Efficient Symbolic Analysis of Bounded Petri Nets Using Interval Decision Diagrams

Von der Fakultät für
Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
genehmigte Dissertation

vorgelegt von
Dipl.-Math. Alexey A. Tovchigrechko
geboren am 15. 3. 1978
in Krasnoarmeisk, Moskau Gebiet, Rußland

Gutachter: Prof. Dr.-Ing. Monika Heiner
Gutachter: Prof. Dr. habil. François Fages
Gutachter: Prof. Dr. rer. nat. Kurt Lautenbach

Tag der mündlichen Prüfung: 16.10.2008

Contents

1 Introduction

1.1 Background

Petri nets [Pet62] are an excellent formalism to model a wide class of concurrent and asynchronous systems. The formalism combines an intuitive graphical notation with a formal definition and a number of advanced analysis methods. Its applications comprise the analysis of systems arising in asynchronous circuit design, communication protocols, distributed software, production systems, flexible manufacturing, and systems biology, to name some examples.

Historically speaking, the Petri net formalism originating from the dissertation of Carl Adam Petri [Pet62] was one of the first approaches introduced to deal with concurrency and synchronization. Since 1970's both the theory and applications of Petri nets have been actively researched. A number of analysis techniques and extensions of the formalism have been proposed. "Petri nets" has become now a generic name for a whole class of models. Place/Transition Petri nets (P/T nets, for short) are the most prominent and best studied class of Petri nets.

Once a system has been modeled as a Petri net, its behavior can be studied with a variety of analysis methods such as *linear algebraic techniques*, techniques based on the *enumeration of reachable states*, and *animation*. Enumerative methods permit to answer any analysis questions for a model with a finite number of states, but have to deal with the *state explosion problem*: the number of reachable states even of a small Petri net can be enormous. Sources for the state explosion are concurrency and a combinatorial explosion occurring, for example, when a model captures that a system has data structures which assume many different values. Linear algebraic methods avoid the state explosion problem and can be applied to models with infinite number of states, but they can not provide all answers. Many of these methods are restricted to special subclasses of P/T nets. Animation is always applicable, but it is akin to testing and is essentially incomplete type of analysis. Being helpful to understand the system under consideration, it can not be relied upon when attempting, for example, to discover all undesirable behaviors.

Temporal logics have proven to be useful for specifying properties of concurrent systems as they can describe the ordering of events in time without introducing the time explicitely. Originally, temporal logics were developed by philosophers to study the way that time is used in natural language arguments. They were first suggested for

the specification of properties and verification of concurrent programs in [Pnu77]. The introduction of temporal-logic *model checking* algorithms [CE81] allowed this type of reasoning to be automated.

There are two possible views regarding the nature of time, which induce two types of temporal logics [Lam80]. In *linear* temporal logics, time is treated as if each moment has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and can be regarded as formulas describing a behavior of a single computation of a system. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logics are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of the system. *Linear Temporal Logic* (*LTL*) [Pnu80] and branching *Computation Tree Logic* (*CTL*) [CE81] are the two most popular and commonly supported temporal logics. Being relatively easy to use, these logics allow specification of many properties of interest, as well as efficient implementation of model checking tools.

Model checking [CGP01] was developed as a technique for the formal verification of hardware and software systems. It has been proven to be a successful method, frequently used to uncover well-hidden bugs in complex sequential circuit designs and communication protocols. Model checking provides means to check whether a *finite state model* of a system satisfies a given specification. The benefit of this restriction is that the verification can be performed fully automatically. The procedure uses normally an exhaustive exploration of all possible states of the model to determine whether it satisfies a property expressed in a temporal logic. An important feature of a model checker is that in case of a negative result, the user is often provided with a trace which can be used as a counterexample for the checked property. Counterexamples can help to determine whether the negative result was caused by an error in the system or comes from incorrect modeling or an incorrect specification. Today, applications of model checking are not limited to the hardware and software verification, it is often used to study properties of models arising in different, not necessary technical areas.

The main drawback of model checking is the state explosion problem. A number of techniques have been proposed to combat the problem. Two of the most successful approaches are *partial-order reductions* [Val91, God91, Pel94] and *symbolic methods*.

The advent of symbolic model checking [BCM$^+$90, McM92] has revolutionized the field of formal verification, transforming it from a purely academic discipline into a industrially applied technique. The key idea underlying symbolic methods is to represent sets of states using their *characteristic functions* and to manipulate them as if they were in bulk. Symbolic methods derive their efficiency from the fact that in many cases of interest large sets of states can be represented concisely by characteristic functions. Typical operations used in symbolic algorithms are computing the set of all successors or all predecessors of states in some set and usual set operations like intersection, union, test

for emptiness. Traditional symbolic algorithms are based on manipulations of boolean functions and *Reduced ordered binary decision diagrams* (ROBDDs) [Bry86].

Binary decision diagrams (BDDs) were studied in [Lee59] and [Ake78] as a data structure for the representation of boolean functions. Reduced ordered binary decision diagrams introduced in [Bry86] are a *canonical form* representation for boolean functions. ROBDDs are often substantially more compact than traditional normal forms, moreover, they can be manipulated very efficiently. Hence, they have become very popular and are widely used for a variety of applications like computer aided design, verification of finite-state concurrent systems, etc. The success of ROBDDs has inspired efforts to improve their efficiency and to expand the range of the applicability. Techniques have been discovered to make representations more compact and to represent other classes of functions.

1.2 Motivation

Considering implementation of efficient model checking tools for Petri nets, the most research efforts have been directed at techniques for 1-bounded P/T nets. Both symbolic [YHTM96, Spr01] and partial-order methods [VHHP95, EH01, Hel02] have been successfully applied, a number of tools are available. Reports on application of symbolic techniques to k-bounded P/T nets can be met in the literature [PRCB94, LR95, ST98, MC99], but the only available tool implementing symbolic CTL model checking is `SMART` [CJMS01]. Though every bounded net can be *unfolded* into 1-bounded P/T net, in practice, the unfolding is usually complicated or results in huge nets, which can not be analyzed efficiently.

In the latest projects running at our chair Petri nets are used to model and analyze biochemical systems [HK04, HKW04, KJH05]. k-bounded P/T nets are used for the *qualitative* analysis of these systems. Many of the arising models can not be analyzed by existing model checking tools, new techniques are required.

The research in this thesis will focus on the different techniques which can improve efficiency of the symbolic analysis of k-bounded P/T nets.

1.3 Organization of Thesis and Contributions

The thesis is organized as follows:

Chapter 2 Basic concepts of the Petri nets theory are presented in this chapter. We introduce then P/T net with extended arcs. This net class is strictly more powerful than the class of P/T nets and includes extensions quite often needed in practice: *inhibitor*, *read* and *reset* arcs. In this thesis we shall concentrate on the analysis of bounded P/T net with extended arcs. Being simple, this net class

provides a possibility to implement a number of analysis techniques efficiently, however, it still allows to model a wide class of systems.

Chapter 3 Boolean functions naturally encode sets of states of 1-bounded P/T nets, but a number of problem arise when they must be used to represent sets of states of k-bounded nets. By contrast, *interval logic functions* [LR95] allow a natural encoding of these sets. In this chapter we introduce an extension of BDDs that we denote as *interval decision diagrams*. This extension was proposed in [LR95] and then, probably independently from [LR95], in [ST98]. *Reduced ordered interval decision diagrams* (ROIDDs) are a canonical form representation for interval logic functions. We discuss basic ROIDD algorithms and consider elaborately how to implement an efficient ROIDD package. This subject was not covered in previous publications on Interval decision diagrams.

Chapter 4 This chapter represents the kernel of the thesis. We discuss how special operations used in the symbolic algorithms can be implemented efficiently and introduce a number of functions for the symbolic manipulation of Petri nets.

Though small decision diagrams can encode large sets of states, not every large set of states can be encoded by a small decision diagram. The breath-first order strategy traditionally applied in symbolic algorithms is not well suited for asynchronous systems. Often, sizes of decision diagrams encoding working sets of symbolic algorithms are much larger during the computation than upon termination. As the efficiency of the operations on decision diagrams depends on their sizes, the performance decreases as big diagrams start to be generated. Straying from the breath-first strategy in the exploration of state spaces can improve the reachability analysis. We study techniques to reduce sizes and a number of intermediate diagrams, and propose then a new saturation approach, which exploits both the structure of k-bounded P/T nets and the structure of ROIDDs. It manages to keep sizes of intermediate diagrams smaller than other approaches and can drastically improve efficiency of the reachability analysis.

Decomposing a graph into its strongly connected components (SCCs) is a fundamental graph problem and has many applications in the analysis of different properties. We propose new algorithms for enumeration of SCCs in sets of states of P/T nets. To implement an algorithm that enumerates *terminal* SCCs we adapt and improve an algorithm introduced in [XB98] for the state classification of finite-state Markov chains. To implement an algorithm that enumerates all SCCs we adapt an algorithm introduced in [BGS00]. A saturation-based implementation allows to significantly improve efficiency of these algorithms.

Finally, we discuss how to analyze basic net properties efficiently.

Chapter 5 In this chapter we make a short introduction into temporal logics and model checking. We concentrate on the temporal logics CTL and LTL and discuss corresponding model checking algorithms.

Chapter 6 Implementation of an efficient symbolic CTL model checker for P/T nets with extended arcs is subject of this chapter. Conventional symbolic CTL algorithms are based on backward breath-first order exploration of the state space. We study how the saturation approach can be employed in CTL algorithms to improve their efficiency. Performance of the symbolic state space exploration depends heavily on the structure of the model. Sometimes, forward state traversals are quite efficient, whereas intermediate decision diagrams created during the backward state space exploration become too large and can not be handled efficiently. A CTL model checking algorithm based mainly on forward state traversals was suggested in [INH96]. We show how this algorithm can be adopted and implemented efficiently.

Chapter 7 In this chapter we discuss how to implement a symbolic LTL model checker for k-bounded P/T nets with extended arcs. Implementation of a symbolic LTL model checker for 1-bounded P/T nets was described in [Spr01]. We are not aware of any previous attempts to implement a symbolic LTL model checker for k-bounded nets. We describe the employed approach and consider a number of algorithms that can outperform the Emerson-Lei algorithm [EL86], which is conventionally used in symbolic LTL model checking. Finally, we discuss how to adapt and improve the "on-the-fly" algorithm introduced in [Spr01].

Chapter 8 This chapter summarizes the achieved results and provides some ideas for future research.

2 Petri Nets

Petri nets [Pet62] are an excellent formalism to model a wide class of concurrent and asynchronous systems. The formalism combines an intuitive graphical notation with a formal definition and a number of advanced analysis methods. Its applications comprise the analysis of systems arising in asynchronous circuit design, communication protocols, distributed software, production systems, flexible manufacturing, and systems biology, to name some examples.

Historically speaking, the Petri net formalism originating from the dissertation of Carl Adam Petri [Pet62] was one of the first approaches introduced to deal with concurrency and synchronization. Since 1970's both the theory and applications of Petri nets have been actively researched. A number of analysis techniques and extensions of the formalism have been proposed. "Petri nets" has become now a generic name for a whole class of models. The classic monographs on Petri nets are [Pet81], [Rei86], [Sta90], [DE95], and [Jen95], [Jen96].

Place/Transition Petri nets (P/T nets, for short) are the most prominent and best studied class of Petri nets. Exactly this class is most often referred as "Petri nets". A P/T net is a directed bipartite graph with two types of nodes called *places* and *transitions*. The nodes are connected by directed weighted arcs. Connections between nodes of the same type are not allowed. Places are usually represented by circles and transitions by rectangles. Places may contain zero or more *tokens* drawn as black dots. A distribution of tokens over the places is called a *marking* of the net. Every marking corresponds to a state of the modeled system. The initial state of the system is represented by the *initial marking* of the net. A place is an *input place* of some transition if there exists an arc from this place to this transition. A place is an *output place* of some transition if there exists an arc from this transition to this place. A transition is called *enabled* if all its input places contain at least a number of tokens specified by the weights of the corresponding arcs. An enabled transition may *fire*. Firing a transition means consuming tokens from the input places and producing tokens in the output places, a new marking is created. The behavior of the system is defined by the set of all markings reachable from the initial state. A *reachability graph* of the net is a graph representing these reachable markings and transitions between them.

For a Petri net, which models some system, properties such as *boundedness*, *liveness*, *reversibility*, and *deadlock-freedom* are often an object of interest. These properties are *decidable* for P/T nets. Most of the analysis techniques either make use of the

reachability graph or involve linear algebraic techniques. For example, the analysis of *strongly connected components* of the reachability graph can be used to decide liveness and reversibility of bounded P/T nets. *Invariants* analysis is an example of the linear algebraic technique. *Structural boundedness* or non-reachability of some particular marking can be decided using P-invariants.

In this chapter P/T nets with *extended arcs* are introduced. Three new arc types between places and transitions are allowed: *inhibitor*, *read* and *reset* arcs. Inhibitor arcs were first introduced in [FA73] to solve a synchronization problem beyond the power of P/T nets. A transition connected with a place by an inhibitor arc can be enabled only if this place contains *less* tokens than the weight of this inhibitor arc. Inhibitor arcs can be used, for example, to model priorities [Hac76] and are needed especially often in the biochemical models [HKW04]. Read arcs [MR95] allow to model that some resource is read, but not consumed by a transition. Reset arcs [AK77] allow a transition to empty a place independently of how many tokens this place contains.

Nets with inhibitor arcs can model Turing machines [Hac76]. This means, every non-trivial problem like boundedness or reachability is not generally decidable for this class of the nets. The only available general analysis methods are based on the reachability graph, provided that the net is bounded.

As the main focus of this thesis is the analysis, we do not consider questions of modeling using Petri nets and refer the interested reader to the monographs mentioned above.

2.1 Definition of P/T Nets

Definition 1 (Net)
A *Net* is a tuple $N = [P, T, F]$ where:

1. P and T are finite sets, satisfying $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.
2. F is a *flow relation*: $F \subseteq (P \times T) \cup (T \times P)$.

A *Net* is a finite *bipartite* directed graph. Elements of P are called *places*, elements of T are called *transitions*. Elements of $P \cup T$ are denoted as *nodes*, elements of the flow relation F are denoted as *arcs*.

Definition 2 (Pre- and Post-set)
Let $N = [P, T, F]$ be a Net. For a node $x \in P \cup T$ two sets of nodes are defined

1. $\bullet x = \{\, y \in P \cup T \mid (y, x) \in F \,\}$ is a *pre-set* of x.
2. $x\bullet = \{\, y \in P \cup T \mid (x, y) \in F \,\}$ is a *post-set* of x.

For a set of nodes $X \subseteq P \cup T$ we define

$$\bullet X = \bigcup_{x \in X} \bullet x \quad \text{and} \quad X \bullet = \bigcup_{x \in X} x \bullet .$$

Places of a net may contain zero or more *tokens*. A distribution of tokens over the places of the net is called a *marking* or a *state* of the net.

Definition 3 (Marking)
Let $N = [P, T, F]$ be a Net. Any mapping $m : P \to \mathbb{N}_0$ from the set of places into the set of natural numbers is denoted as a *marking*, where $m(p)$ defines a number of *tokens* in the place $p \in P$.

