtree below it in the computation-tree. *Linear-time logics* are interpreted over the set of all runs of a process. A formula holds at a state iff all runs starting at this state satisfy the formula.

There are many temporal logics which have more or less expressive power. We consider the most commonly known temporal logics and some interesting fragments. Some logics are more expressive than others, but there are also cases where two logics are incomparable. Figure 3.1 shows a classification of the temporal logics we are going to introduce, with respect to their linear-time or branching-time nature and their expressive power. In this figure a line from a logic to another logic above it means, that the logic above is strictly more expressive than the logic below. The dotted line from the linear-time μ-calculus to the modal μ-calculus has the same meaning, but in this case the transformation is not as cheap as in the other cases. For every linear-time μ-calculus formula there is a modal μ-calculus formula that expresses the same property, but the modal μ-calculus formula is exponentially larger (see Section 3.2). All logics in Figure 3.1 have a different expressive power. For a thorough treatment of temporal logics see [MB96, Eme94, Bra92].

Before we describe the various temporal logics in detail we give the motivation for their definition. Hennessy-Milner Logic and weak linear-time logic are very weak logics and can hardly express any interesting properties. Thus they are almost never used in formal verification. Historically, linear-time temporal logic (LTL) [Pnu77] and computation-tree logic (CTL) [CE81] were defined as extensions of these logics by new operators that increase the expressiveness considerably. Thus it became possible to express interesting properties. Nowadays LTL and CTL are widely known and are often used in formal verification. The logics EF and EG are both very natural fragments of CTL. However, the motivations for their definition are quite different. EF is considered to be a simple but useful logic, because it can still express many interesting properties that are important for the verification of systems. For example EF can express the property 'From every reachable state there is a terminating computation'. The logic EG was not

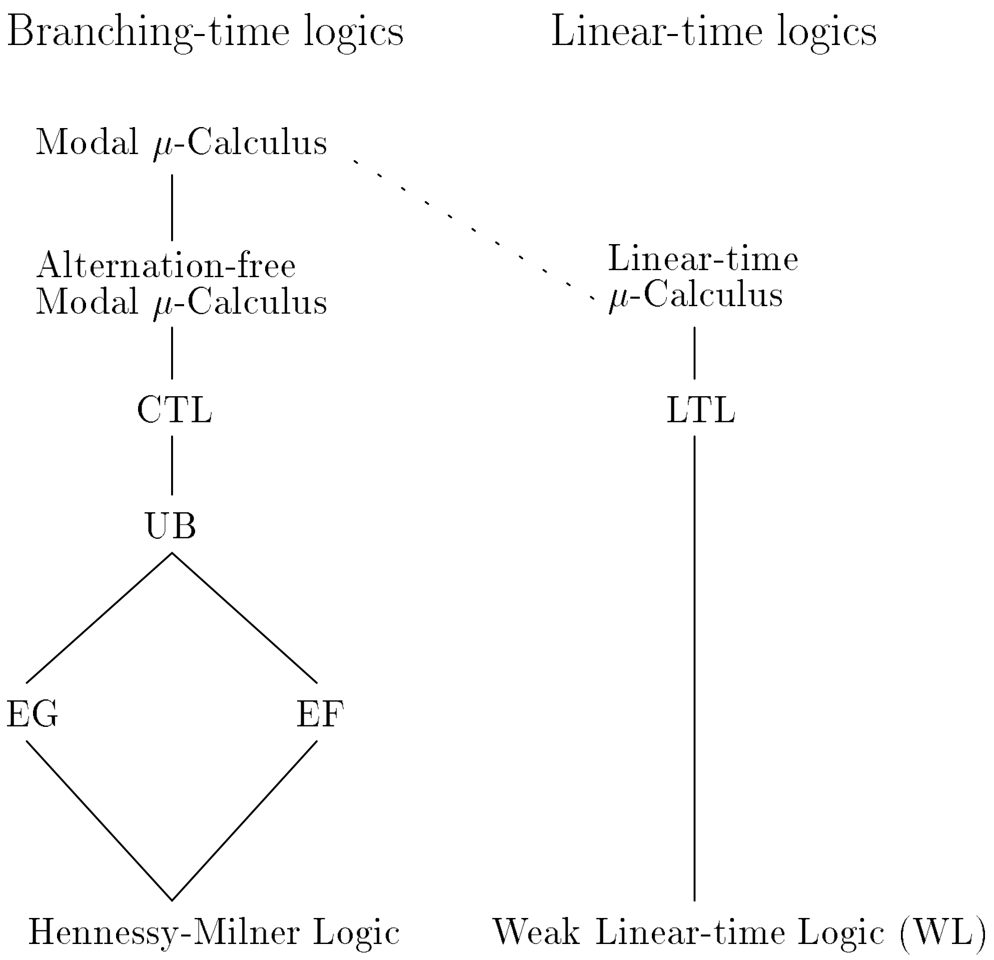


Figure 3.1: Linear and branching-time logics

defined because it is useful, but because it so simple and still so hard. It has been used mostly to prove lower bounds for the decidability and complexity of model checking problems. Some of these results are quite surprising given the limited expressive power of EG. The logic UB is just the smallest common generalization of EF and EG.

The modal μ-calculus [Koz83] and the linear-time μ-calculus are fixpoint logics. They gain their expressive power from minimal and maximal fixpoint operators. In some way they are much more natural than CTL and LTL, because fixpoints are a very elementary concept. The linear-time μ-calculus is only slightly more expressive than LTL. Therefore it is seldom used, because LTL is well-established and many people see no reason to switch to the linear-time μ-calculus. The situation is different for the modal μ-calculus, because it is much more expressive than both CTL and LTL. In fact, the modal μ-calculus is almost too expressive, because the full power of it is almost never needed in practice. Furthermore, it is very hard to understand the meaning of modal μ-calculus formulae with a high nesting-depth of minimal and maximal fixpoint operators. Therefore it has been called 'the machine language of temporal logics', because everything else can be reduced to it, but hardly anyone really understands it completely.

In the most general sense temporal logic formulae are interpreted over the computations of processes given as arbitrary (possibly infinite) labeled transition systems. They can be given a state-based or action-based semantics, or a combination of the two. In state-based semantics formulae are built out of atomic propositions and interpreted according to a valuation that assigns each atomic proposition a set of states in the transition system (the states that satisfy this proposition). The information on top of the arrows is ignored, so it is actually interpreted over an unlabeled transition system. In action-based semantics, the only atomic sentence is true, and the information carried by the states is ignored. In this semantics, logics have relativised next operators, one for each possible label (which is an atomic action). In branching-time logics simple atomic propositions of the form 'action a is enabled' can be expressed with these relativised next operators. This is not the case for linear-time logics.

In the following we will mostly use action-based semantics, for the following reasons: Firstly, it is more natural for our models, which are labeled transition systems. Secondly, atomic propositions of the form 'action a is enabled' are defined for every labeled transition system. This is not the case for stronger atomic propositions which are only defined for some models. For example, propositions of the form 'there are at least k tokens on place p' are only defined for Petri nets and subclasses of Petri nets, but not for other models like context-free processes or PA-Processes. Finally, the decidability of model checking for a temporal logic depends heavily on the set of atomic propositions used. Thus by default we use the minimal action based semantics. However, sometimes (in Chapter 6 and Chapter 8) we use more general sets of atomic propositions, because in these cases this is possible without losing decidability.

3.1 Branching-Time Logics

In branching-time logics, formulae are interpreted on states of a (possibly infinite) labeled transition system. Let Ω be the set of all states. In the framework of (α, β)-PRS, Ω corresponds to the set of process terms $\mathcal{T}$, but for the definition of the logics we stay as general as possible.

3.1.1 Hennessy-Milner Logic

The weakest branching-time logic is *Hennessy-Milner logic*. The formulae have the following syntax:

$$\Phi ::= true \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \langle a\rangle\Phi$$

The denotation $[\![\Phi]\!]$ of a formula Φ is a subset of Ω that is defined inductively as follows:

$$\begin{array}{rcl} [\![true]\!] & := & \Omega \\ [\![\neg\Phi]\!] & := & \Omega - [\![\Phi]\!] \\ [\![\Phi_1 \wedge \Phi_2]\!] & := & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\ [\![\langle a\rangle\Phi]\!] & := & \{s \in \Omega \mid \exists s' \in \Omega.\ s \xrightarrow{a} s' \in [\![\Phi]\!]\} \end{array}$$

The one-step next operator $\langle a\rangle$ is also denoted by $\bigcirc_a$. The operator $\bigcirc$ means a one-step next with any action.

Disjunction can be expressed by conjunction and negation. The atomic proposition 'action a is enabled' can be expressed by the formula $\langle a\rangle true$. We also denote this proposition simply by the term "a".

For any formula Φ and any state $s \in \Omega$, the property $s \in [\![\Phi]\!]$ means 's satisfies Φ' and is also denoted by $s \models \Phi$.

Hennessy-Milner Logic can also be represented without explicit negation. To do this, we need the predicate *false*, disjunction and a second universal one-step next operator. They are defined by

$$\begin{array}{rcl} [\![false]\!] & := & \{\} \\ [\![\Phi_1 \vee \Phi_2]\!] & := & [\![\Phi_1]\!] \cup [\![\Phi_2]\!] \\ [\![\,[a]\Phi\,]\!] & := & \{s \in \Omega \mid \forall s' \in \Omega.\ s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]\} \end{array}$$

We can transform every Hennessy-Milner Logic formula into a formula without negation by pushing the negations inwards. This is possible, because

$$\neg\langle a\rangle\Phi = [a](\neg\Phi)$$

Definition 3.1.1 The size of a formula Φ is defined as the number of atomic propositions in Φ plus the number of operators in Φ.

$$\begin{array}{rcl} size(true) & := & 1 \\ size(\neg\Phi) & := & size(\Phi) + 1 \\ size(\Phi_1 \wedge \Phi_2) & := & size(\Phi_1) + size(\Phi_2) + 1 \\ size(\langle a\rangle\Phi) & := & size(\Phi) + 1 \end{array}$$

For this simple logic the truth or falsity of $s \models \Phi$ only depends on the prefixes of all paths starting at state s that have a length of at most $size(\Phi)$. This is because there are at most $size(\Phi)$ occurrences of the next-operator $\langle a \rangle$ in Φ. Therefore, model checking with Hennessy-Milner logic is decidable for any class of finitely generated transition systems, even for those described by Turing-machines. Hennessy-Milner logic will not play a role in this thesis.

3.1.2 The Logic EF

The branching-time temporal logic EF is an extension of Hennessy Milner logic by the modal operator *EF*, meaning 'for at least one path eventually in the future'. This operator is often denoted by $\Diamond$ and defined as follows:

$$[\![\Diamond\Phi]\!] := \{s \in \Omega \ \mid \ \exists \sigma, s'.\ s \xrightarrow{\sigma} s' \in [\![\Phi]\!]\}$$

where σ is a sequence of actions of arbitrary length.

Another modal operator $\Box$ (meaning "always") can be defined as $\Box := \neg\Diamond\neg$. $\Box\Phi$ means that Φ holds in every reachable state. The modal operator $\Diamond$ significantly increases the expressive power of the logic, because it quantifies over infinitely many sequences of actions of arbitrary length. It has been shown in [JKM98a, JKM98b] that decidability of EF suffices to decide weak bisimulation equivalence between infinite-state processes and finite-state processes. Some other well-known problems can be expressed in fragments of EF.

The *reachability problem* is the problem if a given state is reachable from the initial state.

<u>REACHABILITY</u>

Instance: A finite description of a (possibly infinite) labeled transition system, an initial state s_0 and a given state s in this system.

Question: Is the state s reachable from s_0 ? Formally: Is there a sequence of actions σ s.t. $s_0 \xrightarrow{\sigma} s$?

The problem is not completely expressible in EF with an action-based semantics. However, it is expressible if stronger atomic propositions are introduced. Consider for example propositions of the form "s", where $s \in \Omega$ is a state, which are defined by

$$[\![s]\!] := \{s\}$$

With these atomic propositions the reachability problem is equivalent to the problem

$$s_0 \models \Diamond(s)$$

In action-based semantics a different, but closely related problem is expressible.

Definition 3.1.2 (State Formula)
A *state formula* Ψ is an EF-formula that contains conjunction, disjunction, negation and atomic propositions a (meaning action "a" is enabled), but no modal operators $\Diamond$ or $\langle a \rangle$.

They are called state formulae, because the truth of these formulae at a state only depends on the state and the arcs leading away from this state, but not on the rest of the transition system.

The *reachable property problem* is the problem if there is a reachable state that satisfies certain properties encoded in a state formula.

<u>REACHABLE PROPERTY</u>

Instance: A finite description of a (possibly infinite) labeled transition system, an initial state s_0 and a state formula Ψ.

Question: Is there a reachable state that satisfies Ψ ? Formally: $s_0 \models \Diamond\Psi$?

If strong atomic propositions are allowed, then (as shown above) the reachability problem can be expressed by the reachable property problem. However, in a simple action based semantics this is not the case, since in general there is no state formula whose denotation is a single state. On the other hand in action-based semantics the reachable property problem cannot be expressed by the reachability problem, since there are state formulae whose denotation contains infinitely many states. Thus in a general action-based semantics the two problems are incomparable.

This is different for certain models, especially for Petri nets. Here the reachability problem can be expressed by the reachable property problem, even in a simple action-based semantics. Let Δ be a (P, P)-PRS (which is equivalent to Petri nets; see Chapter 2), t_0 the initial state and t the given state. Let a, b be two new actions (not in $Act(\Delta)$) and $\Delta' := \Delta \cup \{t \xrightarrow{a} t\} \cup \{t \| X \xrightarrow{b} t \| X \mid X \in Var(\Delta)\}$. Then $t_0 \models \Diamond(a \wedge \neg b)$ with respect to Δ' iff t is reachable from t_0 in Δ. The reachable property problem is not expressible by the reachability problem, but by the submarking reachability problem.

It will turn out in the following chapters that both problems have the same decidability and complexity for all models in the PRS-hierarchy. (See Chapter 11 for a summary.)

As mentioned earlier, we use simple action based semantics by default, but we use stronger atomic propositions whenever this is possible without loosing decidability (see Chapters 6 and 8). In Chapter 8 we use a slightly generalized version of EF.

3.1.3 The Logic EG

The branching-time temporal logic EG is the extension of Hennessy Milner logic by the modal operator EG, meaning 'for some path always in the future'. EF and EG are incomparable.

This operator EG is defined as follows:

$$[\![EG\,\Phi]\!] := \{s \in \Omega \ \mid \ \exists \pi \in runs(s)\ \forall \pi' \in prefs(\pi).\ s \xrightarrow{\pi'} s' \Rightarrow s' \in [\![\Phi]\!]\}$$

3.1.4 The Logic UB

The logic UB (for "uniform branching-time") is the extension of Hennessy-Milner Logic by the operators EF and EG. It can be seen as the smallest common generalization of the logics EF and EG. Since these are incomparable, UB is strictly more expressive than both of them.

3.1.5 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) [CE81] is a very popular logic and it is widely used for the verification of finite-state systems. There are several different definitions of it that use different operators. All these definitions have the same expressiveness [MB96]. Here we use a version with a minimal number of operators. It is defined as the extension of Hennessy Milner logic by two operators, the strong until operator $\mathcal{U}$ and the weak until operator $w\mathcal{U}$. The syntax and semantics of these operators is defined as follows:

$$[\![E[\Phi_1\,\mathcal{U}\,\Phi_2]]\!] := \left\{ \begin{array}{lll} s & \mid & \exists \pi \in paths(s).\ s \xrightarrow{\pi} s' \in [\![\Phi_2]\!] \wedge \\ & & \forall \pi'(\neq \pi) \in prefs(\pi).\, s \xrightarrow{\pi'} s'' \Rightarrow s'' \in [\![\Phi_1]\!] \end{array} \right\}$$

Intuitively, this means that there is a path that leads to a state s' that satisfies Φ_2 and all intermediate states on this path satisfy Φ_1. So Φ_1 holds until Φ_2 holds, but Φ_2 must hold eventually.

$$[\![E[\Phi_1\,w\mathcal{U}\,\Phi_2]]\!] := \left\{ \begin{array}{lll} s & \mid & (\exists \pi \in paths(s).\ s \xrightarrow{\pi} s' \in [\![\Phi_2]\!] \wedge \\ & & \forall \pi'(\neq \pi) \in prefs(\pi).\, s \xrightarrow{\pi'} s'' \Rightarrow s'' \in [\![\Phi_1]\!]) \vee \\ & & \exists \pi \in runs(s). \forall \pi' \in prefs(\pi).\ s \xrightarrow{\pi'} s'' \Rightarrow s'' \in [\![\Phi_1]\!] \end{array} \right\}$$

The meaning of $w\mathcal{U}$ is similar to $\mathcal{U}$, except that it allows for the possibility that the path never reaches a state that satisfies Φ_2.

The operators EF and EG can be expressed with these until-operators.

$$EG\,\Phi = E[\Phi\, w\mathcal{U}\, \mathit{false}]$$

and

$$EF\,\Phi = E[\mathit{true}\,\mathcal{U}\,\Phi]$$

Thus CTL subsumes the logic UB.

3.1.6 The Modal μ-Calculus

The modal μ-calculus [Koz83] is a fixpoint logic. It is the extension of Hennessy-Milner logic by variables and fixpoint operators. The semantics of formulae is defined w.r.t. a valuation $\mathcal{V} : Var \mapsto 2^{\Omega}$ that assigns every variable X in the logic a set of states which satisfy it.

$$[\![X]\!]_{\mathcal{V}} := \mathcal{V}(X)$$

The syntax and semantics of the minimal fixpoint operator is defined as follows:

$$[\![\mu X.\Phi]\!]_{\mathcal{V}} := \bigcap\{S \subseteq \Omega \;\mid\; [\![\Phi]\!]_{\mathcal{V}[X:=S]} \subseteq S\}$$

where

$$\mathcal{V}[X := S](X') := \begin{cases} \mathcal{V}(X'), & \text{if } X \neq X' \\ S, & \text{if } X = X' \end{cases}$$

In model checking we use only *closed formulae*. These are the formulae where every variable is bound by a fixpoint operator. Also there is the restriction that every variable occurs within the scope of an even number of negations.

The maximal fixpoint operator ν can be expressed by $\nu X.\Phi := \neg\mu X.\neg\Phi[\neg X/X]$ or defined directly as

$$[\![\nu X.\Phi]\!]_{\mathcal{V}} := \bigcup\{S \subseteq \Omega \;\mid\; S \subseteq [\![\Phi]\!]_{\mathcal{V}[X:=S]}\}$$

The modal μ-calculus is often presented in a form without explicit negation. In this form the negations are pulled inward to the atomic propositions. Formulae then contain conjunction, disjunction, maximal fixpoint and minimal fixpoint, but no negation. In order to achieve this, it is necessary to introduce a second one-step next operator, the so-called "universal one-step next" defined by

$$[\![[a]\Phi]\!] := \{s \;\mid\; \forall s'.\ s \xrightarrow{a} s' \;\Rightarrow\; s' \in [\![\Phi]\!]\}$$

A μ-formula is defined as a formula $\mu X.\Phi$ that starts with the minimal fixpoint operator. A ν-formula is defined analogously. The *alternation-free modal μ-calculus* is defined as the fragment of the modal μ-calculus that satisfies the following two restrictions:

- For every formula $\mu X.\Phi$, the variable X does not occur in any ν-subformula of Φ.
- For every formula $\nu X.\Phi$, the variable X does not occur in any μ-subformula of Φ.

It is easy to see that CTL is a fragment of the alternation-free modal μ-calculus.

$$E[\Phi_1\,\mathcal{U}\,\Phi_2] = \mu X.(\Phi_2 \vee (\Phi_1 \wedge \bigcirc X))$$

$$E[\Phi_1\,w\mathcal{U}\,\Phi_2] = \mu X.(\Phi_2 \vee (\Phi_1 \wedge \bigcirc X)) \;\vee\; \nu X.(\Phi_1 \wedge (\bigcirc X \vee \neg \bigcirc \mathit{true}))$$

3.2 Linear-Time Logics

Formulae in linear-time logics are interpreted over runs of labeled transition systems. A state satisfies a formula if all runs starting at it satisfy the formula.

3.2.1 Weak Linear-Time Logic (WL)

Weak linear-time logic (WL) has the same syntax as Hennessy-Milner logic. However, the interpretation is different. For a transition system with states Ω let $\mathit{runs}(\Omega)$ be the set of all runs (starting at any state).

$$\begin{array}{rcl}
[\![\mathit{true}]\!] & := & \mathit{runs}(\Omega) \\
[\![\neg\Phi]\!] & := & \mathit{runs}(\Omega) - [\![\Phi]\!] \\
[\![\Phi_1 \wedge \Phi_2]\!] & := & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![\langle a\rangle\Phi]\!] & := & \{\pi \in \mathit{runs}(\Omega) \;\mid\; \mathit{firstact}(\pi) = a \;\wedge\; \pi^1 \in [\![\Phi]\!]\}
\end{array}$$

Disjunction can be expressed by conjunction and negation. Unlike in branching-time logics the atomic proposition 'action a is enabled' can not be expressed. Here the formula $\langle a\rangle \mathit{true}$ only means 'action a is the next action in this run'.

It is possible to add atomic propositions by defining

$$[\![a]\!] := \{\pi \in \mathit{runs}(\Omega) \;\mid\; \pi(0) \text{ enables action } a\}$$

These atomic propositions make linear-time logics strictly more expressive. For example LTL (see Subsection 3.2.2) with these propositions is undecidable for Petri nets [Esp97], while normal action based LTL is decidable. Thus by default we use only simple action based linear-time logics.

For any formula Φ and any state $s \in \Omega$ the property $runs(s) \subseteq [\![\Phi]\!]$ means that s satisfies Φ. It is also denoted by $s \models \Phi$.

Just like for Hennessy-Milner Logic, the truth or falsity of $s \models \Phi$ only depends on the prefixes of all paths starting at state s that have a length of at most $size(\Phi)$. Thus model checking with WL is decidable for any class of finitely generated transition systems, even for those described by Turing machines. WL will not play a role in this thesis.

3.2.2 Linear-Time Logic (LTL)

Linear-Time Logic (LTL) [Pnu77] is the extension of WL by the "until"-operators, which are defined in analogy to CTL.

$$[\![\Phi_1 \,\mathcal{U}\, \Phi_2]\!] := \left\{\pi \;\mid\; \exists i.\ \pi^i \in [\![\Phi_2]\!] \;\wedge\; \forall j < i.\ \pi^j \in [\![\Phi_1]\!]\right\}$$

Intuitively, this means that the path has a suffix that satisfies Φ_2 and all prefixes satisfy Φ_1.

$$[\![\Phi_1 \,w\mathcal{U}\, \Phi_2]\!] := \left\{\pi \;\mid\; (\exists i.\ \pi^i \in [\![\Phi_2]\!] \;\wedge\; \forall j < i.\ \pi^j \in [\![\Phi_1]\!]) \;\vee\; \forall i.\ \pi^i \in [\![\Phi_1]\!]\right\}$$

$s \models \Phi \;:\Longleftrightarrow\; runs(s) \subseteq [\![\Phi]\!]$.

It is possible to express the negation of the reachable property problem in LTL with atomic propositions. Let Φ be a state formula (using only atomic propositions, negation and conjunction) and s_0 the initial state. A state that satisfies Φ is not reachable iff

$$s_0 \models (\neg\Phi)\, w\mathcal{U}\, \mathit{false}$$

In the same way the negation of the reachability problem is expressible, but only if stronger atomic propositions are used. These stronger atomic propositions are defined by

$$[\![s]\!] := \{\pi \in runs(\Omega) \;\mid\; \pi(0) = s\}$$

The state s is not reachable from the initial state s_0 iff

$$s_0 \models (\neg s)\, w\mathcal{U}\, \mathit{false}$$

However, these stronger propositions are not normally used for linear-time logics.

Sometimes abbreviations are used for some LTL formulae:

$$\begin{array}{rcl} G\,\Phi & := & \Phi\, w\mathcal{U}\, \mathit{false} \\ F\,\Phi & := & \mathit{true}\, \mathcal{U}\, \Phi \end{array}$$

$G\,\Phi$ means that Φ always holds on a path, and $F\,\Phi$ means that Φ eventually holds on a path.

3.2.3 The Linear-Time μ-Calculus

The linear-time μ-calculus is a fixpoint logic that is defined in analogy to the modal μ-calculus. It is the extension of WL by variables and fixpoint operators.

$$[\![\mu X.\Phi]\!]_{\mathcal{V}} := \bigcap \{S \subseteq runs(\Omega) \;\mid\; [\![\Phi]\!]_{\mathcal{V}[X:=S]} \subseteq S\}$$

Just like the modal μ-calculus it can be presented without negation if, in addition to the normal (strong) next operator, the weak next operator is used.

$$[\![\bigodot_a \Phi]\!] := \{\pi \;\mid\; (\mathit{firstact}(\pi) = a \wedge \pi^1 \in [\![\Phi]\!]) \;\vee\; \mathit{firstact}(\pi) \neq a\}$$

If the subscript is omitted, then it means 'by any action'. Intuitively, $\bigcirc\Phi$ means 'there is a next moment in time and Φ is true at this moment', whereas $\bigodot\Phi$ means 'if there is a next moment in time, then Φ is true at this moment'.

As always in linear-time logics,

$$s \models \Phi \;:\Longleftrightarrow\; runs(s) \subseteq [\![\Phi]\!]$$

Roughly speaking, the intuition for the minimal and maximal fixpoints is the following:

- The minimal fixpoint μ is used to express properties of finite parts of a run. For example the formula $\mu X.(P \vee \bigcirc X)$ means that property P must hold eventually (after a finite number of steps).

- The maximal fixpoint ν is used to express properties that depend on the whole infinite run. For example the formula $\nu X.(P \wedge \bigcirc X)$ means that property P always holds.

Of course the two fixpoints can also be combined. For example the formula $\nu X.\mu Y.(\langle c\rangle X \;\vee\; \bigcirc Y)$ means that the action c occurs infinitely often.

LTL is expressible in the linear-time μ-calculus in the same way that CTL is expressible in the modal μ-calculus.

$$\Phi_1 \,\mathcal{U}\, \Phi_2 = \mu X.(\Phi_2 \vee (\Phi_1 \wedge \bigcirc X))$$

$$\Phi_1 \, w\mathcal{U} \, \Phi_2 = \mu X.(\Phi_2 \vee (\Phi_1 \wedge \bigcirc X)) \;\vee\; \nu X. \left(\Phi_1 \wedge \bigodot X\right)$$

The linear-time μ-calculus can be expressed in the full modal μ-calculus, but only at the cost of an exponential increase in the size of the formula [Bra92].

The fragment of the linear-time μ-calculus that uses only the weak next operator, but not the strong next operator is called the *weak linear-time μ-calculus*. It will be used in Chapter 9.

Sometimes the interpretation of LTL and the linear-time μ-calculus is defined differently. In this other interpretation only the infinite runs of the transition system are considered. Thus the definition is changed to

$$s \models \Phi \;:\Longleftrightarrow\; \{\pi \in \mathit{runs}(s) \;\mid\; \pi \text{ is infinite}\} \subseteq [\![\Phi]\!]$$

We call this the *weak interpretation*, because it is weaker than our definition where both finite and infinite runs are considered. For example the reachability problem cannot be expressed in the weak interpretation. The weak interpretation is equivalent to the *weak linear-time μ-calculus*, the fragment of the linear-time μ-calculus that uses only the weak next operator, but not the strong next operator. This is because in the weak interpretation $[\![\bigcirc \Phi]\!] = [\![\odot \Phi]\!]$. The weak linear-time μ-calculus is used in Chapter 9.

The main difference between the weak linear-time μ-calculus and the normal linear-time μ-calculus is that the negation of the reachability problem cannot be expressed in the weak one. This is smaller restriction than it might seem, because normally the linear-time μ-calculus is only used to verify liveness-properties of systems. These are mostly fairness-properties like 'In every infinite run action a occurs infinitely often'. Such properties only make sense for infinite runs and thus reachability is not needed. Therefore the weak linear-time μ-calculus often suffices.

Chapter 4

Tableau Systems

Tableau systems are a common tool in mathematical logic. The application of tableau systems to temporal logics and verification problems has been initiated by Colin Stirling and Julian Bradfield [BS90, SW90, SW91, BS92a, Sti92, Sti95]. For an overview the reader is referred to [Bra92, Sti96]. Later, tableau systems have been applied to verification problems in order to find structured proof techniques [BEM96, May96b, May97e, BS97] or to achieve decidability results [May97b, May97c].

In this thesis tableau systems will be used in Chapter 8, 9 and 10. The tableau systems of Chapter 8 and Chapter 10 are used in decidability proofs of model checking problems. The tableau system of Chapter 9 is used as a proof method for model checking Petri nets with the weak linear-time μ-calculus. In this chapter we give a brief introduction to tableau systems. For a more thorough treatment see [Bra92] and [Sti96].

Definition 4.0.1 A *tableau* is a proof-tree whose nodes are marked with logical expressions or sets of logical expressions which are called *sequents*. The proof-tree has a unique *root-node* that is marked with the *root-sequent*. The goal of the tableau is to prove the correctness of the root-sequent. The proof-tree is generated by a finite set of *tableau rules* that can be applied to sequents and produce child-nodes that are marked with new sequents.

Tableau rules have the form

$$\frac{A}{B_1 \quad B_2 \quad \ldots \quad B_n}$$

where A is called the *antecedent* and the B_i are called the *succedents*.

Sometimes the applicability and result of a tableau rule also depends on the sequents at several earlier nodes of the same branch in the proof tree. Some tableau rules also have side conditions that must be satisfied. The construction of the tableau can be nondeterministic and thus the tableau for a given root-sequent is not necessarily unique.

Termination conditions are defined on sequents. A node in the tableau whose sequent satisfies a termination condition is a *terminal node*. The construction of the tableau stops at terminal nodes and thus terminal nodes are leaves in the proof-tree.

There are *success conditions* that mark terminal nodes as successful or unsuccessful. The whole tableau is successful if it succeeds in proving the root-sequent. In general the success of the tableau is defined by a function on the success or failure of the terminal nodes. The two most common success conditions for tableaux are the following:

- The tableau is successful iff all terminal nodes are successful. In this case the sequents are often sets of expressions that are interpreted disjunctively, while the branches of the tableau are interpreted conjunctively.
- The tableau is successful iff at least one terminal node is successful. In this case the sequents are often sets of expressions that are interpreted conjunctively [1], while the branches of the tableau are interpreted disjunctively.

A *tableau system* consists of the tableau rules, the termination conditions and the success conditions. It is called *sound* if it can only be successful if the root-sequent is correct. Thus sound tableau systems don't give wrong answers. A tableau system is called *complete* if, for a given correct root sequent, it can always construct a successful tableau. As mentioned earlier the tableau-rules are sometimes nondeterministic. In such cases many different tableaux can be constructed for a given root. In a sound and complete tableau system the root-node is true if and only if at least one of the possible tableaux is successful.

It follows that a tableau system which is sound and complete and for any given root-sequent produces only finitely many different tableaux which are all finite yields a decision procedure. Just construct all the (finitely many) different tableaux for the root-sequent and check if one of them is successful.

In the context of temporal logics the sequents are often sets of expressions of the form $s \vdash \Phi$, where s is a state and Φ is a temporal logic formula. The symbol "$\vdash$"

[1] A set of subgoals that should be proved.

is used instead of "$\models$". This is because the property $s \models \Phi$ is defined semantically while $s \vdash \Phi$ only means that an attempt is made to find a syntactical proof of the property $s \models \Phi$.

Now we show a simple example of a tableau system. We define a tableau system for Hennessy-Milner Logic and arbitrary finitely-branching transition systems (see also Section 3.1.1). The sequents are sets of expressions of the form $s \vdash \Phi$ where s is a state and Φ is a formula. Let Γ stand for sets of such expressions. These sets are interpreted conjunctively, i.e. as sets of subgoals that must be proved. The root-sequent is $\{s_0 \vdash \Phi_0\}$.

The tableau rules are as follows:

$$\wedge \qquad \frac{\{s \vdash \Phi_1 \wedge \Phi_2\} \cup \Gamma}{\{s \vdash \Phi_1, s \vdash \Phi_2\} \cup \Gamma}$$

$$\vee \qquad \frac{\{s \vdash \Phi_1 \vee \Phi_2\} \cup \Gamma}{\{s \vdash \Phi_1\} \cup \Gamma \qquad \{s \vdash \Phi_2\} \cup \Gamma}$$

$$\langle a \rangle \qquad \frac{\{s \vdash \langle a \rangle \Phi\} \cup \Gamma}{\{s_1 \vdash \Phi\} \cup \Gamma \quad \{s_2 \vdash \Phi\} \cup \Gamma \quad \ldots \quad \{s_n \vdash \Phi\} \cup \Gamma} \; (s \xrightarrow{a} s_i)$$

$$[a] \qquad \frac{\{s \vdash [a]\Phi\} \cup \Gamma}{\{s_1 \vdash \Phi, s_2 \vdash \Phi, \ldots, s_n \vdash \Phi\} \cup \Gamma} \; (s \xrightarrow{a} s_i)$$

$$\mathit{true} \qquad \frac{\{s \vdash \mathit{true}\} \cup \Gamma}{\Gamma}$$

A node is a terminal if its sequent is either

1. The empty set.
2. $\{s \vdash \mathit{false}\} \cup \Gamma$
3. $\{s \vdash \langle a \rangle \Phi\} \cup \Gamma$ and there is no s' s.t. $s \xrightarrow{a} s'$.

A terminal of type 1 is successful, but terminals of type 2 or 3 are unsuccessful. In this tableau system the branches are interpreted disjunctively and thus the tableau is successful iff at least one terminal is successful.

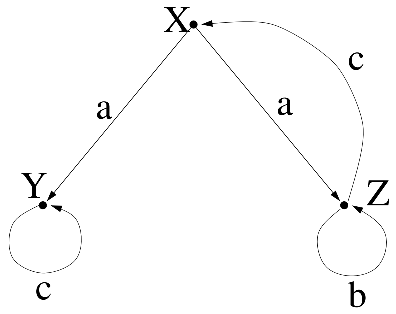
It is easy to see that a tableau constructed by these rules is always finite and $s_0 \models \Phi_0$ iff a tableau with root $\{s_0 \vdash \Phi_0\}$ is successful.

Remark 4.0.2 *Note that it is also possible to construct a different (dual) tableau system for this problem where the sequents are interpreted disjunctively and the branches are interpreted conjunctively. A tableau that is constructed with this other system would be successful if and only if all terminals are successful.*

Example 4.0.3 The following $(1,1)$-PRS defines a finite-state system.

$$X \xrightarrow{a} Y \quad Y \xrightarrow{c} Y \quad X \xrightarrow{a} Z \quad Z \xrightarrow{b} Z \quad Z \xrightarrow{c} X$$

Let X be the initial state. The graphical representation looks like this:



Let $\Phi := \langle a\rangle[b]\mathit{false} \wedge [a]\langle c\rangle \mathit{true}$ be a formula in Hennessy-Milner logic. Intuitively, it means that firstly, there is a state reachable by action a where action b is not enabled and secondly, all states that are reachable via action a enable action c. We now construct the tableau that proves that X satisfies this formula.

$$\begin{array}{c}
\{X \vdash \langle a\rangle[b]\mathit{false} \wedge [a]\langle c\rangle \mathit{true}\ \} \\ \hline
\{X \vdash \langle a\rangle[b]\mathit{false},\ X \vdash [a]\langle c\rangle \mathit{true}\} \\ \hline
\begin{array}{cc}
\begin{array}[t]{c}
\{Y \vdash [b]\mathit{false},\ X \vdash [a]\langle c\rangle \mathit{true}\} \\ \hline
\{X \vdash [a]\langle c\rangle \mathit{true}\} \\ \hline
\{Y \vdash \langle c\rangle \mathit{true},\ Z \vdash \langle c\rangle \mathit{true}\} \\ \hline
\{Y \vdash \mathit{true},\ Z \vdash \langle c\rangle \mathit{true}\} \\ \hline
\{Z \vdash \langle c\rangle \mathit{true}\} \\ \hline
\{X \vdash \mathit{true}\} \\ \hline
\{\} \\ \hline
\text{success, (cond. 1)}
\end{array}
&
\begin{array}[t]{c}
\{Z \vdash [b]\mathit{false},\ X \vdash [a]\langle c\rangle \mathit{true}\} \\ \hline
\{Z \vdash \mathit{false},\ X \vdash [a]\langle c\rangle \mathit{true}\} \\ \hline
\text{failure, (cond. 2)}
\end{array}
\end{array}
\end{array}$$

This tableau is successful, because it has one successful leaf. Thus the transition system with initial state X satisfies the formula Φ.

Chapter 5

Finite-State Systems

Model checking finite-state systems is an important field in both software verification and hardware verification. Since this thesis is about model checking infinite-state systems, we mention the results for finite-state systems only for the sake of completeness. First we describe the results for branching-time logics and then for linear-time logics.

Model checking finite-state systems with the full modal μ-calculus is in $\mathcal{NP} \cap$ co-$\mathcal{NP}$ [EJS93, SW90, SW91, Mad97], but no polynomial algorithm is known so far. The best known algorithms have a complexity of $\mathcal{O}(n^m)$ where n is the size of the transition system and m is the size of the formula [1]. Thus the complexity is polynomial in the size of the system for every fixed formula. Model checking with a fragment, the alternation-free modal μ-calculus (see Section 3.1), can be decided in linear time [SC93, SW90, SW91]. Thus model checking with CTL, UB, EG and EF is polynomial too. The model checker SMV (Symbolic Model Verifier) [CGL94] is a practical tool for model checking finite-state systems with CTL.

Model checking finite-state systems with LTL and the linear-time μ-calculus is *PSPACE*-complete [SC85, Var88]. However, the problem is polynomial in the size of the system for every fixed formula. The model checkers SPIN [Hol91] and PROD [Val92] are practical tools for model checking finite-state systems with LTL. A sound and complete tableau system for finite-state systems and the linear-time μ-calculus is presented in [BEM96].

The following table summarizes the results on the complexity of model checking finite-state systems. We consider two different versions of the model checking

[1] More precisely the complexity is $\mathcal{O}(n^d)$ where d is the alternation-depth of the minimal and maximal fixpoints in the formula

problem (see the definition of the model checking problem in Chapter 3 and the definition of reachability in Subsection 3.1.2):

1. The general case, where the system and the formula are the input.
2. The special case where the formula is fixed and only the system is the input. However, we assume the worst case for the fixed formula.

Finite-state systems	**general**	**fixed formula**
reachability/reachable property	$\in \mathcal{P}$	$\in \mathcal{P}$
EF	$\in \mathcal{P}$	$\in \mathcal{P}$
EG	$\in \mathcal{P}$	$\in \mathcal{P}$
UB	$\in \mathcal{P}$	$\in \mathcal{P}$
CTL	$\in \mathcal{P}$	$\in \mathcal{P}$
alternation-free modal μ-calc.	$\in \mathcal{P}$	$\in \mathcal{P}$
modal μ-calc.	$\in \mathcal{NP} \cap$ co-$\mathcal{NP}$	$\in \mathcal{P}$
LTL	$PSPACE$-complete	$\in \mathcal{P}$
linear-time μ-calc.	$PSPACE$-complete	$\in \mathcal{P}$

Chapter 6

Basic Parallel Processes (BPP)

In this chapter we study model checking problems for Basic Parallel Processes (BPP). As shown in Subsection 2.3.2, BPP are equivalent to communication-free nets, a subclass of Petri nets. Model checking BPP with the logic EG is undecidable [EK95], even for a fixed EG-formula. Thus it is also undecidable for all branching-time logics except for EF (see Section 6.3). In Section 6.1 we show that model checking BPP with the branching-time logic EF is *PSPACE*-complete. In Section 6.2 we show that model checking BPP with LTL and the linear-time μ-calculus is decidable, but at least as hard as the reachability problem for Petri nets. In Section 6.3 we cite other results on BPP and present the general picture.

6.1 Model Checking BPP with EF

As shown in Subsection 2.3.2, BPP are equivalent to communication-free nets, a subclass of Petri nets. While model checking with EF is undecidable for general Petri nets [Esp94, Esp97] (see Chapter 9), it is still decidable for communication-free nets [Esp97]. The proof of the decidability for communication-free nets relied on the fact that for communication-free nets the set of reachable states is effectively semilinear. Thus the model checking problem can be expressed in Presburger arithmetic, which is decidable. The problem with this was that the algorithm relying on Presburger arithmetic requires doubly exponential time, while the problem was only known to be *PSPACE*-hard [Esp97, Esp96]. *PSPACE*-hardness is relatively easy to show, since it holds even for finite-state BPP (notice that the size of the problem is the size of the formula plus the size of the BPP, and not the size of its associated transition system), but the exact complexity of the problem remained open. It was shown in [May96c] that the problem only requires polynomial space, even for infinite-state BPP. Thus it is *PSPACE*-complete.

We present this proof in the terminology of communication-free nets. In fact, this is a somewhat stronger result than for BPP. This is because in communication-free nets of size n we allow arc-weights of up to $\mathcal{O}(2^n)$, while in BPP (or $(1,P)$-PRS) the right hand sides of rules must have size $\mathcal{O}(n)$. This corresponds to arc weights of only $\mathcal{O}(n)$. However, all hardness results also hold for this weaker version.

6.1.1 General Properties of Communication-free Nets

First we prove some general properties of communication-free nets. These results are used later for the model checking problem in Subsection 6.1.2.

Definition 6.1.1 For labeled Petri nets N there is a labeling function $L : T \to Act$ that assigns actions to the transitions. The labeling function L is extended to sequences of transitions in the standard way. $M \xrightarrow{\sigma} M'$ means that a sequence of transitions σ is fireable at a marking M and leads to a new marking M'.