Every marking corresponds to some state of the modeled system. If places of the net are enumerated: $P = \{ p_1, p_2, \ldots, p_n \}$, then any marking of the net can be also represented as a vector $(m(p_1), m(p_2), \ldots, m(p_n))$. From now on we shall not differentiate between these two representations. For vectors we define comparison and addition place-wise

- $m_1 \leq m_2$ if and only if $\forall p \in P \; m_1(p) \leq m_2(p)$.
- $m_1 < m_2$ if and only if $m_1 \leq m_2$ and $\exists p \in P : m_1(p) < m_2(p)$.
- $m = m_1 + m_2$ if and only if $\forall p \in P \; m(p) = m_1(p) + m_2(p)$.

Definition 4 (Place/Transition Net)
A *Place/Transition net* (*P/T net* for short) is a tuple $[P, T, F, V, m_0]$ where:

1. $[P, T, F]$ is a Net.
2. V is a *weight function*, assigning to every arc of the net some positive natural number $V : F \to \mathbb{N}$.
3. m_0 is an *initial marking*.

Note that from now on we shall write $V(x, y)$ meaning actually $V((x, y))$.

Example 1
A P/T net shown in Fig.2.1 is defined with

- Places $\{ p_1, p_2, p_3, p_4, p_5 \}$, transitions $\{ t_1, t_2, t_3, t_4 \}$.
- A flow relation $\{ (p_1, t_1), \; (p_1, t_3), \; (t_2, p_1), \; (t_4, p_1), \; (p_2, t_1), \; (t_2, p_2), \; (p_3, t_2), \; (t_1, p_3), \; (p_4, t_3), \; (t_4, p_3), \; (p_5, t_4), \; (t_3, p_5) \}$.
- A weight function
  $V(p_1, t_1) = 1, \; V(p_1, t_3) = 2, \; V(t_2, p_1) = 1, \; V(t_4, p_1) = 2,$
  $V(p_2, t_1) = 1, \; V(t_2, p_2) = 1, \; V(p_3, t_2) = 1, \; V(t_1, p_3) = 1,$
  $V(p_4, t_3) = 1, \; V(t_4, p_3) = 1, \; V(p_5, t_4) = 1, \; V(t_3, p_5) = 1.$
- An initial marking $(3, 2, 0, 1, 0)$.

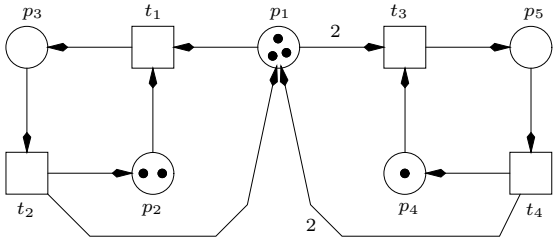


Figure 2.1: A Place/Transition net

2.2 Dynamic Behavior of P/T Nets

In the previous section the structure of P/T nets was defined, now we can discuss their dynamic behavior.

Definition 5 (Enabling and firing vectors)
Let $N = [P, T, F, V, m_0]$ be a P/T net. For every transition $t \in T$ we define mappings t^-, t^+ and Δt for every place $p \in P$

$$t^-(p) = \begin{cases} V(p,t) & \text{if } p \in {\bullet t} \\ 0 & \text{otherwise} \end{cases}$$

$$t^+(p) = \begin{cases} V(t,p) & \text{if } p \in t{\bullet} \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta t(p) = t^+(p) - t^-(p).$$

For a net N with places $P = \{ p_1, p_2, \ldots, p_n \}$ we can also consider the mappings t^-, t^+ and Δt as vectors having n integer elements.

Definition 6 (Enabled transition)
Let $N = [P, T, F, V, m_0]$ be a P/T net and m be a marking of N. A transition $t \in T$ is called *enabled* in the marking m if $m \geq t^-$. We define a function enabled(m) which returns a set of transitions enabled in m:

$$\text{enabled}(m) = \{ t \in T \mid m \geq t^- \}.$$

Definition 7 (Firing)
Let $N = [P, T, F, V, m_0]$ be a P/T net, m be a marking of N and some transition $t \in T$ is enabled in m. The transition t *may fire* in the marking m and create a new marking $m' = m + \Delta t$. We shall denote this as $m \xrightarrow{t} m'$.

The integer vector Δt describes the effect of the firing of the transition t. When t fires, tokens are removed from the places belonging to the pre-set of t and put into the places in the post-set of t. This means that firing of a transition is a *local event*, affecting only the neighboring places of t and leaving all others places of the net without changes.

Definition 8 (Concurrent transitions)
Let $N = [P, T, F, V, m_0]$ be a P/T net and $U \subseteq T$ be a set of transitions. The set U is called *concurrent* in a marking m if $\sum_{t \in U} t^- \leq m$.

If a set U is concurrent in a marking m, then there are enough tokens in m for all transitions of U to fire "at the same time". We say that transitions $t_1, \ldots, t_k$ are concurrent in a marking m, if a set $U = \{ t_1, \ldots, t_k \}$ is concurrent in m.

Definition 9 (Conflict)
Let $N = [P, T, F, V, m_0]$ be a P/T net, m be a marking of N. We say that transitions $t_1, t_2 \in T$ are in *conflict* in m, if they are enabled in m, but not concurrent in m.

Example 2
Consider the P/T net in Fig.2.1. Transitions t_1 and t_3 are concurrent in the initial marking $(3, 2, 0, 1, 0)$, but are in conflict in the marking $(2, 1, 1, 1, 0)$.

Definition 10 (Reachability relation)
Let $N = [P, T, F, V, m_0]$ be a P/T net. We denote as T^* the set of all finite sequences of elements of T. Note that T^* also includes the empty sequence ϵ. For two markings m and m' of N and a sequence $\delta \in T^*$ we define the relation $m \xrightarrow{\delta} m'$ inductively.

1. $m \xrightarrow{\epsilon} m'$ if and only if $m = m'$.
2. $m \xrightarrow{\delta t} m'$ if and only if $\exists m'' : \; m \xrightarrow{\delta} m'' \; \wedge \; m'' \xrightarrow{t} m'$.

We define the *reachability relation* $\xrightarrow{*}$ of N as follows: $m \xrightarrow{*} m'$ if and only if $\exists \delta \in T^* : m \xrightarrow{\delta} m'$. A marking m' is called *reachable* from m if and only if $m \xrightarrow{*} m'$.

Lemma 1 (Monotonicity of Firing)
Let $N = [P, T, F, V, m_0]$ be a P/T net, m and m' be two markings of N. If $m \xrightarrow{\delta} m'$, then $(m + l) \xrightarrow{\delta} (m' + l)$ $\forall l \in \mathbb{N}^{|P|}$.

Proof: By induction over the length of δ. □

Lemma 1 states an important *monotonicity property* of P/T nets: transitions firable from some marking m are also firable from any larger marking. As we shall see later, some decidability results rely on this property.

Definition 11 (Reachability set)
Let $N = [P, T, F, V, m_0]$ be a P/T net. We denote as $\mathcal{R}_N(m)$ the set of all markings of N reachable from m:

$$\mathcal{R}_N(m) = \{\, m' \mid m \xrightarrow{*} m' \,\}.$$

$\mathcal{R}_N(m_0)$ defines the *reachability set* of N, we shall also refer to it as the *state space*.

2.3 Reachability Graph

A reachability graph corresponds to an *interleaving model* of the system behavior. In the interleaving model all events of a single execution are arranged in a linear order called an *interleaving sequence*. Concurrently executed events appear arbitrary ordered with respect to one another.

Definition 12 (Reachability graph)
Let $N = [P, T, F, V, m_0]$ be a P/T net. As a *reachability graph* of N we denote a labeled directed graph $\mathcal{RG}_N = [\mathcal{R}_N(m_0), B_N]$ where:

1. $\mathcal{R}_N(m_0)$ is the set of nodes.
2. B_N is the set of labeled arcs:

$$B_N = \{\, [m, t, m'] \mid \exists\, m, m' \in \mathcal{R}_N(m_0),\ \exists\, t \in T\ :\ m \xrightarrow{t} m' \,\}.$$

A reachability graph of a net N has all reachable markings of N as nodes and its arcs are labeled with transitions of N.

Example 3
The reachability graph for the net from Fig. 2.1 is shown in Fig. 2.2. An information about concurrency gets lost, if only the reachability graph is considered. Using the graph in Fig. 2.2 one can not say if transitions t_1 and t_3 are concurrent in the initial marking or can fire in any arbitrary order. The graph is finite, this, in general, must not be the case for any P/T net.

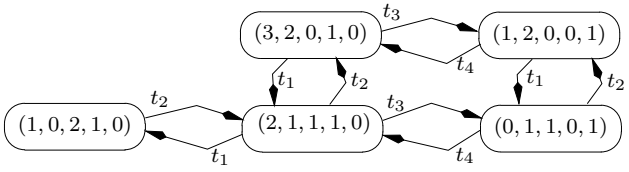


Figure 2.2: A reachability graph

Algorithm 1 (Reachability Graph Construction)

```
1  func DFS (N)
2     R_N:= {m_0}; B_N := ∅; New := ∅
3     push(New, m_0)
4     while New ≠ ∅ do
5        m := pop(New)
6        forall t ∈ enabled(m) do
7           m' := m + Δt
8           B_N := B_N ∪ {[m, t, m']}
9           if m' ∉ R_N then
10             R_N:= R_N ∪ m'
11             push(New, m')
12          fi
13       od
14    od
15    return (R_N, B_N)
16 end
```

A trivial algorithm to construct the reachability graph maintains a set of already explored markings $\mathcal{R}_N$, a set of unexplored markings *New* and a set of arcs B_N. In Algorithm 1 the set *New* is maintained as a *stack*. In this case the reachability graph is constructed in the *depth-first order*. One can replace this stack with a *queue* and get a *breath-first order* construction of the graph.

The complexity of Algorithm 1 is overexponential in the size of a P/T net. The reason for this is the *state explosion* problem: even small P/T nets can have finite but huge reachability graphs. Sources for the state explosion are concurrency (all interleavings have to be represented in the reachability graph) and a combinatorial explosion due to different combinations of tokens in places of the net. Example 4 demonstrates the overexponential state explosion.

Example 4

A P/T net in Fig. 2.3 is bounded in an unobvious but very regular way [Jan83]. The net is parameterized by the weight k of the arc between t_2 and p_3 and the initial marking m of the place p_1. Depending on these two parameters the total number of tokens in the net is bounded to $\max(m, k) = k \cdot f_k(m) + 2$, where f_k is defined inductively:

$$f_k \quad = \quad \begin{cases} k & \text{if } m = 0 \\ f_k(m-1) \cdot k^{f_k(m-1)} & \text{otherwise.} \end{cases}$$

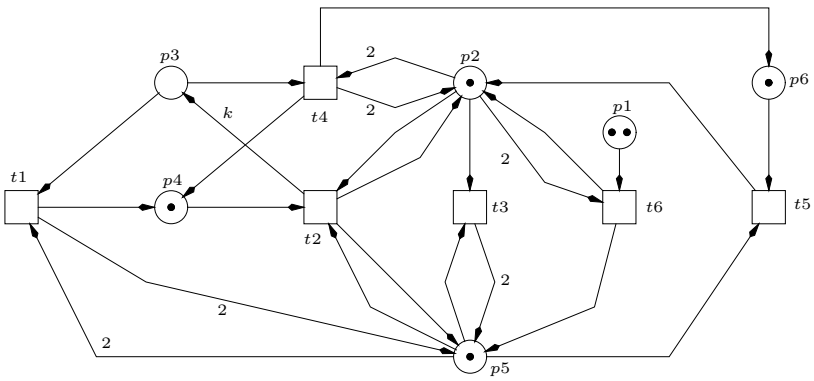


Figure 2.3: A P/T net demonstrating the overexponential growth of $\mathcal{RG}_N$

The maximal number of tokens in a single place is $k \cdot f_k(m)$. Some example values of $\max(m, k)$ are:

$$\begin{array}{lll} \max(1,2) = 18 & \max(1,3) = 245 & \max(1,4) = 4098 \\ \max(2,2) = 4098 & \max(2,3) = 3^{86} + 2 & \max(3,2) = 2^{2060} + 2. \end{array}$$

The number of the reachable markings grows accordingly:

$$\begin{array}{ll} |\mathcal{R}_N|(1,2) = 427 & |\mathcal{R}_N|(1,3) = 39,656 \\ |\mathcal{R}_N|(1,4) = 18,856,970 & |\mathcal{R}_N|(2,2) = 735,951,403. \end{array}$$

2.4 Basic Petri Nets Properties

In this section we define basic P/T net properties and discuss shortly their decidability. For bounded nets we discuss verification of the properties with help of a reachability graph. A more complete consideration of the properties with proofs can be found, for example, in [Sta90]. A good survey on the decidability issues of Petri nets is [EN94].

2.4.1 Boundedness

Definition 13 (Boundedness)
Let $N = [P, T, F, V, m_0]$ be a P/T net, p be a place of N, m be a marking of N and k be some natural number.

1. p is called *k-bounded* if $m(p) \leq k$ in all markings m reachable from m_0.
2. p is called *bounded* if $\exists k : p$ is k-bounded.
3. N is called bounded if all places of N are bounded.

Corollary 1
A P/T net $N = [P, T, F, V, m_0]$ is bounded if and only if its reachability set $\mathcal{R}_N(m_0)$ is finite.

The boundedness of a P/T net is a very important property. Obviously the reachability graph generation procedure (Algorithm 1) does not terminate for unbounded P/T nets. This limits drastically the number of methods available for the analysis of such nets. Note that the boundedness is a *dynamic* property as it treats all reachable states of a net and depends also on the initial marking. However, as we shall see later, it is possible to define a *sufficient* condition for the boundedness using only the structure of a net.

We need the following proposition for the proof of the sufficient condition for unboundedness of a P/T net.

Proposition 1
Let (m_i) be an infinite sequence of markings. There exists an infinite subsequence (m'_i) of (m_i) such that $j < k \Rightarrow m'_j \leq m'_k$.

Proof: By construction. □

Lemma 2 (Boundedness criterion)
Let $N = [P, T, F, V, m_0]$ be a P/T net. N is unbounded if and only if there exist reachable from m_0 markings $m, m_1 \in \mathcal{R}_N(m) : m \xrightarrow{\delta} m_1 \ \wedge \ m_1 > m$.

Proof: "⇐": We prove by contradiction that N can not be k-bounded. From the monotonicity property of P/T nets (see Lemma 1) follows:

$$m \xrightarrow{\delta} m_1 \xrightarrow{\delta} m_2 \xrightarrow{\delta} \cdots m_k \xrightarrow{\delta} m_{k+1} \xrightarrow{\delta} \cdots$$

From $m_1 > m$ follows that there exists some place $p : m_1(p) > m(p)$. This means $m_1(p) > 1,\ m_2(p) > 2,\ \ldots,\ m_{k+1}(p) > k$, so N can not be k-bounded.

"⇒": Consider an infinite reachability graph $\mathcal{RG}_N$. $\mathcal{RG}_N$ has infinitely many nodes, but a number of outgoing arcs of every node is bounded by $|T|$. Consider m_0, an infinite number of paths that do not visit any node twice starts at this node. Infinitely many such paths go at least through one of the outgoing arcs of m_0. We take one of these paths, let m_1 be the last node in it. We repeat for m_1 the same considerations as for m_0 and choose a path with a last node m_2 in it. Continuing this procedure we get an infinite sequence of different markings (m_i) such that $m_0 \xrightarrow{*} m_1 \xrightarrow{*} m_2 \xrightarrow{*} \cdots$ and $m_i \neq m_j$ for $i \neq j$. Because of Proposition 1 there exists a subsequence (m'_j) of (m_i) such that $j < k \Rightarrow m'_j \leq m'_k$. We can select m'_1, m'_2 from (m'_j) such that $m'_1 \xrightarrow{*} m'_2$, $m'_1 \leq m'_2$ and $m'_1 \neq m'_2$. □

The sequence δ in Lemma 2 can be seen as a *token generator*. Karp and Miller proved decidability of boundedness in [KM69]. They showed how to detect token generators by constructing what was later called the *coverability tree*. Finkel improved the coverability tree generation algorithm in [Fin93].

2.4.2 Reachability Problem

Definition 14 (Reachability problem)
Let $N = [P, T, F, V, m_0]$ be a P/T net. The *reachability problem* consists of deciding if a marking m is reachable from m_0.

It was observed [Hac75] that many other net problems are recursively equivalent to the reachability problem, so it became a central issue of the net theory. A quite complicated proof of the decidability was first given in [May81]. A number of simplifications of the proof have been done later, as the last reference see [PW03].

Though the reachability problem is decidable, in general case none of the known algorithms is primitive recursive. If a P/T net is bounded, then a variation of Algorithm 1 can be used to check the reachability of a marking m. The procedure can be terminated as soon as m is discovered. If one is interested not only in the reachability, but also in the shortest way to m, then the discussed above breath-first order version of the algorithm can be used.

2.4.3 Liveness

Definition 15
Let $N = [P, T, F, V, m_0]$ be a P/T net.

1. A marking m of N is called *dead* if $\text{enabled}(m) = \emptyset$.
2. A transition t of N is called *dead* in a marking m if
$$\nexists m' : m \xrightarrow{*} m' \ \wedge \ t \in \text{enabled}(m').$$
3. A transition t of N is called *live in a marking* m if
$$\nexists m' : m \xrightarrow{*} m' \ \wedge \ t \text{ is dead in } m'.$$
4. A transition t of N is called *live* if it is live in m_0.
5. A marking m is called *live* if all transitions $t \in T$ are live in m.
6. A net N is called *live* if m_0 is live.
7. A net N is called *deadlock-free* if $\forall m : m_0 \xrightarrow{*} m \ \text{enabled}(m) \neq \emptyset$.

Dead markings represent those states of the modeled system, in which it can not make any progress. During a verification of a system it is often desirable to detect such states and eliminate them. The deadlock-freedom problem can be reduced in polynomial time to the reachability problem, therefore it is decidable.

Corollary 2
A P/T net $N = [P, T, F, V, m_0]$ is deadlock-free if and only if its reachability graph does not contain nodes without outgoing arcs.

Loosely speaking, a P/T net is live if every transition has always a possibility to fire again. [Hac75] showed that liveness is recursively equivalent to the reachability problem, and thus decidable. If a P/T net is bounded, then the following corollary suggests how its liveness can be checked.

Corollary 3
A transition t of a bounded P/T net $N = [P, T, F, V, m_0]$ is live if and only if in every terminal strongly connected component[1] C of the reachability graph $\mathcal{RG}_N = [\mathcal{R}_N(m_0), B_N]$ there exist nodes $m, m' \in C : [m, t, m'] \in B_N$.

Corollary 4
If a P/T net $N = [P, T, F, V, m_0]$ is not deadlock-free, then all transitions $t \in T$ are not live.

2.4.4 Home States and Reversibility

Definition 16
Let $N = [P, T, F, V, m_0]$ be a P/T net.

1. A marking m of N is called a *home state* if $\forall m' \in \mathcal{R}_N(m_0)\ \exists \delta \in T^* : \delta \neq \epsilon$ and $m' \xrightarrow{\delta} m$.

2. A net N is called *reversible* if m_0 is a home state.

Loosely speaking, a marking is a home state if it is reachable from every reachable state. The home state problem was shown to be decidable in [FEJ89].

Corollary 5
A bounded P/T net $N = [P, T, F, V, m_0]$ is reversible if and only if its reachability graph is strongly connected.

[1] See definitions of strongly connected components (SCCs) in the Appendix. We shall discuss the SCC decomposition of graphs in chapter 4.

2.5 Structural Techniques

Analysis based on the reachability graph suffers from the state explosion problem, and it can not be applied to unbounded nets at all. A number of *structural* techniques that avoid construction of the reachability graph have been developed for P/T nets. Analysis based on *reduction*, *deadlock/trap property* or *decomposition* can be highly effective for the special subclasses of P/T nets like *marked graphs*, *state machines*, *extended free choice nets* or *extended simple nets*, see [Sta90] and [DE95]. We consider here only generic techniques that we shall discuss later as means to improve the analysis based on the reachability.

Definition 17 (Incidence matrix)
Let $N = [P, T, F, V, m_0]$ be a P/T net. The *incidence matrix* $C : P \times T \to \mathbb{Z}$ of N is defined with $C(p,t) = \Delta t(p)$.

Example 5
An incidence matrix for the P/T net from Fig.2.1 is:

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} -1 & 1 & -2 & 2 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Definition 18 (Parikh vector)
Let $N = [P, T, F, V, m_0]$ be a P/T net and $\delta \in T^*$ be some finite transition sequence. The Parikh vector $\Psi(\delta) : T \to \mathbb{N}_0$ assigns to every transition t the number of occurrences of t in δ.

Lemma 3 (State equation)
Let $N = [P, T, F, V, m_0]$ be a P/T net and m, m' be markings of N. For every finite transition sequence $\delta : m \xrightarrow{\delta} m'$ holds the following state equation:

$$m' = m + C \cdot \Psi(\delta).$$

Proof: By induction over the length of δ. □

The state equation can be used to formulate a *necessary*, but not sufficient condition for the reachability of a marking.

Corollary 6 (Necessary condition for the reachability of a marking)
Let $N = [P, T, F, V, m_0]$ be a P/T net. If a marking m is reachable from the initial marking m_0, then the equation $C \cdot x = m - m_0$ has a nonnegative integer solution x.

In other words, if the equation $C \cdot x = m - m_0$ does not have a nonnegative integer solution, then m is definitely not reachable.

Definition 19 (P-invariants)
Let $N = [P, T, F, V, m_0]$ be a P/T net and C be the incidence matrix of N.

1. Every nontrivial integer solution of the linear homogeneous equation $\vec{y} \cdot C = \vec{0}$ is called a *P-invariant* of the net N.
2. A P-invariant $\vec{y}$ is called *semipositive* if $\vec{y} \geq \vec{0}$.
3. If $\vec{y}$ is a *semipositive* P-invariant, we denote the set $\langle \vec{y} \rangle = \{ p \mid \vec{y}(p) > 0 \}$ as a *support* of $\vec{y}$.
4. A semipositive P-invariant $\vec{y}$ is called *minimal* if there exists no other semipositive P-invariant $\vec{j}$ such that $\langle \vec{j} \rangle \subset \langle \vec{y} \rangle$.

Efficient calculation of minimal P-invariants is discussed in [CS89]. From the state equation follows the following corollary.

Corollary 7 (Invariance)
Let $N = [P, T, F, V, m_0]$ be a P/T net and $\vec{y}$ be a P-Invariant of N. Then the equation $\vec{y} \cdot m^{\mathsf{T}} = \vec{y} \cdot m_0^{\mathsf{T}}$ holds for all markings m reachable from the initial marking m_0.

This means if P-invariants are considered as weighting vectors, then Corollary 7 states that a weighted number of tokens remains constant in all reachable markings. P-invariants can be also used to formulate a necessary condition for the reachability of a marking and a sufficient condition for the *structural boundedness* of a P/T net.

Corollary 8 (Necessary condition for the reachability of a marking)
Let $N = [P, T, F, V, m_0]$ be a P/T net, m be a marking of N. If N has a P-invariant $\vec{y}$ such that $\vec{y} \cdot m^{\mathsf{T}} \neq \vec{y} \cdot m_0^{\mathsf{T}}$, then m is not reachable from m_0.

Corollary 9 (Structural boundedness)
Let $N = [P, T, F, V, m_0]$ be a P/T net. If for every place p there exists a semipositive P-invariant $\vec{y_p}$ of N such that $\vec{y_p}(p) > 0$, then N is bounded.

A P/T net covered by P-invariants is bounded, unfortunately this is only a sufficient condition. For example, a net in Fig.2.4 is obviously bounded but does not have any P-invariants.

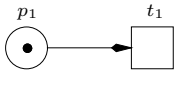


Figure 2.4: A bounded P/T without P-invariants

Definition 20 (Traps)
Let $N = [P, T, F, V, m_0]$ be a P/T. A non-empty set of places $H \subseteq P$ is called a *trap* if $H\bullet \subseteq \bullet H$.

Corollary 10
Let $N = [P, T, F, V, m_0]$ be a P/T net, $H \subseteq P$ be a trap, and m be a marking of N. If there exists a place $p \in H : m(p) > 0$, then $\forall m' : m \xrightarrow{*} m'\ \exists p' \in H : m'(p') > 0$.

If a trap is marked in some marking m, then it remains marked in all markings reachable from m.

2.6 P/T Nets with Extended Arcs

We consider a class of Petri nets with extensions quite often needed in practice. Concerning the expressive power, the introduced class is strictly more powerful than the class of P/T nets.

First, the classical P/T Nets are extended with the ability to handle *context*: transitions do not only produce and consume tokens, but can also have context conditions, specifying something that is needed for a transition to fire, but is not affected by the firing. The new two kinds of arcs are inhibitor and read arcs. *Inhibitor arcs* were first introduced in [FA73] to solve a synchronization problem beyond the power of P/T nets. A transition connected with a place by an inhibitor arc can be enabled only if this place contains *less* tokens than the weight of this inhibitor arc. Inhibitor arcs can be used, for example, to model priorities [Hac76] and are needed especially often in biological models [HKW04]. *Read arcs* were introduced in [MR95]. They allow to model that some resource is read, but not consumed by a transition. Hence the same token can be used by many transitions at the same time. This allows to specify a higher degree of concurrency than it is possible in P/T nets. Read arcs were proposed to model database systems, concurrent constraint programming or frameworks based on a shared memory. In the interleaving model read arcs can be considered as a "syntactic sugar" because they can be replaced by two opposite normal arcs with corresponding weights.

Second, we allow a transition to empty a place independently of how many tokens this place contains by means of *reset arcs* [AK77].

Definition 21 (P/T Net with extended arcs)
A P/T Net with extended arcs is a tuple $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ where:

1. $[P, T, F, V, m_0]$ is a P/T net.
2. $I \subseteq P \times T$ is the *inhibiting* relation.

3. $R \subseteq P \times T$ is the *reading* relation.
4. $Z \subseteq P \times T$ is the *reseting* relation.
5. $V_I : I \to \mathbb{N}$ is the weight function of inhibitor arcs.
6. $V_R : R \to \mathbb{N}$ is the weight function of read arcs.

Definition 22 (Extended enabling vectors)
Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs. In addition to the mapping t^- defined above we define now the mappings t_I^- and t_R^-

$$t_I^-(p) = \begin{cases} V_I(p,t) & \text{if } (p,t) \in I \\ \infty & \text{otherwise} \end{cases}$$

$$t_R^-(p) = \begin{cases} V_R(p,t) & \text{if } (p,t) \in R \\ 0 & \text{otherwise.} \end{cases}$$

Definition 23 (Extended enabling condition)
Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs and m be a marking of N. A transition $t \in T$ is *enabled* in the marking m if

$$m \geq t^- \text{ and } m \geq t_R^- \text{ and } m < t_I^-.$$

Definition 24 (Extended firing)
Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs, m be a marking of N and some transition $t \in T$ be enabled in the marking m. The marking m' created by firing the transition t in m is calculated as follows:

$$m'(p) = \begin{cases} m(p) + \Delta t(p) & \text{if } (p,t) \notin Z \\ t^+(p) & \text{otherwise.} \end{cases}$$

Example 6
Consider a P/T net in Fig.2.5

1. The transition t_0 is connected with p_0 by an inhibitor arc. t_0 is enabled when p_0 has less than two tokens. Firing of t_0 does not change the number of tokens in p_0.
2. The transition t_1 is connected with p_0 by a read arc and with p_1 by a reset arc. t_1 is enabled when p_0 has at least two tokens. If t_1 fires, then the number of tokens in p_0 will not be changed, but p_1 will become empty.
3. The transition t_2 is connected with p_0 by a read and an inhibitor arc, it is connected with p_1 by a reset and a normal arc. t_2 is enabled if p_0 contains exactly two tokens. If t_2 fires, then the number of tokens in p_0 will not be changed, but p_1 will have five tokens.

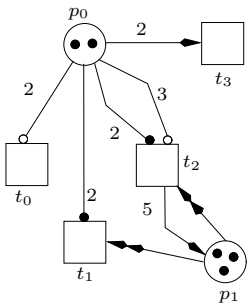


Figure 2.5: A P/T net with extended arcs

4. The transition t_3 is connected with p_0 by a normal arc, it is enabled if p_0 has at least two tokens. When t_3 fires, it consumes two tokens from p_0.

Nets with inhibitor arcs can model Turing machines [Hac76]. This means, every non-trivial problem like boundedness or reachability is not generally decidable for this class of the nets. The problem of the construction of a finite coverability tree for nets with inhibitor arcs is the loss of the monotonicity. It is no longer true that transitions firable from some marking m are also firable from any larger marking. It is shown in [Bus98] that finite coverability trees can be constructed for a constrained subclass of nets with inhibitor arcs. The constraint is defined as follows: it is possible to know a limit for each inhibiting place, in such a way that, if the number of tokens in the place exceeds the limit at some stage of computation, then this place can not be tested for absence of tokens any more.
Obviously, invariant analysis can not take inhibitor arcs into account. A place can be structurally bounded even if it is not covered by some P-invariant.

2.7 Closing Remark

Petri nets are an excellent formalism to model a wide class of concurrent systems. The formalism benefits from the mathematical foundation and exhibits a large number of techniques for analysis of the system properties.

Considering the interleaving semantic, all behaviors of a bounded net are represented by the reachability graph. Though structural techniques like invariants analysis can be used to decide some net properties, in general, only techniques based on the reachability graph can provide all answers about the behavior of a system. Actually, for nets with inhibitor arcs even just being able to tell if the net is bounded is already an undecidable problem. Hence, in practice we are limited to hoping that the reachability graph of a system is not only bounded, but is also small enough to fit in the memory of a

computer. Unfortunately reachability graphs can be really huge even for small nets. Therefore many research efforts have been directed at finding efficient algorithms and data structures to generate, store, and explore reachability graphs. Interval decision diagrams discussed in the next chapter are an example of such a data structure. In chapter 4 we shall introduce algorithms for the analysis of reachability graphs using interval decision diagrams. We shall also discuss how structural properties of nets can be used to improve efficiency of these algorithms.