Let σ be a sequence of transitions. Then $P(\sigma)$ is the Parikh-vector of σ and $E(\sigma)$ is the effect-vector of σ (see Def. 2.3.4). Let σ, σ' be sequences of transitions and $P(\sigma), P(\sigma')$ the corresponding Parikh-vectors. Let $P(\sigma)_i$ be the i-th component of the Parikh-vector $P(\sigma)$. Let m be the number of transitions in the net. Then the sequence σ' is called a *smaller sequence* than σ iff

$$\forall i \in \{1, \ldots, n\}.\ P(\sigma')_i \leq P(\sigma)_i$$

σ is then called a *greater sequence* than σ'.

σ' is called a *subsequence* of σ if it is a smaller sequence and the transitions occur in the same order. This means that σ' can be constructed from σ by removing some occurrences of transitions. Note that σ' is not a single piece of σ, but possibly a composition of many pieces of σ. However, these pieces must be in the same order as in σ. If σ' is a subsequence of σ then σ is called a *supersequence* of σ'.

Definition 6.1.2 Let $N = (S, T, W)$ be a communication-free net with places S, transitions T and a function W that assigns weights to the arcs in the net. The size of N is the space needed to describe it with the numbers in binary coding.

$$n := size(N) := \sum_{(x,y) \in Dom(W)} \log(W(x,y)) + 1$$

It follows that $|S| \leq n$, $|T| \leq n$ and $\forall t \in T\ \forall s \in S.\ W(t,s) \leq 2^n$.

For a marking M of N, $tokens(M) := \sum_{s \in S} M(s)$ is the number of tokens in the marking M. The space needed to describe M is $\mathcal{O}(n \cdot \log(tokens(M)))$.

For two markings M and M' of N we define

$$M' \leq M \iff \forall s \in S.\ M'(s) \leq M(s)$$

Now we define interleavings of sequences of transitions.

Definition 6.1.3 Let σ, σ_1 and σ_2 be sequences of transitions. We define that $\sigma \in interleave(\sigma_1, \sigma_2)$ means that σ is an arbitrary interleaving of σ_1 and σ_2. The formal definition is as follows: Let λ be the empty sequence.

$$\begin{aligned}
interleave(\lambda, \sigma) &:= \{\sigma\} \\
interleave(\sigma, \lambda) &:= \{\sigma\} \\
interleave(t_1\sigma_1, t_2\sigma_2) &:= \{t_1\sigma \mid \sigma \in interleave(\sigma_1, t_2\sigma_2)\} \cup \\
&\ \{t_2\sigma \mid \sigma \in interleave(t_1\sigma_1, \sigma_2)\}
\end{aligned}$$

The generalization of the function *interleave* to n arguments is straightforward.

In communication-free nets tokens can move independently of each other. In the following definitions and lemmas we show that this has many interesting consequences. The following definition is somewhat unusual, because it gives tokens a limited individuality. This is contrary to the normal definitions in Petri net theory, but in this special case it does not violate the standard semantics of Petri nets.

Definition 6.1.4 Let N be a communication-free net with initial marking M_0. If a transition fires then it takes one token from the place in its preset and puts several tokens on the places in its postset. We interpret this so that the transition chooses arbitrarily one of the tokens on the place in its preset and then uses this token. We call this token the *parent-token*. The new tokens that the transition puts on the places in its postset are called the *children* of the parent-token. We distinguish these children from the other tokens on the places in the postset that were already there before the transition fired. In this sense the tokens have a limited individuality. Each time a transition fires a nondeterministic choice is made which of the tokens on the place in the preset is used. We call this choice the *token-choice*.

The formal definition is as follows: We assign each token a pair of labels (l, al) where l is a unique label for this token and al is its *ancestor-label*. A marking M

of N is then a mapping s.t. $M(s) = \{(l_1, al_1), \ldots, (l_k, al_k)\}$, if there are k tokens on place s. In the initial marking M_0 every token is its own ancestor and thus $l = al$ for every token in M_0. However, this is not the case for other markings. When a transition t with preset s fires, then it chooses nondeterministically one of the tokens (l_i, al_i) and removes it from the place s. Then (l_i, al_i) is the *token-choice* made by this occurrence of the transition t. t possibly puts several tokens on the places in its postset. These tokens all have their own unique label, but their ancestor-label is al_i, the same as the ancestor-label of the parent-token. Thus the ancestor-label is inherited by the children.

A sequence of transitions σ then represents many different possible sequences of token-choices made by the transitions in σ. If $M_0 \xrightarrow{\sigma}$ then let $choices(M_0, \sigma)$ be the set of possible sequences of token-choices for σ. $choices(M_0, \sigma)$ is finite, but it depends on M_0.

Let M_0 be the initial marking, M' a marking and σ a sequence of transitions s.t. $M_0 \xrightarrow{\sigma} M'$. Now we fix a possible sequence of token-choices $sc \in choices(M_0, \sigma)$. Then for every token in M' one of the following cases holds:

- The token was already there in M_0, or
- The token was created as a child-token by a transition. Then it has a unique parent-token. This parent-token was either already present in M_0 or was created as a child-token from another parent-token, and so on.

Thus every token in M' has a unique ancestor-token in M_0 (possibly itself). This ancestor-token is uniquely determined by the ancestor-label. Every occurrence of a transition t in σ uses exactly one token. We label this occurrence of t with the ancestor-label of this token. We call this label the *ancestor-label* of this occurrence of t [1]. Every occurrence of a transition has a unique ancestor-label. The ancestor-label is only defined for occurrences of transitions. Different occurrences of the same transition may have different ancestor-labels.

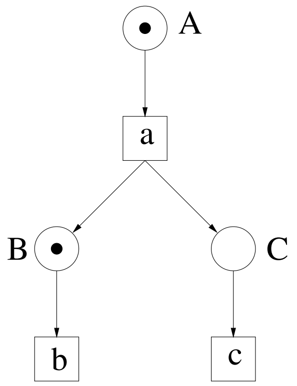
Let $M_0 \xrightarrow{\sigma}$ and let $sc \in choices(M_0, \sigma)$ be a sequence of possible token-choices. If all occurrences of transitions in σ have the same ancestor-label then we call the pair (σ, sc) a *uniform sequence*. We call σ a *1-token initiated sequence* if there is a sequence of token-choices $sc \in choices(M_0, \sigma)$ s.t. (σ, sc) is a uniform sequence.

Now let σ be a 1-token initiated sequence. We say that a place s is the *start* of σ if the ancestor-label of all transitions in the uniform sequence (σ, sc) is a token on place s. The start s of σ is uniquely determined and does not depend

[1] Do not confuse the ancestor-label with the action-label that assigns an atomic action to a transition.

on the choice $sc \in choices(M_0, \sigma)$, because s is the place in the preset of the first transition of σ. It is clear that a 1-token initiated sequence σ with start s is enabled by every initial marking that has at least one token on s. σ is called *removing* if the effect-vector of σ is $(0, \ldots, 0, -1, 0, \ldots, 0)$ where the -1 works on place s. So the effect of a removing sequence σ is just to remove one token from its start s.

Example 6.1.5 Consider the following communication-free net.



There are two ways to interpret the sequence abc. It depends on the token-choice made by the transition b.

1. It can be seen as 1-token initiated sequence abc with start A.
2. It can also be seen as an interleaving of the 1-token initiated sequence ac with start A and the 1-token initiated sequence b with start B.

Lemma 6.1.6 *Let N be a communication-free net with initial marking M_0 and σ a 1-token initiated sequence with start s s.t. $M_0 \xrightarrow{\sigma}$.*

Then $E(\sigma)(s) \geq -1$ and $\forall s' \neq s.\, E(\sigma)(s') \geq 0$.

Proof Let M be the marking that is defined by $M(s) = 1$ and $\forall s' \neq s.\, M(s') = 0$. Then σ is fireable at M. The result follows directly. ∎

Now we show that every sequence can be decomposed (not uniquely) into 1-token initiated sequences.

Lemma 6.1.7 *Let (N, M_0) be a communication-free net with initial marking M_0. Every fireable sequence of transitions σ is an interleaving of finitely many 1-token initiated sequences.*

Proof Choose an arbitrary sequence of token-choices $sc \in choices(M_0, \sigma)$. There are only finitely many tokens in M_0. So there are only finitely many different ancestor-labels $l_1, \ldots, l_k$ for the occurrences of transitions in σ. ($k = tokens(M_0)$). For every ancestor-label l_i let σ_i be the maximal subsequence of σ where all transitions are labeled with l_i. (This means that the ancestor-label l_i only occurs in σ_i and not in the rest of σ.) Then every σ_i is a 1-token initiated sequence that is fireable at M_0 and $\sigma \in interleave(\sigma_1, \ldots, \sigma_k)$. ∎

Thus every sequence σ can be decomposed into 1-token initiated sequences, but this decomposition is not unique, because it depends on the choice of sc. However, the decomposition is unique for every fixed sc. Now we show that the parts of a decomposition are independent of each other.

Lemma 6.1.8 *Let (N, M_0) be a communication-free net with initial marking M_0 and $\sigma \in interleave(\sigma_1, \ldots, \sigma_n)$ a fireable sequence where each σ_i is a 1-token initiated sequence. Let $\{\sigma'_1, \ldots, \sigma'_m\} \subseteq \{\sigma_1, \ldots, \sigma_n\}$.*

Then every sequence $\sigma' \in interleave(\sigma'_1, \ldots, \sigma'_m)$ is also fireable at M_0.

Proof For every $i \in \{1, \ldots, n\}$ the ancestor-label l_i of σ_i is a token in M_0. Thus every σ_i is fireable at M_0. The different sequences σ_i cannot influence each other, since N is a communication-free net. Thus every interleaving of these sequences is fireable at M_0. ∎

A special case of a 1-token initiated sequence is a *cycle*.

Definition 6.1.9 (Cycles/spin-offs)
In communication-free nets every transition t has exactly one place in its preset. Let $Pre(t)$ be the one place in the preset of t and $Post(t)$ the postset of t. A sequence of transitions $\sigma = t_1.t_2.\ldots.t_n$ is a *cycle* iff

1. $\forall i \neq j.\ t_i \neq t_j$
2. $\forall i \in \{1, \ldots, n-1\}.\ Pre(t_{i+1}) \in Post(t_i)$
3. $Pre(t_1) \in Post(t_n)$

It follows that a cycle is a 1-token initiated sequence with start $Pre(t_1)$. If $M \xrightarrow{\sigma} M'$ with a cycle σ, then $M' \geq M$. We can arbitrarily choose any decomposition of M' into M and $(M' - M)$. The tokens in the marking $(M' - M)$ are called *spin-offs*. When a cycle is possible it can be repeated an arbitrary number of times, because the resulting marking is bigger than the original one. A cycle does not change a marking, except that it generates some new tokens (the spin-offs).

The intuition is that in a communication-free net tokens can move freely through the net, because every transition has only one place in its preset and the arc leading from this place to the transition is labeled by 1. When a transition t_i in a cycle fires, then it takes a token from $Pre(t_i)$ and puts some tokens on places in $Post(t_i)$. One regards one token on $Pre(t_{i+1})$ as the continuation of the token that was on $Pre(t_i)$. The other tokens that were put on the places in $Post(t_i)$ are the *spin-offs*. So, informally speaking, a cycle moves a token around and then back to its original place and generates some spin-offs on the way.

We show that sequences that do not contain any cycles have bounded length.

Lemma 6.1.10 *Let (N, M) be a communication-free net with marking M. Let n be the size of N and $x := tokens(M)$. Let σ be a firing sequence starting in M that does not contain any cycle σ' as subsequence. Then*

$$length(\sigma) \leq x\frac{2^{n^2-n}-1}{2^n-1} = \mathcal{O}(x \cdot 2^{n^2})$$

Proof Let m be the number of places in N. Then $m \leq n$. Consider an arbitrary marking M' that is reached from M by a prefix of σ. Every token in M' was either already present in M or it was created by a transition in this prefix of σ. Any path in the net N that does not contain any place twice has a length of at most $m-1$. No subsequence of σ is a cycle. Thus no token can move more than $m-1$ steps. The firing of a transition increases the number of tokens in the net by at most 2^n-1. When a transition fires it replaces a token that can move k steps by $\leq 2^n$ tokens that can only move $\leq k-1$ steps. So only a finite cascade of tokens is possible. Thus the sequence σ has a maximal length of

$$\sum_{i=0}^{m-2} x * (2^n)^i \;=\; x\frac{1-(2^n)^{m-1}}{1-2^n} \;=\; x\frac{2^{nm-n}-1}{2^n-1} \;\leq\; x\frac{2^{n^2-n}-1}{2^n-1}$$

■

Now we define a partial order on markings of nets. Later it'll be used to show that certain classes of smaller markings have the same properties as larger ones.

Definition 6.1.11 First we define a partial order on natural numbers. Let $x, y, y' \in \mathbb{N}$, then

$$y \leq_x y' \;:\Leftrightarrow\; (y \leq y') \wedge (y < y' \Rightarrow y \geq x)$$

For every x relation $\leq_x$ is a partial order on $\mathbb{N}$. Now we define this order on markings of Petri nets. Let N be a Petri net and S the set of its places. For every $x \in \mathbb{N}$ the relation $\leq_x$ on the set of markings of N is defined by

$$M \leq_x M' \;:\Leftrightarrow\; \forall s \in S.\ M(s) \leq_x M'(s)$$

For every x the relation $\leq_x$ is a partial order on the set of markings of N.

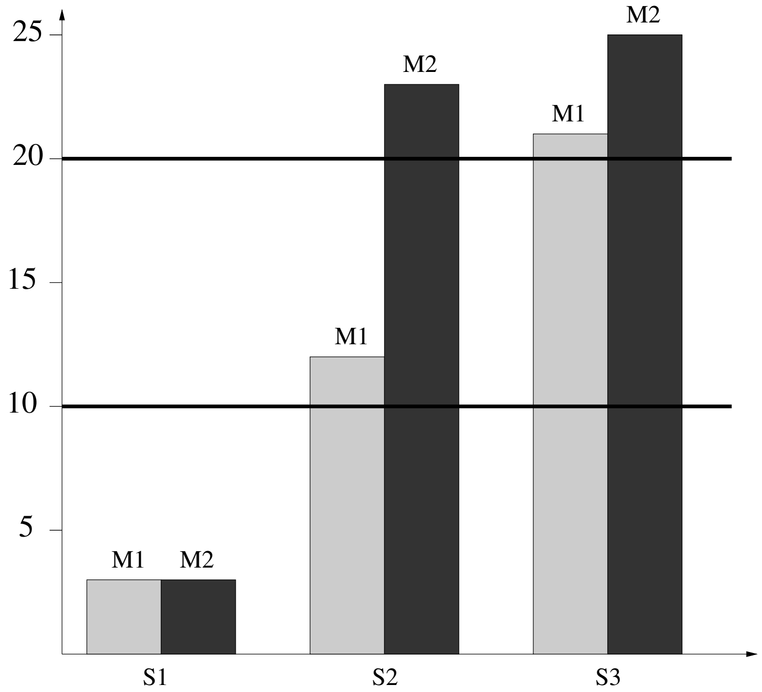


Figure 6.1: $M_1 \leq_{10} M_2$, but not $M_1 \leq_{20} M_2$.

Example 6.1.12 Let there be a net with three places s_1, s_2, s_3 and two markings M_1 and M_2 s.t. $M_1(s_1) = 3$, $M_1(s_2) = 12$, $M_1(s_3) = 21$, $M_2(s_1) = 3$, $M_2(s_2) = 23$ and $M_2(s_3) = 25$. Then $M_1 \leq_{10} M_2$, but not $M_1 \leq_{20} M_2$. Figure 6.1 illustrates this.

The following definitions and lemmas play a central role in the model checking problem. They show that smaller markings can simulate the behavior of larger ones, at least for a limited number of steps.

Definition 6.1.13 Let $N = (S, T, W)$ be a communication-free net and $s \in S$. Then $reach(s)$ is the set of places s' s.t. there is a path in N from s to s'. This can also be defined as a smallest fixpoint.

$$reach(s) := \bigcap\{S' \subseteq S \mid s \in S' \wedge Post(Post(S')) \subseteq S'\}$$

Definition 6.1.14 Let $x \in \mathbb{R}$, $x \geq 0$. Then

$$\begin{aligned} \lfloor x \rfloor &:= max\{y \in \mathbb{N} \mid y \leq x\} \\ \lceil x \rceil &:= min\{y \in \mathbb{N} \mid y \geq x\} \end{aligned}$$

The following lemma is used in the proofs of Lemma 6.1.16 and Lemma 6.1.17.

Lemma 6.1.15 *Let $N = (S, T, W)$ be a communication-free net, $s \in S$, $m := |S|$, $x \in \mathbb{N}$ and M_0 the initial marking with $M_0(s) \geq x$. Let there be a $k \in \mathbb{N}$ s.t. $M_0(s) \geq k > x - \lfloor x/m \rfloor$. Let there be k 1-token initiated sequences $\gamma_1, \ldots, \gamma_k$ with start s s.t. $\forall i \in \{1, \ldots, k\}. E(\gamma_i)(s) = -1$. For every $i \in \{1, \ldots, k\}$ let $\alpha_i := \gamma_1 \ldots \gamma_{i-1}\gamma_{i+1} \ldots \gamma_k$.*

Then there is a $j \in \{1, \ldots, k\}$ s.t.

$$\forall s' \in S.\ (E(\gamma_j)(s') > 0 \ \Rightarrow\ E(\alpha_j)(s') \geq \lfloor x/m \rfloor)$$

Proof We assume the contrary and derive a contradiction. For every $j \in \{1, \ldots, k\}$ there is a place $s'(j)$ s.t. $E(\gamma_j)(s'(j)) > 0$ and $E(\alpha_j) < \lfloor x/m \rfloor$. We know that $s'(j) \neq s$ for all j, because $E(\gamma_j)(s) = -1$. Thus there are only $m - 1$ different choices for $s'(j)$. So there must be a place $s'' \neq s$ s.t. $s'' = s(j)$ for at least $\lceil k/(m-1) \rceil$ different j. We have

$$\begin{aligned} \left\lceil \frac{k}{m-1} \right\rceil &\geq \left\lceil \frac{x - \lfloor x/m \rfloor + 1}{m-1} \right\rceil \\ &\geq \left\lceil \frac{m\lfloor x/m \rfloor - \lfloor x/m \rfloor + 1}{m-1} \right\rceil \\ &= \left\lceil \lfloor x/m \rfloor + \frac{1}{m-1} \right\rceil \\ &= \lfloor x/m \rfloor + 1 \end{aligned}$$

Therefore there are at least $\lfloor x/m \rfloor + 1$ different γ_j with $E(\gamma_j)(s'') > 0$. Thus by Lemma 6.1.6 for every i we have $E(\alpha_i)(s'') \geq \lfloor x/m \rfloor$. We take one j for which $s(j) = s''$ and have a contradiction, because $E(\alpha_j)(s'') < \lfloor x/m \rfloor$. ■

Now we show that smaller markings can, in a limited way, simulate the behavior of larger markings.

Lemma 6.1.16 *Let N be a communication-free net with m places and two markings M_1 and M_2 s.t. $M_1 \leq_x M_2$ for an $x \in \mathbb{N}$. Then for any sequence of transitions σ_2 with $M_2 \xrightarrow{\sigma_2} M_2'$ there is a smaller sequence σ_1 s.t. $M_1 \xrightarrow{\sigma_1} M_1'$ and $M_1' \leq_{\lfloor x/m \rfloor} M_2'$.*

Proof We fix a sequence of token-choices $sc \in \mathit{choices}(M_2, \sigma_2)$. By Lemma 6.1.7 we can decompose σ_2 into 1-token initiated sequences. By Lemma 6.1.6 there are two kinds of 1-token initiated sequences with start s. For every place s let

- $\gamma_1^s, \ldots, \gamma_{n_s}^s$ be the 1-token initiated subsequences of σ_2 with start s s.t. for $1 \leq i \leq n_s$ we have $E(\gamma_i^s)(s) = -1$ and $\forall s' \neq s.\, E(\gamma_i^s)(s') \geq 0$. Let $\gamma_s := \gamma_1^s \ldots \gamma_{n_s}^s$.
- $\delta_1^s, \ldots, \delta_{m_s}^s$ be the 1-token initiated subsequences of σ_2 with start s s.t. $E(\delta_i^s) \geq \vec{0}$ for $1 \leq i \leq m_s$. Let $\delta_s := \delta_1^s \ldots \delta_{m_s}^s$.

It follows that

$$M_2' = M_2 + \sum_s E(\gamma_s) + \sum_s E(\delta_s)$$

By Lemma 6.1.6 no sequence γ_i^s can have a negative effect on a place $s' \neq s$.

Now we construct σ_1 by choosing a subset of these sequences. For every s we choose a subsequence γ_s' of γ_s and add this γ_s' to σ_1. Furthermore all sequences δ_s are added to σ_1. δ_s is added before γ_s', because the effect of δ_s is non-negative on every place. We start with the empty sequence for σ_1. Then for every place s we do the following. As $M_1 \leq_x M_2$ there are two cases:

1. If $M_1(s) = M_2(s)$ then let $\gamma_s' := \gamma_s$ and we add δ and γ_s' to σ_1. The only place on which this sequence can have a negative effect is s. Thus it is fireable at M_1, because it is fireable at M_2 and $M_1(s) = M_2(s)$.
2. If $M_1(s) < M_2(s)$ then $M_1(s) \geq x$. Only the sequences $\gamma_1^s, \ldots, \gamma_{n_s}^s$ can have a negative effect on s. There are two cases:
    (a) If $n_s \leq x - \lfloor x/m \rfloor$ then let $\gamma_s' := \gamma_s$ and we add δ and γ_s' to σ_1. We have $E(\gamma_s')(s) \geq \lfloor x/m \rfloor - x$. Thus the sequence is fireable at M_1 and leaves at least $\lfloor x/m \rfloor$ tokens on the place s. Thus we have $M_1'(s) \geq [x/m]$.

(b) Now we assume $n_s > x - \lfloor x/m \rfloor$. We show that the preconditions of Lemma 6.1.15 are satisfied. $k := n_s$ corresponds to k in Lemma 6.1.15 and the $\gamma_1^s, \ldots, \gamma_{n_s}^s$ correspond to the $\gamma_1, \ldots, \gamma_k$ in Lemma 6.1.15. Then we repeatedly apply Lemma 6.1.15 to remove sequences γ_i^s from $\gamma_1^s \ldots \gamma_{n_s}^s$ until there are only $x - \lfloor x/m \rfloor$ left. Let γ_s' be the concatenation of those γ_i^s that are left. By Lemma 6.1.15 we have for every place $s' \neq s$ either $E(\gamma_s')(s') = E(\gamma_s)(s')$ or at least $E(\gamma_s')(s') \geq \lfloor x/m \rfloor$. Since γ_s' consists of only $x - \lfloor x/m \rfloor$ 1-token initiated sequences γ_i^s we also have $E(\gamma_s')(s) = \lfloor x/m \rfloor - x$. We add δ and γ_s' to σ_1.

Altogether we have the following cases:

1. $M_1(s) = M_2(s)$ and $E(\gamma_s) = E(\gamma_s')$.

2. $M_2(s) > M_1(s) \geq x$ and $E(\gamma_s')(s) \geq \lfloor x/m \rfloor - x$ and

$$\forall s' \neq s.\, E(\gamma_s')(s') \leq_{\lfloor x/m \rfloor} E(\gamma_s)(s')$$

It follows that for every place s

$$\forall s' \neq s.\; E(\gamma_s')(s') \leq_{\lfloor x/m \rfloor} E(\gamma_s)(s')$$

The sequences δ_s have a non-negative effect on every place. We define $\sigma_1 := \delta_{s_1}\gamma_{s_1}' \ldots \delta_{s_n}\gamma_{s_n}'$. By Lemma 6.1.8 and the above conditions σ_1 is fireable at M_1 and we get $M_1 \xrightarrow{\sigma_1} M_1'$.

It remains to show that $M_1' \leq_{\lfloor x/m \rfloor} M_2'$. We show that $M_1'(s) \leq_{\lfloor x/m \rfloor} M_2'(s)$ for every place s. There are two cases:

1. If $M_1(s) = M_2(s)$ then

$$\begin{array}{rcl}
M_1'(s) & = & M_1(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'}')(s) + \sum_{s'} E(\delta_{s'})(s) \\
& = & M_2(s) + E(\gamma_s)(s) + \sum_{s' \neq s} E(\gamma_{s'}')(s) + \sum_{s'} E(\delta_{s'})(s) \\
& \leq_{\lfloor x/m \rfloor} & M_2(s) + E(\gamma_s)(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
& = & M_2'(s)
\end{array}$$

2. If $M_2(s) > M_1(s) \geq x$ then

$$\begin{aligned} M_1'(s) &= M_1(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'}')(s) + \sum_{s'} E(\delta_{s'})(s) \\ &\geq x + E(\gamma_s')(s) \\ &\geq x + (\lfloor x/m \rfloor - x) \\ &= \lfloor x/m \rfloor \end{aligned}$$

Thus $M_1' \leq_{\lfloor x/m \rfloor} M_2'$. ■

Now we show a dual property for larger markings.

Lemma 6.1.17 *Let N be a communication-free net with m places and two markings M_1 and M_2 s.t. $M_1 \leq_x M_2$ for an $x \in \mathbb{N}$. Then for any sequence $M_1 \xrightarrow{\sigma_1} M_1'$ there is a greater sequence σ_2 s.t. $M_2 \xrightarrow{\sigma_2} M_2'$ and $M_1' \leq_{\lfloor x/m \rfloor} M_2'$.*

Proof We fix a sequence of token-choices $sc \in \mathit{choices}(M_1, \sigma_1)$. By Lemma 6.1.7 we can decompose σ_1 into 1-token initiated sequences. By Lemma 6.1.6 there are two kinds of 1-token initiated sequences with start s. For every place s let

- $\gamma_1^s, \ldots, \gamma_{n_s}^s$ be the 1-token initiated subsequences of σ_1 with start s s.t. for $1 \leq i \leq n_s$ we have $E(\gamma_i^s)(s) = -1$ and $\forall s' \neq s.\, E(\gamma_i^s)(s') \geq 0$. Let $\gamma_s := \gamma_1^s \ldots \gamma_{n_s}^s$.
- $\delta_1^s, \ldots, \delta_{m_s}^s$ be the 1-token initiated subsequences of σ_1 with start s s.t. $E(\delta_i^s) \geq \vec{0}$ for $1 \leq i \leq m_s$. Let $\delta_s := \delta_1^s \ldots \delta_{m_s}^s$.

It follows that

$$M_1' = M_1 + \sum_s E(\gamma_s) + \sum_s E(\delta_s)$$

By Lemma 6.1.6 no sequence γ_i^s can have a negative effect on a place $s' \neq s$.

Now we construct σ_2 by choosing a multiset of these sequences (i.e. some sequences γ_i^s occur in σ_2 more than once). For every s we construct a sequence γ_s' s.t. γ_s' is a supersequence of γ_s and add γ_s' to σ_2. Furthermore all sequences δ_s are added to σ_2. δ_s is added before γ_s', because δ_s has a non-negative effect on every place. We start with the empty sequence for σ_2. Then for every place s we do the following. As $M_1 \leq_x M_2$ there are two cases:

1. If $M_1(s) = M_2(s)$ then let $\gamma_s' := \gamma_s$ and we add δ_s and γ_s' to σ_2.

2. If $M_1(s) < M_2(s)$ then $M_1(s) \geq x$. Only the sequences $\gamma_1^s, \ldots, \gamma_{n_s}^s$ can have a negative effect on s. There are two cases:

   (a) If $n_s \leq x - \lfloor x/m \rfloor$ then let $\gamma_s' := \gamma_s$ and we add δ_s and γ_s' to σ_1. We have $E(\gamma_s')(s) = E(\gamma_s)(s) \geq \lfloor x/m \rfloor - x$. Thus the sequence is fireable at M_1 and leaves at least $\lfloor x/m \rfloor$ tokens on the place s.

   (b) Now we assume $n_s > x - \lfloor x/m \rfloor$. We show that the preconditions of Lemma 6.1.15 are satisfied. $k := n_s$ corresponds to k in Lemma 6.1.15 and the $\gamma_1^s, \ldots, \gamma_{n_s}^s$ correspond to the $\gamma_1, \ldots, \gamma_k$ in Lemma 6.1.15. Thus by Lemma 6.1.15 there is a γ_j^s s.t. $\forall s' \neq s.\ (E(\gamma_j^s)(s') > 0 \Rightarrow E(\gamma_s)(s') \geq E(\gamma_1^s \ldots \gamma_{j-1}^s, \gamma_{j+1}^s \ldots \gamma_{n_s}^s) \geq \lfloor x/m \rfloor)$. The intuition is that we have too many tokens on place s in the marking M_2. So we move these surplus tokens away to a place where they do no harm, i.e. to places that contain at least $\lfloor x/m \rfloor$ tokens in the marking M_1'. We can do this by adding several extra copies of γ_j^s to σ_2. Let $w := M_2(s) - M_1(s)$. We define $\gamma_s' := \gamma_s(\gamma_j^s)^w$. Then we have $E(\gamma_s')(s) = E(\gamma_s)(s) - (M_2(s) - M_1(s))$ and $\forall s' \neq s.\ E(\gamma_s)(s') \leq_{\lfloor x/m \rfloor} E(\gamma_s')(s')$. We add δ_s and γ_s' to σ_1.

Altogether we have the following cases:

1. $M_1(s) = M_2(s)$ and $E(\gamma_s) = E(\gamma_s')$.

2. $M_2(s) > M_1(s) \geq x$ and $E(\gamma_s) = E(\gamma_s')$ and $E(\gamma_s)(s) \geq \lfloor x/m \rfloor - x$.

3. $M_2(s) > M_1(s) \geq x$ and $E(\gamma_s')(s) = E(\gamma_s) - (M_2(s) - M_1(s))$ and
$$\forall s' \neq s.\ E(\gamma_s)(s') \leq_{\lfloor x/m \rfloor} E(\gamma_s')(s')$$

It follows that for every place s

$$\forall s' \neq s.\ E(\gamma_s)(s') \leq_{\lfloor x/m \rfloor} E(\gamma_s')(s')$$

The sequences δ_s have a non-negative effect on every place. We define $\sigma_2 := \delta_{s_1}\gamma_{s_1}' \ldots \delta_{s_n}\gamma_{s_n}'$. By Lemma 6.1.8 and the above conditions σ_2 is fireable at M_2 and we get $M_2 \xrightarrow{\sigma_2} M_2'$.

It remains to show that $M_1' \leq_{\lfloor x/m \rfloor} M_2'$. We show that $M_1'(s) \leq_{\lfloor x/m \rfloor} M_2'(s)$ for every place s. We have the same three cases as above:

1. If $M_1(s) = M_2(s)$ then

$$\begin{aligned}
M_1'(s) &= M_1(s) + E(\gamma_s)(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&= M_2(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&\leq_{\lfloor x/m \rfloor} M_2(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'}')(s) + \sum_{s'} E(\delta_{s'})(s) \\
&= M_2'(s)
\end{aligned}$$

2. If $M_2(s) > M_1(s) \geq x$ and $E(\gamma_s) = E(\gamma_s')$ and $E(\gamma_s)(s) \geq \lfloor x/m \rfloor - x$ then

$$\begin{aligned}
M_1'(s) &= M_1(s) + E(\gamma_s)(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&\geq x + E(\gamma_s)(s) \\
&\geq x + (\lfloor x/m \rfloor - x) \\
&= \lfloor x/m \rfloor
\end{aligned}$$

3. Now we consider the third of the cases described above.

$$\begin{aligned}
M_1'(s) &= M_1(s) + E(\gamma_s)(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&= M_1(s) + E(\gamma_s')(s) + (M_2(s) - M_1(s)) + \\
&\quad \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&= M_2(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'})(s) + \sum_{s'} E(\delta_{s'})(s) \\
&\leq_{\lfloor x/m \rfloor} M_2(s) + E(\gamma_s')(s) + \sum_{s' \neq s} E(\gamma_{s'}')(s) + \sum_{s'} E(\delta_{s'})(s) \\
&= M_2'(s)
\end{aligned}$$

Thus $M_1' \leq_{\lfloor x/m \rfloor} M_2'$. ■

Now we show that if $M \xrightarrow{\sigma} M'$ then one can reach a marking M'' with $M'' \leq_k M'$ by a subsequence of bounded length.

Lemma 6.1.18 *Let N be a communication-free net of size n, M a marking of N and $x := tokens(M)$. If $M \xrightarrow{\sigma} M'$ then for every $k \in \mathbb{N}$ there is a subsequence $\tilde{\sigma}$ of σ s.t. $M \xrightarrow{\tilde{\sigma}} \tilde{M}$ and $\tilde{M} \leq_k M'$ and*

$$length(\tilde{\sigma}) \leq n^2k + (x + (2^n - 1)n^2k) * \frac{2^{n^2-n} - 1}{2^n - 1} = \mathcal{O}(2^{n^2}(x + k))$$

Proof The sequence $\tilde{\sigma}$ is the same as σ, except that it possibly contains fewer cycles. What is the maximal number of cycles in $\tilde{\sigma}$ that are needed to reach such a $\tilde{M}$ with $\tilde{M} \leq_k M'$? Cycles just generate new tokens, and at most k new tokens need to be produced per place in N. The number of places in N is $\leq n$. So at most $n * k$ cycles are needed in $\tilde{\sigma}$. In every cycle $\leq n$ transitions are fired, so $\leq (2^n-1)n^2k$ new tokens are produced. So at most $x+(2^n-1)n^2k$ tokens are in the net for moves without cycles. By Lemma 6.1.10 at most $(x+(2^n-1)n^2k)*\frac{2^{n^2-n}-1}{2^n-1}$ non-cyclic moves are possible [2]. By adding the numbers of moves belonging to cycles and the non-cyclic moves we get $length(\tilde{\sigma}) \leq n^2k + (x + (2^n - 1)n^2k) * \frac{2^{n^2-n}-1}{2^n-1} = \mathcal{O}(2^{n^2}(x + k))$. ∎

6.1.2 Model Checking Communication-free Nets

The basic structure of the temporal logic EF (see Chapter 3) is fixed, but the possible atomic propositions can depend on the process model that is analyzed. In this context we use propositions of the form $s \geq k/s \leq k$, meaning 'there are at least/at most k tokens on place s'. It is easy to express the normally used predicates like 'action a is enabled' by these. To do this, just find all places that are in the preset of any transition marked with the action a. As every transition has exactly one place in its preset, action a is enabled iff at least one of these places contains at least one token.

We repeat the syntax of the logic here.

$$\Phi ::= s \geq k \mid s \leq k \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \langle a \rangle \Phi \mid \Diamond\Phi$$

where s ranges over the places of the net N and $k \in \mathbb{N}$. The modal operator $\Box$ can be added by defining $\Box := \neg\Diamond\neg$.

Let $\mathcal{F}$ be the set of all formulae. Let Ω be the set of all markings of N. The denotation $[\![\Phi]\!]$ of a formula Φ is the set of markings of N inductively defined as

[2] This does not necessarily mean that the cycles are done first, and the non-cyclic moves afterwards. Moves belonging to cycles and non-cyclic moves can occur in any order. We consider the worst case where the cycles are done first.

follows:

$$\begin{array}{lcl}
[\![s \geq k]\!] & := & \{M \mid M(s) \geq k\} \\
[\![s \leq k]\!] & := & \{M \mid M(s) \leq k\} \\
[\![\neg\Phi]\!] & := & \Omega - [\![\Phi]\!] \\
[\![\Phi_1 \wedge \Phi_2]\!] & := & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![\langle a\rangle\Phi]\!] & := & \{M \mid \exists M \xrightarrow{a} M' \in [\![\Phi]\!]\} \\
[\![\Diamond\Phi]\!] & := & \{M \mid \exists \sigma.\, M \xrightarrow{\sigma} M' \in [\![\Phi]\!]\}
\end{array}$$

$M \in [\![\Phi]\!]$ is also denoted by $M \models \Phi$.

The model checking problem consists of deducing if $M \models \Phi$ holds for a given communication-free net N with marking M and formula Φ.

Definition 6.1.19 The *nesting-depth* $nd(\Phi)$ of an EF-formula Φ is defined by

$$\begin{array}{rcl}
nd(s \geq k) & := & 0 \\
nd(s \leq k) & := & 0 \\
nd(\neg\Phi) & := & nd(\Phi) \\
nd(\Phi_1 \wedge \Phi_2) & := & max\{nd(\Phi_1), nd(\Phi_2)\} \\
nd(\langle a\rangle\Phi) & := & nd(\Phi) + 1 \\
nd(\Diamond\Phi) & := & nd(\Phi) + 1
\end{array}$$

Definition 6.1.20 $\mathcal{F}_d \subset \mathcal{F}$ is defined as the set of all formulae with a nesting depth of modal operators $\Diamond$ or $\langle a\rangle$ of at most d.

$$\mathcal{F}_d := \{\Phi \in \mathcal{F} \mid nd(\Phi) \leq d\}$$

It follows that formulae in $\mathcal{F}_0$ contain no modal operators.

We show that certain classes of smaller markings satisfy the same EF-formulae as larger ones, provided that these formulae have a limited nesting depth.

Lemma 6.1.21 *Let N be a communication-free net of size $n \geq 2$ and M_1 and M_2 two markings of N. Let $\Phi \in \mathcal{F}_d$ and $\hat{k}$ be the maximal k occurring in a subterm of Φ of the form $s \geq k$ or $s \leq k$. If $M_1 \leq_{(\hat{k}+1)n^d} M_2$ then*

$$M_1 \models \Phi \;\Leftrightarrow\; M_2 \models \Phi$$

Proof By induction on d.

1. If $d = 0$ then Φ doesn't contain any modal operators and $M_1 \leq_{(\hat{k}+1)} M_2$. Thus for all places s and any $k \leq \hat{k}$, $M_1(s) \geq k \Leftrightarrow M_2(s) \geq k$ and $M_1(s) \leq k \Leftrightarrow M_2(s) \leq k$. By induction on the structure of Φ the result follows.

2. Now $d > 0$. We do an induction on the structure of Φ.

   - In the base case $\Phi = s \geq k$ or $\Phi = s \leq k$. Just like in case 1 we have $M_1 \models \Phi \;\Leftrightarrow\; M_2 \models \Phi$.
   - If $\Phi = \neg\Phi'$ then by induction hypothesis. $M_1 \models \Phi' \;\Leftrightarrow\; M_2 \models \Phi'$. It follows directly that $M_1 \models \Phi \;\Leftrightarrow\; M_2 \models \Phi$.
   - If $\Phi = \Phi_1 \wedge \Phi_2$ then by induction hypothesis $M_1 \models \Phi_i \;\Leftrightarrow\; M_2 \models \Phi_i$, $i = 1, 2$. It follows that $M_1 \models \Phi \;\Leftrightarrow\; M_2 \models \Phi$.
   - Now $\Phi = \langle a \rangle \varphi$ for a $\varphi \in \mathcal{F}_{d-1}$.

     $\Rightarrow$ If $M_1 \models \langle a \rangle \varphi$, then there is a M_1' s.t. $M_1 \xrightarrow{a} M_1'$ and $M_1' \models \varphi$. Fire the same transition in M_2 and get $M_2 \xrightarrow{a} M_2'$ with $M_1' \leq_{(\hat{k}+1)n^d-1} M_2'$. As $n \geq 2$ it follows that $M_1' \leq_{(\hat{k}+1)n^{(d-1)}} M_2'$. By induction hypothesis $M_2' \models \varphi$ and therefore $M_2 \models \langle a \rangle \varphi$.

     $\Leftarrow$ If $M_2 \models \langle a \rangle \varphi$ then there is a M_2' s.t. $M_2 \xrightarrow{a} M_2'$ and $M_2' \models \varphi$. The same transition is fireable in M_1 and so we get $M_1 \xrightarrow{a} M_1'$ with $M_1' \leq_{(\hat{k}+1)n^d-1} M_2'$. As $n \geq 2$ it follows that $M_1' \leq_{(\hat{k}+1)n^{(d-1)}} M_2'$. By induction hypothesis $M_1' \models \varphi$ and therefore $M_1 \models \langle a \rangle \varphi$.
   - Now $\Phi = \Diamond\varphi$ for a $\varphi \in \mathcal{F}_{d-1}$.

     $\Rightarrow$ If $M_1 \models \Diamond\varphi$ then there is a sequence σ s.t. $M_1 \xrightarrow{\sigma} M_1'$ and $M_1' \models \varphi$. The number of places in N is $\leq n$. Thus by Lemma 6.1.17 there is a greater sequence σ' s.t. $M_2 \xrightarrow{\sigma'} M_2'$ and $M_1' \leq_{(\hat{k}+1)n^{(d-1)}} M_2'$. By induction hypothesis $M_2' \models \varphi$ and therefore $M_2 \models \Diamond\varphi$.

     $\Leftarrow$ If $M_2 \models \Diamond\varphi$ then there is a sequence σ s.t. $M_2 \xrightarrow{\sigma} M_2'$ and $M_2' \models \varphi$. By Lemma 6.1.16 there is a smaller sequence σ' s.t. $M_1 \xrightarrow{\sigma'} M_1'$ and $M_1' \leq_{(\hat{k}+1)n^{(d-1)}} M_2'$. By induction hypothesis $M_1' \models \varphi$ and therefore $M_1 \models \Diamond\varphi$. ■

We show that in order to decide $M \models \Diamond\Phi$ it suffices to check $M' \models \Phi$ for those M' that can be reached from M by sequences of a certain bounded length.