3 Interval Decision Diagrams

A large number of problems in Computer Science can be reduced to the manipulation of *boolean functions*. *Binary decision diagrams* (*BDDs*) were studied in [Lee59] and [Ake78] as a data structure for the representation of boolean functions. *Reduced ordered binary decision diagrams* (*ROBDDs*) defined in [Bry86] are a *canonical form* representation for boolean functions. ROBDDs are often substantially more compact than traditional normal forms, and they can be manipulated very efficiently. Hence, they have become very popular and are widely used for a variety of applications like computer aided design, verification of finite-state concurrent systems, etc. According to statistics of the scientific literature digital library CiteSeer[1], [Bry86] is one of the most cited articles in Computer Science. A short introduction into boolean functions and ROBDDs can be found in the Appendix.

The success of ROBDDs has inspired efforts to improve their efficiency and to expand the range of the applicability. Techniques have been discovered to make representations more compact and to represent other classes of functions. A number of extensions of ROBDDs have been proposed, here is a small list of those, applied for analysis of (timed) Petri nets: *Zero suppressed binary decision diagrams* [Min93], *Multi-valued decision diagrams* [Kam95], *Natural decision diagrams* [LR95], *Interval decision diagrams* [ST98], *Difference decision diagrams* [ML98], *Data decision diagrams* [CEPA+02].

In this chapter we introduce an extension of BDDs that we denote as *Interval decision diagrams* (IDDs). This extension was proposed in [LR95] and then, probably independently from [LR95], in [ST98]. IDDs can be viewed as a generalization of BDDs. IDDs are directed acyclic graphs with two types of nodes. Like BDDs, IDDs have two terminal nodes, labeled with 0 and 1. In contrast to BDDs, nonterminal nodes in IDDs have a variable number of outgoing arcs labeled with intervals. IDDs can represent *interval logic functions*, induced by expressions of the *interval logic*. This logic was defined in [LR95] to describe sets of marking of P/T nets. *Reduced ordered interval decision diagrams* (ROIDDs) are a canonical form representation for interval logic functions. For many interesting functions ROIDDs provide a compact representation. Furthermore, they allow to define efficient algorithms for manipulation of interval logic functions. We consider algorithms for ROIDDs and describe how they can be implemented efficiently.

[1] `http://citeseer.ist.psu.edu`

3.1 Interval Logic Functions

From now on we shall consider half-open intervals on $\mathbb{N}_0$ which have the form $[a, b) = \{x \in \mathbb{N}_0 \mid x \geq a \wedge x < b\}$. We denote a set of such intervals as $\mathcal{I}$. Note that the empty interval $\emptyset$ and intervals of the form $[a, \infty)$ are considered to belong to the set $\mathcal{I}$.

Definition 25 (Interval logic expressions)
Interval logic expressions consisting of symbols of variables $x_1, \ldots x_n$, the symbol $\in$, and elements of $\mathcal{I}$ are defined recursively.

1. $x_i \in I$ is an *atomic interval logic expression* if x_i is one of the symbols of the variables and I is some interval belonging to the set $\mathcal{I}$.

2. If F and G are interval logic expressions, then $(F \wedge G)$, $(F \vee G)$, and $\neg(F)$ are also interval logic expressions.

We introduce elements of $\mathbb{B}$ as abbreviations for $x_i \in \emptyset$ and $x_i \in [0, \infty)$. Expressions of the form $x_i \triangleleft c$ where $c \in \mathbb{N}_0$ and $\triangleleft \in \{=, \neq, >, <, \geq, \leq, \}$ are also seen as obvious abbreviations. For example, an expression $x_i = c$ abbreviates $x_i \in [c, c + 1)$, $x_i \leq c$ abbreviates $x_i \in [0, c + 1)$, etc.

Definition 26 (Interval logic functions)
Every interval logic expression G induces an *interval logic function* f_G

$$f_G : \mathbb{N}_0^n \to \mathbb{B}, \quad (e_1, \ldots, e_n) \mapsto f_G(e_1, \ldots e_n)$$

where $f_G(e_1, \ldots e_n)$ denotes an element of $\mathbb{B}$ got by replacing variables x_i with e_i followed by the evaluation of logic operations $\wedge, \vee$ and $\neg$.

Operations on interval logic functions are defined as follows:

1. $(f \vee g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \vee g(x_1, \ldots, x_n)$.

2. $(f \wedge g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \wedge g(x_1, \ldots, x_n)$.

3. $(\neg f)(x_1, \ldots, x_n) = \neg f(x_1, \ldots, x_n)$.

Definition 27 (Cofactor)
Let $f = f(x_1, \ldots, x_n)$ be an interval logic function. We denote a function $f|_{x_i=b} = f|_{x_i=b}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ as a *cofactor* of the function f with respect to a variable x_i and some natural number $b \in \mathbb{N}_0$ if

$$f|_{x_i=b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots x_n).$$

Definition 28 (Independence interval)
Let $f = f(x_1, \ldots, x_n)$ be an interval logic function and $I \in \mathcal{I}$ be some interval. I is called an *independence interval* of f with respect to a variable x_i if for all possible values of x_i in I the function f does not depend on x_i

$$f|_{x_i=b} = f|_{x_i=c} \; \forall b, c \in I.$$

We define then $f|_{x_i \in I} = f|_{x_i=b}$ for some $b \in I$.

Definition 29 (Independence interval partition)
Let $P = \{ I_1, \ldots, I_k \}$ be a set of intervals, $I_j \in \mathcal{I}$ for all j, let $f = f(x_1, \ldots, x_n)$ be an interval logic function, and x_i be a variable. The set P is called an *independence interval partition* of $\mathbb{N}_0$ if

1. All $I_1, \ldots, I_k$ are independence intervals of f with respect to x_i.
2. Elements of P are pairwise *disjoint*: $\forall j, m \;\; I_j \cap I_m = \emptyset$.
3. P is a *complete cover* of $\mathbb{N}_0$: $\bigcup\limits_{1 \le j \le k} I_j = \mathbb{N}_0$.

Based on independence interval partitions most of the interval logic functions of interest can be decomposed with respect to a variable x_i in several partial functions describable by cofactors. Each cofactor contributes to the function only in an independence interval with respect to x_i. From now on we consider only those interval logic functions that are decomposable over an interval partition with a finite number of independence intervals. Their partial functions can be composed using the *Bool-Shannon expansion*:

$$f = \bigvee_{1 \le j \le k} x_i \in I_j \wedge f|_{x_i \in I_j}$$

where the intervals $I_1, \ldots, I_k$ form an independence interval partition of $\mathbb{N}_0$ with respect to the variable x_i. As we shall discuss partitions of $\mathbb{N}_0$ only, from now on we shall simply write (independence) interval partition meaning (independence) interval partition of $\mathbb{N}_0$.

Definition 30 (Reduced interval partition)
Let $P = \{ I_1, \ldots, I_k \}$ be an independence interval partition for an interval logic function $f = f(x_1, \ldots, x_n)$ and some variable x_i. P is called *reduced* if

1. It contains no neighbored intervals that can be joined into an independence interval: $\nexists j : I_j \cup I_{j+1}$ is an independence interval of f with respect to x_i.
2. Higher bounds of all intervals build an increasing sequence with respect to indices of the intervals: $\forall j, m, I_j = [a_j, b_j), I_m = [a_m, b_m)$ from $j < m$ follows $b_j < b_m$.

Lemma 4 (Uniqueness of reduced independence interval partitions)
Let $P = \{ I_1, \ldots, I_k \}$ be a reduced independence interval partition for an interval logic function $f = f(x_1, \ldots, x_n)$ and some variable x_i. P is unique.

Proof: By contradiction. □

Example 7

1. a) $x_1 \geq 6 \wedge x_2 > 0 \vee x_3 \leq 8$ is an interval logic expression.
   b) $x_1 > x_2 + 5$ is not an interval logic expression.

2. $f(x_1, x_2, x_3) = x_1 \geq 6 \wedge x_2 > 0 \vee x_3 \leq 8$ is an interval logic function
   a) $f|_{x_1=7}(x_2, x_3) = \ x_2 > 0 \vee x_3 \leq 8$ is a cofactor of f with respect to the variable x_1 and the natural number 7.
   b) $f|_{x_1=2}(x_2, x_3) = x_3 \leq 8$ is a cofactor of f with respect to x_1 and 2.
   c) $[2, 5), [0, 3), [0, 6)$ and $[6, \infty)$ are independence intervals of f with respect to x_1.
   d) $[2, 10)$ and $[5, \infty)$ are not independence intervals of f with respect to x_1.
   e) $P = \{ [3, 5), [0, 3), [5, 6), [6, \infty) \}$ is an independence interval partition with respect to the variable x_1. P is not reduced.
   f) $P = \{ [0, 6), [6, \infty) \}$ is a reduced independence interval partition with respect to the variable x_1.

3.2 Reduced Ordered Interval Decision Diagrams

3.2.1 Interval Decision Diagrams

Definition 31 (Interval decision diagram)
An *interval decision diagram* (*IDD*) for variables $X = \{ x_1, \ldots, x_n \}$ is a tuple $[V, E, v_0]$ where:

1. V is a finite set of nodes.
2. $E \subseteq V \times \mathcal{I} \times V$ is finite set of arcs labeled with intervals on $\mathbb{N}_0$.
3. $[V, E]$ forms a DAG.
4. $v_0 \in V$ is a root of the IDD.

The following conditions must also hold:

1. V contains two *terminal nodes* which have no outgoing arcs. We define for these nodes a labeling function value : $V \to \mathbb{B}$, which labels one node with 0, another with 1.

2. All other nodes $v \in V$ are denoted as *nonterminal nodes*, we define for them a labeling function $\text{var} : V \to X$. Every nonterminal node v
   a) is labeled with a variable $\text{var}(v)$,
   b) has $v_k > 0$ outgoing arcs labeled with intervals $I_j \in \mathcal{I}$. The set $\{I_1, \ldots, I_{v_k}\}$ is an independence interval partition with respect to the variable $\text{var}(v)$.
3. On every path from the root to terminal nodes a variable may appear as label of a node only once.

Assuming that outgoing arcs of a node are ordered, we define also the following functions:

1. $\text{part}(v) = \{ I_1, \ldots, I_{v_k} \}$ returns all labels of the outgoing arcs of the node v.
2. $\text{part}_j(v) \in \mathcal{I},\ 1 \leq j \leq v_k$ returns the label of the j-th outgoing arc of v.
3. $\text{child}_j(v) \in V,\ 1 \leq j \leq v_k$ returns the node connected by the j-th outgoing arc of v.

An example of an IDD is shown in Fig. 3.1. Every IDD with a root v determines an interval logic function f_v in the following manner:

1. If v is a terminal node, then $f_v = \text{value}(v)$.
2. If v is a nonterminal node with $\text{var}(v) = x_i$, then f_v is the function

$$f = \bigvee_{1 \leq j \leq v_k} x_i \in \text{part}_j(v) \wedge f_{\text{child}_j(v)}.$$

$f_{\text{child}_j(v)}$ is either a constant if $\text{child}_j(v)$ is a terminal node, or an interval logic function determined by the IDD with a root $\text{child}_j(v)$.

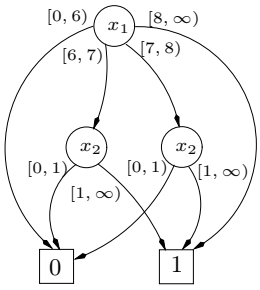


Figure 3.1: An IDD representing $f = (x_1 \geq 8) \vee (x_1 \in [6, 8) \wedge x_2 > 0)$

Every[2] interval logic function $f : \mathbb{N}_0^n \to \mathbb{B}$ can be represented by an IDD with help of the Bool-Shannon expansion. The decomposition must be applied recursively until leaves are reached. The Bool-Shannon decomposition for an interval logic function is demonstrated in Example 8.

Example 8 (Bool-Shannon decomposition)
Consider an interval logic function $f = (x_1 \geq 8) \vee (x_1 \in [6,8) \wedge x_2 > 0)$. First, we decompose f over the variable x_1.

1. $f|_{x_1 \in [0,6)}(x_2) = 0$,
2. $f|_{x_1 \in [6,8)}(x_2) = x_2 > 0$,
3. $f|_{x_1 \in [8,\infty)}(x_2) = 1$.

This means, the root of the IDD is labeled with x_1 and has three outgoing arcs labeled with intervals $[0,6)$, $[6,8)$ and $[8,\infty)$. The arcs labeled with $[0,6)$ and $[8,\infty)$ lead to the terminal nodes 0 and 1 respectively. The arc labeled with $[6,8)$ leads to an IDD representing the function $f|_{x_1 \in [6,8)}(x_2) = x_2 > 0$. We decompose now the function $f|_{x_1 \in [6,8)}$ over x_2.

1. $f|_{x_1 \in [6,8)}|_{x_2 \in [0,1)}() = 0$
2. $f|_{x_1 \in [6,8)}|_{x_2 \in [1,\infty)}() = 1$

The root of the IDD representing the function $f|_{x_1 \in [6,8)}$ is labeled with x_2 and has two outgoing arcs labeled with intervals $[0,1)$ and $[1,\infty)$ leading to the terminal nodes 0 and 1 respectively.
An IDD created during the decomposition of f is shown in Fig.3.2, right. Note that this IDD and the IDD in Fig.3.1 represent the same function.

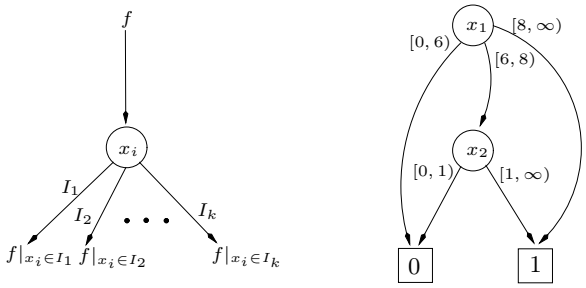


Figure 3.2: Bool-Shannon decomposition for IDDs

[2]Recall the assumption on page 31.

3.2.2 Reduced Ordered IDDs

Definition 32 (Ordered IDDs)
Let $B = [V, E, v_0]$ be an IDD. B is called *ordered* with respect to some variable ordering π if on every path from the root v_0 to terminal nodes all nodes are ordered with respect to their labels: for all non-terminal nodes v, v' if $(v, \_, v') \in E$ then $\mathrm{var}(v) <_\pi \mathrm{var}(v')$.

Definition 33 (Isomorphic IDDs)
Let $B = [V_B, E_B, v_{0_B}]$ and $F = [V_F, E_F, v_{0_F}]$ be two IDDs. B and F are called *isomorphic* if there exists a bijective function $\sigma : V_B \to V_F$, such that if $\sigma(v) = v'$ then

1. either v and v' are both terminal nodes and $\mathrm{value}(v) = \mathrm{value}(v')$,
2. or $\mathrm{var}(v) = \mathrm{var}(v')$ and $\forall j\ \mathrm{part}_j(v) = \mathrm{part}_j(v')$ and $\sigma(\mathrm{child}_j(v)) = \mathrm{child}_j(v')$.

Definition 34 (Reduced IDD)
Let $B = [V, E, v_0]$ be an IDD. B is called *reduced* if

1. The independence interval partition $\mathrm{part}(v)$ of each non-terminal node $v \in V$ is reduced.
2. Each non-terminal node $v \in V$ has at least two different children.
3. There exist no different nodes $v, v' \in V$ such that the subgraphs rooted by v and v' are isomorphic.

An example of a reduced ordered IDD (*ROIDD*) is shown in Fig. 3.2, right.

3.2.3 Canonicity of Reduced Ordered IDDs

Theorem 1 (Canonicity of reduced ordered IDDs)
If some variable ordering π is defined, then for every[3] interval logic function $f(x_1, \ldots, x_n)$ there exists a unique reduced ordered with respect to π IDD, representing this function f.

Proof: The proof resembles the ones in [Bry86, ST98] and is done by induction on the size of the dependence set D_f. The *dependence set* of f is the set of arguments that f depends upon:

$$D_f = \{\, x_i \mid \exists b, c \in \mathbb{N}_0 : f|_{x_i=b} \neq f|_{x_i=c} \,\}.$$

The following statements are assumed without explanations:

1. The path from the root of an ROIDD to a terminal node is unique for given variable values.

[3] Recall the assumption on page 31.

2. Each node in an ROIDD is reachable from the root of the ROIDD.

3. If ROIDD G is reduced, then any subgraph of G is also reduced.

If $|D_f| = 0$, then f must be one of the two constant functions $f = 0$ or $f = 1$. Let G be the ROIDD representing $f = 0$. This ROIDD can not contain terminal nodes labeled with 1, otherwise there would be some variables for which the function evaluates to 1, since all nodes in an ROIDD are reachable from the root. Suppose now that G contains at least one nonterminal node v. This node v must have then only one outgoing arc labeled with $\mathbb{N}_0$, as f does not depend on the variable $\text{var}(v)$. Therefore, v would have just one child, which contradicts to that G is reduced. Hence, the only ROIDD representing the function $f = 0$ consists of a single terminal node labeled with 0. Similarly, the only ROIDD representing the function $f = 1$ consists of a single terminal node labeled with 1.

Suppose, the statement of the theorem holds for any function g having $|D_g| < k$, we prove now that the statement of the theorem holds for a function f with $|D_f| = k$. Let G be an ROIDD representing f, and let $x_i \in D_f$ be the smallest with respect to π variable in D_f. The root v of G must be labeled with x_i. If the root v had $\text{var}(v) <_\pi x_i$, then it would have exactly one outgoing arc labeled with $\mathbb{N}_0$, as x_i is chosen so that f does not depend on any variables $x_j : x_j <_\pi x_i$. Therefore, v would have just one child, which contradicts to that G is reduced. If the node v had $\text{var}(v) >_\pi x_i$, then x_i would not be the element of D_f.

All functions g represented by ROIDDs rooted by nodes w with $\text{var}(w) >_\pi x_i$ satisfy $|D_g| < k$ and are different, otherwise the subgraphs of two equal functions would be isomorphic due to the assumption of the induction. In particular, all children of the root v of G represent different functions and are not isomorphic. Consider now the Bool-Shannon expansion of f with respect to x_i. The independence interval partition of the node v is unique. Consequently, the number of outgoing arcs and their labels are unique. Moreover, the arc with the label I_j leads to the node $\text{child}_j(v)$ which is unique, otherwise there would exist two identical functions represented by ROIDDs rooted by nodes with $\text{var}(v) >_\pi x_i$. □

3.2.4 Variable Ordering

Interval decision diagrams can be viewed as a generalization of binary decision diagrams, hence the same variable ordering issues hold for ROIDDs.

1. Variable ordering can have a great impact on the size of an ROIDD.

2. In general, finding an optimal ordering is infeasible, even checking if a particular ordering is optimal is NP-complete.

3. There exist interval logic functions that have ROIDD representations of exponential size for any variable ordering.

4. The heuristic stating that related variables should be close together in the ordering brings often good results.

Example 9
Let $f = f(a_1, \ldots, a_n, b_1, \ldots, b_n)$ be interval logic function defined as

$$f = \bigwedge_{1 \leq i \leq n} ((a_i = 0 \wedge b_i = 0) \vee (a_i > 0 \wedge b_i > 0)).$$

The number of nodes in the ROIDD representing f will be

- $3n + 2$ if we use the variable ordering π_1 defined as

$$a_1 <_{\pi_1} b_1 <_{\pi_1} a_2 <_{\pi_1} b_2 <_{\pi_1} \ldots <_{\pi_1} a_n <_{\pi_1} b_n.$$

- $3 \cdot 2^n - 1$ if we use the variable ordering π_2 defined as

$$a_1 <_{\pi_2} a_2 <_{\pi_2} \ldots <_{\pi_2} a_n <_{\pi_2} b_1 <_{\pi_2} b_2 <_{\pi_2} \ldots <_{\pi_2} b_n.$$

Consider the case when $n = 2$. Two ROIDDs representing f are shown in Fig. 3.3. The variable ordering π_1 is used for the left ROIDD, π_2 for the right.

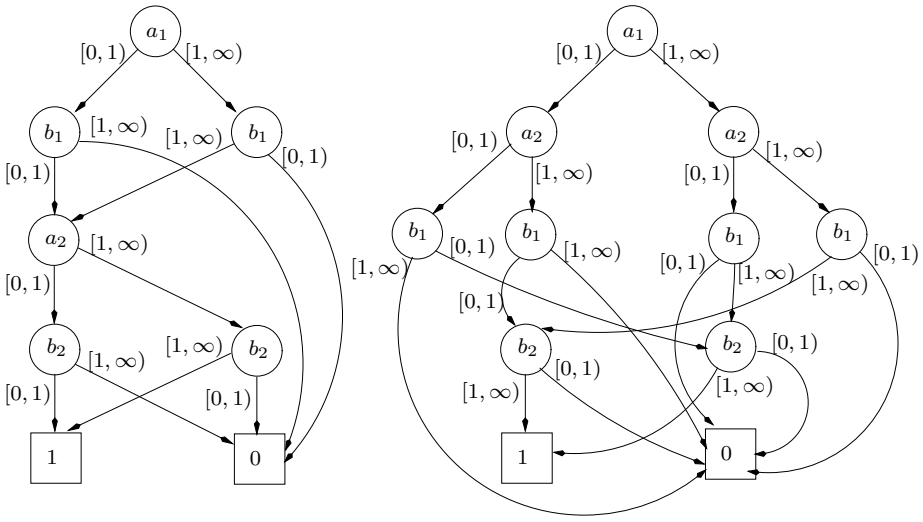


Figure 3.3: ROIDDs with different variable orderings

3.2.5 Shared ROIDDs

Before considering operations on ROIDDs we introduce first the following extension of ROIDDs: a single multirooted DAG is used to represent a collection of interval logic functions. All functions in the collection must be defined over the same set of variables. Additionally, the same variable ordering must be used for these functions.

Definition 35 (Shared ROIDD)
A *Shared ROIDD* is a tuple $[V, E, X, \pi]$ where:

1. V is a finite set of nodes.
2. $E \subseteq V \times \mathcal{I} \times V$ is finite set of arcs labeled with intervals on $\mathbb{N}_0$.
3. $[V, E]$ forms a DAG.
4. X is a set of variables.
5. π is a variable ordering.
6. Every node $v \in V$ is a root of some reduced ordered with respect to π IDD.
7. There exist no different nodes $v, v' \in V$ such that the ROIDDs rooted by v and v' are isomorphic.

Because of the canonicity of ROIDDs two functions in the collection are identical if and only if ROIDDs representing these functions have the same root in the Shared ROIDD. The idea of the similar extension for boolean functions and ROBDDs was first introduced in [BRB90].

Example 10
An example of a Shared ROIDD is shown in Fig.3.4. The following interval logic functions are encoded in this Shared ROIDD:

- $f_0 = 0$,
- $f_1 = 1$,
- $f_2 = x_2 > 0$,
- $f_3 = (x_1 \in [6, 8) \wedge x_2 > 0) \vee (x_1 \geq 8)$,
- $f_4 = (x_1 = 0 \wedge x_2 > 0) \vee x_1 \geq 1$.

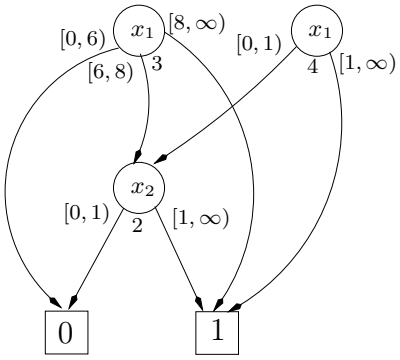


Figure 3.4: A Shared ROIDD

We shall use the Shared ROIDDs in all algorithms for operations on ROIDDs. Notice that all nodes of the Shared ROIDD in Fig. 3.4 are enumerated (the numbers under nonterminal nodes). The terminal nodes get respectively the numbers 0 and 1. We shall use these numbers to address nodes. For simplification of the algorithms we assume that the function var labels terminal nodes with a special variable x such that $x >_\pi \mathrm{var}(v)$ for all nonterminal nodes $v \in V$.

A supplementary function MakeNode is used to insert a new node into the Shared ROIDD R. It takes care that the inserted IDDs are reduced and that no isomorphic nodes are added to R, compare Definitions 34 and 35. The function gets as parameters a variable x, an independence interval partition $P = \{ I_1, \ldots, I_k \}$, and a list of children $C = (c_1, \ldots, c_k)$.

1. First, MakeNode reduces the independence interval partition P, uniting neighboring intervals if the neighboring children are equal. If the reduced interval partition consists of one interval, then no new node should be created, as it would be redundant. The function simply returns the only child left in C.

2. Second, MakeNode uses the hash table *UniqueTable* to check if a node represented by a tuple (x, P, C) already exists in the Shared ROIDD. $\mathit{UniqueTable}[x, P, C]$ is negative if the node does not exist, otherwise it contains the number of the node. If the node is found in the hash table, it is returned, otherwise a new ROIDD node is added to R. The variable *nodesInShIDD* counts the number of nodes in the Shared ROIDD. Notice that *nodesInShIDD* is initialized with 2, as the values 0 and 1 are reserved for the terminal nodes.

Algorithm 2 (MakeNode)

```
1  nodesInShIDD := 2
2  func MakeNode (x, P = {I_1, ..., I_k}, C = (c_1, ..., c_k))
3     while ∃ c_j, c_{j+1} ∈ C such that c_j = c_{j+1} do
4        C:= C \ c_{j+1}
5        I_j := I_j ∪ I_{j+1}
6        P:= P \ I_{j+1}
7     od
8     if |P| = 1 then return c_1 fi
9     res := UniqueTable[x, P, C]
10    if res ≥ 0 then return res fi
11    nodesInShIDD := nodesInShIDD + 1
12    UniqueTable[x, P, C] := nodesInShIDD
13    return nodesInShIDD
14 end
```

3.3 Operations on ROIDDs

Algorithms for ROIDDs, being of course a bit more complicated, resemble closely the algorithms for ROBDDs discussed in section A.5 of the Appendix.

3.3.1 Equivalence Check

Let f and g be two interval logic functions over the same set of variables, and let F and G be ROIDD representations of f and g. The equivalence check of these functions becomes a trivial operation if F and G are saved in one Shared ROIDD. It is enough then to check if F and G have the same root. Obviously, this operation can be done in a constant time.

3.3.2 Apply Operation

Consider Algorithm 3 which is a uniform algorithm for computing all binary logical operations on interval logic functions. The algorithm resembles the Apply [Bry86] algorithm for ROBDDs discussed on page 163.

Let $\star$ be an arbitrary two-argument logical operation, f and g be two interval logic functions over the same set of variables, F and G be ROIDDs representing f and g. We assume that F and G are saved in a Shared ROIDD R. The algorithm calculating $f \star g$ is implemented with help of a recursive function AuxApply which gets roots r_1, r_2 of two ROIDDs as parameters. We denote with f_1 and f_2 interval logic functions represented by ROIDDs rooted by r_1 and r_2. AuxApply(r_1, r_2) returns a root of an

ROIDD representing $f_1 \star f_2$. Several cases depending on the relationship between r_1 and r_2 are possible.

1. If r_1 and r_2 are both terminal nodes, then $f_1 \star f_2 = \text{value}(r_1) \star \text{value}(r_2)$.

2. If $\text{var}(r_1) = \text{var}(r_2)$, then the Bool-Shannon expansion is used to break the problem into subproblems that can be solved then recursively. The function IntersectPartitions gets two reduced independence interval partitions $\text{part}(r_1)$ and $\text{part}(r_2)$ as parameters and returns a new independence interval partition *NewPart* got by intersecting the intervals of $\text{part}(r_1)$ and $\text{part}(r_2)$. Obviously, *NewPart* is an independence interval partition of both f_1 and f_2, the number of intervals $|NewPart|$ is maximally $|\text{part}(r_1)| + |\text{part}(r_2)|$. We can apply the Bool-Shannon expansion and get $|NewPart|$ subproblems:

$$f_1 \star f_2 = \bigvee_{1 \le j \le |NewPart|} x_i \in NewPart_j \wedge (f_1|_{x_i \in NewPart_j} \star f_2|_{x_i \in NewPart_j}).$$

   The root of the resulting IDD will be a node w with $\text{var}(w) = \text{var}(r_1)$, outgoing arcs labeled with intervals I_j of *NewPart* leading to ROIDDs representing functions $f_1|_{x_i \in I_j} \star f_2|_{x_i \in I_j}$. The function MakeNode is used to insert the IDD into R.

3. If $\text{var}(r_1) < \text{var}(r_2)$, then f_2 does not depend on x. In this case the Bool-Shannon expansion simplifies to

$$f_1 \star f_2 = \bigvee_{1 \le j \le |\,\text{part}(r_1)|} x_i \in \text{part}_j(r_1) \wedge (f_1|_{x_i \in \text{part}_j(r_1)} \star f_2)$$

   and the IDD for $f_1 \star f_2$ is computed recursively as in the second case.

4. If $\text{var}(r_1) > \text{var}(r_2)$, then the computation is similar to the previous case.

Each problem of AuxApply can generate $|\,\text{part}(r_1)| + |\,\text{part}(r_2)|$ subproblems, so care must be taken to prevent the algorithm from being exponential. Each subproblem corresponds to a pair of ROIDDs that are subgraphs of the F and G. The number of subgraphs in an ROIDD is limited by its size, hence, the number of subproblems is limited by the product of the size of F and G. A hash table *ResultTable* is used to store the results of previously computed subproblems, the function AuxApply is a so-called *memory function*. Before any recursive calls are made, the *ResultTable* is used to check if the subproblem has been already solved. $ResultTable[r_1, r_2]$ is negative if the result of the operation $\star$ for the subgraphs rooted by r_1 and r_2 is not known yet, nonnegative otherwise. Usage of the memory function allows to keep the algorithm polynomial.