Lemma 6.1.22 *Let N be a communication-free net of size n, M a marking, $x := tokens(M)$, $\Phi \in \mathcal{F}_d$ and $\hat{k}$ be the maximal k in a subterm of Φ of the form $s \geq k$ or $s \leq k$. Then*

$$M \models \Diamond\Phi \;\Leftrightarrow\; \exists\tilde{\sigma}.\, M \xrightarrow{\tilde{\sigma}} \tilde{M} \;\wedge\; \tilde{M} \models \Phi \;\wedge\; length(\tilde{\sigma}) \leq \mathcal{O}((x + \hat{k}) * 2^{n^2} * n^d)$$

Proof There must be a sequence σ s.t. $M \xrightarrow{\sigma} M'$ and $M' \models \Phi$. By Lemma 6.1.18 there is a smaller sequence $\tilde{\sigma}$ s.t. $M \xrightarrow{\tilde{\sigma}} \tilde{M}$, $\tilde{M} \leq_{(\hat{k}+1)n^d} M'$ and

$$length(\tilde{\sigma}) \leq n^2 * ((\hat{k}+1)n^d) + (x + (2^n - 1) * n^2 * (\hat{k}+1)n^d) * \frac{2^{n^2-n} - 1}{2^n - 1}$$

So $M \xrightarrow{\tilde{\sigma}} \tilde{M}$, $length(\tilde{\sigma}) = \mathcal{O}((x+\hat{k}) * 2^{n^2} * n^d)$ and by Lemma 6.1.21 $\tilde{M} \models \Phi$. ■

Esparza [Esp95] showed that for communication-free nets it is decidable in polynomial time if there is a fireable sequence of transitions with a given Parikh-vector.

Lemma 6.1.23 *Let N be a communication-free net with marking M and K a Parikh-vector.*

It can be decided in $\mathcal{O}(n^3)$ time if there is a fireable sequence of transitions σ ($M \xrightarrow{\sigma}$) with Parikh-vector K ($P(\sigma) = K$).

Proof By Esparza in [Esp95]. ■

Now we show that the model checking problem for EF-formulae of nesting depth d is complete for the d-th order in the polynomial time hierarchy. (See [vL90] for the definition of the polynomial time hierarchy.)

Lemma 6.1.24 *Let N be a communication-free net, M a marking of N and $\Phi \in \mathcal{F}_d$. The problem $M \models \Diamond\Phi$ can be solved in Σ^p_{d+1}.*

Proof By induction on d.
Let n be the size of the instance of the problem, as defined in the definition of model checking in Chapter 3. So n is the size of (N, M) plus the size of $\Diamond\Phi$ in binary coding. It follows that N has $\mathcal{O}(n)$ places and if $\hat{k}$ is the maximal k occurring in any subterm of Φ of the form $s \leq k$ or $s \geq k$, then $\hat{k} = \mathcal{O}(2^n)$. Also $x := tokens(M) = \mathcal{O}(2^n)$ and $d = \mathcal{O}(n)$.

1. If $d = 0$ then Φ doesn't contain any modal operators. By Lemma 6.1.22 it suffices to look for a $\tilde{\sigma}$ with $M \xrightarrow{\tilde{\sigma}} \tilde{M}$ s.t. $length(\tilde{\sigma}) = \mathcal{O}((x+\hat{k})2^{n^2} * n^d)$ and $\tilde{M} \models \Phi$. As $\hat{k} = \mathcal{O}(2^n)$ and $x = \mathcal{O}(2^n)$ and $d = \mathcal{O}(n)$ the Parikh-vector of $\tilde{\sigma}$ can be written in polynomial space. Now guess a Parikh-vector of polynomial size. By Lemma 6.1.23 it can be checked in polynomial time if there is a sequence $\tilde{\sigma}$ with this Parikh-vector s.t. $M \xrightarrow{\tilde{\sigma}} \tilde{M}$. It only takes polynomial time to compute the resulting marking $\tilde{M}$ and $\tilde{M}$ can be described in polynomial space. It can be checked in polynomial time if $\tilde{M} \models \Phi$. So the problem can be solved in $\mathcal{NP} = \Sigma^p_1$.

2. Now $d > 0$. Again by Lemma 6.1.22 it suffices to guess a Parikh-vector of polynomial size. Then by Lemma 6.1.23 we can check in polynomial time if there is a fireable sequence $\tilde{\sigma}$ with this Parikh-vector s.t. $M \xrightarrow{\tilde{\sigma}} \tilde{M}$ and compute $\tilde{M}$ in polynomial time. As $tokens(M) = x$ and $length(\tilde{\sigma}) = \mathcal{O}((x + \hat{k})2^{n^2} * n^d)$ one can assume that $tokens(\tilde{M}) = \mathcal{O}(x + 2^n((x + \hat{k})2^{n^2} * n^d))$. It follows that $tokens(\tilde{M}) = \mathcal{O}(2^{n^2})$ and $\tilde{M}$ can be described in polynomial space. It is possible to apply the induction hypothesis and to check if $\tilde{M} \models \Phi$ in polynomial time with the help of a Σ_d^p-oracle. The oracle is used to solve the problem for the subformulae of Φ that have the form $\Diamond\varphi$ with $\varphi \in \mathcal{F}_{d-1}$. Therefore the problem can be solved in $\mathcal{NP}^{\Sigma_d^p} = \Sigma_{d+1}^p$. ■

The following lower bounds for the model checking problem were shown by Esparza in [Esp97].

Lemma 6.1.25 *Let N be a communication-free net, M a marking of N and $\Phi \in \mathcal{F}_d$. The problem $M \models \Diamond\Phi$ is Σ_{d+1}^p-hard.*

Proof (by Esparza in [Esp97])
The problem of the validity of bounded quantified boolean formulae (BQBF) can be reduced to the model checking problem. Example 6.1.27 describes the idea. ■

Lemma 6.1.26 *Let N be a communication-free net, M a marking of N and $\Phi \in \mathcal{F}$. The problem $M \models \Diamond\Phi$ is PSPACE-hard.*

Proof (by Esparza in [Esp97])
The problem of the validity of quantified boolean formulae (QBF) can be reduced to this model checking problem. Example 6.1.27 describes the idea. ■

These hardness results even hold for communication-free nets with a finite state space. They remain true if the logic is restricted to atomic propositions of the form $s > 0$ or 'action a is enabled' instead of $s \geq k / s \leq k$.

Example 6.1.27 For the formula $\exists x_1 \forall x_2 \exists x_3.(x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_2 \wedge x_3)$ the communication-free net of Figure 6.2 is constructed. It is easy to see that

$$\exists x_1 \forall x_2 \exists x_3.(x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_2 \wedge x_3)$$
$$\Longleftrightarrow$$
$$\Diamond(\tilde{x_2} > 0 \wedge \Box(\tilde{x_3} = 0 \vee \Diamond((x_1 > 0 \wedge \bar{x_2} > 0 \wedge \bar{x_3} > 0) \vee (x_2 > 0 \wedge x_3 > 0))))$$

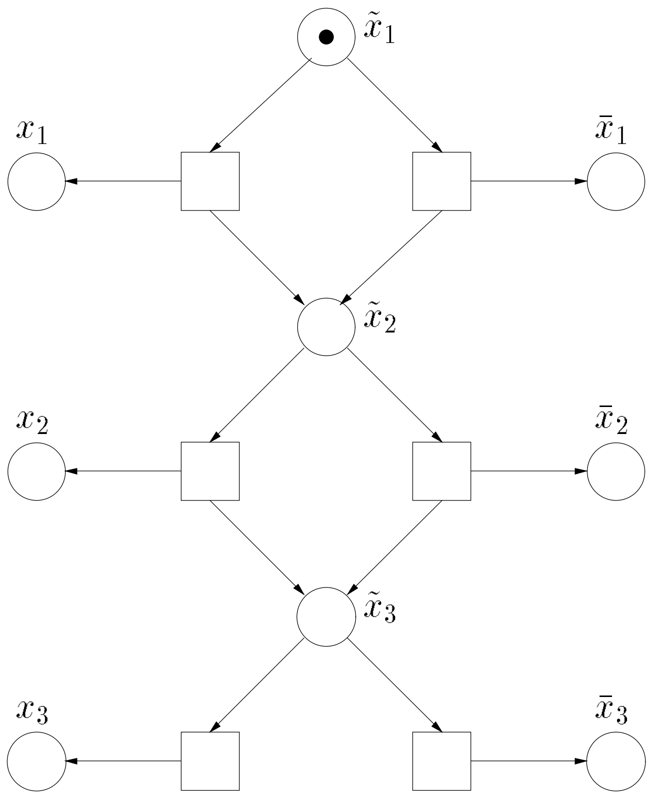


Figure 6.2: Hardness of model checking BPP.

Theorem 6.1.28 *Let N be a communication-free net, M a marking of N and $\Phi \in \mathcal{F}_d$. The problem $M \models \Diamond\Phi$ is Σ^p_{d+1}-complete.*

Proof Directly from Lemma 6.1.24 and Lemma 6.1.25. ■

Theorem 6.1.29 *Model checking BPP with EF is PSPACE-complete.*

Proof BPP are equivalent to communication-free Petri nets. Let N be a communication-free net, M a marking of N and $\Phi \in \mathcal{F}$. The question is if $M \models \Phi$.

Let n be the size of the instance of the model checking problem (the size of (N, M) plus the size of Φ). Let $x := \mathit{tokens}(M)$, $y := \mathit{size}(\Phi)$, $d := \mathit{nd}(\Phi)$ the nesting-depth of Φ and $\hat{k}$ the maximal k occurring in a subterm of Φ of the form $s \geq k$ or $s \leq k$. Thus $x = \mathcal{O}(2^n)$, $y = \mathcal{O}(n)$, $d = \mathcal{O}(n)$ and $\hat{k} = \mathcal{O}(2^n)$. We show by induction on d that the problem can be solved nondeterministically with $\mathcal{O}(n^3 + d * n^2 + y + \log x)$ space.

If $d = 0$ then Φ does not contain the operator $\Diamond$. By induction on the structure of Φ the problem can be solved in time $\mathcal{O}(y)$ and thus in space $\mathcal{O}(y)$.

If $d > 0$ then also use induction on the structure of Φ until subformulae of the form $\Diamond\varphi$ are reached. This requires only $\mathcal{O}(y)$ time and space. The only difficult part are the subproblems of the form $M \models \Diamond\varphi$. Then $\varphi \in \mathcal{F}_{d-1}$ and by Lemma 6.1.22 it suffices to look for a $M \xrightarrow{\sigma} M'$ s.t. $\mathit{length}(\sigma) = \mathcal{O}((x+\hat{k})2^{n^2} * n^{d-1}) = \mathcal{O}(x*2^{n^2} * n^{d-1}) = \mathcal{O}(x*2^{n^2+(d-1)\log n}) = \mathcal{O}(x*2^{n^2})$ and $M' \models \varphi$. The Parikh-vector of σ can be written in space $\mathcal{O}(n(\log x + n^2)) = \mathcal{O}(n^3)$. We nondeterministically guess such a Parikh-vector. By Lemma 6.1.23 it can be decided in time (and space) $\mathcal{O}(n^3)$ if there is a fireable sequence with this Parikh-vector. If yes, then the resulting marking M' can also be computed in polynomial time. Let $x' := \mathit{tokens}(M')$. It follows that $x' \leq x + 2^n\, \mathit{length}(\sigma) = \mathcal{O}(x * 2^{n^2+n}) = \mathcal{O}(x * 2^{n^2})$. By induction hypothesis the problem $M' \models \varphi$ can be decided in space $\mathcal{O}(n^3 + (d-1)n^2 + y + \log x') = \mathcal{O}(n^3 + (d-1)n^2 + y + \log x + n^2) = \mathcal{O}(n^3 + dn^2 + y + \log x)$. There are at most $\mathcal{O}(y)$ such subproblems and thus the whole problem can be solved nondeterministically with $\mathcal{O}(n^3 + dn^2 + y + \log x)$ space.

As $d = \mathcal{O}(n)$, $y = \mathcal{O}(n)$ and $x = \mathcal{O}(2^n)$ the problem is in $\mathit{NSPACE}(\mathcal{O}(n^3))$. By the theorem of Savitch [vL90] it is in $\mathit{DSPACE}(\mathcal{O}(n^6)) \subseteq \mathit{PSPACE}$.

By combining this with Lemma 6.1.26 it follows that the problem is *PSPACE*-complete. ■

So far we have seen that the model checking problem for BPP and EF is *PSPACE*-complete in the general case and Σ_d^p-complete if the formulae are restricted to nesting-depth d. (Note that formulae of a fixed nesting-depth can still be arbitrarily large.) The question about the complexity of the problem in the size of the BPP for every fixed formula is still open. However, there is a linear time algorithm for a slightly more restricted problem.

Theorem 6.1.30 *Let N be a fixed communication-free net and Φ a fixed EF-formula. Then the problem if $M \models \Phi$ for a marking M can be solved in linear time in the size of M.*

Proof Let n be the size of N, $\hat{k}$ the maximal k that occurs in Φ in an atomic proposition of the form $s \geq k$ or $s \leq k$ and let d be the nesting-depth of Φ. So n, $\hat{k}$ and d are fixed. Thus we can also assume that we already know if $M' \models \Phi$ for every M' s.t. $\forall s.\, M'(s) \leq (\hat{k}+1)n^d$, because the number of these markings M' is fixed.

Now the algorithm for deciding $M \models \Phi$ is as follows: We construct a new marking M' by defining for every s

$$M'(s) := \mathit{min}\{M(s),\, (\hat{k}+1)n^d\}$$

This can be done in linear time. We know already if $M' \models \Phi$, because M' satisfies the condition above. By Lemma 6.1.21 we have

$$M \models \Phi \iff M' \models \Phi$$

■

The results on the complexity of model checking BPP with EF can be summarized as follows.

Problem	**Complexity**
general	*PSPACE*-complete
formula restricted to nesting-depth d	Σ_d^p-complete
fixed formula	$\in \Sigma_d^p$
fixed formula and fixed structure of the BPP	$\in \mathcal{P}$

6.2 Model Checking BPP with LTL

In [Hab97] Habermehl solves the model checking problem for Petri nets and BPP and a version of the linear-time μ-calculus (and LTL). In this version only infinite runs of the system are considered. This is weaker than our version (see Section 3.2) where both finite and infinite runs are considered. Habermehl's version has the same expressiveness as the weak linear-time μ-calculus, which is defined in Section 3.2 and used in Section 9.2. He shows that for the **weak** linear-time μ-calculus the model checking problem is *EXPSPACE*-complete in the size of the Petri net and *PSPACE*-complete in the size of the formula. He claims to prove the same result for BPP, but this is only partially correct. It is correct that model checking BPP with the weak linear-time μ-calculus is *EXPSPACE*-complete, but the proof in [Hab97] that the problem is *EXPSPACE*-hard in the size of the BPP is incorrect. In this proof the size of the formula grows together with the size of the BPP. So the complexity of model checking BPP with a fixed linear-time μ-calculus formula is an open question.

Now we consider normal LTL and the normal linear-time μ-calculus as defined in Section 3.2. Model checking with these logics is decidable for Petri nets (see Chapter 9). However, it is at least as hard as the reachability property problem for Petri nets, because reachability of a deadlocked state can be expressed in LTL (see Section 3.2). This is a (potentially) stronger result, because the complexity of reachability for Petri nets is an open question, and it may be harder than *EXPSPACE*.

Since BPP are a subclass of Petri nets, model checking with LTL and the linear-time μ-calculus is decidable. However, reachability for BPP is only $\mathcal{NP}$-complete [Esp95]. This is a much weaker lower bound than the *EXPSPACE*-hardness proved by Habermehl in [Hab97]. The question is now if model checking BPP with full LTL and linear-time μ-calculus is also *EXPSPACE*-complete or harder. The following theorem shows that it is at least as hard as the reachability problem for Petri nets. Note that for this result not even weak atomic predicates (of the form 'action a is enabled') are needed. Only relativised next-operators and the predicate *true* are used.

Theorem 6.2.1 *Model checking BPP with LTL is at least as hard as the reachability problem for Petri nets.*

Proof The reachability problem for Petri nets has the same complexity as the Zero-Reachability Problem [Pet81]. This is the problem, for a Petri net, if the empty marking is reachable. This problem is equivalent to the Deadlock Reachability Problem, the question if a deadlock is reachable. (The reduction is as follows: For every place add a transition that takes a token from this place and puts it back.)

Let (N, M_0) be the Petri net with initial marking M_0. The question is if a deadlock is reachable. We consider a modified deadlock reachability problem where the initial marking M_0 is not a deadlock. This is equivalent to the deadlock reachability problem without restriction. We reduce this modified deadlock reachability problem to the model checking problem for BPP and LTL. We assume w.r. that every transition in N has at least one place in its preset. Note that N is an unlabeled Petri net.

Now we construct a BPP that weakly simulates the net (N, M_0). Replace any transition t in N with preset $\{p_1, \ldots, p_{n_t}\}$ (a multiset) and postset $\{p'_1, \ldots, p'_{m_t}\}$ (a multiset) by a set of new transitions as follows. These new transitions are labeled with atomic actions. We describe them as rules in $(1, P)$-PRS notation.

$$\begin{array}{rcl}
p_1 & \xrightarrow{t_1} & \epsilon \\
p_2 & \xrightarrow{t_2} & \epsilon \\
p_3 & \xrightarrow{t_3} & \epsilon \\
& \vdots & \\
p_{n-1} & \xrightarrow{t_{n_t-1}} & \epsilon \\
p_n & \xrightarrow{t_{n_t}} & p'_1 \| \ldots \| p'_{m_t}
\end{array}$$

The t_i are new atomic actions. Let N' be the new net. All transitions in N' have exactly one place in their preset and thus N' is a communication-free net (which is equivalent to a BPP). M_0 is also a marking of N', since N' has the same places as N.

Now we define an LTL formula such that the runs of (N', M_0) that satisfy this formula are exactly those runs that faithfully simulate the behavior of (N, M_0). Let Γ be the set of all LTL-formulae Φ of the form

$$\Phi := \langle t_i \rangle \langle t_{i+1} \rangle \mathit{true}$$

for transitions t of N and $1 \leq i \leq n_t - 1$, or of the form

$$\Phi := \langle t_{n_t} \rangle \langle t'_1 \rangle \mathit{true}$$

where t and t' are transitions of N. Let A be the set of all actions t_k. A run of N' (see Def. 3.0.15) has length 0 iff it satisfies the LTL-formula

$$\Psi := \neg \left(\bigvee_{a \in A} \langle a \rangle \mathit{true} \right)$$

Then a run of the system (N', M_0) is a faithful simulation of N if it satisfies the following LTL-formula.

$$\Psi_1 := \left(\bigvee_{\Phi \in \Gamma} \Phi \right) \; w\mathcal{U} \; \left(\bigvee_{t} \langle t_{n_t} \rangle \Psi \right)$$

This formula ensures that the run of N' simulates every transition t of N in n_t steps. It also ensures that the run cannot stop during such a simulation series, but only between them.

A run is infinite if it satisfies the following formula.

$$\Psi_2 := (\neg \Psi) \, w\mathcal{U} \, \mathit{false}$$

A deadlock is not reachable in N if and only if all faithful simulation runs in N' are infinite. Thus a deadlock is reachable in N iff

$$\neg \left((N', M_0) \models (\Psi_1 \Rightarrow \Psi_2) \right)$$

■

6.3 Conclusion

Basic Parallel Processes are a weak model of concurrent computation. It can be argued that any decent model of concurrent computation should be at least as powerful as BPP. What makes them interesting is that they are a model for infinite-state concurrent systems that seems to lie just on the "border of decidability". Some problems that are undecidable for more powerful models of concurrent systems are still decidable for BPP. For example strong bisimulation equivalence [CHM93a] and weak bisimulation equivalence to a finite-state LTS [May96c] is decidable for BPP. On the other hand BPP are powerful enough to make some properties undecidable, for example language equivalence [Hir93].

Model checking BPP with most branching-time logics is undecidable. This follows from the result by Esparza and Kiehn [EK95] that model checking BPP with the logic EG is undecidable. Although Esparza and Kiehn do not mention it explicitly, this undecidability result even holds for a fixed EG formula. We briefly explain the idea how to show this.

In [EK95] the undecidability is shown by a reduction of the halting problem for Minsky 2-counter machines where the counters are initially zero, to the model checking problem for BPP and EG. However, in this construction the size of the EG formula is proportional to the size of the finite control of the counter machine. This problem can be overcome by doing the same reduction for the universal 2-counter machine, which has a fixed finite control, but whose counters initially contain arbitrary values. (Any control program can be encoded in these values.) In this case the constructed EG formula is fixed, but the initial state of the BPP represents the initial values in the counters. Thus model checking BPP with EG is undecidable, even for a fixed EG formula.

EF is the only decidable branching-time logic for BPP. It has been shown in Section 6.1, that model checking BPP with EF is *PSPACE*-complete. However, the problem is only Σ_d^p-complete for formulae whose nesting-depth is bounded by d.

Model checking BPP with linear-time logics is decidable and *EXPSPACE*-hard. As shown in Section 6.2, the problem is *EXPSPACE*-complete for the interpretation on infinite runs, but in general it depends on the reachability problem for Petri nets. The complexity for a fixed formula is an open question.

The following table shows the complexity of model checking BPP.

BPP	**general**	**fixed formula**
reachability, reachable property	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
EF	$PSPACE$-complete	$\in \Sigma_d^p$
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alternation-free modal μ-calc.	undecidable	undecidable
modal μ-calc.	undecidable	undecidable
LTL	decidable, $EXPSPACE$-hard	decidable
linear-time μ-calc.	decidable, $EXPSPACE$-hard	decidable

Chapter 7

Pushdown Processes and BPA

As shown in Section 2.3.5, pushdown processes are equivalent to (S, S)-PRS. Model checking pushdown processes has been studied in [BS94, BEM97a, Wal96a, Wal96b]. The main idea in [BEM97a] is to describe sets of states (stack contexts) of a pushdown system with finite (alternating) multi-automata. A polynomial-time/(exponential-time) algorithm is presented in [BEM97a], that, given a set of states described by an (alternating) multi-automaton, computes an (alternating) multi-automaton describing the set of all possible predecessors of these states. It takes only polynomial time to check if a given state is described (recognized) by a given alternating multi-automaton. One does not need alternation for the reachability problem and the reachable property problem. Thus they can be solved in polynomial time.

The results on the complexity of model checking pushdown processes can be classified into two groups: results on branching-time logics and results on linear-time logics.

Model checking pushdown processes with branching-time logics is quite hard. Even for the simple branching-time logic EF the model checking problem is *PSPACE*-complete [BEM97a]. It gets even worse by the fact that even for a fixed EF-formula the problem is *PSPACE*-hard in the size of the pushdown system. For the other branching-time logics the problem is even harder. Walukiewicz [Wal96a, Wal96b] has shown that model checking pushdown processes with the modal μ-calculus is *EXPTIME*-complete. Even for a fixed formula in the alternation-free modal μ-calculus the problem is *EXPTIME*-hard in the size of the pushdown process.

For the other branching-time logics EG, UB and CTL the problem is still open. No better algorithm than the exponential time algorithm of Walukiewicz [Wal96a, Wal96b] is known for them, but the known lower bounds are not as strong as for the alternation-free modal μ-calculus.

It has been shown very recently that model checking with Hennessy-Milner Logic is *PSPACE*-complete for pushdown processes [May98] (but only polynomial for every fixed formula). For EG this is the only known lower bound. For UB and CTL the known lower bound is the same as for EF, namely *PSPACE*-hardness (even for a fixed formula). Thus the exact complexity of model checking with UB and CTL is somewhere between *PSPACE* and *EXPTIME*. Altogether it can be said that model checking with branching-time logics is much more difficult for pushdown processes than for finite-state systems. For all branching-time logics (except for the full modal μ-calculus) model checking finite-state systems is polynomial, while it is at least *PSPACE*-hard for pushdown processes (see Chapter 5 for results on finite-state systems). These results show that completely automated verification of pushdown processes is very hard and thus semiautomatic methods are developed. In [BS97] Burkart and Steffen describe a sound and complete tableau system for pushdown processes and the full modal μ-calculus.

The situation is quite different for linear-time logics. Model checking pushdown processes with LTL and the linear-time μ-calculus is *EXPTIME*-complete [BEM97a]. However, the model checking problem for any fixed formula is polynomial in the size of the pushdown process. The algorithm is only exponential in the size of the formula. It follows that the problem is only slightly harder than for finite-state systems, where it is *PSPACE*-complete but polynomial for any fixed formula (see Chapter 5).

The following table summarizes the complexity results on model checking pushdown processes.

Pushdown processes	**general**	**fixed formula**
reachability, reachable property	$\in \mathcal{P}$	$\in \mathcal{P}$
EF	*PSPACE*-complete	*PSPACE*-complete
EG	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*
UB	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*, *PSPACE*-hard
CTL	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*, *PSPACE*-hard
alt.-free modal μ-calc.	*EXPTIME*-complete	*EXPTIME*-complete
modal μ-calc.	*EXPTIME*-complete	*EXPTIME*-complete
LTL	*EXPTIME*-complete	$\in \mathcal{P}$
linear-time μ-calc.	*EXPTIME*-complete	$\in \mathcal{P}$

Now we consider a subclass of pushdown processes, the context-free processes. Context-free processes were defined in Subsection 2.3.4. They are described by a sequence of symbols which can be interpreted as a stack, but unlike pushdown processes they have no finite control. They are equivalent to $(1, S)$-PRS. The algebra of context-free processes is also called *Basic Process Algebra* (BPA). So they are also called BPA-processes.

First we consider model checking context-free processes with branching-time logics. Although context-free processes are a weaker model than pushdown processes, the known upper bounds for the complexity of model checking are the same in the general case where both the system and the formula are the input. However, there is one important difference when one considers the complexity in the size of the system. Burkart and Steffen showed in [BS92b] that for any fixed formula in the alternation-free modal μ-calculus the model checking problem is only polynomial in the size of the context-free process. The algorithm is only exponential in the size of the formula. More recently, Walukiewicz [Wal96a, Wal96b] has shown that even for every fixed formula in the *full* modal μ-calculus model checking context-free process is decidable in polynomial time. Thus model checking BPA is much easier than for pushdown processes where the problem is *EXPTIME*-hard in the size of the process even for a fixed formula in the alternation-free modal μ-calculus. In practice, the formula is normally very small while the system can be very large. Thus, in practice, model checking context-free processes with branching-time logics is much easier than model checking pushdown processes.

Very recently, some lower bounds have been shown for model checking BPA with branching-time logics. Model checking BPA with Hennessy-Milner Logic is *PSPACE*-complete and model checking BPA with the alternation-free modal μ-calculus is *EXPTIME*-complete [May98]. (Of course these hardness results do not hold for any fixed formula.)

Now we consider model checking context-free processes with linear-time logics. As mentioned above, model checking pushdown systems with LTL and the linear-time μ-calculus is *EXPTIME*-complete, but only polynomial in the size of the system for any fixed formula. The only question that remained was if *EXPTIME*-hardness also holds for context-free processes.

We show now that model checking with LTL is *EXPTIME*-hard even for BPA. We generalize the proof of *EXPTIME*-hardness for pushdown systems and LTL of [BEM97a]. (This proof for pushdown systems is in the appendix of [BEM97a] and can be found in [BEM97b].)

The proof of *EXPTIME*-hardness is done by a reduction of the acceptance problem for linearly bounded alternating Turing machines [vL90]. An alternating Turing machine (ATM) is described by a tuple $(Q, \Sigma, \delta, q_0, l)$, where Q are the states of the finite control, Σ the tape symbols, δ the transition relation, q_0 the initial state and l is a function that labels states as existential, universal, accepting or rejecting. The computation of an ATM is defined just like the computation of a normal Turing machine, but the acceptance condition is more complex. Since the machine is nondeterministic, the computation can be represented as a computation tree in which the branches represent different possible computations. The states of the finite control of the ATM are assigned labels by the function l as existential, universal, accepting or rejecting. Now the states in the computation tree are labeled as accepting or rejecting by the following rules:

1. A leaf of the computation tree is labeled accepting (rejecting) if the finite control of the ATM in this state is accepting (rejecting).

2. An internal node where the finite control is labeled universal (existential) is accepting if any only if all (at least one) of its successor nodes is accepting. Otherwise it is rejecting.

3. A node is labeled undefined if the label cannot be determined by the other rules. (This only happens if there are infinite branches.)

Without loss of generality let $|\delta(q,a)| = 2$ for every universal state q and symbol a. We choose an arbitrary order on the two elements of $\delta(q,a)$ and call them the first and second successor configuration of (q,a). An ATM M is called linearly bounded if there is a constant k, such that for every word w in the language of M, M has an accepting computation that uses at most $k \cdot |w|$ space. We only consider linearly bounded ATMs and thus avoid the problem of infinite branches and undefined labels.

The acceptance problem for linearly bounded alternating Turing-machines is *EXPTIME*-complete [vL90]. Now we are ready to prove the hardness result for the model checking problem.

Theorem 7.0.1 *Model checking BPA with LTL is EXPTIME-hard.*

Proof We reduce the acceptance problem of a linearly bounded ATM to the model checking problem. Let $M = (Q, \Sigma, \delta, q_0, l)$ be the ATM, w the input word and $n := k \cdot |w|$ the length of the tape. Let M's head be over the first cell of the tape. We construct in polynomial time a BPA Δ with initial state I and a LTL-formula Φ s.t. M accepts w iff $I \models \Phi$, w.r.t. Δ.

First we describe the intuition for the construction, then we formally define the BPA Δ and finally we construct the LTL-formula that characterizes exactly the runs of the system that are faithful simulations of M.

The intuition is as follows. A configuration of M is described by words of $\Sigma^* Q \Sigma^*$ of length n. In a configuration $\alpha q \beta$, α is the content of the tape to the left of the head, q is the state of the finite control and β is the content of the tape under the head and to the right of it. The configuration is accepting if q is an accepting state. The computation of the BPA is now defined as an attempt to guess a computation of M. This is a finite or infinite tree of configurations in which every node has at most two successors. The BPA attempts (by guessing nondeterministically) to simulate a traversal of this tree in infix order. In the sequence that describes its state it stores the sequence $\#c_1\#c_2\#\ldots\#c_k$ of configurations $c_1, \ldots, c_k$ describing the path in this tree from the root to the actual configuration. Of course, most of these guesses are wrong or not even meaningful. Later we'll use the LTL formula to enforce a faithful simulation of M.

Now we define the BPA Δ. As the tree is traversed in infix order the BPA always does one of two things:

1. It outputs $\#fc\#$, where c is a configuration of M and f (meaning 'forward') is a special action and writes $\#c\#$ onto the stack. This symbolizes that we enter a node from the parent node in the computation tree of M.

2. It outputs $\#bc^r\#$, where c^r is the reverse of the configuration c and b (meaning 'backward') is a special action, and pops $\#c^r\#$ from the stack.

At the beginning of the execution the BPA outputs $\#fq_0w\#$ and writes $\#q_0w\#$ onto the stack. A computation is successful if it leads to a state where the stack is empty.

The rules describing this are as follows: (We use a shorthand notation where the rules can have strings as labels instead of single actions.)

$$\begin{array}{rcll}
I & \xrightarrow{\#f} & I'.\# & \\
I' & \xrightarrow{q_0 w} & T'_n.w^r.q_0 & \\
T'_n & \xrightarrow{\#f} & T_0.\# & \\
T_i & \xrightarrow{a} & T_{i+1}.a & \text{for } a \in \Sigma,\ 0 \leq i \leq n-2 \\
T_i & \xrightarrow{q} & T'_i.q & \text{for } q \in Q,\ 1 \leq i \leq n \\
T'_i & \xrightarrow{a} & T'_{i+1}.a & \text{for } a \in \Sigma,\ 1 \leq i \leq n-1 \\
T'_n & \xrightarrow{\#b} & \epsilon & \\
a & \xrightarrow{a} & \epsilon & \text{for } a \in \Sigma \\
q & \xrightarrow{q} & \epsilon & \text{for } q \in Q \\
\# & \xrightarrow{\#} & \epsilon & \\
\# & \xrightarrow{\#f} & T_0.\# &
\end{array}$$

It is easy to see that M accepts w if the BPA outputs a string $\#d_0\#d_1\#\ldots\#d_m\#$ such that this string is a faithful simulation of the computation of M on w and is in a state with empty stack (a deadlock) afterwards.

Now we define when a simulation is faithful and construct the LTL formula that characterizes exactly the faithful simulations. The simulation is faithful if the following properties hold for every $0 \leq i \leq m-1$

1. If $d_i = fc$ and c is an existential configuration, then $d_{i+1} = fc'$ and c' is a successor configuration of c.

2. If $d_i = fc$ and c is a universal configuration, then $d_{i+1} = fc'$ and c' is the first successor configuration of c.

3. If $d_i = bc^r$ and $d_{i+1} = fc'$, then c is a universal configuration, and c' is the second successor configuration of c.

4. The configuration in d_i is not a rejecting configuration.

5. If the configuration in d_i is an accepting configuration, then $d_{i+1} = bc^r$ for some c.

These properties can be encoded in LTL. For each symbol $a \in \Sigma \cup Q \cup \{f, b, \#\}$ we define a proposition $p_a := \bigcirc_a true$. We use the abbreviation $\bigcirc^i \Phi$ for $\bigcirc \cdots \bigcirc \Phi$ (i-times).

As defined in Subsection 3.2.2, $G\,\Phi := \Phi\; w\mathcal{U}\, false$ and $F\,\Phi := true\,\mathcal{U}\,\Phi$.

A run of the BPA is a faithful simulation of M if it satisfies the following LTL formula:

$$faithful := G((p_\# \wedge \bigcirc^{n+3} p_\#) \Rightarrow (\Psi_1 \wedge \Psi_2 \wedge \Psi_3 \wedge \Psi_4 \wedge \Psi_5))$$

where $p_\# \wedge \bigcirc^{n+3} p_\#$ expresses that the current state is a #-position different from the last, and $\Psi_1, \Psi_2, \Psi_3, \Psi_4, \Psi_5$ encode parts (1)–(5) of the properties above. It is $n+3$, because n is the length of the configuration and the symbol #, the symbol f (or b) and state q count extra.

M accepts w if there is a run of the BPA that is faithful (satisfies the LTL formula *faithful*) and leads to a state of deadlock. Let

$$\Phi := faithful \;\wedge\; F(\neg \bigcirc true)$$

Therefore M accepts w iff there is a run of the BPA that satisfies Φ. This is true iff not all runs satisfy $\neg\Phi$. Thus M accepts w iff

$$\neg(I \models \neg\Phi)$$

We only show how to construct the formula Ψ_1, since $\Psi_2 - \Psi_5$ are similar.

- "$d_i = fc$" is encoded as $\bigcirc p_f$.
- "c is an existential configuration" is encoded as

$$\bigvee_{j=1}^{n+2} \bigcirc^j (\bigvee_{q \in Q_e} p_q)$$

  where $Q_e \subseteq Q$ is the set of existential states.
- "$d_{i+1} = fc'$" is encoded as $\bigcirc^{n+3} p_f$.

- "c' is a successor configuration of c" is encoded as a disjunction of formulae, one for each possible successor configuration of c. These formulae are in turn a conjunction of formulae of the form

$$(\bigcirc^{j-1}p_{a_1} \wedge \bigcirc^{j}p_q \wedge \bigcirc^{j+1}p_{a_2}) \Rightarrow \bigcirc^{j+(n+3)+k}p_x$$

where $k \in \{-1,0,1\}$, $q \in Q$, $a \in \Sigma$ and $x \in Q \cup \Sigma$ and $3 \leq j \leq n+1$. They are determined only by the transition relation of the ATM.

For example let $a_1 = 0, a_2 = 1$, $q \in Q$ and $(q', 0, L) \in \delta(q, 1)$. Then, for every j, there would be three such formulae with different right hand sides. These right hand sides are

$$\bigcirc^{j+(n+3)-1}p_{q'} \quad \bigcirc^{j+(n+3)}\,p_1 \quad \bigcirc^{j+(n+3)+1}\,p_0$$

■

The following table summarizes the complexity results on model checking BPA.

BPA	**general**	**fixed formula**
reachability/reachable property	$\in \mathcal{P}$	$\in \mathcal{P}$
EF	*PSPACE*-complete	$\in \mathcal{P}$
EG	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
UB	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
CTL	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
alternation-free modal μ-calc.	*EXPTIME*-complete	$\in \mathcal{P}$
modal μ-calc.	*EXPTIME*-complete	$\in \mathcal{P}$
LTL	*EXPTIME*-complete	$\in \mathcal{P}$
linear-time μ-calc.	*EXPTIME*-complete	$\in \mathcal{P}$

Chapter 8

PAD and PA

The process model PAD is defined as (S, G)-PRS in the PRS-hierarchy. As described in Subsection 2.3.7, it can be used to model systems with nondeterminism, parallelism (but no synchronization) and subroutines that can return a value to their caller. A special case of PAD is PA, which is defined as $(1, G)$-PRS in Subsection 2.3.6. PA can model nondeterminism, parallelism and recursion, but, unlike in PAD, the subroutines have no effect on their caller. PA is the smallest natural common generalization of BPP and BPA.

Almost all model checking problems have the same complexity for PAD and PA and thus we consider both of them in this chapter.

Model checking with linear-time logics like LTL and the linear-time μ-calculus is undecidable for PA [BH96], even for a fixed LTL-formula. Thus it is undecidable for PAD too.

Model checking with most branching-time logics is undecidable too. This is because model checking with the logic EG is undecidable for BPP (see Chapter 6), even for a fixed EG formula. The only possible exception is the logic EF, because it is the only branching-time logic that is not stronger than EG. Here we show that model checking PAD with EF is indeed decidable. It has already been shown by the author in [May97b] that model checking PA-processes with EF is decidable. Here we prove the more general result for PAD. Note that the model checking problem for EF is $PSPACE$-hard, because it is $PSPACE$-complete for BPP (see Chapter 6 and [May96c]).

In Section 8.1 we prove that model checking PAD with the logic EF is decidable. In Section 8.2 we prove $\mathcal{NP}$-completeness of the reachability problem for PAD. In Section 8.3 we give some examples of simple verification problems for PA that can be solved in polynomial time. Section 8.4 contains other results and the general picture for PA and PAD.

8.1 Model Checking PAD with $EF^{=}_{DC}$

In this section we prove that model checking PAD with EF is decidable. We use the logic $EF^{=}_{DC}$, a generalized version of EF, because the decidability proof for it has a clearer structure and is easier to understand.

The proof is structured as follows: In Subsection 8.1.1 we define the logic $EF^{=}_{DC}$ and reduce the model checking problem to a simpler form. In Subsection 8.1.2 we show how properties can be decomposed w.r.t. sequential and parallel composition. This is used in Subsection 8.1.3 to construct a tableau system that solves the model checking problem. In Subsection 8.1.4 we show that this tableau system is sound, complete and decidable. Subsection 8.1.5 is about the complexity of the algorithm.

8.1.1 The Temporal Logic $EF^{=}_{DC}$

We use the logic $EF^{=}_{DC}$, an extended version of the logic EF. It uses strong atomic propositions of the form 'The current state is term t' and can thus express reachability. The "=" in the name stands for these strong propositions, because they express that the current state is equal to a given state t. The logic $EF^{=}_{DC}$ can also express weak constraints on sequences of actions. These constraints are called *decomposable constraints* (thus the DC in the name).

Definition 8.1.1 ($EF^{=}_{DC}$)
The syntax of the formulae is as follows:

$$\Phi ::= t \ \mid\ \neg\Phi \ \mid\ \Phi_1 \wedge \Phi_2 \ \mid\ \Diamond_C \Phi$$

where $t \in \mathcal{T}$ is a process term and C is a *decomposable constraint* (see Def. 8.1.3).

Let $\mathcal{F}$ be the set of all $EF^{=}_{DC}$-formulae. Let $\mathcal{T}$ be the set of all processes terms (as in Def. 2.1.2) in the process algebra. The denotation $[\![\Phi]\!]$ of an $EF^{=}_{DC}$-formula Φ is the set of process terms defined inductively by the following rules:

$$\begin{array}{lcl}
[\![t]\!] & := & \{t\} \\
[\![\neg\Phi]\!] & := & \mathcal{T} - [\![\Phi]\!] \\
[\![\Phi_1 \wedge \Phi_2]\!] & := & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![\Diamond_C \Phi]\!] & := & \{t \in \mathcal{T} \mid \exists t', \sigma.\ t \xrightarrow{\sigma} t' \ \wedge\ t' \in [\![\Phi]\!] \ \wedge\ C(\sigma)\}
\end{array}$$

Disjunction can be expressed by conjunction and negation.

The property $t \in [\![\Phi]\!]$ is also denoted by $t \models \Phi$.

Definition 8.1.2 For any $EF^{=}_{\mathcal{DC}}$-formula Φ let $terms(\Phi)$ be the set of process terms used in Φ as atomic propositions.

$$\begin{array}{rcl} terms(t) & := & \{t\} \\ terms(\neg\Phi) & := & terms(\Phi) \\ terms(\Phi_1 \wedge \Phi_2) & := & terms(\Phi_1) \cup terms(\Phi_2) \\ terms(\Diamond_C \Phi) & := & terms(\Phi) \end{array}$$

The logic $EF^{=}_{\mathcal{DC}}$ uses constraints on sequences of actions. These constraints are called *decomposable*, because they can be decomposed with respect to sequential and parallel composition of sequences of actions.

Definition 8.1.3 (Decomposable Constraints)
A set of decomposable constraints $\mathcal{DC}$ is a finite set of predicates on finite sequences of actions that satisfy the following conditions.

1. $\mathcal{DC}$ contains the predicates *true* (all sequences satisfy it) and *false* (no sequence satisfies it).
2. For every predicate $C \in \mathcal{DC}$ it is decidable if C is satisfiable.
3. For every $C \in \mathcal{DC}$ there is a finite index set I and a finite set of decomposable constraints $\{C_i^1, C_i^2 \in \mathcal{DC} \mid i \in I\}$ s.t.
$$\forall \sigma, \sigma_1, \sigma_2.\ \sigma_1\sigma_2 = \sigma \Rightarrow \left(C(\sigma) \iff \bigvee_{i \in I} C_i^1(\sigma_1) \wedge C_i^2(\sigma_2) \right)$$
4. For every $C \in \mathcal{DC}$ there is a finite index set I and a finite set of decomposable constraints $\{C_i' \in \mathcal{DC} \mid i \in I\}$ s.t.
$$\forall \sigma, \sigma'.\ a\sigma' = \sigma \Rightarrow \left(C(\sigma) \iff \bigvee_{i \in I} C_i'(\sigma') \right)$$
5. For every $C \in \mathcal{DC}$ there is a finite index set I and a finite set of decomposable constraints $\{C_i^1, C_i^2 \in \mathcal{DC} \mid i \in I\}$ s.t.
$$\forall \sigma, \sigma_1, \sigma_2. \forall a \in Act.\ \sigma_1 a \sigma_2 = \sigma \Rightarrow \left(C(\sigma) \iff \bigvee_{i \in I} C_i^1(\sigma_1) \wedge C_i^2(\sigma_2) \right)$$

6. For every $C \in \mathcal{DC}$ there is a finite index set I and a finite set of decomposable constraints $\{C_i^1, C_i^2 \in \mathcal{DC} \mid i \in I\}$ s.t.

$$\forall \sigma_1, \sigma_2. \left((\exists \sigma \in \mathit{interleave}(\sigma_1, \sigma_2).\, C(\sigma)) \iff \bigvee_{i \in I} (C_i^1(\sigma_1) \wedge C_i^2(\sigma_2)) \right)$$

$\sigma \in \mathit{interleave}(\sigma_1, \sigma_2)$ means that σ is an arbitrary interleaving of σ_1 and σ_2. The formal definition of the function *interleave* is as follows: Let λ be the empty sequence.

$$\begin{array}{rcl} \mathit{interleave}(\lambda, \sigma) & := & \{\sigma\} \\ \mathit{interleave}(\sigma, \lambda) & := & \{\sigma\} \\ \mathit{interleave}(a_1\sigma_1, a_2\sigma_2) & := & \begin{array}{l} \{a_1\sigma \mid \sigma \in \mathit{interleave}(\sigma_1, a_2\sigma_2)\} \cup \\ \{a_2\sigma \mid \sigma \in \mathit{interleave}(a_1\sigma_1, \sigma_2)\} \end{array} \end{array}$$

Lemma 8.1.4 *If $\mathcal{DC}$ is a set of decomposable constraints, then the closure $\mathcal{DC}'$ of $\mathcal{DC}$ under the boolean operations of conjunction and disjunction is also a set of decomposable constraints.*

Proof The formulae in $\mathcal{DC}'$ can be transformed into disjunctive normal form, such that the formulae in $\mathcal{DC}$ are the atomic formulae. Since $\mathcal{DC}$ is finite, $\mathcal{DC}'$ is finite too. ∎

Remark 8.1.5 *A set of decomposable constraints need not be closed under negation.*

Now we give an example for a set of decomposable constraints. Let $A \subset Act$, be a finite set of atomic actions. For any $a \in A$ let $\#_a(\sigma)$ be the number of occurrences of action a in σ. For $u, v \in \mathbb{N}$ let $[u]_v$ denote u modulo v. We define the following constraints:

1. $\mathit{length}(\sigma) \geq i$ or $\mathit{length}(\sigma) \leq i$ for all $i \leq k$ for some fixed constant k.

2. $\#_a(\sigma) \geq i$ or $\#_a(\sigma) \leq i$ for all $i \leq n$ for some fixed constant n.

3. $[\#_a(\sigma)]_k = i$ for all $i, k \leq m$ for some fixed constant m.

4. $\mathit{first}(\sigma) = a$ for any action $a \in A$.

For any choice of A, k, n, m let $\mathcal{C}_{A,k,n,m}$ denote the closure of the set of these constraints under conjunction and disjunction.

Lemma 8.1.6 *For any A, k, n, m, the set $\mathcal{C}_{A,k,n,m}$ is a set of decomposable constraints. It is even closed under negation.*

Proof Directly from the definitions. ■

Example 8.1.7 The constraint $[\#_a(\sigma)]_2 = 0$ expresses that the number of occurrences of action a in σ is even. Let $\sigma \in \mathit{interleave}(\sigma_1, \sigma_2)$ be an interleaving of two sequences. Then the number of occurrences of the action a in σ is even iff it is either even in both σ_1 and σ_2 or odd in both σ_1 and σ_2. This can be expressed by the following decomposition.

$$\begin{array}{rcl} [\#_a(\sigma)]_2 = 0 & \iff & ([\#_a(\sigma_1)]_2 = 0 \ \wedge \ [\#_a(\sigma_2)]_2 = 0) \ \vee \\ & & ([\#_a(\sigma_1)]_2 = 1 \ \wedge \ [\#_a(\sigma_2)]_2 = 1) \end{array}$$

We use these constraints to show that the usual definition of EF is a fragment of $EF^{=}_{DC}$. The usual $\Diamond$ is just $\Diamond_{true}$. The normal one-step nexttime operator EX is often denoted by $\langle a \rangle$ and defined by

$$[\![\langle a \rangle \Phi]\!] := \{t \mid \exists t \xrightarrow{a} t' \in [\![\Phi]\!]\}$$

It is clear that $\langle a \rangle = \Diamond_C$ with $C := [\mathit{first}(\sigma) = a \wedge \mathit{length}(\sigma) = 1]$. The normal version of EF also does not have atomic propositions t (meaning that the state is equal to t; see Def. 8.1.1), but propositions "a" (meaning that the atomic action a is enabled). This can be expressed by $\langle a \rangle true$, where $true = t \vee \neg t$ for any term t.

It is also possible to express the modal operator $\Box$ (meaning 'always') by defining $\Box_C := \neg \Diamond_C \neg$. $\Box_C \Phi$ then means that Φ holds in all states that are reachable via a sequence of actions σ s.t. $C(\sigma)$.

Definition 8.1.8 The *nesting-depth* $nd(\Phi)$ of an $EF^{=}_{DC}$-formula Φ is defined by

$$\begin{array}{rcl} nd(t) & := & 0 \\ nd(\neg\Phi) & := & nd(\Phi) \\ nd(\Phi_1 \wedge \Phi_2) & := & max\{nd(\Phi_1), nd(\Phi_2)\} \\ nd(\Diamond_C \Phi) & := & nd(\Phi) + 1 \end{array}$$

Definition 8.1.9 $\mathcal{F}_d \subset \mathcal{F}$ is defined as the set of all $EF^{=}_{DC}$-formulae with a nesting-depth of modal operators $\Diamond_C$ of at most d.

$$\mathcal{F}_d := \{\Phi \in \mathcal{F} \mid nd(\Phi) \leq d\}$$

It follows that formulae in $\mathcal{F}_0$ contain no modal operators.

In order to simplify the notation we use some abbreviations:
Let $T = \{t_1, \dots, t_n\} \subseteq \mathcal{T}$ be a finite set of process terms, then

$$t \models -T \ :\Longleftrightarrow \ t \models \neg t_1 \wedge \cdots \wedge \neg t_n$$

For reasons of symmetry we also define

$$t \models T :\Longleftrightarrow \ t \models t_1 \wedge \cdots \wedge t_n$$

Of course this cannot be true if $n \geq 2$.

Definition 8.1.10 We define a subset $\mathcal{F}^c_d \subset \mathcal{F}_d$ of formulae that do not contain disjunction. Thus the formulae in $\mathcal{F}^c_d$ are called *conjunctive formulae.* $\mathcal{F}^c_d$ is defined as the minimal set of formulae Φ_d that are defined by the following grammar.

$$\Phi_0 = T^+ \wedge -T^-$$

for every finite $T^+, T^- \subset \mathcal{T}$ and

$$\Phi_d = T^+ \wedge -T^- \mid \Phi_d \wedge \Diamond_C \Phi_{d-1} \mid \Phi_d \wedge \neg\Diamond_C \Phi_{d-1}$$

for every finite $T^+, T^- \subset \mathcal{T}$ and every decomposable constraint C and every $\Phi_{d-1} \in \mathcal{F}^c_{d-1}$.

It follows that every formula in $\mathcal{F}^c_d$ has the form

$$T^+ \wedge -T^- \wedge \bigwedge_{i \in I} \Diamond_{C_i} \Psi_i \wedge \bigwedge_{j \in J} \neg\Diamond_{D_j} \Upsilon_j$$

where $T^+, T^- \subset \mathcal{T}$, and C_i, D_j are decomposable constraints and $\Psi_i \in \mathcal{F}^c_{d-1}$ and $\Upsilon_j \in \mathcal{F}^c_{d-1}$.

A formula Φ is in *normal form* if $\Phi = \bigvee_{i \in I} \Diamond_{C_i} \Psi_i$ s.t. the Ψ_i are conjunctive formulae.

Lemma 8.1.11 *Any $EF^{=}_{DC}$-formula $\Diamond_C \Phi$ is equivalent to a formula in normal form.*

Proof By induction on the nesting-depth d of modal operators in Φ. The important property here is that $\Diamond_C(\Phi_1 \vee \Phi_2) = \Diamond_C \Phi_1 \vee \Diamond_C \Phi_2$. We transform the subformulae into disjunctive normal form, and then push the disjunctions outwards. ∎

Lemma 8.1.12 *Every model checking problem for $EF^{=}_{DC}$ is decidable iff it is decidable for all formulae $\Diamond_C \Phi$ with $\Phi \in \bigcup_{d \in \mathbb{N}} \mathcal{F}^c_d$.*

Proof If it is decidable for formulae of the form $\Diamond_C \Phi$ with $\Phi \in \mathcal{F}^c_d$, then it is decidable for formulae in normal form and thus by Lemma 8.1.11 for all formulae of the form $\Diamond_C \Psi$, with $\Psi \in \mathcal{F}$. Simple boolean operations yield the decidability of the whole model checking problem. The other direction is trivial. ∎

8.1.2 Decomposition

The key to the construction of the tableau system in Subsection 8.1.3 is that properties of the form $t_1.t_2 \models \Diamond_C \Phi$ or $t_1 \| t_2 \models \Diamond_C \Phi$ can be decomposed into properties of t_1 and properties of t_2. First we give a small example how this is done and then we do it in general.

Example 8.1.13 We show how to do the decomposition for the following simple formula of nesting-depth two:

$$\Phi := \Diamond(\neg u \ \wedge\ \Diamond(v) \ \wedge\ \neg\Diamond(w))$$

where u, v, w are process terms. No decomposable constraints are used, except for the constraint *true* ($\Diamond_{true} = \Diamond$). This formula means that there is a reachable state different from u, s.t. from this state the state v is reachable, but the state w is not reachable.

Let t_1, t_2 be process terms. Then the property

$$t_1 \| t_2 \models \Diamond(\neg u \ \wedge\ \Diamond(v) \ \wedge\ \neg\Diamond(w))$$

is equivalent to

$$\begin{array}{l}
(t_1 \models \Diamond(\epsilon) \ \wedge\ t_2 \models \Phi) \ \vee \\
(t_1 \models \Phi \ \wedge\ t_2 \models \Diamond(\epsilon)) \ \vee \\
\exists \sigma_1, \sigma_2, t_1', t_2'.\ \ t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \ \wedge \\
\qquad\qquad\qquad \displaystyle\bigwedge_{\alpha_1 \| \alpha_2 = u} (t_1' \neq \alpha_1 \vee t_2' \neq \alpha_2) \ \wedge\ t_1' \| t_2' \models \Diamond(v) \ \wedge\ t_1' \| t_2' \models \neg\Diamond(w)
\end{array}$$

where α_1, α_2 are process terms. This is equivalent to

$$\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\exists \sigma_1, \sigma_2, t_1', t_2'. \quad t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \ \wedge \bigwedge_{\alpha_1 \| \alpha_2 = u} (t_1' \neq \alpha_1 \vee t_2' \neq \alpha_2) \ \wedge \\
&\qquad \bigvee_{\beta_1 \| \beta_2 = v} (t_1' \models \Diamond(\beta_1) \ \wedge \ t_2' \models \Diamond(\beta_2)) \ \wedge \\
&\qquad \neg \left(\bigvee_{\gamma_1 \| \gamma_2 = w} t_1' \models \Diamond(\gamma_1) \wedge t_2' \models \Diamond(\gamma_2) \right)
\end{aligned}$$

This is equivalent to

$$\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\exists \sigma_1, \sigma_2, t_1', t_2'. \quad t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \wedge \bigwedge_{\alpha_1 \| \alpha_2 = u} (t_1' \neq \alpha_1 \vee t_2' \neq \alpha_2) \ \wedge \\
&\qquad \bigvee_{\beta_1 \| \beta_2 = v} (t_1' \models \Diamond(\beta_1) \ \wedge \ t_2' \models \Diamond(\beta_2)) \ \wedge \\
&\qquad \bigwedge_{\gamma_1 \| \gamma_2 = w} (t_1' \models \neg\Diamond(\gamma_1) \vee t_2' \models \neg\Diamond(\gamma_2))
\end{aligned}$$

Now we transform this expression into disjunctive normal form. We define the set F of all functions f that assign to every pair (α_1, α_2) s.t. $\alpha_1 \| \alpha_2 = u$, a value in $\{1, 2\}$. For every $f \in F$ let $A_f^1 := \{\alpha_1 \mid f((\alpha_1, \alpha_2)) = 1\}$ and $A_f^2 := \{\alpha_2 \mid f((\alpha_1, \alpha_2)) = 2\}$. Then the expression is equivalent to

$$\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\exists \sigma_1, \sigma_2, t_1', t_2'. \quad t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \wedge \bigvee_{f \in F} (t_1' \notin A_f^1 \ \wedge \ t_2' \notin A_f^2) \ \wedge \\
&\qquad \bigvee_{\beta_1 \| \beta_2 = v} (t_1' \models \Diamond(\beta_1) \ \wedge \ t_2' \models \Diamond(\beta_2)) \ \wedge \\
&\qquad \bigwedge_{\gamma_1 \| \gamma_2 = w} (t_1' \models \neg\Diamond(\gamma_1) \vee t_2' \models \neg\Diamond(\gamma_2))
\end{aligned}$$

In the same way we define the set G of all functions g that assign to every pair (γ_1, γ_2) s.t. $\gamma_1 \| \gamma_2 = w$, a value in $\{1, 2\}$. For every $g \in G$ let $B_g^1 := \{\gamma_1 \mid g((\gamma_1, \gamma_2)) = 1\}$ and $B_g^2 := \{\gamma_2 \mid g((\gamma_1, \gamma_2)) = 2\}$.

Then the expression is equivalent to

$$
\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\exists \sigma_1, \sigma_2, t_1', t_2'. \quad t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \ \wedge \bigvee_{f \in F} (t_1' \models -A_f^1 \ \wedge \ t_2' \models -A_f^2) \ \wedge \\
&\qquad \bigvee_{\beta_1 || \beta_2 = v} (t_1' \models \Diamond(\beta_1) \ \wedge \ t_2' \models \Diamond(\beta_2)) \ \wedge \\
&\qquad \bigvee_{g \in G} \left(\bigwedge_{\gamma_1 \in B_g^1} t_1' \models \neg\Diamond(\gamma_1) \ \wedge \ \bigwedge_{\gamma_2 \in B_g^2} t_2' \models \neg\Diamond(\gamma_2) \right)
\end{aligned}
$$

This is equivalent to

$$
\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\exists \sigma_1, \sigma_2, t_1', t_2'. \ t_1 \xrightarrow{\sigma_1} t_1' \wedge t_2 \xrightarrow{\sigma_2} t_2' \ \wedge \\
&\bigvee_{f \in F, \beta_1 || \beta_2 = v, g \in G} \quad t_1' \models -A_f^1 \wedge t_1' \models \Diamond(\beta_1) \wedge \bigwedge_{\gamma_1 \in B_g^1} t_1' \models \neg\Diamond(\gamma_1) \wedge \\
&\qquad\qquad t_2' \models -A_f^2 \wedge t_2' \models \Diamond(\beta_2) \wedge \bigwedge_{\gamma_2 \in B_g^2} t_2' \models \neg\Diamond(\gamma_2)
\end{aligned}
$$

Finally, this is equivalent to

$$
\begin{aligned}
&(t_1 \models \Diamond(\epsilon) \ \wedge \ t_2 \models \Phi) \vee \\
&(t_1 \models \Phi \ \wedge \ t_2 \models \Diamond(\epsilon)) \vee \\
&\bigvee_{f \in F, \beta_1 || \beta_2 = v, g \in G} \quad t_1 \models \Diamond(-A_f^1 \wedge \Diamond(\beta_1) \wedge \bigwedge_{\gamma_1 \in B_g^1} \neg\Diamond(\gamma_1)) \ \wedge \\
&\qquad\qquad t_2 \models \Diamond(-A_f^2 \wedge \Diamond(\beta_2) \wedge \bigwedge_{\gamma_2 \in B_g^2} \neg\Diamond(\gamma_2))
\end{aligned}
$$

This is a boolean combination of properties of t_1 and properties of t_2.

Now we show how the decomposition is done in the general case. In order to simplify the presentation, we define the following sets of expressions. Let $\mathcal{DC}$ be a set of decomposable constraints, $T \subset \mathcal{T}$ a finite set of process terms and $d \in \mathbb{N}$.

$$\begin{array}{rcl}
\mathit{Cform}(d,T,\mathcal{DC}) & := & \left\{ \begin{array}{l} \left(\bigwedge_{i\in I} t_i \models \Diamond_{C_i}\Phi_i \ \wedge \ \bigwedge_{i\in J} t'_j \models \neg\Diamond_{D_j}\Psi_j\right) \ \mid \\ \forall i,j.\ t_i, t'_j \in T, C_i, D_j \in \mathcal{DC}, \Phi_i \in \mathcal{F}^c_d, \Psi_j \in \mathcal{F}^c_{d-1} \end{array} \right\} \\
\mathit{Cform}'(d,T,\mathcal{DC}) & := & \text{like } \mathit{Cform}(d,T,\mathcal{DC}), \text{ except that } \Psi_j \in \mathcal{F}^c_d \\
\mathit{Dform}(d,T,\mathcal{DC}) & := & \left\{ \bigvee_{i\in I} F_i \ \mid \ F_i \in \mathit{Cform}(d,T,\mathcal{DC}) \right\}
\end{array}$$

The next two lemmas show the decomposition of properties for sequential composition. The general idea is that properties of the form $t_1.t_2 \models \Diamond_C \Phi$ are decomposed into properties of t_1 and properties of t_2. However, the details are more complex. It does not always suffice to use properties of t_1 and properties of t_2, but sometimes also properties of other terms are needed. These other terms are the terms that occur in Φ as atomic propositions and the terms that occur in the rules of the PAD-process. Fortunately, these are only finitely many.

We defined that sequential composition is left-associative, so if we write $t_1.t_2$, then the term t_2 is either a single variable or a parallel composition. The following lemma describes the decomposition for the case that t_2 is a single variable.

Lemma 8.1.14 *Let t be a process term, X a process variable, Δ a PAD, Φ a formula in $\mathcal{F}^c_d$ that contains only constraints from a set $\mathcal{DC}$ of decomposable constraints and $C \in \mathcal{DC}$. Let $T := \{\epsilon, t, X\} \cup \mathit{terms}(\Phi) \cup \{r \mid (l \xrightarrow{a} r) \in \Delta\}$*

Then an expression $F \in \mathit{Dform}(d,T,\mathcal{DC})$ can be effectively constructed s.t.

$$t.X \models \Diamond_C \Phi \iff F$$

Proof by induction on d.

$\Phi = (T^+ \wedge - T^- \wedge \bigwedge_{i\in I} \Diamond_{C_i}\Phi_i \wedge \bigwedge_{j\in J} \neg\Diamond_{D_j}\Psi_j)$ for some $T^+, T^- \subset \mathcal{T}$, $\Phi_i, \Psi_j \in \mathcal{F}^c_{d-1}$ and $C_i, D_j \in \mathcal{DC}$ decomposable constraints. In the base case $d = 0$ the sets I and J are empty and we solve the problem without referring to the induction hypothesis.

If $|T^+| \geq 2$ then $t.X \models \Diamond_C \Phi$ is equivalent to *false*.

If $|T^+| = 1$ s.t. the term in T^+ is not $t'.X$ for some t', then $t.X \models \Diamond_C \Phi$ is equivalent to

$$\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i} \Phi)$$
$$\vee$$
$$\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j} \Phi)$$

where the C_k^i, D_k^j are the decompositions of C as defined in Def. 8.1.3 (cases 3 and 5). This expression is the F that we are looking for. It is in $Dform(d, T, \mathcal{DC})$.

Now we consider the case that $T^+ = \{u.X\}$ for some term u. If $u.X \in T^-$ then $t.X \models \Diamond_C \Phi$ is equivalent to *false*. Otherwise $t.X \models \Diamond_C \Phi$ is equivalent to

$$\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i} \Phi)$$
$$\vee$$
$$\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j} \Phi)$$
$$\vee$$
$$t \models \Diamond_C(u) \wedge \bigwedge_{i \in I} u.X \models \Diamond_{C_i} \Phi_i \wedge \bigwedge_{j \in J} u.X \models \neg \Diamond_{D_j} \Psi_j$$

where the C_k^i, D_k^j are the decompositions of C as defined in Def. 8.1.3 (cases 3 and 5). This is the expression F that we are looking for. It is in $Dform(d, T, \mathcal{DC})$.

Now we consider the case that $T^+ = \{\}$. Then $t.X \models \Diamond_C \Phi$ is equivalent to

$$\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i} \Phi)$$
$$\vee$$
$$\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j} \Phi)$$
$$\vee$$
$$\exists \sigma, t'. \left(\begin{array}{l} t \xrightarrow{\sigma} t' \wedge C(\sigma) \wedge (\forall (\alpha.X) \in T^-.\ t' \neq \alpha) \wedge \\ \bigwedge_{i \in I} t'.X \models \Diamond_{C_i} \Phi_i \wedge \bigwedge_{j \in J} t'.X \models \neg \Diamond_{D_j} \Psi_j \end{array} \right)$$

where the C_k^i, D_k^j are the decompositions of C as defined in Def. 8.1.3 (cases 4 and 5).

If $d = 0$ then $I = J = \{\}$ and the induction hypothesis is not needed. If $d > 0$ then by induction hypothesis there are expressions $F_i, G_j \in \mathit{Dform}(d-1, (T - \{t\}) \cup \{t'\}, \mathcal{DC})$ s.t. the above expression is equivalent to

$$\begin{gathered}
\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i}\Phi) \\
\vee \\
\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j}\Phi) \\
\vee \\
\exists \sigma, t'.\ t \xrightarrow{\sigma} t' \wedge C(\sigma) \wedge (\forall (\alpha.X) \in T^-.\ t' \neq \alpha)\ \wedge \bigwedge_{i \in I} F_i \wedge \bigwedge_{j \in J} \neg G_j
\end{gathered}$$

By transformation to disjunctive normal form there are finite index sets K, N_k, N_k', M_k and formulae $\Phi_i', \Psi_i' \in \mathcal{F}^c_{d-1}$ and decomposable constraints $E_i, E_i' \in \mathcal{DC}$ and $H_j \in \mathit{Cform}'(d-1, T - \{t\}, \mathcal{DC})$ s.t. the above expression is equivalent to

$$\begin{gathered}
\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i}\Phi) \\
\vee \\
\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j}\Phi) \\
\vee \\
\exists \sigma, t'. \left(\begin{array}{l} t \xrightarrow{\sigma} t' \wedge C(\sigma) \wedge (\forall (\alpha.X) \in T^-.\ t' \neq \alpha) \wedge \\ \bigvee_{k \in K} \left[\bigwedge_{i \in N_k} t' \models \Diamond_{E_i}\Phi_i' \bigwedge_{i \in N_k'} t' \models \neg\Diamond_{E_i'}\Psi_i' \bigwedge_{j \in M_k} H_j \right] \end{array} \right)
\end{gathered}$$

Note that the expressions H_j do not contain the terms t or t'. This is equivalent to

$$\begin{gathered}
\bigvee_{i \in I} (t \models \Diamond_{C_1^i}(\epsilon) \wedge X \models \Diamond_{C_2^i}\Phi) \\
\vee \\
\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} (t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j}\Phi) \\
\vee \\
\bigvee_{k \in K} \left[t \models \Diamond_C \left(-\{\alpha \mid \alpha.X \in T^-\} \wedge \bigwedge_{i \in N_k} \Diamond_{E_i}\Phi_i' \bigwedge_{i \in N_k'} \neg\Diamond_{E_i'}\Psi_i' \right) \wedge \bigwedge_{j \in M_k} H_j \right]
\end{gathered}$$

This is the expression F that we are looking for. It is in $\mathit{Dform}(d, T, \mathcal{DC})$. ∎

The following lemma does the same decomposition for the case that the second component in the sequential composition is itself a parallel composition.

Lemma 8.1.15 *Let t_1, t_2, t_3 be process terms, Δ a PAD, Φ a formula in $\mathcal{F}_d^c$ that contains only constraints from a set $\mathcal{DC}$ of decomposable constraints and $C \in \mathcal{DC}$. Let $T := \{\epsilon, t_1, t_2 \| t_3\} \cup terms(\Phi) \cup \{r \mid (l \xrightarrow{a} r) \in \Delta\}$*

Then an expression $F \in Dform(d, T, \mathcal{DC})$ can be effectively constructed s.t.

$$t_1.(t_2 \| t_3) \models \Diamond_C \Phi \iff F$$

Proof The proof is similar to Lemma 8.1.14 with only the following differences:

1. Leave out the part
$$\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} t \models \Diamond_{D_1^j}(l) \wedge r \models \Diamond_{D_2^j} \Phi$$
of the disjunction, and

2. Substitute $(t_2 \| t_3)$ for X everywhere. ■

Now we show an analogous property for parallel composition.

Lemma 8.1.16 *Let t_1, t_2 be process terms, Δ a PAD, Φ a formula in $\mathcal{F}_d^c$ that contains only constraints from a set $\mathcal{DC}$ of decomposable constraints and $C \in \mathcal{DC}$. Let $T := \{\epsilon, t_1, t_2\} \cup terms(\Phi)$.*

Then an expression $F \in Dform(d, T, \mathcal{DC})$ can be effectively constructed s.t.

$$t_1 \| t_2 \models \Diamond_C \Phi \iff F$$

Proof by induction on d.
Φ has the form $(T^+ \wedge -T^- \wedge \bigwedge_{i \in I} \Diamond_{C_i} \Phi_i \wedge \bigwedge_{j \in J} \neg \Diamond_{D_j} \Psi_j)$ for some $T^+, T^- \subset \mathcal{T}$ and $\Phi_i, \Psi_j \in \mathcal{F}_{d-1}^c$.

If $|T^+| \geq 2$ then $\Diamond_C \Phi$ is equivalent to *false*.

Now we consider the case that $T^+ = \{t\}$ for some term t. If $t \in T^-$ then $t_1 \| t_2 \models \Diamond_C \Phi$ is equivalent to *false*. Otherwise it is equivalent to

$$\bigvee_{k \in K} \left(\begin{array}{l} (t_1 \models \Diamond_{C_k'}(\epsilon) \wedge t_2 \models \Diamond_{C_k''} \Phi) \vee \\ (t_2 \models \Diamond_{C_k'}(\epsilon) \wedge t_1 \models \Diamond_{C_k''} \Phi) \end{array} \right)$$
$$\vee$$
$$\bigvee_{l \in L} \bigvee_{\alpha_1 \| \alpha_2 = t} \left(\begin{array}{l} t_1 \models \Diamond_{D_l'}(\alpha_1) \wedge t_2 \models \Diamond_{D_l''}(\alpha_2) \wedge \\ \bigwedge_{i \in I} t \models \Diamond_{C_i} \Phi_i \wedge \bigwedge_{j \in J} t \models \neg \Diamond_{D_j} \Psi_j \end{array} \right)$$

where the C'_k, C''_k, D'_l, D''_l are the decompositions of C as defined in Def. 8.1.3 (case 6). This is the F that we are looking for. It is in $Dform(d, T, \mathcal{DC})$.

Now we consider the case that $T^+ = \{\}$. Then $t_1 \| t_2 \models \Diamond_C \Phi$ is equivalent to

$$\bigvee_{k\in K} \left(\begin{array}{l} (t_1 \models \Diamond_{C'_k}(\epsilon) \wedge t_2 \models \Diamond_{C''_k}\Phi) \vee \\ (t_2 \models \Diamond_{C'_k}(\epsilon) \wedge t_1 \models \Diamond_{C''_k}\Phi) \end{array} \right)$$

$$\vee$$

$$\bigvee_{l\in L} \exists \sigma_1, \sigma_2, t'_1, t'_2. \left(\begin{array}{l} t_1 \xrightarrow{\sigma_1} t'_1 \wedge t_2 \xrightarrow{\sigma_2} t'_2 \wedge D'_l(\sigma_1) \wedge D''_l(\sigma_2) \wedge \\ \displaystyle\bigwedge_{\alpha_1\|\alpha_2\in T^-} (t'_1 \neq \alpha_1 \vee t'_2 \neq \alpha_2) \wedge \\ \displaystyle\bigwedge_{i\in I} t'_1\|t'_2 \models \Diamond_{C_i}\Phi_i \wedge \bigwedge_{j\in J} t'_1\|t'_2 \models \neg\Diamond_{D_j}\Psi_j \end{array} \right)$$

where the C'_k, C''_k, D'_l, D''_l are the decompositions of C as defined in Def. 8.1.3 (case 6). In the base case $d = 0$ we have $I = J = \{\}$ and don't need the induction hypothesis. For $d > 0$, by induction hypothesis, there are formulae $F_i, G_j \in Dform(d-1, \{t'_1, t'_2\} \cup terms(\Phi), \mathcal{DC})$ such that $t'_1\|t'_2 \models \Diamond_{C_i}\Phi_i \iff F_i$ and $t'_1\|t'_2 \models \Diamond_{D_j}\Psi_j \iff G_j$. Now we transform the expression into disjunctive normal form. We define the set *Func* of all functions

$$f : \{(\alpha_1, \alpha_2) \mid \alpha_1\|\alpha_2 \in T^-\} \mapsto \{1, 2\}$$

that assign to every pair (α_1, α_2) s.t. $\alpha_1\|\alpha_2 \in T^-$, a value in $\{1, 2\}$. For every $f \in Func$ let $A^1_f := \{\alpha_1 \mid f((\alpha_1, \alpha_2)) = 1\}$ and $A^2_f := \{\alpha_2 \mid f((\alpha_1, \alpha_2)) = 2\}$. Then the expression is equivalent to

$$\bigvee_{k\in K} \left(\begin{array}{l} (t_1 \models \Diamond_{C'_k}(\epsilon) \wedge t_2 \models \Diamond_{C''_k}\Phi) \vee \\ (t_2 \models \Diamond_{C'_k}(\epsilon) \wedge t_1 \models \Diamond_{C''_k}\Phi) \end{array} \right)$$

$$\vee$$

$$\bigvee_{l\in L} \exists \sigma_1, \sigma_2, t'_1, t'_2. \left(\begin{array}{l} t_1 \xrightarrow{\sigma_1} t'_1 \wedge t_2 \xrightarrow{\sigma_2} t'_2 \wedge D'_l(\sigma_1) \wedge D''_l(\sigma_2) \wedge \\ \displaystyle\bigvee_{f\in Func} (t'_1 \notin A^1_f \wedge t'_2 \notin A^2_f) \wedge \\ \displaystyle\bigwedge_{i\in I} F_i \wedge \bigwedge_{j\in J} \neg G_j \end{array} \right)$$

By transformation to disjunctive normal form there must be finite index sets O and $M(o), M'(o), N(o), N'(o)$ for every $o \in O$ and formulae $\Phi'_n, \Psi'_{n'}, \Phi''_m, \Psi''_{m'} \in \mathcal{F}^c_{d-1}$ and decomposable constraints $E_n, E'_{n'}, F_m, F'_{m'} \in \mathcal{DC}$ s.t. the condition is equivalent to

$$\bigvee_{k\in K}\left(\begin{array}{l}(t_1 \models \Diamond_{C'_k}(\epsilon) \wedge t_2 \models \Diamond_{C''_k}\Phi) \vee \\ (t_2 \models \Diamond_{C'_k}(\epsilon) \wedge t_1 \models \Diamond_{C''_k}\Phi)\end{array}\right)$$

$$\vee$$

$$\bigvee_{l\in L} \exists \sigma_1,\sigma_2,t'_1,t'_2. \left[\begin{array}{l} t_1 \xrightarrow{\sigma_1} t'_1 \wedge t_2 \xrightarrow{\sigma_2} t'_2 \wedge D'_l(\sigma_1) \wedge D''_l(\sigma_2) \wedge \\ \bigvee_{f\in Func} (t'_1 \notin A^1_f \wedge t'_2 \notin A^2_f) \wedge \\ \bigvee_{o\in O}\left(\begin{array}{l} \bigwedge_{n\in N(o)} t'_1 \models \Diamond_{E_n}\Phi'_n \bigwedge_{n'\in N'(o)} t'_1 \models \neg\Diamond_{E'_{n'}}\Psi'_{n'} \\ \bigwedge_{m\in M(o)} t'_2 \models \Diamond_{F_m}\Phi''_m \bigwedge_{m'\in M'(o)} t'_2 \models \neg\Diamond_{F'_{m'}}\Psi''_{m'} \end{array}\right)\end{array}\right]$$

This is equivalent to

$$\bigvee_{k\in K}\left(\begin{array}{l}(t_1 \models \Diamond_{C'_k}(\epsilon) \wedge t_2 \models \Diamond_{C''_k}\Phi) \vee \\ (t_2 \models \Diamond_{C'_k}(\epsilon) \wedge t_1 \models \Diamond_{C''_k}\Phi)\end{array}\right)$$

$$\vee$$

$$\bigvee_{l\in L, f\in Func, o\in O}\left[\begin{array}{l} t_1 \models \Diamond_{D'_l}\left(-A^1_f \wedge \bigwedge_{n\in N(o)} \Diamond_{E_n}\Phi'_n \bigwedge_{n'\in N'(o)} \neg\Diamond_{E'_{n'}}\Psi'_{n'}\right) \wedge \\ t_2 \models \Diamond_{D''_l}\left(-A^2_f \wedge \bigwedge_{m\in M(o)} \Diamond_{F_m}\Phi''_m \bigwedge_{m'\in M'(o)} \neg\Diamond_{F'_{m'}}\Psi''_{m'}\right)\end{array}\right]$$

This is the expression F that we are looking for. It is in $\mathit{Dform}(d, T, \mathcal{DC})$. ∎

8.1.3 The Tableau System

We show the decidability of the model checking problem for PAD and $EF^{\equiv}_{DC}$ by induction on the nesting-depth d of the formula. We describe a tableau system that solves the model checking problem for formulae $\Diamond_C\Phi$ with $\Phi \in \mathcal{F}^c_d$ under the condition that we can already solve the problem for formulae $\Diamond_C\Psi$ with $\Psi \in \mathcal{F}^c_{d-1}$. This is because we use properties of the form $t' \models \Diamond_C\Psi$ for $\Psi \in \mathcal{F}^c_{d-1}$ as side conditions in the construction of the tableau. By induction hypothesis we can assume this. In the base case of $d = 0$ the condition is trivially satisfied, as $\mathcal{F}^c_{-1} = \{\}$. (See Chapter 4 for an introduction to tableau systems.)

Every node in the tableau is a set of expressions of the form $t \vdash \Phi$, where t is a process term and Φ an $EF^{\equiv}_{DC}$-formula. We use the symbol $\vdash$ in the tableau instead of $\models$. The expression $t \vdash \Phi$ means that one attempts to prove the property $t \models \Phi$. The meaning of $t \models \Phi$ is defined semantically (Def. 8.1.1). The sets of expressions that form the tableau nodes are denoted by Γ and interpreted as sets of subgoals that should be proved. These subgoals are interpreted conjunctively. The branches in the tableau are interpreted disjunctively, so the tableau is successful iff there is at least one successful branch. Every branch in the tableau can be seen as an attempt to construct a proof.

The following tableau rules are meant to be applied to a problem of the form $t \models \Diamond_C \Phi$ with $\Phi \in \mathcal{F}^c_d$. In the rules Induct1–Induct4 we apply the induction hypothesis that we can already solve the problem for formulae of a smaller nesting-depth.

SEQ1 $\quad \dfrac{\{t.X \vdash \Diamond_C \Phi\} \cup \Gamma}{\{F\} \cup \Gamma}$ where F is from Lemma 8.1.14

SEQ2 $\quad \dfrac{\{t_1.(t_2 \| t_3) \vdash \Diamond_C \Phi\} \cup \Gamma}{\{F\} \cup \Gamma}$ where F is from Lemma 8.1.15

PAR $\quad \dfrac{\{t_1 \| t_2 \vdash \Diamond_C \Phi\} \cup \Gamma}{\{F\} \cup \Gamma}$ where F is from Lemma 8.1.16

STEP1 $\quad \dfrac{\{X \vdash \Diamond_C \Phi\} \cup \Gamma}{\{X \vdash \Phi\} \cup \Gamma \quad \{\bigvee_{i \in I_1} t_1 \vdash \Diamond_{D^i_1} \Phi\} \cup \Gamma \ \ldots \ \{\bigvee_{i \in I_n} t_n \vdash \Diamond_{D^i_n} \Phi\} \cup \Gamma}$

if $C(\lambda)$, where λ is the empty sequence, $(X \stackrel{a_k}{\to} t_k) \in \Delta$, $k = 1, \ldots, n$ and the D^i_k are the decompositions of C (Def. 8.1.3 (case 4)).

STEP2 $\quad \dfrac{\{X \vdash \Diamond_C \Phi\} \cup \Gamma}{\{\bigvee_{i \in I_1} t_1 \vdash \Diamond_{D^i_1} \Phi\} \cup \Gamma \quad \ldots \quad \{\bigvee_{i \in I_n} t_n \vdash \Diamond_{D^i_n} \Phi\} \cup \Gamma}$

if not $C(\lambda)$, $(X \stackrel{a_k}{\to} t_k) \in \Delta$, $k = 1, \ldots, n$ and the D^i_k are decompositions of C (Def. 8.1.3 (case 4)).

Unsat $\quad \dfrac{\{t \vdash \Diamond_C \Phi\} \cup \Gamma}{\{\mathit{false}\}}$ if C is unsatisfiable

conj1 $\quad \dfrac{\{t \vdash \Phi \wedge \Psi\} \cup \Gamma}{\{t \vdash \Phi, t \vdash \Psi\} \cup \Gamma}$

conj2 $\quad \dfrac{\{F \wedge G\} \cup \Gamma}{\{F, G\} \cup \Gamma}$

disj1 $\quad \dfrac{\{t \vdash \Phi \vee \Psi\} \cup \Gamma}{\{t \vdash \Phi\} \cup \Gamma \quad \{t \vdash \Psi\} \cup \Gamma}$

disj2 $\quad \dfrac{\{F \vee G\} \cup \Gamma}{\{F\} \cup \Gamma \quad \{G\} \cup \Gamma}$

Induct1 $\dfrac{\{t \vdash \Diamond_C \Psi\} \cup \Gamma}{\Gamma}$ if $\Psi \in \mathcal{F}^c_{d-1}$ and $t \models \Diamond_C \Psi$

Induct2 $\dfrac{\{t \vdash \Diamond_C \Psi\} \cup \Gamma}{\{\mathit{false}\}}$ if $\Psi \in \mathcal{F}^c_{d-1}$ and not $t \models \Diamond_C \Psi$

Induct3 $\dfrac{\{t \vdash \neg\Diamond_C \Psi\} \cup \Gamma}{\Gamma}$ if $\Psi \in \mathcal{F}^c_{d-1}$ and not $t \models \Diamond_C \Psi$

Induct4 $\dfrac{\{t \vdash \neg\Diamond_C \Psi\} \cup \Gamma}{\{\mathit{false}\}}$ if $\Psi \in \mathcal{F}^c_{d-1}$ and $t \models \Diamond_C \Psi$

Term1 $\dfrac{\{t \vdash T^+\} \cup \Gamma}{\Gamma}$ if $T^+ = \{t\}$ or $T^+ = \{\}$

Term2 $\dfrac{\{t \vdash T^+\} \cup \Gamma}{\{\mathit{false}\}}$ if $T^+ \neq \{\} \wedge T^+ \neq \{t\}$

Term3 $\dfrac{\{t \vdash -T^-\} \cup \Gamma}{\Gamma}$ if $t \notin T^-$

Term4 $\dfrac{\{t \vdash -T^-\} \cup \Gamma}{\{\mathit{false}\}}$ if $t \in T^-$

In order to avoid any unnecessary growth of the proof tree, we define that the rules with names in capital letters (PAR, SEQ1, SEQ2, STEP1 and STEP2) have a **lower** precedence than the other rules. So in the construction of a branch of the proof tree we only use such a rule if none of the others is applicable.

Lemma 8.1.17 *For any instance of a tableau-rule, the antecedent is true iff at least one of the succedents is true.*

Proof This follows immediately from the definition of the tableau-rules and Lemma 8.1.14, Lemma 8.1.15 and Lemma 8.1.16. ■

Definition 8.1.18 (Termination conditions)
A node in the tableau consisting of a set of formulae Γ is a *terminal node* if one of the following conditions is satisfied:

1. $\Gamma = \{\}$

2. *false* $\in \Gamma$.
3. There is a previous node in the same branch that is marked with the same set Γ.

Terminal nodes of type 1 are successful, while terminal nodes of types 2,3 are unsuccessful.

The construction of a branch of the tableau stops when a terminal node is reached. The branch is successful if this terminal node is successful. The tableau is successful if there is at least one successful branch.

The intuition is that every branch in the tableau is an attempt to construct a proof. A terminal node of type 1 means that all subgoals have been solved. A terminal node of type 2 means that this attempt to construct a proof failed. A terminal node of type 3 means that the proof is 'running in circles'. If there is a proof, then it can be found elsewhere in the tableau by a shorter branch.

The construction of the tableau starts with a root-node of the form $\{t \models \Diamond_C \Phi\}$ where t is a process term and $\Phi \in \mathcal{F}^c_d$. The tableau for a given root is not unique, because the sequents are sets of expressions and the element to which a rule is applied is chosen nondeterministically. However, all tableaux are equivalent semantically, because the order in which subgoals are solved does not matter.

8.1.4 Decidability

In this section we show that the tableau system of the previous section is sound and complete and produces only finite tableaux for any given root. Thus it yields a decision procedure for the model checking problem for PAD and $EF^{=}_{DC}$.