Algorithm 3 (Binary Operation on IDDs)

```
1  func Apply (⋆, F, G)
2     func AuxApply (r1, r2)
3        if r1 ∈ {0,1} ∧ r2 ∈ {0,1} then return r1 ⋆ r2 fi
4        if ResultTable[r1, r2] ≥ 0 then return ResultTable[r1, r2] fi
5        if var(r1) = var(r2) then
6           NewPart := IntersectPartitions(part(r1), part(r2))
7           forall Ij ∈ NewPart, Ik ∈ part(r1), Il ∈ part(r2) do
8              if Ij ∩ Ik ∩ Il ≠ ∅ then
9                 NewChildj := AuxApply(childk(r1), childl(r2))
10             fi
11          od
12          res := MakeNode(var(r1), NewPart, NewChild)
13       elseif var(r1) < var(r2) then
14          NewPart := part(r1)
15          forall Ij ∈ NewPart do
16             NewChildj := AuxApply(childj(r1), r2)
17          od
18          res := MakeNode(var(r1), NewPart, NewChild)
19       else  /* var(r1) > var(r2) */
20          NewPart := part(r2)
21          forall Ij ∈ NewPart do
22             NewChildj := AuxApply(childj(r2), r1)
23          od
24          res := MakeNode(var(r2), NewPart, NewChild)
25       fi
26       ResultTable[r1, r2] := res
27       return res
28    end
29
30    begin
31       B.root := AuxApply(F.root, G.root)
32       return B
33    end
```

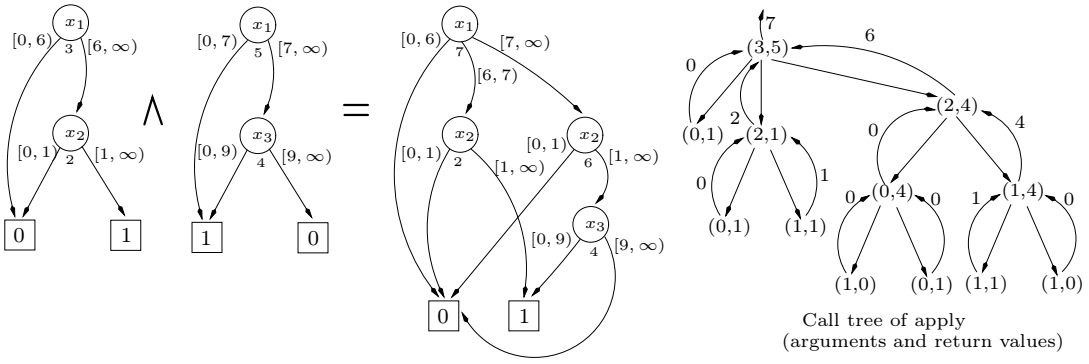


Figure 3.5: Calculation of $f_1 \wedge f_2$ using Apply

Example 11 (Apply)
Fig.3.5 demonstrates the calculation of $f_1 \wedge f_2$ using Apply for $f_1 = x_1 \geq 6 \wedge x_2 > 0$ and $f_2 = x_1 < 7 \vee x_3 < 9$. ROIDDs for f_1, f_2, and $f_1 \wedge f_2$ are shown on the left, the call tree of the function AuxApply is on the right. Notice that we use Shared ROIDDs to store all the ROIDDs and that the resulting ROIDD reuses nodes 2 and 4.

3.3.3 Negation

Let g be an interval logic function, G be an ROIDD representing g, and let G be saved in a Shared ROIDD. Algorithm 4 calculating $\neg g$ is implemented with help of a recursive function AuxNeg which gets a root r of an ROIDD as a parameter. Let f be an interval logic function represented by the ROIDD rooted by r. AuxNeg(r) returns a root of an ROIDD representing the function $\neg f$. Two cases are possible.

1. If r is a terminal node, then $\neg f = \neg\, \mathrm{value}(r)$.

2. If r is a nonterminal node, then the Bool-Shannon expansion

$$\neg f = \bigvee_{1 \leq j \leq |\,\mathrm{part}(r)|} \mathrm{var}(r) \in \mathrm{part}(r)_j \wedge (\neg f|_{\mathrm{var}(r) \in \mathrm{part}(r)_j})$$

   is used to break the problem into $|\,\mathrm{part}(r)|$ subproblems that are solved then recursively. The root of the resulting ROIDD will be a node w with $\mathrm{var}(w) = \mathrm{var}(r)$, outgoing arcs labeled with intervals I_j of $\mathrm{part}(r)$ leading to ROIDDs representing functions $\neg f|_{\mathrm{var}(r) \in I_j}$.

Though each problem of the function AuxNeg can generate $|\,\mathrm{part}(r)|$ subproblems, the total number of subproblems is limited by the size of G. AuxNeg is implemented like

Algorithm 4 (Negation)

```
1  func Neg (G)
2     func AuxNeg (r)
3        if r ∈ { 0, 1 } then return ¬r fi
4        if ResultTable[r] ≥ 0 then return ResultTable[r] fi
5        NewPart := part(r)
6        forall I_j ∈ NewPart do
7           NewChild_j := AuxNeg(child_j(r))
8        od
9        res := MakeNode(var(r), NewPart, NewChild)
10       ResultTable[r] := res
11       return res
12    end
13    begin
14       G'.root := AuxNeg(G.root)
15       return G'
16    end
```

AuxApply as a memory function, this allows to keep the algorithm linear in the size of G. Actually, a call to $\mathrm{Neg}(G)$ returns an ROIDD G' which differs from G only by interchanged terminal nodes.

3.3.4 Cofactors

Let g be an interval logic function, $x \in X$ be a variable, $c \in \mathbb{N}_0$ be some natural number, and G be an ROIDD representing g. We assume that G is saved in a Shared ROIDD. Algorithm 5 calculating $g|_{x=c}$ is implemented with help of a recursive function AuxCofactor which gets a root r of an ROIDD as a parameter. Let f be an interval logic functions represented by the ROIDD rooted by r. AuxCofactor(r) returns a root of an ROIDD representing the function $f|_{x=c}$. Three cases depending on the relationship between $\mathrm{var}(r)$ and x are possible.

1. If $\mathrm{var}(r) < x$, then the Bool-Shannon expansion

$$f|_{x=c} = \bigvee_{1 \le j \le |\,\mathrm{part}(r)|} \mathrm{var}(r) \in \mathrm{part}(r)_j \wedge (f|_{\mathrm{var}(r) \in \mathrm{part}(r)_j}|_{x=c})$$

   is used to break the problem into $|\,\mathrm{part}(r)|$ subproblems that are solved then recursively. The root of the resulting IDD will be a node w with $\mathrm{var}(w) = \mathrm{var}(r)$, outgoing arcs labeled with intervals I_j of $\mathrm{part}(r)$ leading to ROIDDs representing function $f|_{\mathrm{var}(r) \in I_j}|_{x=c}$.

Algorithm 5 (Cofactors)

```
1  func Cofactor (G, x, val)
2     func AuxCofactor (r)
3        if var(r) < x then
4           if ResultTable[r] ≥ 0 then return ResultTable[r] fi
5           NewPart := part(r)
6           forall I_j ∈ NewPart do
7              NewChild_j := AuxCofactor(child_j(r), x, val)
8           od
9           res := MakeNode(var(r), NewPart, NewChild)
10          ResultTable[r] := res
11          return res
12       elseif var(r) = x then
13          forall I_j ∈ part(r) do
14             if val ∈ I_j then return child_j(r) fi
15          od
16       else  /* var(r) > x */
17          return r
18       fi
19    end
20    begin
21       G'.root := AuxCofactor(G.root)
22       return G'
23    end
```

2. If $\mathrm{var}(r) = x$, then $f|_{x=c} = f|_{\mathrm{var}(r) \in \mathrm{part}(r)_j}$ if $c \in \mathrm{part}(r)_j$.

3. If $\mathrm{var}(r) > x$, then f does not depend on x and $f|_{x=c} = f$. Note that this case includes also the terminal nodes.

Though each problem of the function AuxCofactor can generate $|\,\mathrm{part}(r)|$ subproblems, the total number of subproblems is limited by the size of G. Again, implementation of AuxCofactor as a memory function allows to keep the algorithm linear in the size of G.

3.3.5 Construction of ROIDDs

Consider a function Construct that takes an interval logic function f as an argument and returns an ROIDD that represents f. We define this function inductively.

1. If f is induced by an atomic interval logic expression, then Construct(f) returns one of the elementary ROIDDs shown in Fig. 3.6.

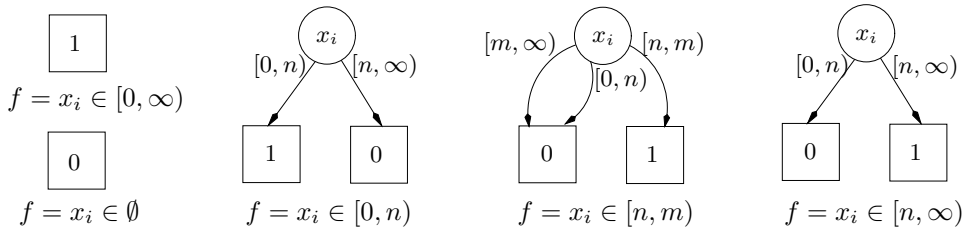


Figure 3.6: Elementary ROIDDs

2. If $f = f_1 \star f_2$ where $\star \in \{ \wedge, \vee \}$, then

$$\text{Construct}(f) = \text{Apply}(\star, \text{Construct}(f_1), \text{Construct}(f_2)).$$

3. If $f = \neg f_1$, then $\text{Construct}(f) = \text{Neg}(\text{Construct}(f_1))$.

3.4 Efficient Implementation of an ROIDD Package

We have implemented the ROIDD package in C++. The implementation has been tested on Linux, SUN and lately on Windows (with Cygwin). We have applied the approved techniques [BRB90, YBO+98, Noa99] used in the implementations of ROBDD packages and considered also the ROIDD specific problems. In this section we discuss the techniques briefly.

3.4.1 Shared ROIDDs and Garbage Collection

Recall that a global hash table *UniqueTable* used in the function MakeNode allows a node to be found in a Shared ROIDD. We merge data structures of the hash table and ROIDDs forming a Shared ROIDD.

```
1  struct SharedROIDD
2      nodes : array of Node
3      uniqueTable : array of unsigned
4      extRefs: list of RefCount
5      firstFree: unsigned
6  end
```

```
1  struct Node
2      var : unsigned
3      children: list of unsigned
4      arcs: list of unsigned
5      nextFree : unsigned
6      nextInUTable : unsigned
7      mark : bool
8  end
```

All ROIDD nodes are saved in the array *nodes*. For every ROIDD node v we store an index of a variable $\text{var}(v)$, a list of children, and a list of labels of outgoing arcs. We shall discuss problems of storing these lists later in this section.

firstFree is an index of the first free ROIDD node, free nodes are linked using the member *nextFree*. A hash function is used to compute a key k for the tuple $[x, P, C]$ when a node represented by this tuple must be found in the Shared ROIDD. $uniqueTable[k]$ contains an index of the first ROIDD node with the hash value k, all ROIDD nodes with the same hash key are linked using the member *nextInUTable*.

We denote ROIDDs representing functions of a user of the package as *externally referenced*. The member *extRefs* keeps count of references on the roots of externally referenced ROIDDs. The *garbage collection* is triggered automatically if all free nodes are exhausted during an ROIDD operation. First, garbage collection marks all nodes reachable from the roots of externally referenced ROIDDs using a simple recursive function and the member *mark*. Second, all not marked nodes are linked in the free list and the ROIDD operation is repeated.

3.4.2 Cache Management and Special Operations

Recall that hash tables *ResultTable* storing the results of already calculated subproblems were used to prevent the algorithms on ROIDDs from being exponential. Because of the usage of Shared ROIDDs the results saved in the hash tables remain valid even across top-level calls to ROIDD operations. Hence, the hash tables can be initialized only once at the initialization of the package and after garbage collections, but not at each call to an ROIDD operation.

It was proposed in [BRB90] to implement *ResultTable* as a *hash-based cache*. In hash-based caches a newer entry overwrites the old one when a collision occurs. Obviously, a hash-based cache requires less memory and is faster, as no collision management is needed. Each hash-based cache entry stores a key and a result of the operation. For example, in the function AuxApply indices of roots r_1, r_2 and an index of the operation $\star$ form a key and an index of *res* is the result. The usage of hash-based caches introduces the possibility of recalculating previous results and can lead in the worst case to the exponential complexity of ROIDD operations. A gain both in terms of average memory usage and time when appropriate hash functions and sizes of caches are used allows to warrant such a risk.

To improve efficiency we do not use the function Apply for the most often used logic operations like disjunction, conjunction, etc. Instead, they are implemented as dedicated ROIDDs operations. Consider, for example, the disjunction, we use the following facts in the implementation of the function Disjunction

1. $f_1 \vee f_2 = f_2 \vee f_1$.
2. $f \vee 1 = 1$, $f \vee 0 = f$, $f \vee f = f$.
3. If $f = f_1 \vee f_2$, then $f \vee f_1 = f$ and $f \vee f_2 = f$.

Hence, the line *3* of Algorithm 3 is replaced in the Disjunction with

- if $r_1 > r_2$ then swap(r_1, r_2) fi
- if $r_1 = 0$ then return r_2 fi
- if $r_1 = 1$ then return 1 fi
- if $r_1 = r_2$ then return r_1 fi

and the line *26* is replaced with

- *ResultTable*[r_1, r_2] := *res*
- if *res* < r_1 then *ResultTable*[*res*, r_1] := *res* else *ResultTable*[r_1, *res*] := *res* fi
- if *res* < r_2 then *ResultTable*[*res*, r_2] := *res* else *ResultTable*[r_2, *res*] := *res* fi

3.4.3 Complement Arcs and Dynamic Variable Ordering

Complement arcs and *dynamic variable ordering* are two features *not* implemented in our ROIDD package.

Recall that ROIDDs F and F' representing interval logic functions f and $\neg f$ differ only by interchanged terminal nodes. This fact could be exploited by using *complement arcs*. A complement arc is an ordinary arc with an extra bit (complement bit), set to indicate that the connected node is to be interpreted as the complement of the ordinary node. Hence, $\neg f$ can be represented by only a compliment arc to F, avoiding creation of all nodes of F'. Nevertheless, the cost of the implementation and management of complement arcs is not justified when the package aims to symbolic analysis of bounded Petri nets. The negation $\neg f$ is used quite seldom there. It is usually replaced with a calculation of $g \wedge \neg f$, which corresponds to a set difference operation, as we shall see in the next chapter. Instead of complement arcs, we implement this operation as an effective dedicated ROIDD operation, like the Disjunction, discussed above.

Recall that the size of ROIDDs can depend critically on the variable ordering. Dynamic variable ordering algorithms [Rud93] try to adjust the ordering at running time. As we have mentioned, there exist no efficient algorithms to check if a variable ordering is optimal, so dynamic variable ordering techniques are implemented using quite expensive brute-force algorithms. During the garbage collection the package attempts to reduce the storage requirement and reorders the variables by performing a series of swaps between adjacent variables. As noticed in [YBO+98], if a good variable ordering is defined using heuristics, then dynamic variable ordering can not improve the analysis times. As we shall see, the structure of Petri nets can be used to define quite good variable orderings. Furthermore, usage of dynamic variable ordering would complicate the algorithms exploiting the locality of Petri nets discussed in the next chapter, so we forbear from the implementation of dynamic ordering in our package.