Lemma 8.1.19 *If the root node has the form $\{t \vdash \Diamond_C \Phi\}$, for $\Phi \in \mathcal{F}^c_d$, then for every node in a tableau with this root at least one of the following conditions is satisfied:*

1. *A tableau rule is applicable*
2. *The node is a terminal node.*

Proof The only problematic cases are the expressions of the form $t \vdash \neg\Diamond_C \Phi$. If such an expression occurs, then it must be due to the rules SEQ1, SEQ2 or STEP1. By definition of these rules and Lemma 8.1.14 and Lemma 8.1.15 we know that $\Phi \in \mathcal{F}^c_{d-1}$. Then the rules Induct3 or Induct4 are applicable, because we assumed (by induction hypothesis) that we can already solve the problem for formulae of a smaller nesting depth. ■

Lemma 8.1.20 *The tableau is finite for every instance of the model checking problem.*

Proof If only process terms of a bounded size are used as atomic propositions, then there are only finitely many formulae in $\mathcal{F}_d^c$ for any fixed d. The tableau rules and the proofs of Lemmas 8.1.14, 8.1.15 and 8.1.16 show that this precondition is satisfied. Any set $\mathcal{DC}$ of decomposable constraints is finite. There are only finitely many rules $(t_1 \xrightarrow{a} t_2) \in \Delta$ with only finitely many subterms of the terms t_2. So there are only finitely many different sets of expressions of the form $t \vdash \Phi$ in the tableau. Therefore the branches of the tableau can only have finite length, because of termination condition 3. Since the tableau is finitely branching, the result follows. ∎

Now we prove the soundness and completeness of the tableau. The following lemma shows the soundness.

Lemma 8.1.21 *Let $\Phi \in \mathcal{F}_d^c$ and $C \in \mathcal{DC}$. If there is a successful tableau with root $\{t \vdash \Diamond_C \Phi\}$, then $t \models \Diamond_C \Phi$.*

Proof A successful tableau has a successful branch ending with a node marked by the empty set of expressions. Since these sets are interpreted conjunctively this node is true. By repeated application of Lemma 8.1.17 all its ancestor-nodes must be true and thus the root-node must be true. ∎

We need some new definitions to show the completeness. These definitions only apply to this particular tableau system in this chapter.

Definition 8.1.22 A *valid sequent* Γ in a tableau is a set of expressions which evaluate to *true*.

For example if $(t \vdash \Diamond_C \Phi) \in \Gamma$ then $t \models \Diamond_C \Phi$. If $(F \wedge G) \in \Gamma$ then F and G evaluate to *true*.

It follows from the construction of the tableau system that every expression in a valid sequent is a disjunction of conjunctions of expressions of the form $t \vdash \Diamond_C \Phi$ or $t \vdash \neg \Diamond_C \Phi$.

Now we define a total order on valid sequents.

Definition 8.1.23 For an expression $t \vdash \Diamond_C \Phi$ with $t \models \Diamond_C \Phi$ we define

$$xnorm(t \vdash \Diamond_C \Phi) := min\{length(\sigma) \ \mid \ t \xrightarrow{\sigma} t' \in [\![\Phi]\!] \wedge C(\sigma)\}$$

and

$$ynorm(t \vdash \Diamond_C \Phi) := size(t)$$

For an expression F in a valid sequent we define

$$xnorm(F) := max\{xnorm(t \vdash \Diamond_C \Phi) \mid t \vdash \Diamond_C \Phi \text{ is subterm of } F,\ nd(\Phi) = d\}$$

and

$$ynorm(F) := max\{ynorm(t \vdash \Diamond_C \Phi) \mid t \vdash \Diamond_C \Phi \text{ is subterm of } F,\ nd(\Phi) = d\}$$

and

$$znorm(F) := size(F)$$

where $size(F)$ is just the number of letters/symbols needed to write F. The norm of F is a triple, which is defined by

$$norm(F) := (xnorm(F), ynorm(F), znorm(F))$$

These norms are ordered lexicographically. The order is well-founded.

For a valid sequent Γ let

$$\gamma_{x,y,z} := |\{F \in \Gamma \ \mid \ norm(\Gamma) = (x, y, z)\}|$$

Since Γ is valid and finite, there is a largest x s.t. $\gamma_{x,y,z} \neq 0$ for some y, z. This largest x will be called x_{max}. It depends on Γ. Also for every $x \leq x_{max}$ there is a largest y (called $y(x)$) s.t. $\gamma_{x,y,z} \neq 0$ for some z. Finally, for every x, y there is a largest $z(x, y)$ s.t. $\gamma_{x,y,z} \neq 0$.

We define a well-founded ordering on valid sequents. Let Γ and Γ' be two valid sequents and $\gamma_{x,y,z}$ and $\gamma'_{x,y,z}$ be defined as above. Then

$$\Gamma < \Gamma' \ :\Leftrightarrow \ \exists(x, y, z).\gamma_{x,y,z} < \gamma'_{x,y,z} \ \wedge \ \forall(x', y', z') \geq_{lex} (x, y, z).\ \gamma_{x',y',z'} = \gamma'_{x',y',z'}$$

The intuition is that if a tableau-rule is applied to a valid sequent Γ, then there is at least one valid succedent sequent that is smaller. This is because an expression $F \in \Gamma$ is replaced with several others with a lower norm. Since the ordering is well-founded, the process must eventually terminate.

Note that these definitions do not apply to non-valid sequents.

Lemma 8.1.24 *Let Γ be a valid sequent. Then every tableau with root Γ has at least one successful branch that ends with the empty sequent.*

Proof By Lemma 8.1.17 every tableau with root Γ has at least one branch that only contains valid sequents. Choose one such branch of minimal length. We show that the order of the sequents on this branch must strictly decrease. We do this by showing that every application of a tableau rule to a valid sequent yields a smaller sequent.

SEQ1,SEQ2 It follows from the construction of the expressions in Lemma 8.1.14 and Lemma 8.1.15 that in these expressions one of two cases holds:

1. The remaining sequence is shorter (lower $xnorm$) or
2. The remaining sequence has the same length and the terms are smaller (lower $ynorm$).

Thus the succedent sequent is smaller.

PAR It follows from the construction of the expression in Lemma 8.1.16 that the terms are always smaller (since t_1, t_2 are smaller than $t_1 \| t_2$). The $xnorm$ is the same or smaller and the $ynorm$ is smaller. Thus the succedent is smaller.

STEP1,STEP2 Here we have two sub-cases:

- In the first branch of the rule STEP1 the sequence has length 0. In the succedent the $xnorm$ and $ynorm$ are the same, but the $znorm$ is smaller.
- In the other branches of STEP1 and all branches of STEP2 we choose the valid succedent that corresponds to the shortest sequence that leads to a state that satisfies Φ. In this succedent the sequence is shorter and thus the $xnorm$ is smaller.

In both cases the succedent is smaller.

Unsat This rule is never applied in this branch, because all sequents are valid.

conj1,conj2 For these rules the succedent is smaller, because the $znorm$ decreases.

disj1,disj2 For these rules the succedent is smaller, because the $znorm$ decreases.

Induct,Term For the rules Induct1,Induct3 and Term1,Term3 the succedent must be smaller, because expressions are removed from the sequent. The rules Induct2,Induct4,Term2,Term4 are never applied in this branch, because all sequents are valid.

The construction of this branch cannot be stopped by termination condition 3, because the order strictly decreases. Since the order of the sequents strictly decreases on this branch, it must eventually end with the empty sequent and thus it is successful. ∎

Corollary 8.1.25 *If $t \models \Diamond_C \Phi$ for $\Phi \in \mathcal{F}^c_d$ and $C \in \mathcal{DC}$ then every tableau with root $\{t \vdash \Diamond_C \Phi\}$ is successful.*

Proof The root-sequent is valid. By Lemma 8.1.24 every tableau must have a branch that ends with the empty sequent. This branch is successful and thus the tableau is successful. ∎

Lemma 8.1.26 *Let t be a process term, Δ a PAD, $\Phi \in \mathcal{F}^c_d$, $\mathcal{DC}$ a set of decomposable constraints and $C \in \mathcal{DC}$. Then the following conditions are equivalent:*

- $t \models \Diamond_C \Phi$
- *A tableau with root $\{t \vdash \Diamond_C \Phi\}$ is successful.*
- *Every tableau with root $\{t \vdash \Diamond_C \Phi\}$ is successful.*

Proof Directly from Lemma 8.1.21 and Corollary 8.1.25. ∎

Theorem 8.1.27 *The model checking problem for $EF^{=}_{DC}$ and PAD is decidable.*

Proof By Lemma 8.1.12 it suffices to prove decidability for formulae of the form $\Diamond_C \Phi$ with Φ in $\mathcal{F}^c_d$ for any d. We prove this by induction on d. By Lemma 8.1.26 and Lemma 8.1.20 it suffices to construct a finite tableau. During the construction we must decide problems of the form $t' \models \Diamond_C \Psi$ for $\Psi \in \mathcal{F}^c_{d-1}$. In the base case $d = 0$ this is trivial, since $\mathcal{F}^c_{-1} = \emptyset$. For $d > 0$ this is possible by induction hypothesis. ∎

Since EF is weaker than $EF^{=}_{DC}$, we get the following corollary.

Corollary 8.1.28 *Model checking PAD with EF is decidable.*

8.1.5 Complexity

We have shown that the model checking problem for the branching-time temporal logic EF and the process model PAD is decidable. The exact complexity of the problem is an open problem. While for the special case of BPP the problem is *PSPACE*-complete [May96a, May96c] (see Section 6.1), the algorithm for PA in [May97b] and the one for PAD described here have superexponential complexity. The algorithm described here is a generalization of the one in [May97b], but not a generalization of the algorithm for BPP in [May96c]. The *PSPACE*-algorithm for BPP in [May96c] uses a bounded search, while the algorithms for PA and PAD work by decomposition. For a formula of nesting-depth d the complexity of the algorithm derived from the tableau system is d-times exponential. This is because there are d-times exponentially many different EF-formulae of nesting depth d. The decompositions of Lemma 8.1.14, Lemma 8.1.15 and Lemma 8.1.16 introduce expressions of d-times exponential size. So the overall complexity of the algorithm is $O(tower(n))$, where $tower(0) := 0$ and $tower(i+1) := 2^{tower(i)}$.

The best known lower bound for both PAD and PA is *PSPACE*-hardness. This lower bound is inherited from BPP, because PAD and PA subsume BPP. However, there is still a difference between PAD and PA. For PAD the problem is *PSPACE*-hard in the size of the system for a fixed formula, because this holds for pushdown processes and PAD subsumes pushdown processes (see Chapter 7). PA does not subsume pushdown processes and the best known lower bound is the same as for BPP: The problem is Σ_d^p-hard for formulae of nesting depth d. The interesting part is now the complexity of model checking PA with any fixed formula. Unlike for PAD, there is no hardness result for this problem. In the following we show that model checking PA with any fixed $EF_{DC}^{=}$-formula $\diamond_C \Phi$, where Φ has nesting-depth d, is in Σ_{d+1}^p. So there is a real difference between PAD and PA. It is an open question if model checking PA with a fixed formula can be done in polynomial time.

Theorem 8.1.29 *For every fixed $EF_{DC}^{=}$-formula $\diamond_C \Phi$, where Φ has nesting-depth d, the model checking problem for PA can be solved in Σ_{d+1}^p.*

Proof We use the same tableau system as for PAD. However, it becomes simpler for PA, because some cases can never occur. The most important difference between PAD and PA is in the sequential decomposition in Lemma 8.1.14. In PA the parts of the expression of the form

$$\bigvee_{j \in J,\ \left(l.X \xrightarrow{a} r\right) \in \Delta} t \models \diamond_{D_1^j}(l) \wedge r \models \diamond_{D_2^j} \Phi$$

are empty, because in PA no rule in Δ has sequential composition on the left-hand side. These expressions are the only ones where terms of rules in Δ are introduced into the $EF^{=}_{DC}$-formulae, so this does never occur for PA.

Consider a tableau for a PA-process Δ with initial state t_0 and an $EF^{=}_{DC}$-formula $\Diamond_C\Phi$. The root node has the form $\{t \vdash \Diamond_C\Phi\}$ and every sequent in the tableau is a set of expressions of the form $t' \vdash \Phi'$ where t' is a process term and Φ' is a formula. For PA the size of these formulae Φ' is bounded by a constant that depends only on the size of $\Diamond_C\Phi$ and does not grow with the size of Δ or t_0. Thus there is constant c that depends only on $\Diamond_C\Phi$ s.t. the number of different formulae Φ' that can occur in the tableau is bounded by c. Since we consider the model checking problem for a fixed formula, c is a constant. Every term t' that occurs in a subgoal of the form $t' \vdash \Phi'$ is either a subterm of t_0 or a subterm of the right hand side of some rule in Δ. Thus t' has size $\mathcal{O}(n)$. Furthermore, $Var(\Delta)$ has size $\mathcal{O}(n)$. Thus there are only $\mathcal{O}(n)$ different subgoals of the form $X \vdash \Phi'$.

By Lemma 8.1.26 it does not matter which of the possible tableaux we construct for a given root. Thus we can safely restrict the nondeterminism in the construction of the tableau. We do this by the requirement that the most recently introduced subgoal must be solved first. Thus we assign a priority to every subgoal $t' \vdash \Phi'$ in a sequent. The most recently created subgoals have the highest priority. For every sequent Γ let $first(\Gamma)$ be the subgoal with the highest priority. Now we can modify the tableau system by introducing a stronger version of termination condition 3 (see Def. 8.1.18): A branch is unsuccessful if it ends with a sequent Γ s.t. there is a previous sequent Γ' with $first(\Gamma) = X \vdash \Phi' = first(\Gamma')$ and the subgoal $X \vdash \Phi'$ in $first(\Gamma')$ was created (possibly in several steps) from $first(\Gamma)$. It is easy to see that such a branch must be unsuccessful, because the subgoal $first(\Gamma')$ has been attacked first and should have been solved already. Instead it was reduced to itself, and the proof is running in circles. In a subgoal of the form $t' \vdash \Phi'$ the term t' has size $\mathcal{O}(n)$. t' is decomposed in the tableau, and thus we get a subgoal of the form $X \vdash \Phi'$ after at most $\mathcal{O}(n)$ steps. There are only $\mathcal{O}(n)$ different such subgoals. So one can assume that every subgoal can be solved in $\mathcal{O}(n^2)$ steps. In particular the one goal in the root sequent can be solved in $\mathcal{O}(n^2)$ steps. Thus if the root-sequent is true, then there must be a successful branch of length $\mathcal{O}(n^2)$. The branching degree of the tableau is $\leq max(2, n) = \mathcal{O}(n)$, because there are only $\mathcal{O}(n)$ rules in Δ. Every sequent in the tableau can be described in $\mathcal{O}(n)$ space, because all the terms in the subgoals are disjoint subterms of t_0 or the right-hand sides of rules in Δ. Thus, if the root-sequent is true, then there must be a successful branch that can be described in $\mathcal{O}(n^3)$ space. The instance of the model checking problem has answer 'yes' iff at least one branch is successful. Thus it suffices to nondeterministically guess a branch of size $\mathcal{O}(n^3)$.

In order to verify that this branch is indeed a valid successful branch we need to decide side consitions of the form $t'' \models \Diamond_C \Phi''$ or $t'' \models \neg\Diamond_C \Phi''$ where Φ'' has a smaller nesting-depth (at most $d-1$). By induction hypothesis this can be done in Σ_d^p. Therefore the validness and successfulness of the branch can be verified in polynomial time with the help of a Σ_d^p-oracle. Thus the problem is in Σ_{d+1}^p. ■

8.2 Reachability for PAD

As shown in Chapter 3, the reachability problem can be expressed in the logic EF by $t_0 \models \Diamond(t)$. Since state formulae have nesting depth 0, the tableau system of the previous section yields an exponential time algorithm. A more careful analysis shows that the problem is $\mathcal{NP}$-complete.

Theorem 8.2.1 *The reachability problem and the reachable property problem for PAD are $\mathcal{NP}$-complete.*

Proof The question is if $t_0 \models \Diamond(t)$. Consider the tableau system defined in the previous section. In this special case the induction hypothesis is not needed an the rules Induct1–Induct4 are never used. Every branch in the tableau stands for an attempt to construct a proof of reachability. The branching degree of the tableau may be exponential, but since the problem is decomposed into smaller subproblems, any node can be described in polynomial space. (This is not true for the general model checking problem. There the branching degree is d-times exponential and any node can be described in $(d-1)$-times exponential space.)

The important observation is now that if $t_0 \models \Diamond(t)$, then there is a successful branch of polynomial length. (There may be other successful branches of exponential length.) All (polynomially many) expressions in the set that constitutes a node in the tableau have the form $t_i' \vdash \Diamond(t_i'')$ for terms t_i', t_i'', $1 \leq i \leq k$. Let the size of a node be defined as the sum of the sizes of these terms. This size cannot increase on a branch. It can decrease in some applications of the rules SEQ1 and SEQ2 (where the ends of t' and t'' match). There must be a successful branch where such a decrease occurs every polynomially many steps. If this is not the case, then the branch does unnecessary work (like too many applications of the rules STEP1 and STEP2). Only polynomially many applications of STEP-rules lie between each decrease, because there are only polynomially many right hand sides of rules. Thus if more STEP-rules were applied, the branch would partly be running in circles. So in a short successful branch the size of the nodes decreases at least once every polynomially many steps. Thus the branch has polynomial length and can be described in polynomial space.

The $\mathcal{NP}$-algorithm is now to guess a branch of polynomial length and to use the tableau-rules to verify if it is indeed a valid successful branch. $\mathcal{NP}$-hardness follows from the fact that PAD subsumes BPP and reachability is $\mathcal{NP}$-complete for BPP (see Chapter 6).

The proof for the reachable property problem is similar. ∎

8.3 Simple Verification Problems for PA

As mentioned in Subsection 2.3.6, PA-processes are $(1, G)$-PRS in the PRS-hierarchy. They can describe nondeterminism, sequential- and parallel composition and recursion, but no communication between components. They are the smallest natural common generalization of BPP and BPA. PA is a special case of PAD. Intuitively, the only difference is that in PA the subroutines have no effect on their caller (see also Section 2.2).

In this section we study two simple verification problems for PA-processes. They can be seen as special cases of the model checking problem for PA-processes and the logic EF. However, the algorithm in Section 8.1 has a very high complexity (see also [May97b]). Thus it is useful to consider these subproblems here, because they can be solved in polynomial time. In practice, it often suffices to check very simple properties of systems, and it is not necessary to use a full temporal logic to express them. The verification problems in this section have also been solved with tableau methods in [May97e].

8.3.1 Partial Deadlock

The *Partial deadlock problem* is the problem if there is a reachable state where certain given actions are disabled.

<u>PARTIAL DEADLOCK</u>

Instance: A labeled transition system with initial state s_0 and a finite set of atomic actions A.

Question: Is it possible to reach a state s s.t. $\not\exists a \in A, s'.\ s \xrightarrow{a} s'$?

In the special case of $A = Act$ this is the problem if a deadlocked state is reachable.

This problem can also be formulated in the logic EF. Let $A = \{a_1, \ldots, a_n\}$, then the question is equivalent to $s_0 \models \Diamond(\neg a_1 \wedge \cdots \wedge \neg a_n)$. Of course this is much simpler than general model checking with EF.

For general Petri nets the partial deadlock problem is equivalent to the problem of deciding if a state is reachable where certain places are unmarked. This problem has the same complexity as the reachability problem. So the partial deadlock reachability problem is decidable for Petri nets, but at least *EXPSPACE*-hard [Lip76, May84].

The situation is different for PA. The reachability problem for PA is $\mathcal{NP}$-complete, because it is $\mathcal{NP}$-complete for BPP [Esp95] and for PAD (see Section 8.2). However, the partial deadlock problem can be decided in polynomial time.

First we define some predicates.

Definition 8.3.1 Let $A \subseteq Act$ be a set of actions. Sequences of actions are denoted by σ.

$$\begin{array}{lll} \mathcal{E}(t) & :\Longleftrightarrow & \exists \sigma.\ t \xrightarrow{\sigma} \epsilon \\ D_A(t) & :\Longleftrightarrow & \exists \sigma, t'.\ (t \xrightarrow{\sigma} t' \wedge \ \not\exists a \in A.\ t' \xrightarrow{a}) \\ S_A(t) & :\Longleftrightarrow & \exists \sigma, t'.\ (t \xrightarrow{\sigma} t' \neq \epsilon \wedge \ \not\exists a \in A.\ t' \xrightarrow{a}) \end{array}$$

It is clear that $D_A(t) \iff S_A(t) \vee \mathcal{E}(t)$.

Intuitively, $\mathcal{E}(t)$ means that the process t can terminate, $D_A(t)$ means that t can reach a partial deadlock and $S_A(t)$ means that t can get 'stuck', i.e. reach a deadlock without terminating. The partial deadlock problem for a PA-process t is to decide if $D_A(t)$ holds.

Lemma 8.3.2 *The following logical equalities follow directly from the definitions. They are used to decompose the problem.*

$$\begin{array}{rcl} D_A(t) & \Longleftrightarrow & S_A(t) \vee \mathcal{E}(t) \\ S_A(t_1.t_2) & \Longleftrightarrow & S_A(t_1) \vee (\mathcal{E}(t_1) \wedge S_A(t_2)) \\ S_A(t_1 \| t_2) & \Longleftrightarrow & (S_A(t_1) \wedge D_A(t_2)) \vee (D_A(t_1) \wedge S_A(t_2)) \\ \mathcal{E}(t_1.t_2) & \Longleftrightarrow & \mathcal{E}(t_1) \wedge \mathcal{E}(t_2) \\ \mathcal{E}(t_1 \| t_2) & \Longleftrightarrow & \mathcal{E}(t_1) \wedge \mathcal{E}(t_2) \end{array}$$

Lemma 8.3.3 *Let Δ be a set of PA-rules. It is possible to decide the property $\mathcal{E}(X)$ for all variables X in $\mathcal{O}(n^3)$ time.*

Proof We use a marking algorithm. First mark all variables X s.t. there is a rule $(X \xrightarrow{a} \epsilon) \in \Delta$. This can be done in $\mathcal{O}(n)$ time.

The following step is repeated until no new variable is marked:

Consider every rule $(Y \xrightarrow{a} t) \in \Delta$ s.t. Y is unmarked so far. If all variables in t are marked then mark Y.

Checking if all variable in t are marked can be done in $\mathcal{O}(n)$ time. As there are $\mathcal{O}(n)$ rules in Δ this must be done at most $\mathcal{O}(n)$ times in each step. There are $\mathcal{O}(n)$ variables and thus we need at most $\mathcal{O}(n)$ steps. So the algorithm requires at most $\mathcal{O}(n^3)$ time. ∎

Lemma 8.3.4 *If $\mathcal{E}(X)$ is already known for every variable X then the property $\mathcal{E}(t)$ for a term t can be decided in linear time in the size of t.*

Proof $\mathcal{E}(t)$ holds if and only if $\mathcal{E}(X)$ for every variable X that occurs in t. ∎

Theorem 8.3.5 *The partial deadlock problem for PA-processes is decidable in $\mathcal{O}(n^3)$ time.*

Proof Let t_0 be the initial state, Δ the set of rules and $A \subseteq Act$. We can assume w.r. that t_0 is a single variable X_0, because otherwise we could add a new variable X_0 and a new rule $X \xrightarrow{a} t_0$. The problem is if $D_A(X_0)$ holds.

1. First we decide $\mathcal{E}(X)$ for all variables X in time $\mathcal{O}(n^3)$ with the algorithm of Lemma 8.3.3.

2. Now we decide $S_A(X)$ for all variables X. First mark all variables X s.t. $\not\exists a \in A.\, X \xrightarrow{a}$. This can be done in $\mathcal{O}(n^2)$ time.

   Repeat the following step until no new variable is marked:

   For all rules $Y \xrightarrow{a} t_1.t_2$ or $Y \xrightarrow{a} t_1 \| t_2$ s.t. Y is unmarked use the equations of Lemma 8.3.2 and the already acquired knowledge about $\mathcal{E}(Z)$ and $S_A(Z)$ for other variables Z to prove $S_A(Y)$. In addition to the normal failure cases this fails when we encounter unmarked variables. It it succeeds them mark Y.

   There are $\mathcal{O}(n)$ variables and thus this step is done at most $\mathcal{O}(n)$ times. Each step can be done in $\mathcal{O}(n^2)$ time and thus the whole procedure can be done in $\mathcal{O}(n^3)$ time.

3. As we know the values of $\mathcal{E}(X)$ and $S_A(X)$ for every variable X we can decide $D_A(X_0)$ in constant time. ∎

8.3.2 Livelock

The *Livelock problem* is the problem if a state can be reached, such that from then on certain given actions can never become enabled again.

LIVELOCK

Instance: A labeled transition system with initial state s_0 and a finite set of atomic actions A.

Question: Is is possible to reach a state s s.t. no state s' that is reachable from s enables any action in A ?

Note that in the special case of $A = Act$ this is the same as the deadlock problem.

The livelock problem can be expressed in the logic EF. For $A = \{a_1, \ldots, a_n\}$, the question is if

$$s_0 \models \Diamond\Box(\neg a_1 \wedge \cdots \wedge \neg a_n)$$

We show that the livelock problem for PA can be solved in polynomial time. First we define some predicates.

Definition 8.3.6 Let $A \subseteq Act$ be a set of actions.

$$\begin{array}{rcl}
\mathcal{E}(t) & :\Longleftrightarrow & \exists \sigma.\ t \xrightarrow{\sigma} \epsilon \\
En_A(t) & :\Longleftrightarrow & \exists a \in A.\ t \xrightarrow{a} \\
N_A(t) & :\Longleftrightarrow & \nexists \sigma, t'.\ (t \xrightarrow{\sigma} t' \ \wedge \ En_A(t')) \\
R_A(t) & :\Longleftrightarrow & \exists \sigma, t'.\ (t \xrightarrow{\sigma} t' \ \wedge \ N_A(t')) \\
RI_A(t) & :\Longleftrightarrow & \exists \sigma, t'.\ (t \xrightarrow{\sigma} t' \ \wedge \ N_A(t') \ \wedge \ \neg\mathcal{E}(t'))
\end{array}$$

Intuitively, $N_A(t)$ means that t will never be able to do any action from A, and $R_A(t)$ means that a state t' is reachable from t s.t. for t' all actions in A are disabled forever. $RI_A(t)$ means that a state t' is reachable from t s.t. t' has no terminating computation and all actions in A are disabled forever. This can also be expressed in EF. Let $A := \{a_1, \ldots, a_n\}$.

$$\begin{array}{rcl}
N_A(t) & = & t \models \Box(\neg a_1 \wedge \cdots \wedge \neg a_n) \\
R_A(t) & = & t \models \Diamond\Box(\neg a_1 \wedge \cdots \wedge \neg a_n) \\
RI_A(t) & = & t \models \Diamond(\neg\Diamond(\epsilon) \ \wedge \ \Box(\neg a_1 \wedge \cdots \wedge \neg a_n))
\end{array}$$

The livelock problem is to check if $R_A(t_0)$ holds for a PA-process with initial state t_0 and set of rules Δ. W.r. we can assume that t_0 is a single variable X_0, because otherwise we could just add a new variable X_0 and a rule $X_0 \rightarrow t_0$.

The algorithm proceeds in four steps:

1. Decide $\mathcal{E}(X)$ for all variables X.

2. Decide $N_A(X)$ for all variables X using the previously collected information.

3. Decide $R_A(X)$ and $RI_A(X)$ for all X using the previously collected information.

4. Decide $R_A(t_0)$ by using the previously collected information.

The first step uses the algorithm described in Lemma 8.3.3 It requires $\mathcal{O}(n^3)$ time.

For the second step we define $P_A(t) := \neg N_A(t)$ and use a marking algorithm to decide $P_A(X)$ for every variable X.

Start Mark all variables X s.t. $\exists a \in A.\ X \xrightarrow{a}$.

Step For every unmarked variable X and every rule $(X \rightarrow t) \in \Delta$ do the following: If $P'_A(t)$ then mark X.

P'_A is defined by

$$\begin{array}{rcl} P'_A(t_1 \| t_2) & := & P'_A(t_1) \vee P'_A(t_2) \\ P'_A(t_1.t_2) & := & P'_A(t_1) \vee (\mathcal{E}(t_1) \wedge P'_A(t_2)) \\ P'_A(X) & := & \text{if } X \text{ is marked then } \mathit{true} \text{ else } \mathit{false} \end{array}$$

Repeat Step until no new variable is marked.

At the end $P_A(X)$ holds if X is marked. Thus $N_A(X)$ is true iff X is not marked at the end. The evaluation of an instance of $P'_A(t)$ can be done in $\mathcal{O}(n)$ time. There are $\mathcal{O}(n)$ rules in Δ, and thus at most $\mathcal{O}(n)$ instances of $P'_A(t)$ are called in Step. The Step is done at most $\mathcal{O}(n)$ times and thus the algorithm requires only $\mathcal{O}(n^3)$ time.

The third step is a marking algorithm that uses two different markings R_A and RI_A. If a variable X is marked by R_A (RI_A) then it means that it is already known that $R_A(X)$ ($RI_A(X)$) is true.

Start If $N_A(X)$ then mark X with R_A. If $N_A(X)$ and $\neg\mathcal{E}(X)$ then mark X with RI_A.

Step For every variable X and every rule $(X \to t) \in \Delta$ do: If $R'_A(t)$ then mark X with R_A. If $RI'_A(t)$ then mark X with RI_A. The functions R'_A and RI'_A are defined as follows:

$$\begin{array}{rcl} R'_A(t_1 \| t_2) & := & R'_A(t_1) \wedge R'_A(t_2) \\ R'_A(t_1.t_2) & := & RI'_A(t_1) \vee (\mathcal{E}(t_1) \wedge R'_A(t_2)) \\ R'_A(X) & := & \text{if } X \text{ is marked by } R_A \text{ then } \mathit{true} \text{ else } \mathit{false} \end{array}$$

$$\begin{array}{rcl} RI'_A(t_1 \| t_2) & := & (RI'_A(t_1) \wedge R'_A(t_2)) \vee (R'_A(t_1) \wedge RI'_A(t_2)) \\ RI'_A(t_1.t_2) & := & RI'_A(t_1) \vee (\mathcal{E}(t_1) \wedge RI'_A(t_2)) \\ RI'_A(X) & := & \text{if } X \text{ is marked by } RI_A \text{ then } \mathit{true} \text{ else } \mathit{false} \end{array}$$

Repeat Step until no new variable is marked.

Every evaluation of an instance of $R'_A(t)$ or $RI'_A(t)$ can be done in $\mathcal{O}(n)$ time, because of Lemma 8.3.4. At most $\mathcal{O}(n)$ instances are called in every Step, because there are only $\mathcal{O}(n)$ rules in Δ. The Step is done at most $\mathcal{O}(n)$ times because there are only $\mathcal{O}(n)$ variables. Thus the algorithm requires at most $\mathcal{O}(n^3)$ time.

The last step can be done in constant time, because we assumed that the initial state X_0 is a variable. $R_A(X_0)$ holds if X is marked by R_A.

Theorem 8.3.7 *The livelock problem for PA is decidable in $\mathcal{O}(n^3)$ time.*

Proof Each of the four steps of the algorithm can be done in $\mathcal{O}(n^3)$ time. ■

8.4 Conclusion

The reachability problem and the reachable property problem are $\mathcal{NP}$-complete for BPP (see Chapter 6) and for PAD (see Section 8.2). Thus they are also $\mathcal{NP}$-complete for PA.

Model checking with most branching-time logics is undecidable for PA and PAD, since model checking with the temporal logic EG is undecidable for BPP, even for a fixed EG-formula (see Section 6.3). The only exception is the logic EF, which is incomparable to EG. It has been shown in [May97b] that model checking PA with EF is decidable. This proof has been generalized to PAD in Section 8.1. The exact complexity is open. The general complexity of the algorithm is $\mathcal{O}(tower(n))$ and it is d-times exponential for formulae of nesting-depth d. The best known lower bound is *PSPACE*-hardness, but there is a difference between PA and PAD.

- PAD subsumes pushdown processes and inherits their lower bound (see Chapter 7). Thus the model checking problem for EF and PAD is *PSPACE*-hard in the size of the process even for a fixed EF-formula.

- PA does not subsume pushdown processes and only inherits the lower bound from BPP (see Chapter 6). Thus model checking PA with EF is *PSPACE*-hard, but only Σ_d^p-hard for formulae of nesting-depth d. As shown in Theorem 8.1.29 model checking PA is in Σ_d^p for every fixed EF-formula $\Diamond_C \Phi$ of nesting-depth d. Thus there is a real difference between PA and PAD.

Model checking PA with all linear-time logics (except for WL) is undecidable. This has been shown by Bouajjani and Habermehl in [BH96]. In fact, not even full PA is needed for this result. It suffices to take two context-free processes and a finite-state process running in parallel. The two context-free processes serve as counters and the finite-state system as a finite control. While PA itself can not enforce a synchronization between these three components, this can be achieved with an LTL formula. This LTL formula characterizes all faithful runs of the system which synchronize correctly. So it is possible to reduce the halting problem for Minsky 2-counter machines to the model checking problem, thus proving its undecidability.

Although Bouajjani and Habermehl do not explicitly mention it in [BH96], the model checking problem is even undecidable for two BPA processes that run in parallel and a fixed LTL-formula. This is because the finite control of the universal 2-counter machine can be encoded in a LTL-formula. The two BPA processes serve as counters and their initial states represent the initial values in the counters. The LTL-formula encodes the fixed finite control of the universal 2-counter machine and enforces the synchronization. The undecidability result carries over to PAD, because it subsumes PA.

The following table shows the complexity of model checking PAD.

PAD	**general**	**fixed formula**
reachability, reachable property	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
EF	$\in DTIME(tower(n))$, $PSPACE$-hard	$\in$ d-$EXPTIME$ $PSPACE$-hard
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alt.-free modal μ-calc.	undecidable	undecidable
modal μ-calc.	undecidable	undecidable
LTL	undecidable	undecidable
linear-time μ-calc.	undecidable	undecidable

The results on PA are almost the same. The only exception is the complexity of model checking with EF.

PA	**general**	**fixed formula**
reachability, reachable property	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
EF	$\in DTIME(tower(n))$, $PSPACE$-hard	$\in \Sigma_d^p$
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alt.-free modal μ-calc.	undecidable	undecidable
modal μ-calc.	undecidable	undecidable
LTL	undecidable	undecidable
linear-time μ-calc.	undecidable	undecidable

Chapter 9

Petri Nets

Petri nets are a very popular model for concurrent systems. As shown in Chapter 2, they are equivalent to (P,P)-PRS. In Section 9.1 we show that (except for reachability) no branching-time logic is decidable for Petri nets. Section 9.2 is about model checking Petri nets with linear-time logics (LTL and the linear-time μ-calculus). While this is decidable, it is at least as hard as the reachability problem for Petri nets.

9.1 Branching-Time Logics

The reachability problem and the submarking reachability problem (and thus the reachable property problem) are decidable and *EXPSPACE*-hard for Petri nets [May84, Lip76].

Petri nets are equivalent to (P,P)-PRS and thus they subsume BPP, which are $(1,P)$-PRS (see Chapter 2). Since model checking with the logic EG is undecidable for BPP (see Section 6.3), it is undecidable for Petri nets too. This result even holds for a fixed EG-formula.

Model checking Petri nets with EF is undecidable too. This was first proved by Esparza in [Esp94]. The proof there contains a slight error, which was corrected in [Esp97]. The idea is to prove undecidability by a reduction of the *reachability set containment problem* to the model checking problem.

REACHABILITY SET CONTAINMENT

Instance: Two Petri nets N_1 and N_2 having the same number of places and a bijection f between the sets of places of N_1 and N_2. f can be extended to a bijection on markings in the obvious way. Let M_0^1, M_0^2 be the initial markings of N_1, N_2.

Question: Does the following property hold? For every reachable marking M of N_1, $f(M)$ is a reachable marking of N_2.

Rabin showed that this problem is undecidable by a reduction of Hilbert's 10th problem. A proof by Jančar [Jan94, Jan95] uses a reduction of the halting problem for counter machines.

We sketch the reduction of the reachability set containment problem to the model checking problem. It is similar to the one in [Esp97], but slightly simpler. We assume that the transitions in N_1, N_2 are not labeled with atomic actions.

1. Put N_1 and N_2 side by side.
2. Add a place A and arcs from A to every transition in N_1 and back. Put one token on A.
3. Add a new transition t and a place B and arcs from A to t and from t to B. The transition t is labeled with the atomic action a. Place B is initially unmarked.
4. Add arcs from B to every transition in N_2 and back.
5. For every pair of places $(s, f(s))$ add a transition t_s and arcs from s to t_s, $f(s)$ to t_s, B to t_s and t_s to B.
6. For every place s in N_1 add a transition t'_s labeled with action b and arcs from s to t'_s and back. Do the same for N_2.

Figure 9.1 illustrates this construction.

Remember that the formula 'a' means that the atomic action a is enabled (see Subsection 3.1.1).

Lemma 9.1.1 *An instance of the reachability set containment problem has answer 'yes' if and only if the newly constructed Petri net satisfies the EF-formula*

$$\Box(a \ \Rightarrow \ \Diamond(\neg a \ \wedge \ \neg b))$$

Proof Directly from the definition of the net and the interpretation of the formula. ■

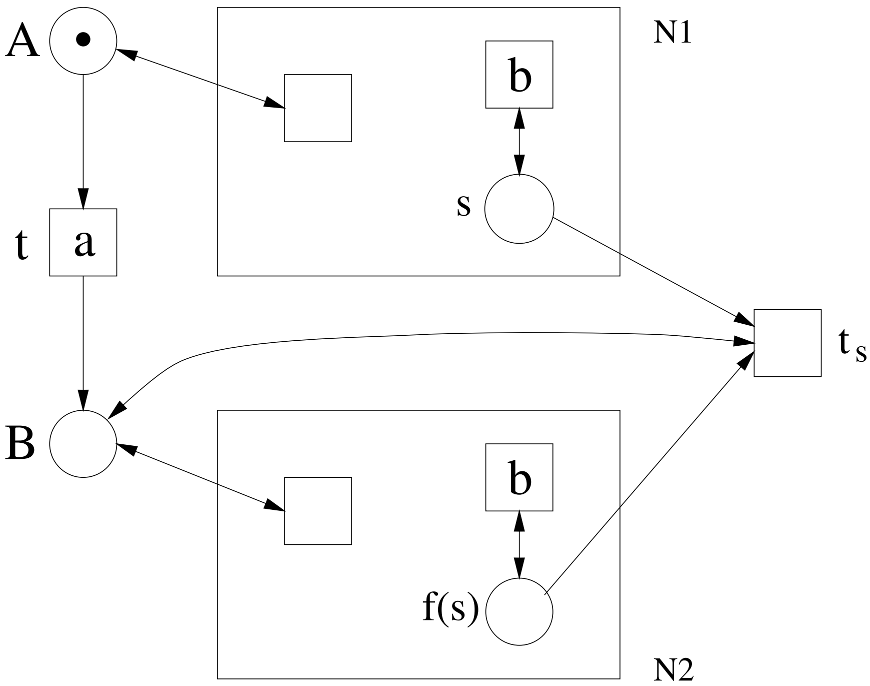


Figure 9.1: Reducing reachability set containment to model checking with EF

Theorem 9.1.2 *Model checking with EF is undecidable for Petri nets, even for a fixed EF-formula.*

Proof Directly from the undecidability of the reachability set containment problem for Petri nets and Lemma 9.1.1. ■

9.2 Linear-Time Logics

9.2.1 The Complexity of the Problem

Model checking Petri nets with LTL and the linear-time μ-calculus is decidable, but at least as hard as the reachability problem for Petri nets [Esp94].

The hardness result is easy to prove. As shown in Section 3.2, the reachable property problem can be encoded in LTL. It was also shown that for Petri nets the reachable property problem subsumes the reachability problem. (This is not true for some other models.) This hardness result even holds for a fixed LTL-formula, since the formula

$$(\bigcirc true)\, w\mathcal{U}\, false$$

expresses the property that no deadlock is reachable. For Petri nets this problem has the same complexity as reachability [Pet81].

The decidability proof in [Esp94] uses the fact that every formula in LTL or the linear-time μ-calculus can be effectively transformed into a Büchi-automaton that describes exactly the same ω-sequences (possibly infinite sequences) of actions [Var88]. (The reverse translation is also possible [Dam92].) So one constructs the Büchi-automaton for the formula and synchronizes this automaton with the Petri net. (Note that, unlike some other models, Petri nets are closed under synchronization with finite-state systems.) For this new system one has to solve finitely many instances of the following problems:

- Is it possible to reach a given state? This is the reachability problem.
- Is there an infinite path that visits a certain state in the Büchi-automaton infinitely often. This problem can be reduced to the problem of the existence of certain infinite runs in Petri nets. It has been show in [Yen92] that it can be done in exponential space.

The problem with the algorithm in [Esp94] is that it has non-elementary complexity and is therefore hardly useful in practice. (The problem is at least as hard as the reachability problem for Petri nets and therefore at least *EXPSPACE*-hard). Even more important is that the algorithm yields hardly any insight on why a property holds. Thus it cannot be used as a proof method in semiautomatic verification.

We consider the model checking problem for Petri nets and the weak linear-time μ-calculus. This logic is a version of the linear-time μ-calculus that is interpreted only on infinite runs. It is equivalent to the fragment of the linear-time μ-calculus without the strong nexttime operator (see Section 3.2 for details on this temporal logic). This model checking problem is *EXPSPACE*-complete [Hab97]. Thus it is hard to solve with fully automated methods.

Here we present a tableau system for this model checking problem. While it cannot provide a more efficient algorithm, it can serve as a basis for a semiautomatic approach to solve this problem. This tableau system is a generalization of the tableau system for the linear-time μ-calculus and finite-state systems by Bradfield, Esparza and Mader [BEM96] and Stirling and Walker [SW91]. It can be used as a proof method and gives the user much better insight why a property holds. In semiautomatic verification this allows the user to apply his/her knowledge about the system in the verification process.

9.2.2 Preliminaries

The linear-time μ-calculus and its interpretation are defined in Section 3.2. The following definitions are important in the construction of the tableau system.

Definition 9.2.1 For all ordinals $\alpha \in Ord$, the *fixpoint approximants* $\mu^\alpha Z.\Phi$ and $\nu^\alpha Z.\Phi$ are defined by: $\mu^0 Z.\Phi = \mathit{false}$ and $\nu^0 Z.\Phi = \mathit{true}$, $\sigma^{\alpha+1} Z.\Phi = \Phi[\sigma^\alpha Z.\Phi/Z]$, $\mu^\lambda Z.\Phi = \bigvee_{\alpha \leq \lambda} \mu^\alpha Z.\Phi$, $\nu^\lambda Z.\Phi = \bigwedge_{\alpha \leq \lambda} \nu^\alpha Z.\Phi$, where λ is a limit ordinal.

Proposition 9.2.2 (Knaster-Tarski) $\mu Z.\Phi = \bigvee_\alpha \mu^\alpha Z.\Phi$, $\nu Z.\Phi = \bigwedge_\alpha \nu^\alpha Z.\Phi$.

Definition 9.2.3 The μ-signature $\mu\text{-}sig(\sigma, \Phi)$ of a run σ w.r.t. a formula Φ (where $\sigma \models \Phi$) is the lexicographically least sequence $\zeta_1, \ldots, \zeta_k$ such that $\sigma \models \Phi[\mu^{\zeta_i} Z_i.\Phi_i/\mu Z_i.\Phi_i]$ where $\mu Z_i.\Phi_i$ are the μ-subformulae of Φ in order of depth, i.e. in some (fixed) order such that subformulae appear after any containing subformulae. The containing formula is called higher and the contained subformula is called lower.

Dually, the ν-signature of $\sigma \not\models \Phi$ is the least sequence s.t. $\sigma \not\models \Phi[\nu^{\zeta_i} Z_i.\Phi_i/\nu Z_i.\Phi_i]$.

The *closure* of a formula in the weak linear-time μ-calculus is the set of its subformulae modulo unfolding of fixpoint operators.

Definition 9.2.4 (Closure)
The closure $Cl(\Phi)$ of a formula Φ is defined as follows:

$$
\begin{array}{rcl}
Cl(Q) & := & \{Q\} \\
Cl(\odot_A \Phi) & := & \{\odot_A \Phi\} \cup Cl(\Phi) \\
Cl(\Phi_1 \wedge \Phi_2) & := & \{\Phi_1 \wedge \Phi_2\} \cup Cl(\Phi_1) \cup Cl(\Phi_2) \\
Cl(\Phi_1 \vee \Phi_2) & := & \{\Phi_1 \vee \Phi_2\} \cup Cl(\Phi_1) \cup Cl(\Phi_2) \\
Cl(\sigma X.\Phi) & := & \{\sigma X.\Phi\} \cup Cl(\Phi[\sigma X.\Phi/X])
\end{array}
$$

where Q is an atomic proposition and $\sigma \in \{\mu, \nu\}$.

These preliminary definitions apply to finite or infinite systems. For a thorough treatment of finite systems we refer to [BEM96] and [SW91]. Here we are interested in infinite systems described by general Petri nets.

Definition 9.2.5 (ω-markings of Petri nets)
A labeled Petri net $N = (S, T, W, L, Act)$ consists of a finite set of places S, a finite set of transitions T, a function $W : S \times T \cup T \times S \rightarrow \mathbb{N}$ that assigns weights to the arcs, a set of actions Act and a labeling function $L : T \rightarrow Act$ that assigns actions to the transitions.

Markings of nets will be denoted by M. As a technicality markings will be mappings $S \mapsto (\mathbb{N} \cup \{\omega\})$ instead of $S \mapsto \mathbb{N}$, where ω is the first limit ordinal. (It follows that for every $k \in \mathbb{N}$ we have $k < \omega$, $\omega + k = \omega$ and $\omega - k = \omega$.)

In the rest of this section we will use the weak linear-time μ-calculus (see Section 3.2). It does not contain the strong nexttime operator $\bigcirc$. This is done in order to make it impossible to express the state of deadlock with the calculus and to avoid having to solve the reachability problem for Petri nets in the tableau. As mentioned earlier the model checking problem is decidable for the full linear-time μ-calculus [Esp94], but so far there exist no tableau methods for solving the reachability problem for Petri nets.

This is not a big restriction, since normally the linear-time μ-calculus is only used to verify liveness-properties of systems. These are mostly fairness-properties like 'In every infinite run action a occurs infinitely often'. Such properties only make sense for infinite runs and thus most systems that are verified with the linear-time μ-calculus are deadlock-free anyway. Therefore one can as well use the weak linear-time μ-calculus.

Definition 9.2.6 We only use the weak nexttime operator $\odot$. As an abbreviation we use finite sets of actions as subscripts instead of single actions. For a set of actions $A = \{a_1, \ldots, a_n\}$ let

$$\underset{A}{\odot} := \underset{a_1}{\odot} \vee \underset{a_2}{\odot} \vee \cdots \vee \underset{a_n}{\odot}$$

We use atomic propositions in the calculus. Let M denote markings of nets. The atomic propositions $Q \in \mathcal{Z}_C$ must satisfy two conditions. Let $\mathcal{W}(Q)$ be the set of markings that satisfy Q.

Q1 $M \in \mathcal{W}(Q) \;\Rightarrow\; \forall M' \leq M.\ M' \in \mathcal{W}(Q)$

Q2 $(M + \omega M') \notin \mathcal{W}(Q) \;\Rightarrow\; \exists k \in \mathbb{N}\; \forall k' \geq k.\; (M + k'M') \notin \mathcal{W}(Q)$.

The conditions Q1 and Q2 imposed on the atomic propositions basically amount to the condition that every atomic proposition has the form

$$P := \{x_1 \leq k_1, x_2 \leq k_2, \ldots, x_n \leq k_n\}$$

where $x_1, \ldots, x_n$ are the places in the net and $k_1, \ldots, k_n \in \mathbb{N} \cup \{\omega\}$. A marking M satisfies the atomic proposition P iff $\forall i \in \{1, \ldots, n\}.M(x_i) \leq k_i$. In Subsection 9.2.9 we show that the tableau system can be generalized to a larger class of atomic propositions.

The following three subsections closely follow the presentation in [BEM96]. The main new points are in in Subsection 9.2.6 and the rest of the chapter.

9.2.3 The Sequents

An important difference between the modal μ-calculus and the linear-time μ-calculus is the treatment of disjunction. In a tableau system for the **modal** μ-calculus (see [SW91]) the sequents have the form $M \vdash \Phi$, which means that the state s satisfies the formula Φ. So $M \models \Phi \vee \Psi$ means $M \in [\![\Phi]\!] \cup [\![\Psi]\!]$ and therefore implies either $M \models \Phi$ or $M \models \Psi$. Thus the two rules

$$\frac{M \vdash \Phi \vee \Psi}{M \vdash \Phi} \qquad \frac{M \vdash \Phi \vee \Psi}{M \vdash \Psi}$$

are complete.

For the **linear** time μ-calculus the situation is different, because here $M \models \Phi \vee \Psi$ means $\{M | \sigma(0) = M\} \subseteq [\![\Phi]\!] \cup [\![\Psi]\!]$. As $\{\sigma | \sigma(0) = M\}$ is a set of runs (that can have more than one element) we can no longer infer $M \models [\![\Phi]\!]$ or $M \models [\![\Psi]\!]$: some runs starting at M may satisfy Φ but not Ψ, while others may satisfy Ψ and not Φ. The solution is to allow sets of formulae in the right hand side of the sequent that are interpreted disjunctively (see [BEM96]). This way, the rule

$$\frac{M \vdash \Phi \vee \Psi}{M \vdash \Phi, \Psi}$$

is sound and complete.

Now the tableau consists of nodes of the form $M \vdash \Phi_1, \ldots, \Phi_k$, where the formulae $\Phi_1, \ldots, \Phi_k$ are interpreted disjunctively. The branches in the tableau are interpreted conjunctively, so the tableau is successful iff all branches are successful. (This is different from other tableau systems in Chapters 8 or 10 where the branches are interpreted disjunctively.)

9.2.4 The Basic Rules

The rules for the tableau can be divided into two groups: the basic rules and the special rules. While the basic rules are sufficient for finite-state systems, the special rules are needed for Petri nets. We will define and discuss the basic rules first since they are more intuitive, and make the necessary adjustments and extensions by the special rules later. To simplify the notation we define that $\Gamma, \Delta, \ldots$ denote sequences of formulae (i.e. $\Gamma = \Phi_1, \Phi_2, \ldots, \Phi_n$ and $\odot_A \Gamma = \odot_A \Phi_1, \ldots, \odot_A \Phi_n$).

$$\wedge \qquad \frac{M \vdash \Gamma, \Phi \wedge \Psi}{M \vdash \Gamma, \Phi \quad M \vdash \Gamma, \Psi}$$

$$\vee \qquad \frac{M \vdash \Gamma, \Phi \vee \Psi}{M \vdash \Gamma, \Phi, \Psi}$$

$$\mathrm{Q} \qquad \frac{M \vdash \Gamma, Q}{M \vdash \Gamma} \quad \text{where } M \notin \mathcal{W}(Q)$$

$$\odot \qquad \frac{M \vdash \odot_{A_1}\Gamma_1, \ldots, \odot_{A_n}\Gamma_n}{M_1 \vdash \Delta \ \ldots \ M_k \vdash \Delta} \quad \text{where } \Delta = \Gamma_1, \ldots, \Gamma_k \text{ and}$$

$$\{M_1, \ldots, M_k\} = \left\{ M' \mid \exists a \in \cap_{i=1}^{n} A_i.\ M \xrightarrow{a} M' \right\}$$

$$\sigma Z \qquad \frac{M \vdash \Gamma, \sigma Z.\Phi}{M \vdash \Gamma, \Phi[\sigma Z.\Phi/Z]}$$

Additionally we use the following 'cleanup-procedure' after each rule application: If a formula Φ occurs in a sequence Γ more than once, then delete all occurrences but the first.

Lemma 9.2.7 *The antecedent of a rule is true if and only if all its consequents are true.*

Proof Trivially from the definitions. ■

For these basic rules the result of the application of a rule to a sequent is completely determined by the sequent. In other words, the child-nodes are completely determined by the parent-node. We'll see later that this is not the case for the special rules. There several ancestors (in the path from the root to the sequent) must be taken into account.

9.2.5 Paths and Internal Paths

A *proof tree* is a tree of sequents constructed by the iterated application of rules, starting with a root $M_0 \vdash \Phi_0$. Associated with a path π in a proof tree is a sequence $\sigma = t_1, t_2, \ldots, t_n$ of transitions arising from the applications of the $\odot$-rule in π. We denote this by $\sigma = trans(\pi)$.

In the tableau each node is assigned a unique label n_i. Let $n_i : M_i \vdash \Phi_i$ and $n_j : M_j \vdash \Phi_j$ be two nodes in the tableau. By $n_i \simeq n_j$ we mean that $M_i = M_j$ and $\Phi_i = \Phi_j$. We write $n_j \ll n_i$ if n_j occurs earlier than n_i in the path from the root to n_i. It follows that $\ll$ is a partial order on the set of nodes in a tableau.

The price we pay for allowing sets of formulae in the right hand side of a sequent is that a path in the proof tree has a more complex internal structure: a set of *internal paths* describing the dependencies between formulae at different nodes. The path

$$\dfrac{n_1 : M \vdash (\odot_{\{a,b\}}\Phi \wedge \Psi) \vee \odot_{\{a,c\}}\Psi}{\dfrac{n_2 : M \vdash \odot_{\{a,b\}}\Phi \wedge \Psi, \odot_{\{a,c\}}\Psi}{\dfrac{n_3 : M' \vdash \Phi \wedge \Psi, \Psi}{n_4 : M' \vdash \Phi, \Psi}}}$$

has the following internal paths:

$$\begin{array}{ccc}
n_1 : (\odot_{\{a,b\}}\Phi \wedge \Psi) \vee \odot_{\{a,c\}}\Psi & \qquad & n_1 : (\odot_{\{a,b\}}\Phi \wedge \Psi) \vee \odot_{\{a,c\}}\Psi \\
\downarrow & & \downarrow \\
n_2 : \odot_{\{a,b\}}\Phi \wedge \Psi & & n_2 : \odot_{\{a,c\}}\Psi \\
\downarrow & & \downarrow \\
n_3 : \Phi \wedge \Psi & & n_3 : \Psi \\
\downarrow & & \downarrow \\
n_4 : \Phi & & n_4 : \Psi
\end{array}$$

Intuitively, the truth of a sequent depends on the structure of the internal paths starting at it, particularly on which μ or ν-variables are unfolded in those paths.

Definition 9.2.8 (Internal paths, internal circuits)
Let π be a path of the proof tree. An *internal path* of π is a finite or infinite

sequence of triples $(n_1, M_1, \Phi_1)(n_2, M_2, \Phi_2), \ldots$ s.t. Φ_1 appears in n_1, and for any two consecutive pairs $(n_1, M_i, \Phi_i), (n_{i+1}, M_{i+1}, \Phi_{i+1})$, one of the following cases holds:

- n_{i+1} is a child of n_i, no rule is applied to Φ_i and $\Phi_{i+1} = \Phi_i$, or
- n_{i+1} is a child of n_i, some rule different from Q is applied to Φ_i, and Φ_{i+1}, M_{i+1} are the formula/marking given by the rule application.

An *internal circuit* of a finite path $\pi = n_1 n_2 \ldots n_k$ such that $n_1 : M_1 \vdash \Gamma$, $n_k : M_k \vdash \Gamma$ and $M_1 \leq M_k$ is a finite sequence of internal paths of π

$$((n_1, M_1, \Phi_1) \ldots (n_k, M_k, \Phi_k)) \quad ((n_1, M_1, \Phi_{k+1}) \ldots (n_k, M_k, \Phi_{2k})) \ldots$$
$$\ldots ((n_1, M_1, \Phi_{jk+1}) \ldots (n_k, M_k, \Phi_{(j+1)k})) \quad \text{for } j \in \mathbb{N}$$

such that $\Phi_{ik+1} = \Phi_{ik}$, $\Phi_1 = \Phi_{(j+1)k}$ and $M_1 \leq M_k$ and $\Phi_1 \in \Gamma$.

The *characteristic* of a finite internal path is the highest variable that is unfolded (by the σZ-rule) or the symbol $\perp$ if no variable is unfolded; the characteristic of an infinite internal path is the highest variable that is unfolded infinitely often. If the characteristic of an internal path is a ν-variable (μ-variable), then we say that the path has ν-characteristic (μ-characteristic). For a path $n \ldots n'$ the set $Int(n, n')$ is defined as the set of triples (Φ, Φ', Z) such that there exists an internal path $(n, \Phi) \ldots (n', \Phi')$ with characteristic Z.

It is easy to see that if the formula at the root of the proof tree is guarded (every variable occurs within the scope of a next-operator $\odot$), then the characteristic of any internal circuit is always different from $\perp$.

9.2.6 The Special Rules

Before defining the special rules we must make some additions. We assign each node a label consisting of a finite set of pairs $\in \mathbb{N}^l \times N$, where l is the number of places of the Petri net and N the set of nodes in the tableau. The label of the root node is the empty set. For a node n with state M, label D and sequence of formulae Γ we write $n(D) : M \vdash \Gamma$. If the label is of no concern then we just write "?" for it.

Child-nodes that are created by basic rule applications do not inherit these labels; they are only introduced by the special rules.

The special rules are introduced to deal with the problems arising from the fact the Petri nets can have infinite state spaces. They ensure the finiteness of the tableau. The intuition is as follows:

Let M be a marking and $M' \geq M$. If there is an unsuccessful run (one that doesn't satisfy the formula) starting at M, then the same run can also start at M'. So the chance to find an unsuccessful run is better if the start-marking is larger. A new marking that contains ω on some places is introduced to represent infinitely many reachable markings with arbitrarily high numbers of tokens in some places. Note that the ω does not mean that there are infinitely many tokens on this place, but only that there are reachable markings with arbitrarily high numbers of tokens on this place. So the ω-rule is as follows:

$$\omega \qquad \frac{n_2(D) : M_2 \vdash \Gamma}{n_2(D) : M_2 + \omega(M_2 - M_1) \vdash \Gamma}$$

With the following side-condition: There is a previous node $n_1(?) : M_1 \vdash \Gamma$ s.t. $n_2 \gg n_1$, $M_2 \geq M_1$ and there is a place s s.t. $M_1(s) < M_2(s) \neq \omega$

A tableau can be seen as an attempt to construct an unsuccessful run. The M-rule is introduced to cut off branches that don't give any new information on the problem if an unsuccessful run can possibly be constructed.

$$M \qquad \frac{n(D) : M \vdash \Gamma}{n(D \cup \{(\delta, n'')\}) : M \vdash \Gamma}$$

With the following side-condition: There are two ancestors $n'(?) : M \vdash \Gamma$ and $n''(?) : M \vdash \Gamma$ s.t. $n \gg n' \gg n''$ and $Int(n'', n') = Int(n'', n)$ and δ is the effect-vector of the sequence of transitions fired between n' and n (see Def. 2.3.4).

If the conditions for the M-rule are satisfied then the path from node n' to node n gives us no new information for the construction of an unsuccessful run. This is because of the condition $Int(n'', n') = Int(n'', n)$. The only thing worth remembering are the changes δ in the marking of the net. By adding the vector to the label of the node we remember that we could insert this piece of the branch as often as we want, and change the marking by δ. This is necessary, because later in the tableau it might turn out that we should have inserted this piece of the branch between n' and n a certain number of times in order to be able to construct an infinite unsuccessful run. However, at the point where the M-rule is applied we don't know yet how often to insert this part of the branch.

Remark 9.2.9 *The special rules only have one antecedent and one succedent. Once the succedent is constructed the antecedent is no longer relevant. So the tableau could be simplified by just replacing the antecedent by the succedent in the case of the special rules. (See [May96b]).*

The basic rules are always applied at the end of the sequence of formulae which form a sequent. We define that the special rules take precedence over the basic rules and the ω-rule takes precedence over the M-rule.

Lemma 9.2.10 *The tableau for a given root is unique.*

Proof Directly from the definition. ■

Now we can define the terminal nodes.

Definition 9.2.11 A node $n(?) : M \vdash \Gamma$ is a *terminal* if any of the following conditions is satisfied:

1. $\Gamma = Q$ and $M \notin \mathcal{W}(Q)$
2. $\Gamma = \odot_{A_1}\Gamma_1, \ldots, \odot_{A_n}\Gamma_n$ and $\nexists a \in \bigcap_{i=1}^{n} A_i, M'.\ M \xrightarrow{a} M'$
3. $Q \in \Gamma$ and $M \in \mathcal{W}(Q)$
4. n has an ancestor $n' \simeq n$ s.t. $n' \ll n$ and
   - every internal circuit of the path $n' \ldots n$ has μ-characteristic, and
   - Let δ_0 be the effect-vector of the sequence of fired transitions between n' and n. Let
     $$\{\delta_1, \ldots, \delta_k\} := \{\delta \mid \exists \tilde{n}.\ n' \ll \tilde{n}(D) \ll n\ \wedge\ \exists(\delta, \hat{n}) \in D.\ \hat{n} \gg n'\}$$
     There are $x_1, \ldots, x_k \in \mathbb{N}$ s.t. $\delta_0 + x_1\delta_1 + \cdots + x_k\delta_k \geq \vec{0}$.
5. There are nodes $n''(?) : M \vdash \Gamma$ and $n'(D) : M \vdash \Gamma$ s.t. $n'' \ll n' \ll n$. Let π_1 be the path between n'' and n' and π_2 the path between n' and n. Let δ be the effect-vector of the sequence of transitions fired in π_2. It must hold that $\pi_1 = \pi_2$ and $\exists \tilde{n}.\ (\delta, \tilde{n}) \in D$.

Terminals of type 1 and 4 are unsuccessful, and terminals of type 2,3 and 5 are successful.

The tableau is a finite proof tree whose leaves (and no other nodes) are terminals. It is successful iff all its terminals are successful.

The intuition behind the definition of the special rules and the terminals is the following: Each path of the tableau can be seen as an attempt to construct a *false* run of the system, i.e. a run that does not satisfy the formula at the root. The terminals identify the points at which we have gathered enough information either to construct such a run (unsuccessful terminal) or to give up searching the continuations of the path (successful terminal), because either they all lead to true runs, or a false run can be found in a different and shorter branch. Let π be a path of the tableau ending in a terminal n, and let $\sigma = trans(\pi)$.

1. If n is of type 1 then it is of the form $M \vdash Q$, and no nun starting at M satisfies Q. Therefore any run of the form $\sigma\sigma'$ is false.

2. If n is of type 2 then any run of the form $\sigma\sigma'$ is a true run. This is due to the definition of $\odot_A$ since σ has no continuations σ' starting with an action in $\bigcap_{i=1}^{n} A_i$.

3. If n is of type 3 then any run of the form $\sigma\sigma'$ is true.

4. If n is of type 4 then an infinite false run can be constructed. Basically this is because in any chain of dependencies corresponding to this run some μ-variable is unfolded infinitely often. If the highest variable that is unfolded infinitely often is a μ-variable then this run does not satisfy the formula (see Subsection 3.2.3). The details will be explained in the proof in Section 9.2.7.

5. If n is of type 5 then nothing new has happened between n' and n. This is because the same path has already occurred earlier in the tableau between n'' and n'. Even the effect-vector of transitions fired between n' and n has already been recorded in the label of n'. Basically this means that if any false run can be found, then it can be found elsewhere in the tableau in an easier (shorter) way.

9.2.7 Soundness and Completeness

First we show that the tableau is always finite. The following general lemma is very useful for decidability problems about Petri nets. It was proved by Dickson in [Dic13].

Lemma 9.2.12 (Dickson's Lemma)
Given an infinite sequence of vectors $M_1, M_2, M_3, \ldots$ in $\mathbb{N}^k$ there are $i < j$ s.t. $M_i \leq M_j$ ($\leq$ taken componentwise).

Lemma 9.2.13 *For any given root the tableau is finite.*

Proof Let τ be the tableau with root $M_0 \vdash \Phi_0$ and m the number of symbols in Φ_0. It is easy to see that the size of the closure $Cl(\Phi_0)$ of Φ_0 is bounded by m. Therefore at most 2^m different sequents Γ can occur in nodes of the tableau and there are at most 2^{m^3} different Int relations. Let t be the number of transitions in the Petri net. Then each node has at most $max\{2, t\}$ children.

Assume that there is an infinite path in the tableau. Because of the special rule ω and Dickson's lemma (9.2.12) the number of different markings M occurring in nodes of the tableau is finite. Thus there are only finitely many different paths between different nodes with the same sequent $M \vdash \Gamma$.

Because of the special rule M all the effect-vectors of these paths will eventually be stored in the labels of the nodes. So the path will end by termination condition 5, a contradiction.

Thus every path in the tableau has finite length. As each node has only finitely many children the tableau is finite. ∎

Lemma 9.2.14 *If* $M_0 \models \Phi_0$ *then the tableau with root-node* $n_0(\{\}) : M_0 \vdash \Phi_0$ *is successful.*

Proof Starting with the root-node $n_0(\{\}) : M_0 \vdash \Phi_0$, apply the rules until the tableau is constructed. The construction terminates by Lemma 9.2.13.

We will assume that there exists an unsuccessful terminal $n(D) : M \vdash \Gamma$ and derive a contradiction. There are two cases:

1. n is of type 1. Then n is of the form $n(D) : M \vdash Q$ and M doesn't satisfy Q. Therefore n is a false node. By condition Q1 and Q2 from Def. 9.2.6 it follows that we could construct another tableau without using the ω-rule that has a path leading to a node $n_2(?) : M' \vdash Q$ s.t. $M' \leq M$ and $\forall s \in S.\ M'(s) \neq M(s) \;\Rightarrow\; M(s) = \omega$ and M' fails Q. This is a contradiction, because by Lemma 9.2.7 the node n_2 should be true.

2. If n is of type 4 then because of condition Q1 and Q2 for any $k \in \mathbb{N}$ it is possible to construct another tableau without using the M- and ω-rules s.t. this tableau contains two nodes $n_1(?) : M_1 \vdash \Gamma \;\ll\; n_2(?) : M_2 \vdash \Gamma$, $M_2 \geq M_1$, $\forall s \in S.M(s) = \omega \;\Rightarrow\; M_1(s) \geq k$ and every internal circuit of the path $n_1 \dots n_2$ has μ-characteristic.

   Let $\sigma = trans(n_1 \dots n_2)$. By our assumption the run σ^ω starting at M_1 satisfies some formula of Γ. Let $\{\Phi_1, \dots, \Phi_l\}$ be the satisfied formulae. Let

$\vec{\sigma}$ be the effect-vector of σ. We know that $\vec{\sigma} \geq \vec{0}$. An internal path starting with Φ_1 of the form $M_1 \vdash \Phi_1 \ldots M_2 = (M_1 + \vec{\sigma}) \vdash \Phi_x \ldots (M_1 + i\vec{\sigma}) \vdash \Phi_y \ldots$ must be periodic. Especially some formula Φ_i must occur infinitely often. Now construct this periodic internal path $\pi = M_1 \vdash \Phi_i \ldots (M_1 + 1 * m\vec{\sigma}) \vdash \Phi_i \ldots (M_1 + j * m\vec{\sigma}) \vdash \Phi_i \ldots$. The construction is guided by inductively associating to each pair $M \vdash \Phi$ a suffix ρ_i of σ^ω s.t. $\rho_i(0) = M$ and ρ_i satisfies Φ. For the initial pair $M_1 \vdash \Phi_i$ this is σ^ω itself. Now we define how to select the $(x+1)$-th element $M' \vdash \Phi'$ and ρ_{x+1}, given the x-th element $M \vdash \Phi$ and ρ_x.

- If $\Phi = Q$ and the Q-rule is applied, then $M \vdash \Phi$ is the last node of π. In the original tableau there is a corresponding marking M_ω s.t. $M \leq M_\omega$ and $\forall s \in S. M_\omega(s) \neq M(s) \Rightarrow M_\omega(s) = \omega \;\wedge\; M(s) \geq k'$. Notice that $k' \in \mathbb{N}$ is finite, but we can choose it arbitrarily high, because we can choose k arbitrarily high and $\vec{\sigma} \geq \vec{0}$. As $M_\omega \not\models Q$ it follows that $M \not\models Q$, because of condition Q2 defined in Def. 9.2.6.
- If $\Phi = \Psi \wedge \Upsilon$ and the $\wedge$-rule is applied to Φ, then $M' = M$, Φ' is either Ψ or Υ, according to the choice in the path from n_1 to n_2, and $\rho_{i+1} = \rho_i$.
- If $\Phi = \Psi \vee \Upsilon$ and the $\vee$-rule is applied to Φ, then $M' = M$, $\rho_{i+1} = \rho_i$ and

$$\Phi' = \begin{cases} \Psi, & \text{if } \mu\text{-}sig(\rho_i, \Psi) \leq \mu\text{-}sig(\rho_i, \Upsilon) \\ \Upsilon, & \text{otherwise} \end{cases}$$

- If $\Phi = \odot_A \Psi$ then the $\odot$-rule is applied and $\Phi' = \Psi$, $\rho_{i+1} = \rho_i^{(1)}$ and M' is the state corresponding to $\rho_{i+1}(0)$.
- If $\Phi = \sigma Z.\Psi$ the σZ-rule is applied and $M' = M$, $\Phi' = \Psi[\sigma Z.\Psi/Z]$ and $\rho_{i+1} = \rho_i$.

There are two possible sub-cases:

(a) π is finite.
Then the last node must be of the form $M \vdash Q$, and the Q-rule is applied. Therefore no run starting at M satisfies Q. This is a contradiction, as the node $M \vdash Q$ should be true.

(b) π is infinite.
Let Z be the characteristic of π. Then Z is also the characteristic of some internal circuit of $n_1 \ldots n_2$, and therefore a μ-variable. Assign to each element $M \vdash \Phi$ of π (with corresponding run ρ) a truncated prefix of $\mu\text{-}sig(\rho, \Phi)$ by removing all ordinals corresponding to μ-variables

lower than Z. Let T_π be the sequence of truncated signatures associated with π.

The sequence T_π is non-increasing, because no variable higher than Z is ever unfolded, and because of the way we defined the internal path where the $\vee$-rule was applied. As we have shown before an infinite number of sequents of the form $M_1 + j * m\vec{\sigma} \vdash \Phi_i$ for $i = 1, 2, \ldots$ occur in π, s.t. each has the associated run σ^ω. So the associated truncated μ-signatures are the same. This is a contradiction, because the truncated μ-signature should decrease as the variable Z is unfolded between two occurrences of this sequent. ∎

Lemma 9.2.15 *If the tableau with root* $n_0(\{\}) : M_0 \vdash \Phi_0$ *is successful, then* $M_0 \models \Phi_0$.

Proof Assume that there is a successful tableau τ for $M_0 \vdash \Phi_0$, but $M_0 \not\models \Phi_0$. We will derive a contradiction.

Assuming that $M_0 \not\models \Phi_0$ there must be a run σ_0 starting at M_0 s.t. $\sigma_0 \notin [\![\Phi_0]\!]$. We will use this run to show the existence of an unsuccessful terminal, contradicting the success of τ.

To do this, we first use σ_0 to construct a (possibly infinite) path π' in a tableau τ' that is constructed without using the special rules (i.e. by the basic rules only). Using this path we will then prove the existence of a finite unsuccessful path π in the tableau τ that is constructed by all rules.

To serve as guide during the construction of the path $\pi' = n_0 n_1 n_2 \ldots$, we inductively associate to each node n_i a suffix ρ_i of σ_0 s.t. the state of n_i is $\rho_i(0)$ and ρ_i fails every formula of n_i. The suffix associated with the root n_0 is σ_0. If n_i is a terminal of type 1, 2, 3 or 4 then (ρ_i, n_i) is the last element of π. Otherwise its successor n_{i+1} and associated suffix ρ_{i+1} are chosen as follows:

- If the $\odot$-rule is applied to n_i, then $\rho_{i+1} = \rho_i^{(1)}$, and n_{i+1} is the child of n_i having $\rho_{i+1}(0)$ as state;
- If the $\wedge$-rule is applied to n_i, then $\rho_{i+1} = \rho_i$ and n_{i+1} is a child of n_i s.t. the ν-signature of ρ_i is preserved (if ρ fails $\Phi \wedge \Psi$ with ν-signature ξ, then ρ fails either Φ or Ψ with ν-signature ξ).
- If one of the rules $\vee$, Q or σZ is applied then $\rho_{i+1} = \rho_i$ and n_{i+1} is the only child of n_i.

As no special rules are used, the labels of all nodes are empty. It follows from Lemma 9.2.7 that every node of π' is false. There are two cases:

1. π' is infinite.
   As there are only finitely many subformulae of Φ_0, there are only finitely many different sequents Γ in the tableau τ'. So π' must contain an infinite subsequence $n_{m_1}, n_{m_2}, \ldots$ s.t. $n_{m_i}(\{\}) : M_{m_i} \vdash \Gamma$. Let $\Gamma = \Phi_1, \ldots, \Phi_n$. We assign to each node n_{m_i} a vector of signatures

   $$\bar{x}_i = (\nu\text{-}sig(\Phi_1, \rho_{m_i}), \ldots, \nu\text{-}sig(\Phi_n, \rho_{m_i}))$$

   By Dickson's Lemma there are two indices $i \leq j$ s.t. $\bar{x}_i \leq \bar{x}_j$ and $M_{m_i} \leq M_{m_j}$. Note that the relation between $\bar{x}_i$ and $\bar{x}_j$ is the pointwise order on vectors, while the order on their components is the lexicographic order.

   Now we prove that every internal circuit of $n_{m_i} \ldots n_{m_j}$ has μ-characteristic.

   Assume there is an internal circuit γ of $n_{m_i} \ldots n_{m_j}$ with ν-characteristic Z. Assign to each element (n_k, Φ_k) of γ a prefix of $\nu\text{-}sig(\Phi_k, \rho_k)$ obtained by removing all ordinals corresponding to ν-variables lower than Z. Let T_γ be the sequence of truncated signatures corresponding to γ. We claim that T_γ is non-increasing. Let n_k and n_{k+1} be two consecutive nodes of γ. If $n_k \neq n_{m_j}$ then n_{k+1} is a successor of n_k and the truncated signature cannot increase when moving from n_k to n_{k+1}, because no variable higher than Z is ever unfolded and because of the way the branch is chosen at the $\wedge$-nodes. If $n_k = n_{m_j}$, then $n_{k+1} = n_{m_i+1}$ and since $\bar{x}_i \leq \bar{x}_j$ the truncated signature cannot increase as well.

   Since Z is unfolded somewhere in γ, the last element of T_γ is lexicographically less than the first. This contradicts the assumption that $\bar{x}_i \leq \bar{x}_j$. Therefore Z must be a μ-variable.

   Using these properties we will now construct a tableau τ'' by using the basic rules and the ω-rule, but omitting the M-rule. The condition Q1 from Def. 9.2.6 ensures that τ'' contains two nodes $n_1(\{\}) : M \vdash \Gamma \ll n_2(\{\}) : M \vdash \Gamma$, s.t. every internal circuit between them has μ-characteristic. (M will possibly contain ωs). (The condition Q1 ensures that the application of the ω-rule does not make atomic propositions true which were false before.) There are two cases:

   (a) Either the ω-rule has been applied before n_{m_i} and n_1 corresponds to n_{m_i} and n_2 to n_{m_j}.
   (b) Or the ω-rule is applied at n_{m_j}. Let π_ω be the path from n_{m_i} to n_{m_j}. Then n_1 corresponds to the modified n_{m_j} and n_2 corresponds to the node that is reached from n_1 via π_ω.

   Note that now both n_1 and n_2 have the same marking M (which can contain ωs).

Let δ be the effect-vector of the sequence of transitions fired between n_1 and n_2. As $M_{m_i} \leq M_{m_j}$ we know that $\delta \geq \vec{0}$.

Now we construct the tableau τ''' using all rules and show that it must contain a terminal of type 4 that occurs at the same place as n_2, or even before. Note that n_2 would be a candidate for such a terminal, were it not for the M-rule and termination condition 5. We will start with the tableau τ'' and successively cut out segments of the path leading from the root to n_2, thus obtaining a shorter path leading to a type 4 terminal. We repeat this until termination condition 5 is not satisfied anywhere in this path. Thus we obtain a path of the tableau that could have been constructed from scratch by using all rules. Let n'', n' and n be the nodes that satisfy the conditions of the M-rule and π_1 be the path from the root to n_1 in τ''. Note that no application of the ω-rule takes place between n'' and n, as well as between n_1 and n_2. There are four cases:

(a) $n \ll n_1$
Let α be the path from n' to n. In the path from n to n_1 we can cut out all the subpaths equal to α, thus obtaining a shorter path. n_2 still satisfies the conditions to be a type 4 terminal.

(b) $n' \ll n_1 \ll n$
Let α be the path from n to n_2 and β the path from n_1 to n. Let n_3 be the node that can be reached from n' with the path $\alpha\beta$. It follows that $n_3 \ll n_2$ and n_3 is a type 4 terminal.

(c) $n'' \ll n_1 \ll n'$
Let α be the path from n to n_2 and β the path from n_1 to n. Let n_3 be the node that can be reached from n'' with the path $\alpha\beta$. It follows that $n_3 \ll n_2$ and n_3 is a type 4 terminal.

(d) $n_1 \ll n''$
Let α be the path from n' to n. In the path from n to n_2 we can now cut out all subpaths equal to α. All internal circuits of the path $n_1 \dots n_2$ still have μ-characteristic, because $Int(n'', n') = Int(n'', n)$. Let δ' be the effect-vector of the sequence of transitions fired in α. n_2 is still a type 4 terminal, because n now carries the additional label (δ', n'') and $n'' \gg n_1$.

So τ''' must contain an unsuccessful terminal. Since the tableau for a given root is unique it follows that $\tau''' = \tau$. This is a contradiction, as τ is successful.

2. π' is finite.
Let n be the last node of π'. n cannot be a terminal of type 5, because all

labels are empty in τ'. n cannot be a terminal of type 2 or 3, because n is false. So n must be a terminal of type 1 or 4.

(a) If n is of type 1 then it must have the form $n(\{\}) : M \vdash Q$ s.t. $M \notin \mathcal{W}(Q)$. There is a path π in τ corresponding to a subsequence σ_0' of σ_0 s.t. πs last node is $n'(?) : M' \vdash Q$ and $\exists M''.\ M' = M + \omega M''$. It follows from condition Q1 in Def. 9.2.6 that $M' \notin \mathcal{W}(Q)$. So n' is an unsuccessful node in τ, a contradiction.

(b) If n is of type 4, then we have the same situation as in case 1. ■

9.2.8 Examples

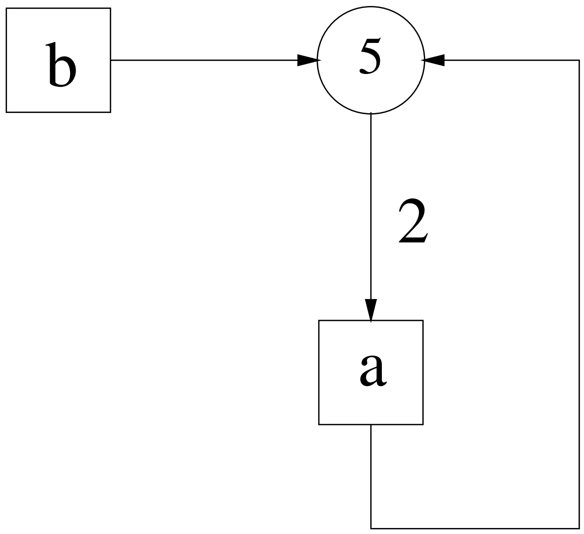


Figure 9.2: A simple Petri net

First we consider a very simple example. The weak linear-time μ-calculus formula

$$\mu x.\ \odot_a x$$

means that there is no infinite run that contains only actions a.

The Petri net in Figure 9.2 satisfies this formula. The tableau that proves this has just one branch. A marking of the net is described by the number of tokens on the place.

$$\begin{array}{rrcr} & (5) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\sigma Z} & (5) & \vdash & \odot_a x \\ \xrightarrow{\odot} & (3) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\sigma Z} & (3) & \vdash & \odot_a x \\ \xrightarrow{\odot} & (1) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\sigma Z} & (1) & \vdash & \odot_a x \end{array}$$

The last node is a successful terminal of type 2.

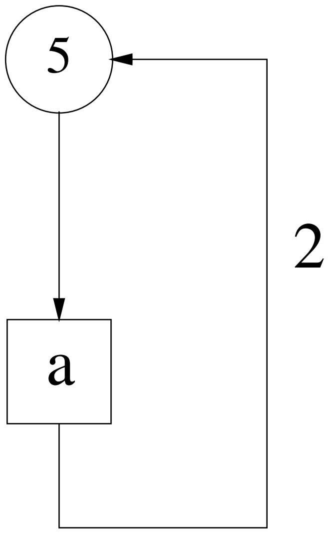


Figure 9.3: The modified Petri net

Now we modify this example. The Petri net in Figure 9.3 does not satisfy the formula $\mu x.\odot_a x$. The tableau that proves this has just one branch. A marking of the net is described by the number of tokens on the place.

$$\begin{array}{rrcr} & n_1 : (5) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\sigma Z} & n_2 : (5) & \vdash & \odot_a x \\ \xrightarrow{\odot} & n_3 : (6) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\omega} & n_4 : (\omega) & \vdash & \mu x.\odot_a x \\ \xrightarrow{\sigma Z} & n_5 : (\omega) & \vdash & \odot_a x \\ \xrightarrow{\odot} & n_6 : (\omega) & \vdash & \mu x.\odot_a x \end{array}$$

The node n_6 is an unsuccessful terminal of type 4. This is because $n_4 \simeq n_6$, the path between n_4 and n_6 has μ-characteristic and the effect-vector of the transitions that are fired between n_4 and n_6 is $(+1)$.

Note that in the last two examples the M-rule was not used. Now we study a more complex example where this rule must be applied. Consider the following weak linear-time μ-calculus formula.

$$\nu x.\, \odot_a (x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y)))$$

Intuitively, the meaning is that there are **no** constants $n, m \in \mathbb{N}$ s.t. a run $a^n (bc^m d)^\omega$ is possible.

Does the system in Figure 9.4 satisfy this formula?

The answer is "No". The tableau that proves this is quite large, so we only describe one unsuccessful branch. Markings of this net are now described by 4-tuples that contain the number of tokens on places A, B, C, D. So the initial marking M is $M := (1,0,1,0)$. Let $M_2 := (1,1,1,1)$, $M_3 := (1,\omega,1,\omega)$, $M_4 := (1,\omega,0,\omega)$.

$$
\begin{array}{lrcr}
 & M & \vdash & \nu x.\, \odot_a (x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y))) \\
\xrightarrow{\sigma Z} & M & \vdash & \odot_a (x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y))) \\
\xrightarrow{\odot} & M_2 & \vdash & x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y)) \\
\xrightarrow{\wedge+\omega} & M_3 & \vdash & \nu x.\, \odot_a (x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y))) \\
\xrightarrow{\sigma Z} & M_3 & \vdash & \odot_a (x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y))) \\
\xrightarrow{\odot} & M_3 & \vdash & x \wedge \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y)) \\
\xrightarrow{\wedge} & n_1 : M_3 & \vdash & \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y)) \\
\xrightarrow{\sigma Z} & M_3 & \vdash & \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y)) \\
\xrightarrow{\odot} & n_2 : M_4 & \vdash & \nu z.\, \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\sigma Z} & M_4 & \vdash & \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\odot} & M_4 & \vdash & z \wedge \odot_d y \\
\xrightarrow{\wedge} & n_3 : M_4 & \vdash & \nu z.\, \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\sigma Z} & M_4 & \vdash & \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\odot} & M_4 & \vdash & z \wedge \odot_d y \\
\xrightarrow{\wedge+M} & n_4(\{((0,2,0,-1), n_2)\}) : M_4 & \vdash & \nu z.\, \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\sigma Z} & M_4 & \vdash & \odot_c (z \wedge \odot_d y) \\
\xrightarrow{\odot} & M_4 & \vdash & z \wedge \odot_d y \\
\xrightarrow{\wedge} & M_4 & \vdash & \odot_d y \\
\xrightarrow{\odot} & M_3 & \vdash & \mu y.\, \odot_b (\nu z.\, \odot_c (z \wedge \odot_d y))
\end{array}
$$

At the node n_4 the M-rule is applied, because n_2, n_3 and n_4 are the same and $Int(n_2, n_3) = Int(n_2, n_4)$. The effect-vector of the transitions fired between n_3 and n_4 is just the effect of the transition that is labeled with the action c.