3.4.4 Handling of Partitions and Lists of Children

Most of the techniques described above were successfully applied in implementations of ROBDD packages [BRB90, Noa99], handling of intervals and lists of children is an ROIDD specific issue.

Obviously, every partition $P = \{ [0, a_1), [a_1, a_2), \ldots, [a_n, \infty) \}$ of $\mathbb{N}_0$ can be uniquely represented by the sequence of natural numbers $a_1, a_2, \ldots, a_n$. Hence, we can store a list of children of an ROIDD node and labels of its outgoing arcs using lists of unsigned integers. The list implementation must be, of course, memory efficient, as well as allow fast operations needed by ROIDD algorithms. The most often used operations are:

- creation and deletion of lists,
- copying and comparison of lists,
- appending and removing list elements,
- functions like IntersectPartitions.

All ROIDD algorithms can be written without loss of efficiency in such a way that they access elements of lists sequentially. We can exploit this fact and use *singly linked* lists. The data structure we use is sketched in the listing.

```
1  struct ListNode
2      data: unsigned
3      next: reference to ListNode
4  end
```

```
1  struct UList
2      first: reference to ListNode
3      last: reference to ListNode
4  end
```

Lists storing labels of outgoing arcs of a node are always sorted. Hence, the function IntersectPartitions can be implemented as a simple merge operation of two sorted lists.

References to the first and last element of a list are stored in the *next* field of nodes referenced by members *first* and *last*. This allows fast appending of elements at the beginning and at the end of the list. Moreover, $last.data$ is used to store the length of the list, the $first.data$ is used for the *reference counting*. We implement a *lazy copy* and *shared list nodes*. If a copy operation is used to create a list, then elements of the old list are not copied to the new one, instead, the new list shares all its nodes with the old list.

The operation comparing list l_1 and l_2 is implemented as follows:

1. It checks first if lists share the same elements: $l_1.first = l_2.first$.
2. If the lists do not share the same elements, then their lengths are compared: $l_1.last.data = l_2.last.data$.

3. If the lengths as well as all elements of the lists are equal, then a list with with a smaller reference count is modified. Suppose it is the list l_1.
   a) If l_1 does not share elements with other lists ($l_1.\mathit{first}.\mathit{data} = 1$), then all its elements are freed. Otherwise, its reference count is decremented.
   b) The references *first* and *last* of l_1 are adjusted to make l_1 and l_2 share nodes and the reference count is incremented.

Frequent allocations and deletions of small objects like list nodes lead to high memory fragmentation, its inefficient usage and, in the long run, to very high memory requirements. To avoid the problem, we use a *pool* of list nodes. The pool allocates large chunks of memory and undertakes the management of (free) nodes. When a list operation needs a new node, it requests it from the pool. Not needed nodes are returned back into the pool.

Usage of shared lists and the pool of nodes allowed to decrease the memory requirements of the package up to the order of magnitude on some examples. We leave a search for even more compact and efficient data structures for the storage of partitions and lists of children as a possible topic for future research.

3.5 Closing Remark

We have considered Interval decision diagrams (IDDs), a generalization of Binary decision diagrams. IDDs can represent functions induced by expressions of the interval logic, which was actually introduced to describe sets of markings of Petri nets. With the definition of reduced ordered interval decision diagrams (ROIDDs) we have got a canonical representation of interval logic functions, which is also compact for many interesting functions. We have discussed how effective algorithms for ROIDDs can be defined and implemented. In the next chapter we consider how ROIDDs can be used for compact representation and manipulations of large sets of markings of bounded Petri nets.

4 Symbolic Analysis of Bounded Petri Nets Using ROIDDs

The advent of *symbolic model checking* [BCM+90, McM92] has revolutionized the field of formal verification, transforming it from a purely academic discipline into a industrially applied technique. The key idea underlying symbolic methods is to represent sets of states using their *characteristic functions* and to manipulate them as if they were in bulk. Such an approach becomes less dependent on the number of states in a set and can be even applied to infinite systems. Symbolic methods derive their efficiency from the fact that in many cases of interest large sets of states can be represented concisely by characteristic functions. Typical operations used in symbolic algorithms are computing the set of all successors or all predecessors of states in some set and usual set operations like intersection, union, test for emptiness. Traditional symbolic algorithms are based on manipulations of boolean functions and ROBDDs [Bry86].

ROBDDs have been applied first to the analysis of Petri nets in [PRCB94]. Boolean functions encode naturally sets of markings of 1-bounded nets. Zero suppressed decision diagrams [Min93] are perfectly suited for the symbolic analysis of such nets. When boolean functions must be used to represent sets of markings of k-bounded nets, a set of boolean variables must be assigned to every place of the net using either *one-hot*, *binary* [PRCB94] or *dense encoding* [PCP99]. This leads often to a number of problems: to save memory and computing power, the coding should be selected so, that it covers no more than a necessary integer range, which, in general, can be not known in advance or can actually be the goal of the analysis. The number of ROBDD variables, which is a critical parameter in efficiency of ROBDD algorithms, can grow very fast. Typical symbolic operations like computation of successors states become unnatural and expensive. To avoid these problems a number of alternative approaches have been suggested.

Multi-valued decision diagrams (MDDs) [Kam95] are used for the analysis of Petri nets in the tool `SMART` [CJMS01]. The approach can be coarsely explained as follows. A net has to be partitioned into a number of subnets. Local state spaces are enumerated using explicit techniques. Every local marking is assigned to a natural number, a global marking corresponds to a tuple of natural numbers. MDDs are used then to encode sets of global markings, *Kronecker operators* are used to encode transitions. An efficient *saturation* algorithm [CLS01, CMS03] that exploits the locality of Petri nets is used for the computation of the global state space, computations of local states spaces can be

done "on-the-fly". The approach aims primary at the analysis of large-sized nets that exhibit a highly asynchronous behavior and can be easily partitioned into a number of loosely coupled subnets having small local state spaces. Most of the nets we face, especially those, modeling biochemical systems [HK04, KJH05], are relatively small-sized k-bounded nets which usually do not allow such a partitioning.

Arithmetic constraints have been proposed as a symbolic representation in [BGP97, BF99, Bul00]. The approaches aim primary at the analysis of infinite systems. The presented experimental results do not look very promising for the general analysis of Petri nets. The complexity of the underlying algorithms is very high, presence of variables with finite domains results often in the very inefficient constraint-based representation.

Interval logic functions [LR95] introduced in the previous chapter allow a natural encoding of sets of markings of k-bounded nets. ROIDDs provide a compact representation of interval logic functions, moreover, they allow a natural and efficient implementation of the special operations needed in symbolic algorithms.

Though small decision diagrams can encode large sets of states, not every large set of states can be encoded by a small decision diagram. The breath-first order strategy traditionally applied in symbolic algorithms is not well suited for asynchronous systems. Often, sizes of decision diagrams encoding working sets of symbolic algorithms are much larger during the computation than upon termination. As the efficiency of the operations on decision diagrams depends on their sizes, the performance decreases as big diagrams start to be generated. Straying from the breath-first strategy in the exploration of state spaces can improve the reachability analysis. In this chapter we study techniques to reduce sizes and the number of intermediate diagrams. We propose then a new saturation algorithm which exploits both the structure of k-bounded Petri nets and the structure of ROIDDs. Sizes of decision diagrams depend heavily on the used variable ordering. We discuss heuristics to get a good variable ordering using the structural information of a Petri net.

Decomposing a graph into its strongly connected components (SCCs) has many applications in the analysis of different properties. The first symbolic algorithms for SCC decomposition were based on the computation of the transitive closure (TC) of the transition relation [MMB93]. This operation is often very expensive and algorithms based on the reachability analysis [XB98, XB99, BGS00] were shown to be superior over the TC-algorithms. They are based on the observation that an SCC containing some state can be computed as an intersection of a set of states that can reach this state with a set of states that are reachable from it. An algorithm introduced in [XB98] for the state classification of finite-state Markov chains can be adapted to the enumeration of terminal SCCs in sets of markings of Petri nets. Moreover, it benefits immediately from the new saturation strategy.

An efficient implementation of the reachability analysis allows an efficient check of the basic Petri net properties.

4.1 Fundamental Isomorphism

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs, and let n be the number of places of N. With M_N we denote a set containing all possible markings of N. Recall that the interval logic was introduced to describe sets of markings. Let $f = f(x_1, \ldots, x_n)$ be some interval logic function. A set of markings M described by the function f is defined as follows:

$$M = \{\, m \in M_N \mid f(m(p_1), \ldots, m(p_n)) = 1 \,\}.$$

We denote with $\mathcal{F}_n$ the set of all interval logic functions with n arguments and with $\mathcal{M}$ the set of all sets of markings described by functions of the set $\mathcal{F}_n$. The least element of $\mathcal{M}$ is the empty set, described by the function $f = 0$. The greatest element is the set M_N, described by the function $f = 1$.

Proposition 2
It can be shown that boolean algebra $[\mathcal{F}_n, \vee, \wedge, \neg, 0, 1]$ *and* $[\mathcal{M}, \cup, \cap, \bar{}, \emptyset, M_N]$ *are isomorphic*[1].

The Proposition 2 states that reasoning in terms of sets of markings and set operations is isomorphic to reasoning in terms of interval logic functions and logic operations on them. We shall denote an interval logic function describing a set of markings M as a *characteristic function* of M and write it as χ_M. Let $M_1, M_2 \in \mathcal{M}$ be two sets of markings then

- $\chi_{M_1 \cup M_2} = \chi_{M_1} \vee \chi_{M_2}$,
- $\chi_{M_1 \cap M_2} = \chi_{M_1} \wedge \chi_{M_2}$,
- $\chi_{\overline{M_1}} = \neg \chi_{M_1}$.

Example 12
Consider a P/T net shown in Fig. 2.1 (page 14). We can treat p_i as a place of the net, as well as a variable of a characteristic function $\chi = \chi(p_1, \ldots, p_5)$.

1. The initial marking $(3, 2, 0, 1, 0)$ is described by the function

$$\chi_{m_0} = p_1 = 3 \wedge p_2 = 2 \wedge p_3 = 0 \wedge p_4 = 1 \wedge p_5 = 0.$$

2. A characteristic function of the reachability set $\mathcal{R}_N(m_0)$ is

$$\begin{aligned} \chi_{\mathcal{R}_N(m_0)} = \quad & (p_1 = 3 \wedge p_2 = 2 \wedge p_3 = 0 \wedge p_4 = 1 \wedge p_5 = 0) \vee \\ & (p_1 = 2 \wedge p_2 = 1 \wedge p_3 = 1 \wedge p_4 = 1 \wedge p_5 = 0) \vee \\ & (p_1 = 1 \wedge p_2 = 0 \wedge p_3 = 2 \wedge p_4 = 1 \wedge p_5 = 0) \vee \\ & (p_1 = 0 \wedge p_2 = 1 \wedge p_3 = 1 \wedge p_4 = 0 \wedge p_5 = 1) \vee \\ & (p_1 = 0 \wedge p_2 = 2 \wedge p_3 = 0 \wedge p_4 = 0 \wedge p_5 = 1). \end{aligned}$$

[1] Boolean algebra are introduced in the Appendix.

3. A function $\chi_{p_4=1} = p_4 = 1$ describes a set of all possible markings where the place p_4 has one token. Obviously, this set is infinite. A function $\chi_{p_4=1} \wedge \chi_{\mathcal{R}_N(m_0)}$ describes a (clearly finite) set of markings reachable from m_0 where p_4 has one token.

Characteristic functions can be also used to represent binary relations between sets. Given sets M,$M' \in \mathcal{M}$, to represent a binary relation $R \subseteq M \times M'$ it is necessary to use two different sets of variables to identify the elements of each set. Taking the set $\{ x_1, \ldots, x_n \}$ for M and $\{ x'_1, \ldots, x'_n \}$ for M', the characteristic function of R is defined as

$$\chi_R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) = 1 \Leftrightarrow$$
$$\exists (m, m') \in R : \chi_m(x_1, \ldots, x_n) = 1 \wedge \chi_{m'}(x'_1, \ldots, x'_n) = 1.$$

Like in the previous chapter, we limit the set $\mathcal{F}_n$ to those interval logic functions that are decomposable over interval partitions with a finite number of independence intervals. The limited set of functions still contains most of the functions of interest. For example, it obviously contains functions describing any sets of reachable markings of a bounded P/T net. As we know, ROIDDs provide a canonical form representation for interval logic functions, so they can be used as an efficient data structure for storage and fast manipulations on sets of markings.

4.2 Symbolic Manipulation of Petri Nets using ROIDDs

4.2.1 Symbolic Operators

As reasoning in terms of sets of markings is isomorphic to reasoning in terms of interval logic functions, for convenience, we shall usually write and discuss algorithms using the set notation. Symbolic algorithms for Petri nets operate on sets of markings applying the operators shown in Table 4.1. As mentioned at the end of the previous chapter, logic operations on interval functions are implemented as efficient dedicated ROIDD operations, hence, the application of basic operators is usually a cheap operation.

Let N be a P/T net with extended arcs, and let n be the number of places of N. Pick(M) returns some marking m belonging to the set of markings $M \in \mathcal{M}$. We implement the function as a special ROIDD operation, consider Algorithm 6. Actually, we just have to find $c_1, \ldots, c_n \in \mathbb{N}_0$ such that $\chi_M|_{x_1=c_1}| \ldots |_{x_n=c_n} = 1$ and construct then an ROIDD for the function $\chi_m = \bigwedge_{1 \leq i \leq n}(x_i = c_i)$. The function iPick gets an ROIDD G_M encoding the set M[2] and returns an ROIDD G_m encoding m. As usual, we assume that G_M and G_m are saved in the same Shared ROIDD. The algorithm is

[2]For the sake of brevity, we shall write "an ROIDD G_M encoding a set M" instead of "an ROIDD G_M representing a characteristic function χ_M which describes a set of markings M".

Basic operators:			Special operators:		
$\cap$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Pick	:	$\mathcal{M} \to M_N$
$\cup$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Fire	:	$\mathcal{M} \times T \to \mathcal{M}$
$\setminus$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	RevFire	:	$\mathcal{M} \times T \to \mathcal{M}$
$\overline{\cdot}$	:	$\mathcal{M} \to \mathcal{M}$	Img	:	$\mathcal{M} \to \mathcal{M}$
$=$	:	$\mathcal{M} \times \mathcal{M} \to \mathbb{B}$	PreImg	:	$\mathcal{M} \to \mathcal{M}$

Table 4.1: Symbolic operators

implemented using a recursive function AuxPick that gets a root r of an ROIDD and an index i of an ROIDD variable as arguments.

When discussing algorithms on ROIDDs in this chapter, we shall assume that the ordering π is defined as $x_1 <_\pi x_2 <_\pi \ldots <_\pi x_n$ and that a function $\mathrm{Pl}(x_i) : X \to P$ returns a place assigned to the variable x_i.

In AuxPick several cases depending on the relationship between r and i are possible.

1. The case $r = 0$ occurs only if M is an empty set, then G_m also contains only the terminal node 0.

2. The end of recursion is reached if $i = n + 1$, in this case AuxPick always returns the terminal node 1.

3. If $\mathrm{var}(r) > x_i$, then M is an infinite set in which a place $\mathrm{Pl}(x_i)$ may contain any number of tokens. Hence, we can randomly select some $c_i \in \mathbb{N}_0$. The root of the resulting IDD has three outgoing arcs, an arc labeled with the interval $[c, c + 1)$ leads to an ROIDD constructed by the recursive call to AuxPick, the two other arcs lead to 0.