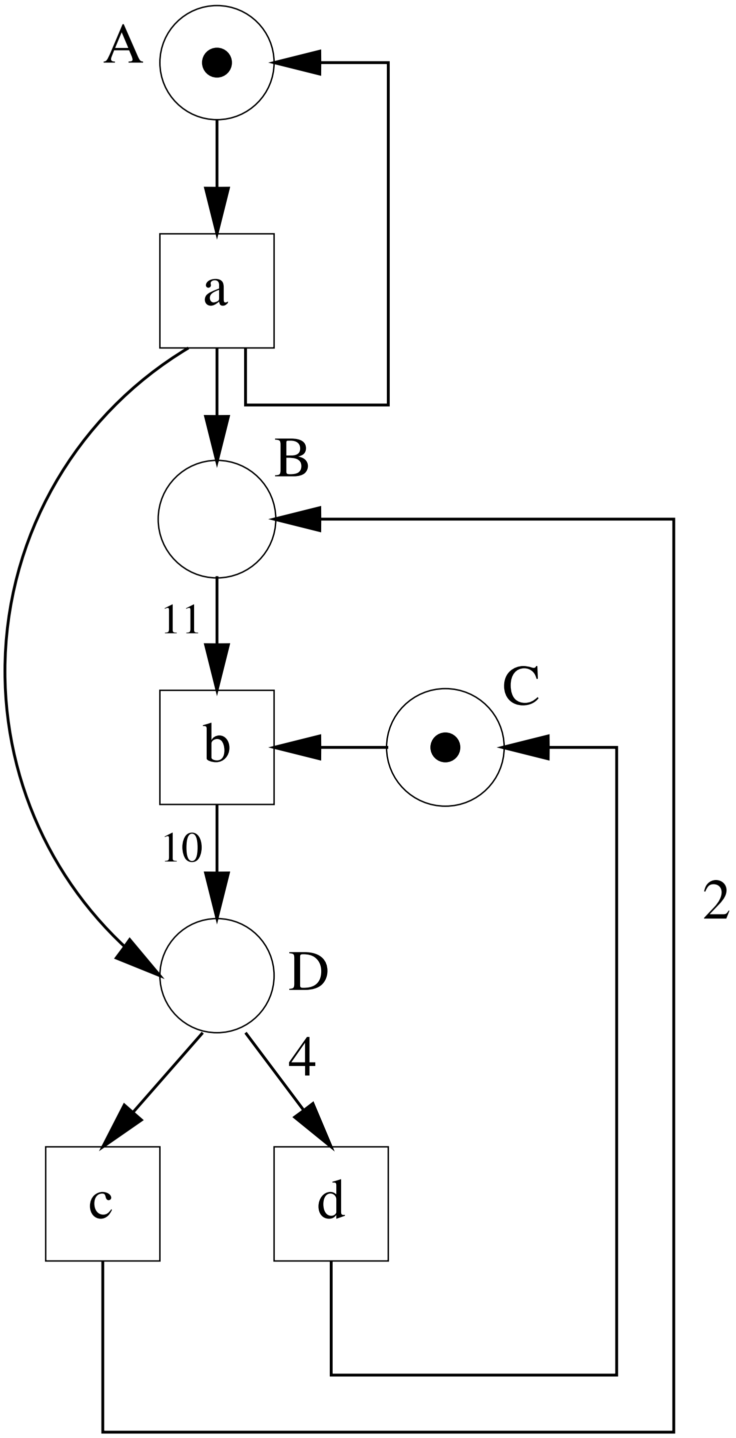


Figure 9.4: A more complex Petri net

The last node in this path already occurred earlier at n_1, and all internal paths between them (only one in this case) have μ-characteristic, because the highest unfolded variable y is a μ-variable. The only problem is that the effect-vector of the sequence of transitions fired between node n_1 and the last node is not positive. Let $E(b)$ be the effect-vector of the transition labeled with the action b, and similarly for the actions c and d. The effect vector of the transitions that are fired between n_1 and the last node is $E(b) + 3 * E(c) + E(d) = (0, -5, 0, 3)$. Fortunately we have a label at n_3 with an entry marked with n_2, which is below n_1 (and thus we may use it). The question is now if there is a $k \in \mathbb{N}$ s.t. $(0, -5, 0, 3) + k * (0, 2, 0, -1) \geq \vec{0}$? We see that there is one (in this case only one) solution, $k = 3$. Thus termination-condition 4 is satisfied and the branch is unsuccessful. Thus the system does not satisfy the formula.

It can be verified that even a slight change of the system makes it satisfy the formula. If the arc from B to b is labeled by 13 instead of 11, then no infinite sequence of the form $(bc^n d)^\omega$ is possible, although such sequences of arbitrary length are possible if enough a's are done first. Thus the above construction is impossible. Any infinite path must contain infinitely many a's and in the tableau each a-action is accompanied by an unfolding of the outermost ν-variable x. Therefore the modified system satisfies the formula.

9.2.9 Extensions

The restrictions Q1 and Q2 imposed on the atomic propositions in Def. 9.2.6 basically amount to the condition that every atomic proposition has the form

$$P := \{x_1 \leq k_1, x_2 \leq k_2, \ldots, x_n \leq k_n\}$$

where $x_1, \ldots, x_n$ are the places in the net and $k_1, \ldots, k_n \in \mathbb{N} \cup \{\omega\}$. A marking M satisfies the atomic proposition P iff $\forall i \in \{1, \ldots, n\}.M(x_i) \leq k_i$.

A possible generalization is to allow conditions of the form $s_i = k_i$ instead of $s_i \leq k_i$. We cannot express reachability even with these new atomic proposition, because they are not closed under complement. In order to express deadlock reachability we would need propositions of the form $s_i \geq k_i$, which we don't have.

The only problem with these new atomic propositions is the application of the ω-rule. We can no longer assume that if a marking M fails an atomic proposition P, then every marking $M' \geq M$ also fails P. So we can no longer assume that whenever the ω-rule is applicable, an infinite path could be created leading to markings with arbitrary high numbers of tokens in some places. This is because now it might be possible to terminate this path by termination condition 3. The solution is to modify the side condition for the ω-rule appropriately. It is now:

There is a previous node $n_1(?) : M_1 \vdash \Gamma$ s.t. $n_2 \gg n_1$ and

- $M_2 \geq M_1$ and there is a place s s.t. $M_1(s) < M_2(s) \neq \omega$ and
- Let $M_1', \ldots, M_m'$ be the states of nodes in the tableau between n_1 and n_2 where the Q-rule is applied and $P_1, \ldots, P_m$ the corresponding atomic propositions. $\forall i \in \{1, \ldots, m\}. \forall k \in \mathbb{N}.\ M_i' + k(M_2 - M_1) \notin \mathcal{W}(P_i)$.

Note that this condition is still decidable for these atomic propositions.

Now it can be seen that the tableau method can be generalized for atomic propositions satisfying the following weaker conditions:

P1 $M \notin \mathcal{W}(Q) \;\Rightarrow\; M + \omega M' \notin \mathcal{W}(Q)$

P2 It is decidable for a marking M and a vector $\delta \geq \vec{0}$ if there is an $i \in \mathbb{N}$ s.t. $M + i\delta \in \mathcal{W}(Q)$.

Q2 $M + \omega M' \notin \mathcal{W}(Q) \;\Rightarrow\; \exists k \in \mathbb{N}\ \forall k' \geq k.\ (M + k'M') \notin \mathcal{W}(Q)$.

9.2.10 Related Work

We have presented a tableau system for model checking Petri nets with the weak linear-time μ-calculus, a fairly expressive fragment of the linear-time μ-calculus. It uses the technique of examining internal paths that was first used for finite-state systems in [BEM96]. Our tableau system is only for the weak linear-time μ-calculus and Petri nets. However, it yields a tableau system for the full linear-time μ-calculus if only finite-state systems are considered. Now we describe this idea.

Finite-state systems can be modeled by Petri nets by assigning a place to each state and a transition to each arc. The initial marking puts only one token on the place that corresponds to the initial state in the finite-state system. The net only contains one token at every reachable marking. The only problem that remains is that the weak linear-time μ-calculus does not contain the strong nexttime operator. However, in this special case, we can encode the strong nexttime operator with the weak nexttime operator and atomic propositions. For any atomic action a we define the atomic proposition P_a s.t.

$$M \in \mathcal{W}(P_a) \;:\Leftrightarrow\; \exists M'.\ M \xrightarrow{a} M' \wedge\ \nexists M'', b \neq a.\ M \xrightarrow{b} M''$$

These atomic propositions satisfy the conditions $Q1$ and $Q2$ from Definition 9.2.6, because the net only contains one token at every reachable marking. Of course this is not the case for general nets. Now we can express the strong nexttime operator. For every formula Φ and every marking M we have

$$M \models \bigcirc_a \Phi \;\Leftrightarrow\; M \models \odot_a \Phi \wedge P_a$$

The tableau system we presented is decidable. However, it is not intended to be used as a decision procedure, but rather as a proof method. The advantage of the tableau system is that it gives the user better insight and more control over the proof. This allows the user to apply his/her knowledge about the system by guiding the search in the tableau and helping to avoid unnecessary branches (see Subsection 9.2.8). Thus tableau systems are particularly useful for computer-assisted verification, i.e. theorem provers with human interaction like PVS [ORSv95] or "Isabelle" [Pau94].

9.3 Conclusion

The complexity of model checking problems for Petri nets is summarized in the following table. Again we distinguish the general problem and the problem for fixed formulae. Note that for Petri nets the problem if the empty marking is reachable has the same complexity as the general reachability problem [Pet81]. The same holds for the problem if a deadlock is reachable [Pet81]. Thus the complexity of the reachability problem for a fixed given marking is the same as the complexity of the general reachability problem.

Petri nets	**general**	**fixed formula**
reachability, reachable property	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
EF	undecidable	undecidable
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alternation-free modal μ-calc.	undecidable	undecidable
modal μ-calc.	undecidable	undecidable
LTL	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
linear-time μ-calc.	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard

Chapter 10

PRS and PAN

The most general and most expressive class of systems in the PRS-hierarchy is (G, G)-PRS (here simply called PRS). PRS have been introduced in [May97c]. They subsume all the other models in the PRS-hierarchy and are strictly more general (see Section 2.5).

PRS and PAN are both extensions of Petri nets by sequential composition, which can be interpreted as the possibility to call subroutines. However, unlike in PAN, PRS-subroutines can influence the behavior of their caller after their termination. This can be interpreted as the possibility to return a value to the caller (see Section 2.3 and Section 10.3), which makes PRS much more useful than PAN for modeling real programs. The results on decidability and complexity of model checking are the same for PAN and PRS and thus we consider only PRS in this chapter.

Model checking PRS or PAN with any of the temporal logics defined in Chapter 3 is undecidable (except for the trivial logics WL and Hennessy-Milner logic). This is because model checking Petri nets with EF is undecidable [BE97, Esp97] (see Chapter 9), model checking BPP with EG is undecidable [EK95] (see Chapter 6) and model checking PA with LTL or linear-time μ-calculus is undecidable [BH96] (see Chapter 8). The other logics (UB, CTL, modal μ-calculus) are more general than EF and EG, and thus also undecidable for PRS and PAN.

The only verification problems that are still decidable for PRS are *reachability* and *reachable property* (see Subsection 3.1.2). Thus PRS are not Turing-powerful. In Section 10.1 we show that the reachability problem is decidable for PRS. Section 10.2 shows the decidability of the reachable property problem. Thus it is decidable if there is a reachable state that satisfies certain properties that can be encoded in a simple logic. In Section 10.3 we describe applications of these algorithms.

10.1 The Reachability Problem

In this section we show that the reachability problem is decidable for PRS. Thus PRS are not Turing-powerful.

For Petri nets reachability is decidable and *EXPSPACE*-hard [May84, Lip76]. Here we show that reachability is decidable for PRS by reducing the problem to the reachability problem for Petri nets. As the atomic actions are not important for reachability, we'll ignore them for the rest of this section and write just $t_1 \rightarrow t_2$ instead of $t_1 \xrightarrow{a} t_2$.

We prove the decidability of reachability in two steps. First we show that it suffices to decide the problem for a special class of PRS, the PRS in transitive normal form (see below). Then we solve the problem for this subclass of PRS.

Definition 10.1.1 For a PRS Δ and process terms $t, t' \in \mathcal{T}$ we define

$$t \succ^{\Delta} t' \;:\Longleftrightarrow\; \exists \sigma.\, t \xrightarrow{\sigma} t'$$

where σ is a sequence of applications of rules in Δ. If Δ is fixed, then we just write $t \succ t'$.

Δ is in *normal form* iff all rules in Δ have one of the following two forms:

Par-Rule $X_1 \| X_2 \| \ldots \| X_i \rightarrow Y_1 \| Y_2 \| \ldots \| Y_k,\;\; i, k \in \mathbb{N}$.

Seq-Rule $X_1.X_2 \rightarrow Y$ or $X \rightarrow Y_1.Y_2$ or $X \rightarrow Y$.

where X, Y, X_i, Y_i are process variables.

The only rules that are both seq-rules and par-rules are of the form $X \rightarrow Y$. The following relations $\succ^{\Delta}_{par}$ and $\succ^{\Delta}_{seq}$ are only technicalities used in the proofs.

$$t \succ^{\Delta}_{par} t' \;:\Longleftrightarrow\; \exists \sigma.\, t \xrightarrow{\sigma} t' \text{ and all rules used in } \sigma \text{ are par-rules from } \Delta$$

$$t \succ^{\Delta}_{seq} t' \;:\Longleftrightarrow\; \exists \sigma.\, t \xrightarrow{\sigma} t' \text{ and all rules used in } \sigma \text{ are seq-rules from } \Delta$$

A PRS Δ is in *transitive normal form* iff it is in normal form and for all $X, Y \in Var$

$$X \succ^{\Delta} Y \;\Rightarrow\; (X \rightarrow Y) \in \Delta$$

Proposition 10.1.2 *Let Δ be a PRS in transitive normal form and t_1, t_2 process terms that do not contain the operator for sequential composition. It is decidable if $t_1 \succ^{\Delta}_{par} t_2$.*

Proof This follows directly from the decidability of the reachability problem for Petri nets [May84]. ■

The reachability problem for PRS is reducible to the reachability problem for PRS in normal form.

Lemma 10.1.3 *Let Δ be a PRS using only variables from the finite set $Var(\Delta)$. Let $t_1, t_2 \in \mathcal{T}$ be two terms containing only variables from $Var(\Delta)$.*

Then a PRS Δ' in normal form and terms t'_1 and t'_2 can be effectively constructed s.t. Δ', t'_1 and t'_2 use only variables from the finite set V' (with $Var(\Delta) \subseteq V' \subset Var$) and

$$t_1 \succ^{\Delta} t_2 \iff t'_1 \succ^{\Delta'} t'_2$$

Proof Let k_i be the number of rules $(t_1 \to t_2)$ in Δ that are neither par-rules nor seq-rules and $size(t_1) + size(t_2) = i$. Let n be the maximal i s.t. $k_i \neq 0$. (n exists because Δ is finite). We define

$$Norm(\Delta) := (k_n, k_{n-1}, \ldots, k_1)$$

These norms are ordered lexicographically. Δ is in normal form iff $Norm(\Delta) = (0, \ldots, 0)$. Now we describe a procedure that transforms Δ into a new PRS Δ' and terms t_1, t_2 into t'_1, t'_2 s.t. $Norm(\Delta') <_{lex} Norm(\Delta)$ and $t_1 \succ^{\Delta} t_2 \iff t'_1 \succ^{\Delta'} t'_2$.

Remember that we assume that sequential composition is left-associative. So when we write $t_1.t_2$ then t_2 is either a single variable or a parallel composition. If Δ is not in normal form, then there exists a rule in Δ that is neither a seq-rule nor a par-rule. We call such rules *"bad rules"*. There are five types of bad rules:

1. The bad rule is $u \to u_1.u_2$. Let Z, Z_1, Z_2 be new variables. We get Δ' in three steps. Initially Δ' is Δ.

   (a) Replace the bad rule by the following rules

   $$u \to Z \quad Z \to Z_1.Z_2 \quad Z_1 \to u_1$$

   (b) Then substitute Z_2 for u_2 in Δ', t_1 and t_2 (thus we get t'_1, t'_2).

   (c) Finally add the rule $Z_2 \to u_2$ to Δ'.

2. The bad rule is $u \to u_1 \| u_2$. Let Z_1, Z_2 be new variables. We get Δ' by substituting Z_1 for u_1 and Z_2 for u_2 everywhere and then replacing the bad rule by the following rules
$$u \to Z_1 \| Z_2 \quad Z_1 \to u_1 \quad Z_2 \to u_2.$$

3. The bad rule is $u_1 \| (u_2.u_3) \to u_4$. Let Z_1, Z_2 be new variables. We get Δ' by replacing the bad rule by the following rules
$$u_1 \to Z_1 \quad u_2.u_3 \to Z_2 \quad Z_1 \| Z_2 \to u_4$$
$t'_1 := t_1$ and $t'_2 := t_2$.

4. The bad rule is $u_1.(u_2 \| u_3) \to u_4$. Let Z be a new variable. Δ' and t'_1, t'_2 are constructed as follows:
   (a) Substitute Z for $(u_2 \| u_3)$ in all rules and in t_1 and t_2.
   (b) Then add the rules $Z \to u_2 \| u_3$ and $u_2 \| u_3 \to Z$.

5. The bad rule is $u_1.X \to u_2$, where u_1 is not a single variable. Let Z be a new variable. We get Δ' by replacing the bad rule with the following two rules
$$u_1 \to Z \quad Z.X \to u_2$$
$t'_1 := t_1$ and $t'_2 := t_2$.

In all these cases $Norm(\Delta') <_{lex} Norm(\Delta)$ and $t_1 \succ^{\Delta} t_2 \iff t'_1 \succ^{\Delta'} t'_2$. Repeated application of this procedure yields a PRS in normal form. ■

The following lemma will be used to prove the correctness of the algorithm in Lemma 10.1.5.

Lemma 10.1.4 *Let Δ be a PRS in normal form. If there are variables X, Y s.t. $X \succ^{\Delta} Y$ and $(X \to Y) \notin \Delta$, then there are also variables X', Y' with $(X' \to Y') \notin \Delta$ and $X' \succ^{\Delta}_{par} Y'$ or $X' \succ^{\Delta}_{seq} Y'$.*

Proof It follows from the preconditions that we can choose a pair of variables X', Y' s.t. $(X' \to Y') \notin \Delta$ and $X' \xrightarrow{\sigma} Y'$ for a sequence σ of minimal length. More precisely the length of σ is minimal over the choice of X', Y' and σ.

Now we show that $X' \succ^{\Delta}_{par} Y'$ or $X' \succ^{\Delta}_{seq} Y'$. We do this by assuming the contrary and deriving a contradiction. We say that a rule is trivial if it has the form $(X'' \to Y'')$. We assume that σ contains both seq-rules and par-rules that are nontrivial. There are two cases:

1. The last nontrivial rule in σ is a par-rule. If a seq-rule $Z_1 \to Z_2.Z_3$ occurs in σ then there is a subsequence σ' of σ and a variable Z_4 s.t. $Z_2.Z_3 \xrightarrow{\sigma'} Z_4$. This contradicts the minimality of the length of σ.

2. The last nontrivial rule in σ is a seq-rule. This seq-rule must have the form $Z_1.Z_2 \to Z$. The first nontrivial par-rule that occurs in σ must have the form $Z' \to Z'_1\|\ldots\|Z'_n$. Then there is a subsequence σ' of σ and a variable Z'' s.t. $Z' \xrightarrow{\sigma'} Z''$. This contradicts the minimality of the length of σ.

Thus σ consists either only of applications of par-rules (and thus $X' \succ^{\Delta}_{par} Y'$) or only of seq-rules (and thus $X' \succ^{\Delta}_{seq} Y'$). ■

Lemma 10.1.5 *Let Δ be a PRS in normal form. Then a PRS Δ' in transitive normal form can be effectively constructed s.t.*

$$\forall t_1, t_2 \in \mathcal{T}.\ t_1 \succ^{\Delta'} t_2 \iff t_1 \succ^{\Delta} t_2$$

Proof It suffices to find all pairs of variables X, Y s.t. $X \succ^{\Delta} Y$ and to add the rules $(X \to Y)$ to Δ. By Lemma 10.1.4 it suffices to check $X \succ^{\Delta}_{par} Y$ and $X \succ^{\Delta}_{seq} Y$. This is decidable because of Proposition 10.1.2 and the decidability of the reachability problem for pushdown processes (see Chapter 7). Lemma 10.1.4 basically says that while there are new rules to add we can find at least one to add.

The algorithm is as follows:

$\Delta' := \Delta$; flag := true;
While flag **do**
  flag := false;
  For every pair of variables X, Y with $(X \to Y) \notin \Delta'$ **do**
    If $X \succ^{\Delta'}_{par} Y$ or $X \succ^{\Delta'}_{seq} Y$ **then** $(\Delta' := \Delta' \cup (X \to Y)$; flag := true) **fi**;
  od;
od;

■

Theorem 10.1.6 *The reachability problem is decidable for PRS.*

Proof Let Δ be a PRS and $t_1, t_2 \in \mathcal{T}$. The question is if $t_1 \succ^{\Delta} t_2$.

We construct a new PRS Δ' by adding new variables X_1 and X_2 and rules $X_1 \to t_1$ and $t_2 \to X_2$. It follows that

$$t_1 \succ^{\Delta} t_2 \;\Leftrightarrow\; X_1 \succ^{\Delta'} X_2$$

Then we use Lemma 10.1.3 and transform Δ' into a PRS Δ'' in normal form. Normally the terms X_1, X_2 would also change in this transformation, but since they are single variables they stay the same [1]. It follows that

$$t_1 \succ^{\Delta} t_2 \;\Leftrightarrow\; X_1 \succ^{\Delta''} X_2$$

Then we use Lemma 10.1.5 to transform Δ'' into a PRS Δ''' in transitive normal form. It follows that

$$t_1 \succ^{\Delta} t_2 \;\Leftrightarrow\; X_1 \succ^{\Delta'''} X_2$$

Since Δ''' is in transitive normal form we have

$$t_1 \succ^{\Delta} t_2 \;\Leftrightarrow\; X_1 \succ^{\Delta'''} X_2 \;\Leftrightarrow\; (X_1 \to X_2) \in \Delta'''$$

The condition $(X_1 \to X_2) \in \Delta'''$ is trivial to check. ■

10.2 The Reachable Property Problem

In the previous section the problem was if one given state is reachable. Here we consider the question if there is a reachable state that has certain properties. This problem was defined in Subsection 3.1.2 as the "reachable property problem". Unlike for reachability, the atomic actions are important for this problem.

The denotation $[\![\Phi]\!]$ of a state formula Φ is a (possibly infinite) set of process terms. To simplify the notation we use sets of actions. Let $A := \{a_1, \ldots, a_k\} \subseteq Act$, then

$$\begin{aligned} [\![A]\!] &:= [\![a_1]\!] \cap \cdots \cap [\![a_k]\!] \\ [\![-A]\!] &:= [\![\neg a_1]\!] \cap \cdots \cap [\![\neg a_k]\!] \end{aligned}$$

By transformation to disjunctive normal form every state-formula Φ can be written as

$$(A_1^+ \wedge -A_1^-) \vee \cdots \vee (A_n^+ \wedge -A_n^-)$$

[1]It wouldn't matter if they changed.

where $A_i^+, A_i^- \subseteq Act$. The modal operator $\Diamond$ is defined as usual.

$$[\![\Diamond\Phi]\!] := \{t \mid \exists \sigma, t'.\ t \xrightarrow{\sigma} t' \in [\![\Phi]\!]\}$$

Let $t \in \mathcal{T}$ be a process term. For $t \in [\![\Phi]\!]$ we also write $t \models \Phi$. The reachable property problem is if $t_0 \models \Diamond\Phi$ for a state formula Φ and a PRS Δ with initial state t_0.

We prove the decidability of the reachable property problem for PRS in two steps. First we solve the problem for PRS in transitive normal form, and then we use this result to prove the decidability in the general case.

Let there be a PRS Δ **in transitive normal form** with initial state t_0 and Φ a state-formula. We now describe a tableau system that decides the problem $t_0 \models \Diamond\Phi$. (See Chapter 4 for the definition of tableau systems.) As Φ can be transformed into disjunctive normal form and

$$t \models \Diamond(\Phi_1 \vee \Phi_2) \iff t \models \Diamond(\Phi_1) \vee t \models \Diamond(\Phi_2)$$

it suffices to show decidability for formulae of the form $\Diamond(A^+ \wedge -A^-)$. The nodes in the tableau will be sets of expressions (subgoals), which will be interpreted conjunctively. Γ denotes sets of expressions. The branches are interpreted disjunctively. The tableau is successful iff there is a successful leaf.

For technical reasons we introduce a new operator ∇ that is defined by

$$[\![\nabla\Phi]\!] := \{t \mid \exists \sigma, t' \neq \epsilon.\ t \xrightarrow{\sigma} t' \in [\![\Phi]\!]\}$$

Now we define the tableau-rules. Every node in the tableau consists of a finite set of expressions. These expressions have either of the following forms:

- $t \vdash \Diamond(A^+ \wedge -A^-)$, where t is a process term and A^+ and A^- are finite sets of atomic actions.
- $t \vdash \nabla(A^+ \wedge -A^-)$, where t is a process term and A^+ and A^- are finite sets of atomic actions.
- $t_1 \succ t_2$, where t_1 and t_2 are process terms.
- $(t_1 \rightarrow t_2) \in \Delta$, where t_1 and t_2 are process terms.

We describe the transformations of single elements of these sets. So the expression "$\cup\,\Gamma$" should be appended to every node, where Γ denotes a set of expressions. We leave this out to simplify the notation. Because of space constraints in the following tableau-rules we write different branches below each other instead of beside each other. So a rule of the form

$$\begin{array}{c} A \\ \hline B_1 \\ B_2 \\ \vdots \\ B_n \end{array}$$

means

$$\frac{A}{B_1 \quad B_2 \quad \ldots \quad B_n}$$

where the B_i stand for different branches.

Note that the following tableau-rules are only correct because Δ is in transitive normal form.

SP1
$$\begin{array}{c} \{(t_1.(t_2\|t_3))\|t_4 \vdash \Diamond(A^+ \wedge -A^-)\} \\ \hline \{t_1 \succ \epsilon,\ t_2\|t_3\|t_4 \vdash \Diamond(A^+ \wedge -A^-)\} \\ \vdots \\ \{t_1 \vdash \nabla(A_1^+ \wedge -A^-),\ t_4 \vdash \Diamond(A_2^+ \wedge -A^-)\} \\ \vdots \end{array}$$
where $A^+ = A_1^+ \cup A_2^+$

SP2
$$\begin{array}{l} \{(t_1.Y)\|t_2 \vdash \Diamond(A^+ \wedge -A^-)\} \\ \hline \{t_1 \succ \epsilon,\ Y\|t_2 \vdash \Diamond(A^+ \wedge -A^-)\} \\ \{t_1 \vdash \nabla(A_1^+ \wedge -A^-),\ t_2 \vdash \Diamond(A_2^+ \wedge -A^-)\} \\ \{t_1 \succ X,\ X.Y \vdash (A_1^+ \wedge -A^-),\ t_2 \vdash \Diamond(A_2^+ \wedge -A^-)\} \\ \{t_1 \succ X,\ (X.Y \to Z) \in \Delta,\ Z\|t_2 \vdash \Diamond(A^+ \wedge -A^-)\} \end{array} \quad (X \in Var(\Delta))$$
with a separate branch for every A_1^+, A_2^+ s.t. $A^+ = A_1^+ \cup A_2^+$

SP3
$$\begin{array}{c} \{(t_1.(t_2\|t_3))\|t_4 \vdash \nabla(A^+ \wedge -A^-)\} \\ \hline \{t_1 \succ \epsilon,\ t_2\|t_3\|t_4 \vdash \nabla(A^+ \wedge -A^-)\} \\ \vdots \\ \{t_1 \vdash \nabla(A_1^+ \wedge -A^-),\ t_4 \vdash \Diamond(A_2^+ \wedge -A^-)\} \\ \vdots \end{array}$$
where $A^+ = A_1^+ \cup A_2^+$

$$\text{SP4} \quad \frac{\{(t_1.Y)\|t_2 \vdash \nabla(A^+ \wedge -A^-)\}}{\begin{array}{l}\{t_1 \succ \epsilon,\ Y\|t_2 \vdash \nabla(A^+ \wedge -A^-)\}\\ \{t_1 \vdash \nabla(A_1^+ \wedge -A^-),\ t_2 \vdash \Diamond(A_2^+ \wedge -A^-)\}\\ \{t_1 \succ X,\ X.Y \vdash (A_1^+ \wedge -A^-),\ t_2 \vdash \Diamond(A_2^+ \wedge -A^-)\}\\ \{t_1 \succ X,\ (X.Y \to Z) \in \Delta,\ Z\|t_2 \vdash \nabla(A^+ \wedge -A^-)\}\end{array}} \ (X \in Var(\Delta))$$

with a separate branch for every A_1^+, A_2^+ s.t. $A^+ = A_1^+ \cup A_2^+$

$$\text{PAR1} \quad \frac{\{t \vdash \Diamond(A^+ \wedge -A^-)\}}{\cdots \left\{\begin{array}{l}\forall k \in K.\ Y_k \vdash \nabla(A_k^+ \wedge -A^-),\\ \forall k \in K'.\ Y_k'.Z_k \vdash (A_k^+ \wedge -A^-),\\ \forall i \in I.\ U_i \vdash \nabla(-A^-),\\ \forall i \in I'.\ U_i'.V_i \vdash (-A^-)\end{array}\right\} \cdots}$$

where $t \in P$ is a term without sequential composition
and $(X_i \to Y_i.Z_i) \in \Delta$, $(i = 1, \ldots, k)$ are seq-rules and
$\exists t' \in P.\ t \succ_{par} (t'\|X_1\| \ldots \|X_k\|t'')$ with $t' \models (A_0^+ \wedge -A^-)$
and t'' is a parallel composition of atomic terms in $M = \{W_1, \ldots, W_j\}$
and $(W_i \to U_i.V_i) \in \Delta$, $(i = 1, \ldots, j)$ are seq-rules and
$I \cup I' = \{1, \ldots, j\}$
$\forall i \in I'.\ U_i \succ U_i'$
and $A^+ = A_0^+ \cup A_1^+ \cup \cdots \cup A_k^+$, $0 \leq k \leq 2^{|A^+|}$
and $K \cup K' = \{1, \ldots, k\}$
and $\forall k \in K'.\ Y_k \succ Y_k'$

$$\text{PAR2} \quad \frac{\{t \vdash \nabla(A^+ \wedge -A^-)\}}{\cdots \left\{\begin{array}{l}\forall k \in K.\ Y_k \vdash \nabla(A_k^+ \wedge -A^-),\\ \forall k \in K'.\ Y_k'.Z_k \vdash (A_k^+ \wedge -A^-),\\ \forall i \in I.\ U_i \vdash \nabla(-A^-),\\ \forall i \in I'.\ U_i'.V_i \vdash (-A^-)\end{array}\right\} \cdots}$$

where $t \in P$ is a term without sequential composition
and $(X_i \to Y_i.Z_i) \in \Delta$, $(i = 1, \ldots, k)$ are seq-rules and
$\exists t' \in P.\ t \succ_{par} (t'\|X_1\| \ldots \|X_k\|t'')$ with $t' \models (A_0^+ \wedge -A^-)$
and t'' is a parallel composition of atomic terms in $M = \{W_1, \ldots, W_j\}$
and $(W_i \to U_i.V_i) \in \Delta$, $(i = 1, \ldots, j)$ are seq-rules and
$I \cup I' = \{1, \ldots, j\}$
$\forall i \in I'.\ U_i \succ U_i'$
and $A^+ = A_0^+ \cup A_1^+ \cup \cdots \cup A_k^+$, $0 \leq k \leq 2^{|A^+|}$
and $K \cup K' = \{1, \ldots, k\}$
and $\forall k \in K'.\ Y_k \succ Y_k'$
and $k > 0$ or $t'\|t'' \neq \epsilon$

E1 $\dfrac{\{t \succ t'\} \cup \Gamma}{\Gamma}$ if $t \succ t'$

E2 $\dfrac{\{(X.Y \to Z) \in \Delta\} \cup \Gamma}{\Gamma}$ if $(X.Y \to Z) \in \Delta$

In the rules SP1,SP2,SP3,SP4 we have to consider all different (but only finitely many) ways of partitioning A^+ into A_1^+ and A_2^+. In PAR1 and PAR2 the dots symbolize all different ways of choosing k, the set M, the rules $(X_i \to t_i)$ and the partitioning of A^+ into $A_0^+, \ldots, A_k^+$. Again there are only finitely many.

Lemma 10.2.1 *If the side conditions of an instance of a rule are satisfied, then the antecedent is true if and only if one of its succedents is true.*

Proof Directly from the definitions and the condition that the PRS Δ is in transitive normal form. The only difference between PAR1 and PAR2 is the condition $k \neq 0 \vee t' \| t'' \neq \epsilon$. This ensures that the reachable state that satisfies $(A^+ \wedge -A^-)$ is not ϵ. ■

Definition 10.2.2 (Termination conditions)
A node marked with a set of expressions Γ is a terminal node iff one of the following conditions is satisfied.

1. Γ is empty.
2. $\Gamma = \Gamma' \cup \{t \succ t'\}$ for some $t, t' \in \mathcal{T}$ and **not** $t \succ t'$.
3. $\Gamma = \Gamma' \cup \{(X.Y \to Z) \in \Delta\}$ and **not** $(X.Y \to Z) \in \Delta$.
4. The same node Γ occurred earlier on the same branch.

Terminals of type 1 are successful, while terminals of types 2,3 and 4 are unsuccessful.

Note that, since the sequents are sets, the tableau for a given root is not unique. However, there are only finitely many for a given root.

The following definition and lemma by Jančar [Jan90] are used to show that the tableau can be effectively constructed.

Definition 10.2.3 For a given Petri net N the set L_N of formulae is defined as follows:

- There is one variable M that stands for a marking of the net.
- A term is either
  - a term $M(p)$ where p is a place, or
  - a constant $c \in \mathbb{N}$, or
  - of the form $t_1 + t_2$.
- A formula is either
  - an atomic formula $t_1 < t_2$ or $t_1 \leq t_2$, where t_1, t_2 are terms, or
  - of the form $f_1 \& f_2$ where f_1, f_2 are formulae.

For a concrete marking M, $f(M)$ denotes the instance of f with this M. The semantics is natural.

Lemma 10.2.4 *([Jan90])*
For a Petri net N with initial marking M_0 it is decidable if there is a reachable marking M s.t. $f(M)$.

Lemma 10.2.5 *For a given root all possible tableaux can be effectively constructed.*

Proof

1. The side conditions of the rules are decidable: For rule E1 this follows from Theorem 10.1.6. For PAR1 and PAR2 this follows from Lemma 10.2.4.
2. The tableau is finitely branching: This is because there are only finitely many different ways to partition A^+ into subsets and because Δ is finite.
3. The tableau is finite: Let t_0 be the state in the root-node. There are only finitely many different subterms of t_0. As Δ is finite there are only finitely many different seq-rules. Only finitely many variables are used in Δ, thus $Var(\Delta)$ is finite. Only finitely many different formulae of the form $\Diamond(A^+ \wedge -A^-)$ or $\nabla(A^+ \wedge -A^-)$ can occur in the tableau. Therefore there are only finitely many different nodes in the tableau. Thus the construction of the tableau must terminate, because of termination condition 4.

4. There are only finitely many different tableaux for a given root: This is because all sequents in any tableau for a given root are sets whose cardinality is bounded by a constant c that depends only on the PRS Δ and the size of the root. The only nondeterminism in the construction is in which subgoal in a sequent is solved first. ∎

Now we prove the soundness and completeness of the tableau system.

Lemma 10.2.6 *If there is a successful tableau with root $\{t \vdash \Diamond(A^+ \wedge -A^-)\}$, then $t \models \Diamond(A^+ \wedge -A^-)$.*

Proof If the tableau is successful, then it has a branch that ends with a successful (empty) node. This node is certainly true. By repeated application of Lemma 10.2.1 the root-node must be true as well. ∎

Lemma 10.2.7 *Let $Op \in \{\Diamond, \nabla\}$. Let there be a node of the form $\{t \vdash Op(A^+ \wedge -A^-)\} \cup \Gamma$, s.t. $t \models Op(A^+ \wedge -A^-)$ and Γ is a set of subgoals that are all true.*

Then every tableau with this root has a branch leading to a node Γ' s.t. Γ' is a true set of subgoals that has developed from Γ.

Proof First we describe the proof for a particular tableau where the most recently created subgoals are solved first.

We do the proof by induction on lexicographically ordered pairs (x, y) s.t.

$$(x, y) := (length(\sigma), size(t))$$

where σ is a sequence of minimal length s.t. $t \xrightarrow{\sigma} t'$ and $t' \models (A^+ \wedge -A^-)$ (and $t' \neq \epsilon$ if $Op = \nabla$). Such a sequence must exist, because $t \models Op(A^+ \wedge -A^-)$.

If $(x, y) = (0, 0)$ then $t = \epsilon$ and $A^+ = \{\}$. The rule PAR1 is applicable and the one child-node is Γ.

Otherwise we apply a tableau-rule to $\{t \vdash Op(A^+ \wedge -A^-)\}$. By Lemma 10.2.1 at least one child-node must be true. Choose the true child-node that corresponds to σ. The tableau-rule replaces the expression by several other expressions with (x', y') s.t. $(x', y') <_{lex} (x, y)$. For the rules SP1, SP2, SP3 and SP4 in the child node x is lower or equal and y is smaller. With the induction hypothesis and rules E1,E2 we can solve all newly created subgoals and arrive at a node Γ. For the rules PAR1 and PAR2 the second component y may have increased in the child-node, but the first component x is always smaller. Thus by induction hypothesis we can solve all newly created subgoals and arrive at a node Γ. This construction cannot be interrupted by termination condition 4, because this would contradict the minimality of the length of σ.

The above construction is for a particular tableau where the most recently created subgoals are solved first. In other tableaux we might have applications of tableau-rules to other expressions in Γ between the steps we described above. However, if we stay on a branch of true sequents (it must exist by the preconditions) then our sequents have the form $M \cup \Gamma'$ where M are the subgoals created from $\{t \vdash Op(A^+ \wedge -A^-)\}$ and Γ' is a true development of Γ. Eventually we will solve the subgoals in M and reach a node Γ' where Γ' is a true development of Γ. ∎

Corollary 10.2.8 *If $t \models \Diamond(A^+ \wedge -A^-)$, then every tableau with the root-node $\{t \vdash \Diamond(A^+ \wedge -A^-)\}$ is successful.*

Proof We apply Lemma 10.2.7 for the special case of $\Gamma = \{\}$. Since Γ' is a development of Γ it must be empty too. Thus the tableau is successful by termination condition 1. ∎

Lemma 10.2.9 *Let Δ be a PRS in transitive normal form with initial state t_0 and $(A^+ \wedge -A^-)$ a state formula. Then the following properties are equivalent.*

1. $t_0 \models \Diamond(A^+ \wedge -A^-)$
2. *A tableau with root $\{t \vdash \Diamond(A^+ \wedge -A^-)\}$ is successful.*
3. *Every tableau with root $\{t \vdash \Diamond(A^+ \wedge -A^-)\}$ is successful.*

Proof Directly from Lemma 10.2.6 and Corollary 10.2.8. ∎

So far we have only considered the reachable property problem for PRS in transitive normal form. For the general case more work is needed. It is not possible to apply the same algorithms as in Lemma 10.1.3 and Lemma 10.1.5 to transform a PRS into transitive normal form, because these transformations do not preserve the properties we want to check. A generalized version of Lemma 10.1.3 is necessary.

Lemma 10.2.10 *Let Δ be a PRS that uses only variables from the finite set $Var(\Delta) \subset Var$ and $t \in \mathcal{T}$ a process term.*

Then a PRS Δ' in normal form and a term t' can be effectively constructed s.t. for every state formula Φ, $t \models \Diamond\Phi$ with respect to Δ iff $t' \models \Diamond(\Phi \wedge \neg\gamma)$ with respect to Δ'. (γ is a new action.)

Proof Let k_i be the number of rules $(t_1 \rightarrow t_2)$ in Δ that are neither par-rules nor seq-rules and $size(t_1) + size(t_2) = i$. Let n be the maximal i s.t. $k_i \neq 0$. (n exists because Δ is finite). We define

$$Norm(\Delta) := (k_n, k_{n-1}, \ldots, k_1)$$

These norms are ordered lexicographically. Δ is in normal form iff $Norm(\Delta) = (0, \ldots, 0)$. Now we describe a procedure that transforms Δ into a new PRS Δ' and t into t', with the above properties. For this we introduce two completely new atomic actions γ and τ that do not occur in Δ, t and the state formula Φ.

If Δ is not in normal form, then there exists a rule in Δ that is neither a seq-rule nor a par-rule. We call such rules *bad rules*. There are five types of bad rules:

1. The bad rule is $u \xrightarrow{a} u_1.u_2$, where u_2 is either a single variable or a parallel composition. Let Z, Z_1, Z_2 be new variables. We get Δ' in three steps:
   (a) First replace the bad rule by the following rules
   $$u \xrightarrow{a} Z \quad Z \xrightarrow{\gamma} Z_1.Z_2 \quad Z_1 \xrightarrow{\gamma} u_1$$
   (b) Then we substitute Z_2 for u_2 in Δ' and t (obtaining t').
   (c) Finally, we add the rules $Z_2 \xrightarrow{\gamma} u_2$ and $u_2 \xrightarrow{\tau} Z_2$.
2. The bad rule is $u \xrightarrow{a} u_1 \| u_2$. Let Z_1, Z_2 be new variables. We get Δ' through the following steps:
   (a) Replace the bad rule by the following rule
   $$u \xrightarrow{a} Z_1 \| Z_2$$
   (b) Then we add the rules
   $$Z_1 \xrightarrow{\gamma} u_1 \quad Z_2 \xrightarrow{\gamma} u_2$$
3. The bad rule is $u_1 \| (u_2.u_3) \xrightarrow{a} u_4$. Let Z_1, Z_2, Z_3 be new variables. We get Δ' in the following steps:
   (a) First replace the bad rule by the following rules
   $$u_1 \xrightarrow{\tau} Z_1 \quad u_2.u_3 \xrightarrow{\tau} Z_2 \quad Z_1 \| Z_2 \xrightarrow{\tau} Z_3 \quad Z_1 \xrightarrow{\gamma} Z_1 \quad Z_2 \xrightarrow{\gamma} Z_2 \quad Z_3 \xrightarrow{a} u_4$$
   (b) Then for all actions b that are enabled by the term $u_1 \| (u_2.u_3)$ with respect to Δ add to Δ' a rule $Z_3 \xrightarrow{b} Z_3$.

(c) For every rule $(l \xrightarrow{x} r)$ where the term $u_1 \| (u_2.u_3)$ occurs as a subterm of l add a new rule $(l' \xrightarrow{x} r)$ where l' is obtained from l by replacing all occurrences of $u_1 \| (u_2.u_3)$ by Z_3 [2].

(d) $t' := t$.

4. The bad rule is $u_1.(u_2 \| u_3) \xrightarrow{a} u_4$. Let Z be a new variable. Δ' and t' are constructed as follows:

   (a) Substitute Z for $(u_2 \| u_3)$ in all rules in Δ and in t (thus obtaining t').

   (b) Then add the rules $Z \xrightarrow{\gamma} u_2 \| u_3$ and $u_2 \| u_3 \xrightarrow{\tau} Z$.

5. The bad rule is $u_1.X \xrightarrow{a} u_2$, where u_1 is not a single variable. Let Z_1, Z_2 be new variables. We get Δ' by the following steps:

   (a) First replace the bad rule with the following rules

$$u_1 \xrightarrow{\tau} Z_1 \quad Z_1 \xrightarrow{\gamma} Z_1 \quad Z_1.X \xrightarrow{\tau} Z_2 \quad Z_2 \xrightarrow{a} u_2$$

   (b) Then for all actions b that are enabled by the term $u_1.X$ with respect to Δ add to Δ' a rule $Z_2 \xrightarrow{b} Z_2$.

   (c) For every rule $(l \xrightarrow{x} r)$ where the term $u_1.X$ occurs as a subterm of l add a new rule $(l' \xrightarrow{x} r)$ where l' is obtained from l by replacing all occurrences of $u_1.X$ by Z_2 [3].

   (d) $t' := t$.

In all cases $Norm(\Delta') <_{lex} Norm(\Delta)$ and the property of the state formulae is preserved. Repeated application of this procedure yields a PRS in normal form.
■

Now we can prove decidability for the general case.

Theorem 10.2.11 *The reachable property problem is decidable for PRS.*

Proof Let there be a PRS Δ with initial state t_0 and Φ a state-formula. The question is if $t_0 \models \Diamond\Phi$.

First we apply Lemma 10.2.10 to get a PRS Δ' in normal form and a t_0' s.t. $t_0 \models \Diamond\Phi$ w.r.t. Δ iff $t_0' \models \Diamond(\Phi \wedge \neg\gamma)$ w.r.t. Δ'. Then we use the algorithm in

[2] Note that we keep the old rule $(l \xrightarrow{x} r)$.

[3] We keep the old rule $(l \xrightarrow{x} r)$.

Lemma 10.1.5 to transform the PRS Δ' into an equivalent PRS Δ'' in transitive normal form. All new rules that are added in this process are labeled with the special new action τ. It follows that $t_0 \models \Diamond\Phi$ w.r.t. Δ iff $t_0' \models \Diamond(\Phi \wedge \neg\gamma)$ w.r.t. Δ''. It suffices to show decidability for formulae of the form $\Diamond(A^+ \wedge -A^-)$. Since Δ'' is in transitive normal form we can apply the tableau system. By Lemma 10.2.5 all possible tableaux with root $\{t_0' \vdash \Diamond(A^+ \wedge -A^-)\}$ can be effectively constructed. It follows from Lemma 10.2.9 that it suffices to construct one tableau. The property holds if and only if it is successful. ∎

This result can also be used to decide the deadlock reachability problem. Let Δ be a PRS with initial state t_0 and $Act(\Delta)$ the (finite!) set of actions used in Δ. A deadlock is reachable iff $t_0 \models \Diamond(-Act(\Delta))$.

10.3 Application

We consider the example from Section 2.4 again. With the algorithm from Section 10.1 we can verify the system by checking the following properties. Let X be the initial state.

1. It is possible to reach the state T. This means that the computation can terminate and return the result *true*.

2. It is not possible to reach the state $X\|Z$. This means that process Z can never run in parallel with process X. Of course this must hold, because process Z is always called as a subroutine of process X.

3. It is not possible to reach the state $(W\|T).X$. Remember that we introduced different symbols for boolean values to force a conjunctive or disjunctive interpretation. This property shows (partly) that the two interpretations cannot get into conflict with each other. (Later we show this fully.)

With the algorithm from Section 10.2 we can do further verification: Let $Act(ex)$ be the set of all actions used in the example. It is possible to reach a state where $decomp_2$ is the only possible action

$$X \models \Diamond(decomp_2 \wedge -(Act(ex) - \{decomp_2\}))$$

but there is no reachable state where $decomp_1$ and $decomp_2$ are the only possible actions.

$$X \models \neg\Diamond(decomp_1 \wedge decomp_2 \wedge -(Act(ex) - \{decomp_1, decomp_2\}))$$

Now we show that the conjunctive and disjunctive interpretations of boolean values can never get into conflict. To do this we add some new rules to the system of the example from Section 2.4.

$$\begin{aligned} W\|F &\xrightarrow{\;conflict\;} \epsilon \\ R\|F &\xrightarrow{\;conflict\;} \epsilon \\ W\|T &\xrightarrow{\;conflict\;} \epsilon \\ R\|T &\xrightarrow{\;conflict\;} \epsilon \end{aligned}$$

Then the conjunctive and disjunctive interpretations of boolean values can get into conflict if and only if the action *conflict* can ever become enabled. With the algorithm from Section 10.2 we can show that for this modified system the property

$$X \not\models \Diamond(conflict)$$

holds and thus action *conflict* can never become enabled.

The algorithms for the reachability problem and the reachable property problem for PRS rely on the reachability problem for Petri nets and are thus not primitive recursive. So it might seem that they are not applicable in practice because of their very high complexity. However, there are three arguments in their favor:

1. In many examples the system is not very large and the structure of the Petri nets that are contained in them is often simple.

2. In a large PRS there may be many Petri nets as substructures, but often each of these Petri nets is quite small. These Petri nets are either not connected with each other at all, or their influence on each other is very limited. Thus they yield small subproblems that can be solved in acceptable time.

3. Finally, the reachability problem for Petri nets has been studied for many years and ways of dealing with it have been developed. There are semi-decision procedures that give yes/no/don't know answers in acceptable time [CH78, Mur89, ME96]. These algorithms mostly use constraints to represent sets of states and approximate the behavior of the system.

Therefore the algorithms of Section 10.1 and Section 10.2 can still be useful in practice to verify systems that are modeled with PRS.

10.4 Conclusion

The reachability problem and the reachable property problem are the only decidable verification problems for PRS. Since PRS subsumes Petri nets, these problems are at least as hard as the reachability problem for Petri nets and thus *EXPSPACE*-hard. Like for Petri nets, this hardness result even holds for the question if the empty state ϵ is reachable.

Model checking Petri nets with EF is undecidable [BE97, Esp97] (see Chapter 9), model checking BPP with EG is undecidable [EK95] (see Chapter 6) and model checking PA with LTL or linear-time μ-calculus is undecidable [BH96] (see Chapter 8). Thus model checking PAN or PRS with these logics is undecidable. The other logics (UB, CTL, modal μ-calculus) are more general than EF and EG, and thus also undecidable for PAN and PRS.

PAN/PRS	**general**	**fixed formula**
reachability, reachable property	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
EF	undecidable	undecidable
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alternation-free modal μ-calc.	undecidable	undecidable
modal μ-calc.	undecidable	undecidable
LTL	undecidable	undecidable
linear-time μ-calc.	undecidable	undecidable

Chapter 11

Summary

In the Chapters 5 – 10 the results about the complexity of model checking are given individually for each process-model in the PRS-hierarchy. Now we present a view on these results from the perspective of the logics.

We also show the limits of the decidability of model checking with each logic in the PRS-hierarchy. In the Figures 11.1,11.2, 11.3, and 11.4 this is described graphically.

11.1 Branching-Time Logics

11.1.1 Reachability and Reachable Property

As shown in Chapter 10, the reachability problem and the reachable property problem are decidable for all models in the PRS-hierarchy, so none of these models is Turing-powerful. It has also turned out that, although the two problems are not completely equivalent for every model, they have the same complexities for every model in the PRS-hierarchy. Interestingly, there are three groups of models: For models with only sequential composition (BPA, pushdown processes) reachability is polynomial. For models with parallel composition but no synchronization (BPP, PA, PAD) it is $\mathcal{NP}$-complete. For the other models it is decidable, but at least as hard as the reachability problem for Petri nets, and thus *EXPSPACE*-hard. The reachability problem for a fixed given state might be easier for BPP, PA and PAD, but not for Petri nets, PAN and PRS.

Reachability	**general**	**fixed state**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	$\in \mathcal{P}$	$\in \mathcal{P}$
pushdown processes	$\in \mathcal{P}$	$\in \mathcal{P}$
BPP	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
PA	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
PAD	$\mathcal{NP}$-complete	$\in \mathcal{NP}$
Petri nets	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
PAN	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
PRS	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard

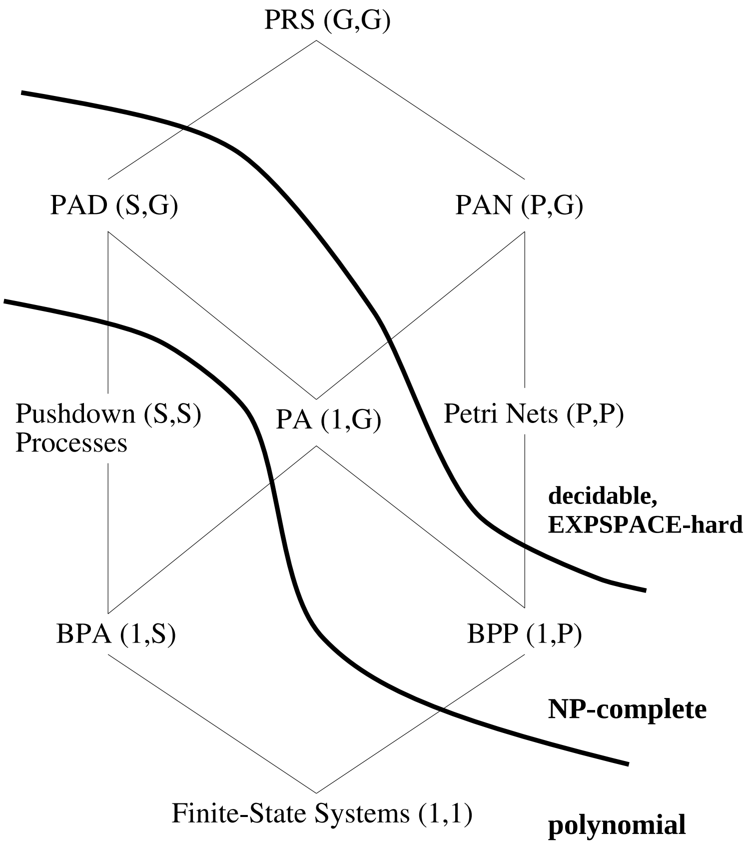


Figure 11.1: The complexity of reachability.

11.1.2 EF

EF is the branching-time logic with the easiest model checking problem. Model checking with EF is decidable for several models (PAD, PA and BPP), where model checking with any other branching time logic is undecidable. One of the reasons for this is that the logic allows a limited decomposition: $\Diamond(\Phi_1 \vee \Phi_2) = \Diamond(\Phi_1) \vee \Diamond(\Phi_2)$. Such a decomposition is not possible in other logics (not even in EG). The following table shows the complexity of the model checking problem for EF. Like in the previous chapters we distinguish the general model checking problem and the problem for a fixed formula.

EF	**general**	**fixed formula**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	*PSPACE*-complete	$\in \mathcal{P}$
pushdown processes	*PSPACE*-complete	*PSPACE*-complete
BPP	*PSPACE*-complete	$\in \Sigma_d^p$
PA	$\in DTIME(tower(n))$, *PSPACE*-hard	$\in \Sigma_d^p$
PAD	$\in DTIME(tower(n))$, *PSPACE*-hard	$\in$ d-*EXPTIME*, *PSPACE*-hard
Petri nets	undecidable	undecidable
PAN	undecidable	undecidable
PRS	undecidable	undecidable

So far, there is no hardness result for any branching-time logic and BPA. The known algorithms for PA and PAD have a very high complexity ($O(tower(n))$), but this is mostly in the size of the formula.

Figure 11.2 shows the limits of the decidability of EF. Model checking with EF is decidable for all models below the line and undecidable for those above it.

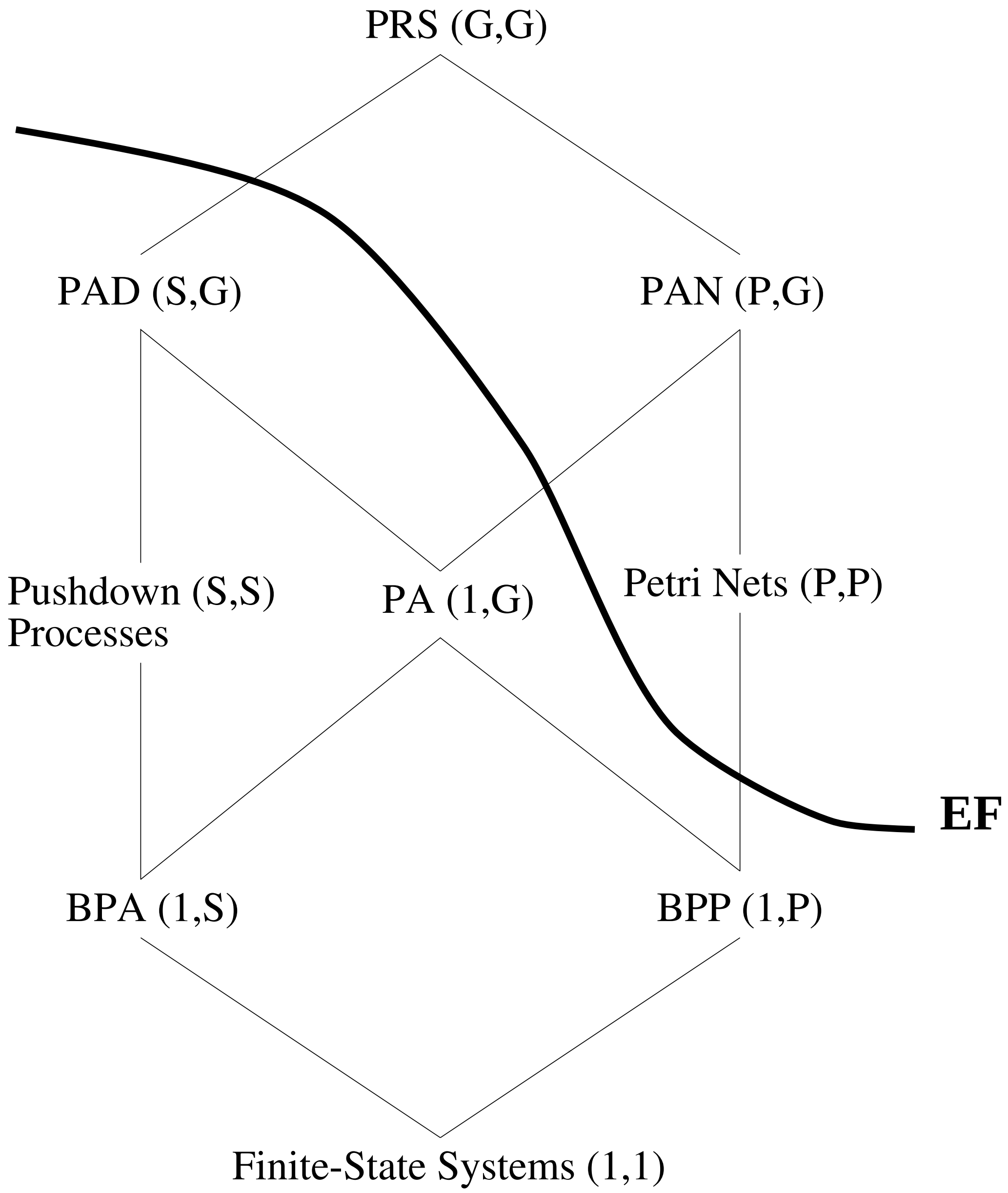


Figure 11.2: Limits of the decidability of model checking with EF.

11.1.3 EG

The logic EG is a simple fragment of CTL (and UB), but model checking with EG is a lot harder than with EF. In the rest of this chapter we'll see that for all models in the PRS-hierarchy decidability of model checking with EG coincides with decidability of model checking with the full modal μ-calculus. (See Figure 11.3.) It is an open question if this also holds for computational complexity. It is unlikely however, since the modal μ-calculus is much more expressive than EG.

EG	**general**	**fixed formula**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
pushdown processes	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*
BPP (and higher)	undecidable	undecidable

11.1.4 UB

The logic UB is a combination of EF and EG. The results on the complexity of model checking reflect this. It seems that the operators *EF* and *EG* have no strong interaction that increases the expressiveness.

UB	**general**	**fixed formula**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
pushdown processes	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*, *PSPACE*-hard
BPP (and higher)	undecidable	undecidable

11.1.5 CTL

The following table of complexities is the same as for UB. However, it is not certain yet that UB and CTL always have the same complexity. Model checking BPA with UB (or EG) is possibly easier than model checking BPA with CTL.

CTL	**general**	**fixed formula**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	$\in$ *EXPTIME*, *PSPACE*-hard	$\in \mathcal{P}$
pushdown processes	$\in$ *EXPTIME*, *PSPACE*-hard	$\in$ *EXPTIME*, *PSPACE*-hard
BPP (and higher)	undecidable	undecidable

11.1.6 Alternation-free Modal μ-Calculus

For the alternation-free modal μ-calculus there is a stronger hardness result than for CTL. Model checking pushdown processes with the alternation-free modal μ-calculus is *EXPTIME*-complete while for CTL it is only known to be between *PSPACE* and *EXPTIME*.

Alt.-free modal μ-calc.	**general**	**fixed formula**
finite-state systems	$\in \mathcal{P}$	$\in \mathcal{P}$
BPA	*EXPTIME*-complete	$\in \mathcal{P}$
pushdown processes	*EXPTIME*-complete	*EXPTIME*-complete
BPP (and higher)	undecidable	undecidable

11.1.7 Modal μ-Calculus

The full modal μ-calculus is the only logic considered here for which model checking finite-state systems is not known to be polynomial. However, it is in $\mathcal{NP} \cap$ co-$\mathcal{NP}$, which argues that a subexponential algorithm might exist.

Modal μ-calculus	**general**	**fixed formula**
finite-state systems	$\in \mathcal{NP} \cap$ co-$\mathcal{NP}$	$\in \mathcal{P}$
BPA	*EXPTIME*-complete	$\in \mathcal{P}$
pushdown processes	*EXPTIME*-complete	*EXPTIME*-complete
BPP (and higher)	undecidable	undecidable

Except for the logic EF, all branching-time logics have the same limits of decidability in the PRS-hierarchy. Figure 11.3 illustrates this.

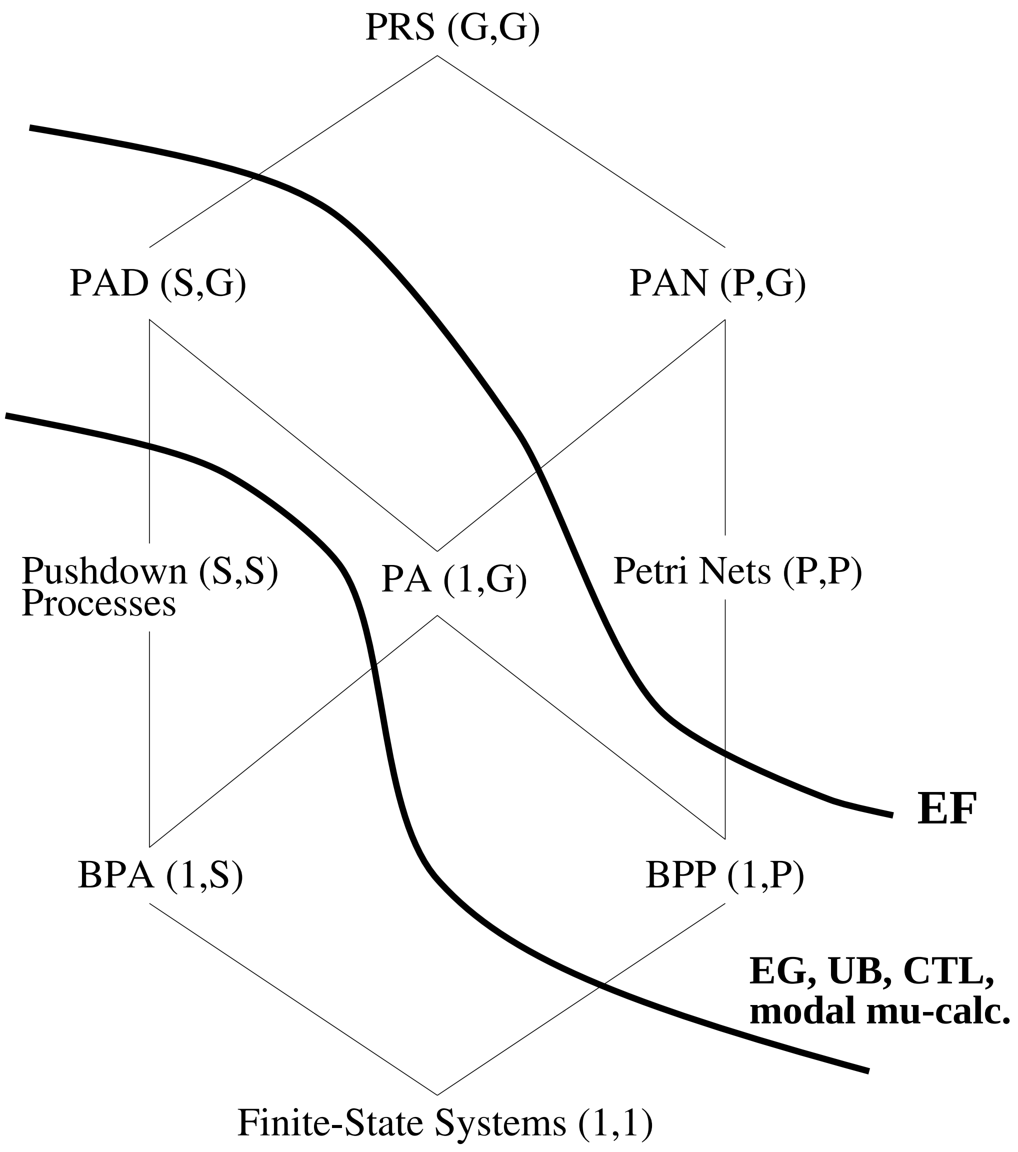


Figure 11.3: Limits of the decidability of branching-time logics.

11.2 Linear-Time Logics

LTL and the linear-time μ-calculus have the same decidability and complexity for all models in the PRS-hierarchy. This is not too surprising, since they have almost (but not quite) the same expressive power. Unlike for branching-time logics, strict lower bounds are known for BPA.

LTL/Linear-time μ-calc.	**general**	**fixed formula**
finite-state systems	*PSPACE*-complete	$\in \mathcal{P}$
BPA	*EXPTIME*-complete	$\in \mathcal{P}$
pushdown processes	*EXPTIME*-complete	$\in \mathcal{P}$
BPP	decidable, *EXPSPACE*-hard	decidable
PA	undecidable	undecidable
PAD	undecidable	undecidable
Petri nets	decidable, *EXPSPACE*-hard	decidable, *EXPSPACE*-hard
PAN	undecidable	undecidable
PRS	undecidable	undecidable

These complexity results are quite different from those for any branching-time logic. On the one hand model checking finite-state systems is harder and model checking systems with both sequential and parallel composition (PA,PAD,PAN and PRS) is undecidable. But on the other hand model checking Petri nets is decidable, unlike for any branching-time logic. Another nice point is that model checking purely sequential systems (BPA and pushdown processes) is polynomial for every fixed formula.

Figure 11.4 shows the limits of the decidability of model checking with linear-time logics. These limits are quite different from those for any branching-time logic.

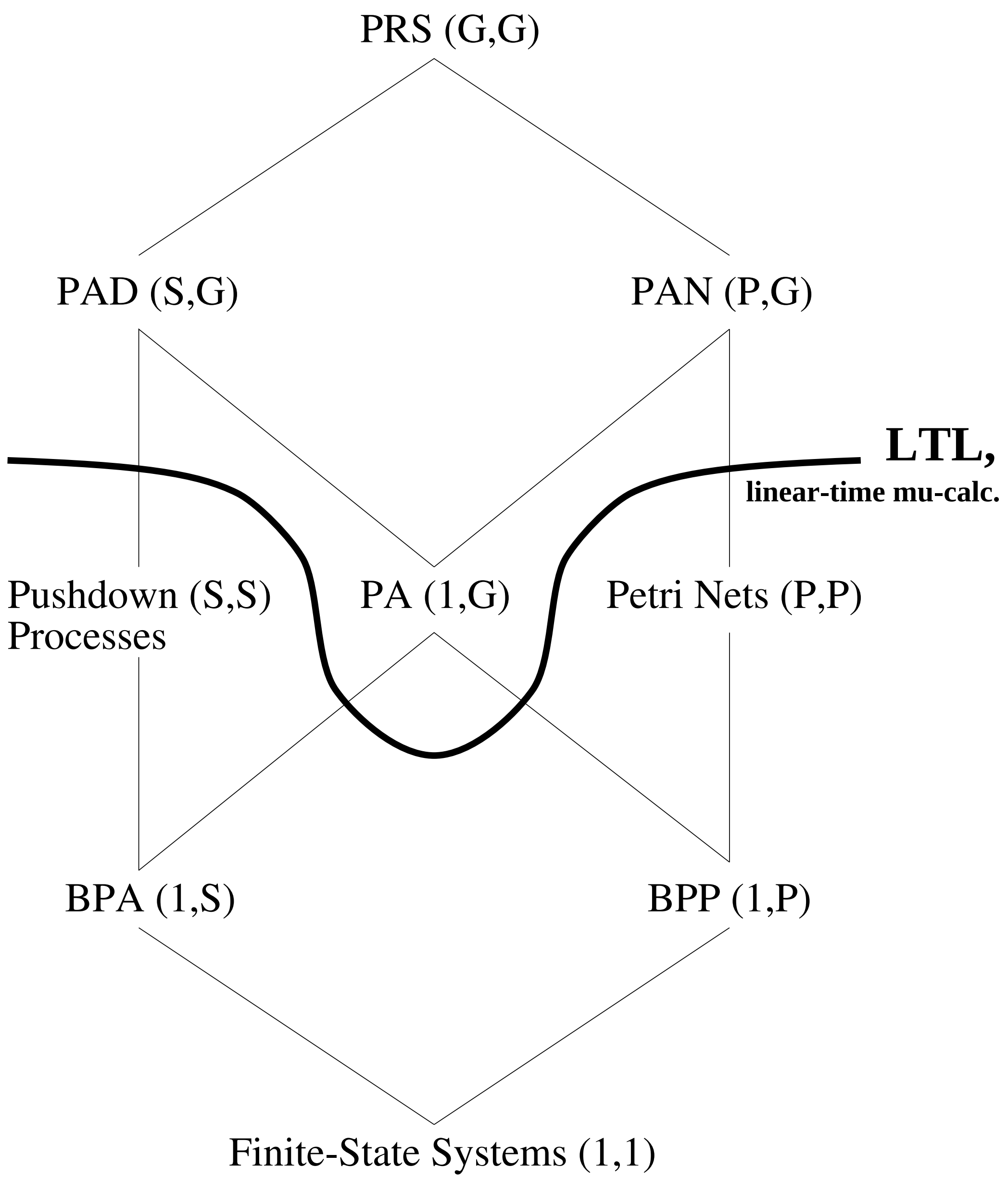


Figure 11.4: Limits of the decidability of linear-time logics.

Chapter 12

Conclusion and Final Remarks

There are five main conclusions from the results that are presented in this thesis.

1. None of the models in the PRS-hierarchy is Turing-powerful.
2. Model checking with EF is much easier than for any other branching-time logic. It is decidable for many more models and often has a lower complexity.
3. All other branching-time logics are decidable for the same models in the PRS-hierarchy.
4. Linear-time logics and branching-time logics are completely different with respect to decidability and complexity of model checking.
5. Most model checking problems for infinite-state systems are *PSPACE*-hard. However, the complexity in the size of the system is often lower. It is even linear in some cases.

The results on the complexity of model checking infinite-state systems look discouraging at first. Many problems are *PSPACE*-hard, *EXPTIME*-hard or even *EXPSPACE*-hard. However, a closer look shows that the situation is not as bad as it might seem. There are still many things that can be done in practice.

Firstly, it is not always necessary to use a full temporal logic to specify the properties that must be checked in order to verify a system. In practice it often suffices to check simple properties of systems, which can be much easier. Some of these simple verification problems are decidable in polynomial time (see Section 8.3).

Secondly, not even *PSPACE*-hard problems are always as bad as they seem. The complexity of the model checking problem depends on two parameters: the size of the system and the size of the formula. In practice, the system can be very large, but the formula is usually very small. Thus the complexity in the size of the system is the important part. In many cases the complexity in the size of the system is much lower than the complexity of the general problem. Roughly speaking, there are two classes of model checking problems:

1. Model checking problems that are hard in the size of the system even for a small fixed formula. The model checking problems for pushdown processes (and PAD) and branching-time logics belong to this class, since they are *PSPACE*/*EXPTIME*-hard, even for small fixed formulae (see Chapter 7).

   Model checking parallel systems like Petri nets with linear-time logics is also in this class, since it is at least as hard as reachability for Petri nets and thus *EXPSPACE*-hard, even for a small fixed LTL-formula (see Chapters 6 and 9).

   Model checking BPP with EF is almost (but not quite) as hard. The model checking problem is not *PSPACE*-hard for any fixed formula, but complete for the d-th order in the polynomial time hierarchy for formulae of nesting depth $\leq d$. Of course this lower bound also holds for PA.

2. There are model checking problems that are only hard in the size of the formula. Model checking BPA with all branching-time logics is polynomial in the size of the system for any fixed formula (see Chapter 7). This shows that BPA and pushdown processes are quite different in model checking, although they describe the same class of languages (Chomsky-2).

   The model checkers SPIN [Hol91] and PROD [Val92] work with finite-state systems and LTL. This problem is also *PSPACE*-complete, but polynomial in the size of the system for any fixed formula. Model checking BPA and pushdown processes with linear-time logics (see Chapter 7) is also polynomial in the size of the system for any fixed formula. So these problems are tractable in practice and model checkers like SPIN [Hol91] and PROD [Val92] could be generalized to handle them. Similar tools could be developed for BPA and the modal μ-calculus.

Finally, verification doesn't have to be completely automatic. Normally a systems designer knows quite a lot about the structure of the system that he/she wants to verify. If the user can use his knowledge about the system in the verification process, then the problem becomes a lot easier. This is because fully automatic verification algorithms often spend a lot of time proving (implications of) properties that are trivial for the user, for example "process t_1 cannot have any influence on the behavior of process t_2 before action b has occurred". So semiautomatic verification methods are a promising direction. They can be implemented in theorem provers with human interaction, like PVS [ORSv95] and "Isabelle" [Pau94]. In these semiautomatic methods it is necessary that the user can understand and influence the verification process. Tableau systems like in Section 9.2 and in [Bra92, BEM96, BS97, And94] provide the theoretical basis for this.

Bibliography

[And94] Henrik Reif Andersen. On model checking infinite-state systems. In *Logical Foundations of Computer Science – LFCS'94*, volume 813 of *LNCS*. Springer Verlag, 1994.

[BCS95] O. Burkart, D. Caucal, and B. Steffen. An elementary bisimulation decision procedure for arbitrary context-free processes. In *MFCS'95*, volume 969 of *LNCS*. Springer Verlag, 1995.

[BCS96] O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*. Springer Verlag, 1996.

[BE97] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 5, 1997.

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. Verifying infinite state processes with sequential and parallel composition. In *Proceedings of POPL'95*, pages 95–106. ACM Press, 1995.

[BEM96] J. Bradfield, J. Esparza, and A. Mader. An effective tableau system for the linear time μ-calculus. In F. Meyer auf der Heide and B. Monien, editors, *Proceedings of ICALP'96*, volume 1099 of *LNCS*. Springer Verlag, 1996.

[BEM97a] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: application to model checking. In *International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*. Springer Verlag, 1997.

[BEM97b] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. Technical report, VERIMAG, 1997. ftp://ftp.imag.fr/imag/SPECTRE/ODED/pda.ps.gz.

[BH96] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and Petri nets. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*. Springer Verlag, 1996.

[BK85] J. A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science (TCS)*, 37:77–121, 1985.

[Bra92] J. Bradfield. *Verifying Temporal Properties of Systems*. Birkhäuser, 1992.

[BS90] J. Bradfield and C. Stirling. Verifying temporal properties of processes. volume 458 of *LNCS*, pages 115–125. Springer Verlag, 1990.

[BS92a] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science (TCS)*, 96:157–174, 1992.

[BS92b] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. of CONCUR'92*, volume 630 of *LNCS*, pages 123–137, 1992.

[BS94] O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *CONCUR'94*, volume 836 of *LNCS*, pages 98–113. Springer Verlag, 1994.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking optimal of pushdown processes. *Nordic Journal of Computer Science*, 1995.

[BS97] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *Proceedings of ICALP'97*, volume 1256 of *LNCS*. Springer Verlag, 1997.

[BW90] J.C.M. Baeten and W.P. Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, 18, 1990.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Journal of Theoretical Computer Science*, 106:61–86, 1992.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. volume 131 of *LNCS*, pages 52–71, 1981.

[CGL94] E. Clarke, O. Grumberg, and D. Long. *Verification tools for finite-state concurrent systems*, volume 803 of *LNCS*. Springer Verlag, 1994.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages*. ACM-Press, 1978.

[CHM93a] S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for Basic Parallel Processes. In E. Best, editor, *Proceedings of CONCUR 93*, volume 715 of *LNCS*. Springer Verlag, 1993.

[CHM93b] S. Christensen, Y. Hirshfeld, and F. Moller. Decomposability, decidability and axiomatisability for bisimulation equivalence on Basic Parallel Processes. In *Proceedings of LICS'93*. IEEE Computer Society Press, 1993.

[Chr93] S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, Edinburgh University, 1993.

[CHS92] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivelence is decidable for all context-free processes. In W.R. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *LNCS*. Springer Verlag, 1992.

[Dam92] M. Dam. Fixed points of Büchi automata. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'92)*, volume 652 of *LNCS*, pages 39–50. Springer Verlag, 1992.

[Dic13] L.E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with distinct factors. *American Journal of Mathematics*, 35:413–422, 1913.

[EJS93] E. Emerson, C.S. Jutla, and A. Sistla. On model checking for fragments of μ-calculus. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 385–396. Springer Verlag, 1993.

[EK95] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and Basic Parallel Processes. In *CAV'95*, volume 939 of *LNCS*, pages 353–366. Springer Verlag, 1995.

[Eme94] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science : Volume B, FORMAL MODELS AND SEMANTICS*. Elsevier, 1994.

[Esp94] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Trees in Algebra and Programming – CAAP'94*, volume 787 of *LNCS*. Springer Verlag, 1994.

[Esp95] J. Esparza. Petri nets, commutative context-free grammars and Basic Parallel Processes. In Horst Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *LNCS*. Springer Verlag, 1995.

[Esp96] J. Esparza. More infinite results. In B. Steffen and T. Margaria, editors, *Proceedings of INFINITY'96*, number MIP-9614 in Technical report series of the University of Passau. University of Passau, 1996.

[Esp97] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[Hab97] P. Habermehl. On the complexity of the linear-time mu-calculus for Petri nets. In *Proceedings of the International Conference on Application and Theory of Petri Nets, Toulouse, France*, LNCS. Springer Verlag, 1997.

[Hir93] Y. Hirshfeld. Petri nets and the equivalence problem. In *Proceedings of CSL'93*, volume 832 of *LNCS*, pages 165–174. Springer Verlag, 1993.

[HJM94] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimulation of normed context free processes. Technical report, LFCS report series 94-286, Edinburgh University, 1994.

[HJM96] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial-time algorithm for deciding bisimulation equivalence of normed Basic Parallel Processes. *Journal of Mathematical Structures in Computer Science*, 1996.

[Hol91] G.J. Holzman. *Design and validation of computer protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[Jan90] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Comuter Science*, 74:71–93, 1990.

[Jan94] P. Jančar. Decidability questions for bisimilarity of Petri nets and some related problems. In *Proceedings of STACS'94*, volume 775 of *LNCS*. Springer Verlag, 1994.

[Jan95] P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148:281–301, 1995.

[JE96] P. Jančar and J. Esparza. Deciding finiteness of Petri nets up to bisimulation. In F. Meyer auf der Heide and B. Monien, editors, *Proceedings of ICALP'96*, volume 1099 of *LNCS*. Springer Verlag, 1996.

[JKM98a] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. In *Proc. of ICALP'98*, LNCS. Springer Verlag, 1998. To appear.

[JKM98b] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. Technical Report I9805, TU-München, 1998.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.

[Lip76] R. Lipton. The reachability problem requires exponential space. Technical Report 62, Department of Computer Science, Yale University, January 1976.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, TU-München, 1997.

[May84] E. Mayr. An algorithm for the general Petri net reachability problem. *SIAM Journal of Computing*, 13:441–460, 1984.

[May96a] Richard Mayr. Some results on Basic Parallel Processes. Technical Report TUM-I9616, TU-München, March 1996.

[May96b] Richard Mayr. A tableau system for model checking Petri nets with a fragment of the linear time μ-calculus. Technical Report TUM-I9634, TU-München, October 1996.

[May96c] Richard Mayr. Weak bisimulation and model checking for Basic Parallel Processes. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, volume 1180 of *LNCS*. Springer Verlag, 1996.

[May97a] Richard Mayr. Combining Petri nets and PA-processes. In Martin Abadi and Takayasu Ito, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, volume 1281 of *LNCS*. Springer Verlag, 1997.

[May97b] Richard Mayr. Model checking PA-processes. In *International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*. Springer Verlag, 1997.

[May97c] Richard Mayr. Process rewrite systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 7, 1997. Proceedings of Expressiveness in Concurrency (EXPRESS'97).

[May97d] Richard Mayr. Semantic reachability. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 5, 1997.

[May97e] Richard Mayr. Tableau methods for PA-processes. In D. Galmiche, editor, *Analytic Tableaux and Related Methods (TABLEAUX'97)*, volume 1227 of *LNAI*. Springer Verlag, 1997.

[May98] Richard Mayr. Strict lower bounds for model checking BPA. May 1998.

[MB96] F. Moller and G. Birtwistle, editors. *Logics for Concurrency*, volume 1043 of *LNCS*. Springer Verlag, 1996.

[ME96] S. Melzer and J. Esparza. Checking system properties via integer programming. In H.R. Nielson, editor, *Proc. of ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 250–264. Springer Verlag, 1996.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mol96] Faron Moller. Infinite results. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*. Springer Verlag, 1996.

[Mur89] T. Murata. Petri nets: Properties, analysis und applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[ORSv95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer Verlag, 1994.

[Pet81] J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*. IEEE, 1977.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[SC93] B. Steffen and R. Cleaveland. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. *International Journal on Formal Methods in System Design*, 1, 1993.

[Ste93] B. Steffen. Generating data flow analysis algorithms from modal specifications. *International Journal on Science of Computer Programming*, 21:115–139, 1993.

[Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.

[Sti95] C. Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors, *Proceedings of CONCUR'95*, volume 962 of *LNCS*, pages 1–11, 1995.

[Sti96] C. Stirling. Modal and temporal logics for processes. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *LNCS*, pages 149–237. Springer Verlag, 1996.

[SW90] C. Stirling and D. Walker. CCS, liveness, and local model checking in the linear time μ-calculus. In *Proceedings of the First International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 166–178. Springer Verlag, 1990.

[SW91] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297 – 322, 1992.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In *Conference Record of the 15th Annual Symposium on Principles of Programming Languages (POPL'88)*, pages 250–259. ACM Press, 1988.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*. Elsevier, 1990.

[Wal96a] I. Walukiewicz. Pushdown processes: games and model checking. In *International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*. Springer Verlag, 1996.

[Wal96b] I. Walukiewicz. Pushdown processes: games and model checking. Technical Report RS-96-54, BRICS, Aarhus, Denmark, 1996. Longer version of a CAV'96 paper.

[Yen92] H. Yen. A unified approach for deciding the existence of certain Petri net paths. *Information and Computation*, 96(1):119–137, 1992.

List of Figures

Index