4. If $\mathrm{var}(r) = x_i$, we choose one of the outgoing arcs of r which do not lead to 0. c_i is randomly selected from the interval labeling this arc. The resulting IDD is constructed as described in the previous case.

Implementation of the following functions is discussed in the next section. Usually, Img and PreImg are the most expensive operations.

- The function $\mathrm{Fire}(M,\ t)$ returns a set of markings M' obtained by firing the transition t in the set of markings M

$$\mathrm{Fire}(M, t) = \{\, m' \in M_N \mid \exists m \in M : m \xrightarrow{t} m' \,\}.$$

Algorithm 6 (Picking a state)

```
1  func iPick (G_M)
2     func AuxPick (r, i)
3        if r = 0 ∨ i = n + 1 then return r fi
4        if var(r) > x_i then
5           c := oneof (ℕ_0)
6           r' := AuxPick(r, i + 1)
7        else  /* var(r) = x_i */
8           k := oneof ({j | child_j(r) ≠ 0})
9           c := oneof (part_k(r))
10          r' := AuxPick(child_k(r), i + 1)
11       fi
12       NewChild := {0, r', 0}
13       NewPart := {[0, c), [c, c + 1), [c + 1, ∞)}
14       res := MakeNode(x_i, NewPart, NewChild)
15       return res
16    end
17    begin
18       G_m.root := AuxPick(G_M.root, 1)
19       return G_m
20    end
```

- $\mathrm{Img}(M)$ returns a set of markings M' obtained by firing all transitions of the net in the set of markings M

$$\mathrm{Img}(M) = \{\, m' \in M_N \mid \exists m \in M, \exists t \in T : m \xrightarrow{t} m' \,\}.$$

- $\mathrm{RevFire}(M,\, t)$ returns a set of markings M' from which M can be reached if the transition t fires

$$\mathrm{RevFire}(M, t) = \{\, m' \ \in M_N \mid \exists m \in M : m' \xrightarrow{t} m \,\}.$$

- $\mathrm{PreImg}(M)$ returns a set of markings M' from which M can be reached, if any transition of the net fires

$$\mathrm{PreImg}(M) = \{\, m' \ \in M_N \mid \exists m \in M, \exists t \in T : m' \xrightarrow{t} m \,\}.$$

4.2.2 Enabling and Firing

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs and let $t \in T$ be some transition of N. With $E_t \in \mathcal{M}$ we denote a set of markings in which t can be enabled, a characteristic function for this set can be defined as

$$\chi_{E_t} = \bigwedge_{p_i \in {}^\bullet t} (p_i \geq t^-(p_i) \ \wedge \ p_i < t_I^-(p_i) \ \wedge \ p_i \geq t_R^-(p_i)).$$

Here we treat p again as a place of the net, as well as a variable of the characteristic function. Correspondingly, a set of markings D_t in which t can not be enabled is described by the function $\chi_{D_t} = \neg\chi_{E_t}$.

As mentioned above, characteristic functions can be used to represent binary relations between sets. In the traditional symbolic approach that employs ROBDDs this fact is used to encode a *transition relation* of a net. The function Img is implemented then using standard ROBDD operations. This technique is not very well suited for Petri nets. Consider, for example, a transition t that adds two tokens to a place p. We have to represent explicitely all possible pairs of a place's state and its successor after firing of the transition t, i.e.

$$\{(p, p')\} = \{(0, 2), (1, 3), \ldots, (n-2, n)\}.$$

The transition relation becomes too large and firing becomes a very inefficient operation. Notice that we also have to introduce an upper bound n to keep the relation finite. To avoid these problems firing can be implemented as a special operation on decision diagrams. This approach was first applied in [YHTM96] for the analysis of 1-bounded Petri nets using ZBDDs. The implementation of the function Fire resembled the Apply algorithm, the function got a decision diagram G_M encoding a set of markings M and a list of adjacent places of the transition t as arguments. A decision diagram $G_{M'}$ encoding the resulting set M' was constructed during a single traversal of G_M. Computation of a set $K = \mathrm{Img}(M)$ was done then as $K = \bigcup_{t \in T} \mathrm{Fire}(M, t)$. This technique was shown to be very efficient and was applied also in [Rid97] and [Noa99]. A special type of diagrams called *Predicate action diagrams* (PAD) was introduced in [ST98]. Instead of the function Fire, the function Img was implemented as a special operation on ROIDDs, it got a decision diagram G_M and a PAD encoding all transitions of the net as arguments.

We shall use *action lists* which encode single transitions and implement the function Fire as a special ROIDD operation. Action lists naturally support enabling and firing rules of P/T nets with extended arcs, compared to simple lists of places they allow a more flexible implementation of the function Fire. For example, this implementation can be also reused in the function RevFire. As we shall see, implementation of the function Fire instead of the function Img as a special operation on ROIDDs allows application of different traversal techniques which can enormously speedup the construction and exploration of state spaces.

For every transition t connected with n_t places $\{p_{1_t}, \ldots, p_{n_t}\}$ we construct an action list al using enabling and firing rules of P/T nets with extended arcs (recall section 2.6).

The list consists of n_t elements $\{al_1, \ldots, al_{n_t}\}$ having the following structure:

- $al_i.var$ is an ROIDD variable assigned to the place p_{i_t}: $\mathrm{Pl}(al_i.var) = p_{i_t}$
- $al_i.enInterval \in \mathcal{I}$ determines how many tokens the place p_{i_t} may contain if t is enabled: $al_i.enInterval = [t^-(p_{i_t}), \infty) \cap [0, t_I^-(p_{i_t})) \cap [t_R^-(p_i), \infty)$
- $al_i.action = \begin{cases} \text{SHIFT} & \text{if } (p_{i_t}, t) \notin Z \\ \text{ASSIGN} & \text{otherwise} \end{cases}$
- $al_i.shift \in \mathbb{Z}$ is used when $al_i.action = \text{SHIFT}$, $al_i.shift = \Delta t(p_{i_t})$
- $al_i.asgnInterval \in \mathcal{I}$ is used when $al_i.action = \text{ASSIGN}$, $al_i.asgnInterval = [t^+(p_{i_t}), t^+(p_{i_t}) + 1)$.

Elements of the action list are sorted with respect to the ordering defined for ROIDD variables. The action list $revAl$ to be used by the implementation of the function RevFire can be easily constructed from the list al:

- $revAl_i.var = al_i.var$
- $revAl_i.action = al_i.action$
- $revAl_i.enInterval = \begin{cases} al_i.enInterval + al_i.shift & \text{if } al_i.action = \text{SHIFT} \\ al_i.asgnInterval & \text{otherwise} \end{cases}$
- $revAl_i.shift = -al_i.shift$
- $revAl_i.asgnInterval = al_i.enInterval$.

Example 13
Consider a P/T net with extended arcs in Fig. 2.5 (page 26). Action lists created for transitions of the net are presented in Table 4.2. We assume that ROIDD variables x_0 and x_1 are assigned correspondingly to the places p_0 and p_1 and that $x_0 <_\pi x_1$.

	var	*enInterval*	*action*	*shift*	*asgnInterval*
t_0	x_0	$[0, 2)$	SHIFT	0	
t_1	x_0	$[2, \infty)$	SHIFT	0	
	x_1	$[0, \infty)$	ASSIGN		$[0, 1)$
t_2	x_0	$[2, 3)$	SHIFT	0	
	x_1	$[0, \infty)$	ASSIGN		$[5, 6)$
t_3	x_0	$[2, \infty)$	SHIFT	-2	

Table 4.2: Action lists for the net from Fig. 2.5

Let us consider Algorithm 7. The function iFire gets an ROIDD G_M encoding a set M and a transition t as arguments and returns an ROIDD encoding the set of markings obtained by firing t in M. The algorithm is implemented with help of a recursive function AuxFire which gets a root r of an ROIDD and an action list al as arguments. The implementation resembles the Apply algorithm discussed in the previous section. In AuxApply, the construction of the resulting ROIDD is determined by two ROIDDs and an operation $\star$. In AuxFire, the action list al replaces one of the ROIDDs and $\star$. Recall here that elements of al are sorted accordingly to the variable ordering defined for ROIDDs. The action list encodes also how the independence interval partitions of new nodes must be constructed. As usual, we assume that G_M and the resulting ROIDD $G_{M'}$ are saved in the same Shared ROIDD R. We denote with a the first element of the action list al. Several cases depending on the relationship between r and al are possible.

1. The end of recursion is reached if the action list al is empty or r is a terminal node labeled with 0. In this case r can be returned as a result of the function.

2. If $\mathrm{var}(r) < a.var$, then the action list does not determine how the nodes with the variable $\mathrm{var}(r)$ must be constructed, hence $|\,\mathrm{part}(r)|$ children of the root of the resulting IDD are simply created using recursive calls to AuxFire. The function MakeNode is used then to insert the IDD into R.

3. If $\mathrm{var}(r) = a.var$, then the resulting IDD is constructed as defined by the action list. So, $a.enInterval$ determines a set C' of children of r that must be used in recursive calls to AuxFire

$$C' = \{\mathrm{child_j}(r) \mid \mathrm{part_j}(r) \cap a.enInterval \neq \emptyset\}.$$

   $a.action$ determines how the root r' of the resulting IDD is constructed.

   - If $a.action$ = ASSIGN, then r' can have up to three children. We construct first an outgoing arc labeled by $a.asgnInterval$ which leads to a node computed as

$$\bigvee_{\mathrm{child}_j(r) \in C'} \mathrm{AuxFire}(\mathrm{child}_j(r), \mathrm{tail}(al)).$$

     Other arcs leading to the terminal node 0 will be constructed by the function CompletePartition.
   - If $a.action$ = SHIFT, then r' can have up to $|\,\mathrm{part}(r)| + 2$ children. We compute $|C'|$ of them using recursive calls to the function AuxFire. The function $\mathrm{Shift}(P{=}(I_1, \ldots, I_n), val)$ shifts all intervals in the list P on $val \in \mathbb{Z}$ and replaces negative bounds of intervals (if such appear) with 0.

The function CompletePartition $(P{=}(I_1, \ldots, I_n), C = (c_1, \ldots, c_n))$ guarantees that intervals in the list P form a partition of $\mathbb{N}_0$, modifying if needed the lists P and C. If an interval including 0 is appended to P, then 0 is appended at the head of the list C, if an interval including ∞ is appended to P, then 0 is appended at the end of the list C.

4. A case when $\mathrm{var}(r) > a.var$ occurs if M is an infinite set of markings in which a place $\mathrm{Pl}(a.var)$ can contain any number of tokens. The computation is then a simplified version of the previous case.

As usual, to prevent the algorithm from being exponential AuxFire is implemented as a memory function.

4.2.3 Reachability Analysis

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs, and let $M \in \mathcal{M}$ be some set of markings. A function FwdReach(M) returns a set of markings reachable from markings in the set M

$$\mathrm{FwdReach}(M) = \{\, m' \in M_N \mid \exists m \in M : m \xrightarrow{*} m' \,\}.$$

We define also a complementary function BwdReach(M) which returns a set of markings from which markings in the set M are reachable

$$\mathrm{BwdReach}(M) = \{\, m' \in M_N \mid \exists m \in M : m' \xrightarrow{*} m \,\}.$$

Given some set $M \in \mathcal{M}$, we can apply one of the two strategies to find out if M is reachable from the initial marking m_0.

- We can use the forward strategy: compute first the reachability set $\mathcal{R}_N(m_0) = \mathrm{FwdReach}(m_0)$ and then check whether it intersects M.
- Alternatively, we can compute the set $B = \mathrm{BwdReach}(M)$ and check if m_0 belongs to B.

It can happen that one of the approaches is more efficient and terminates earlier. Notice that a possible drawback of the backward strategy is that it can explore too many markings not present in the $\mathcal{R}_N(m_0)$, we shall discuss this problem later in section 4.3.3.

Consider Algorithm 8 which implements the function FwdReach using a *symbolic breath-first search*. The algorithm maintains a set of already reached markings *Reached* and a set of unexplored markings *New*, both initially equal to M. Iteratively, the successors of markings in *New* are added to the set *Reached*. A set *Old* contains markings

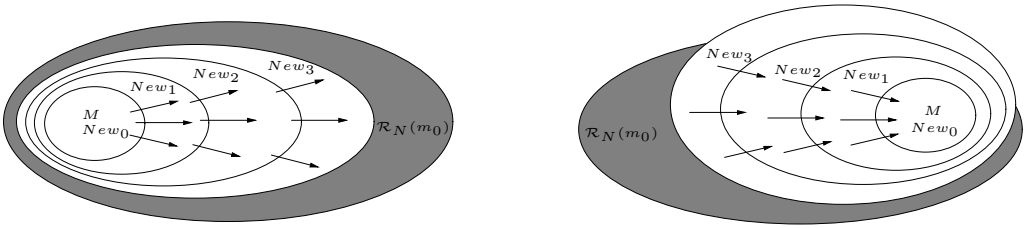
Algorithm 7 (Firing as a special ROIDD operation)

```
1  func iFire (G_M, t)
2     func AuxFire (r, al)
3        if al = ∅ ∨ r = 0 then return r fi
4        a := head(al)
5        if ResultTable[r, a] ≠ ∅ then return ResultTable[r, a] fi
6        if var(r) < a.var then
7           NewPart := part(r)
8           forall I_j ∈ NewPart do
9              NewChild_j := AuxFire(child_j(r), al)
10          od
11          res := MakeNode(var(r), NewPart, NewChild)
12       elseif var(r) = a.var then
13          if a.action = ASSIGN then
14             NewPart_1 := a.asgnInterval
15             NewChild_1 := 0
16             forall I_j ∈ part(r) do
17                if I_j ∩ a.enInterval ≠ ∅ then
18                   NewChild_1 := AuxApply(∨, NewChild_1, AuxFire(child_j(r), tail(al))
19                fi
20             od
21          else
22             NewPart := Intersect(part(r), a.enInterval)
23             forall I_j ∈ NewPart, I_k ∈ part(r) do
24                if I_j ∩ I_k ≠ ∅ then
25                   NewChild_j := AuxFire(child_k(r), tail(al))
26                fi
27             od
28             Shift(NewPart, a.shift)
29          fi
30          CompletePartition(NewPart, NewChild)
31          res := MakeNode(var(r), NewPart, NewChild)
32       else  /* var(r) > a.var */
33          if a.action = ASSIGN then
34             NewPart_1 := a.asgnInterval
35          else
36             NewPart_1 := a.enInterval
37             Shift(NewPart, a.shift)
38          fi
39          NewChild_1 := AuxFire(r, tail(al))
40          CompletePartition(NewPart, NewChild)
41          res := MakeNode(a.var, NewPart, NewChild)
42       fi
43    fi
44    ResultTable[r, a] := res
45    return res
46 end
47 begin
48    G_M'.root := AuxFire(G_M.root, t.al)
49    return G_M'
50 end
```

Algorithm 8 (Forward reachability using BFS)

```
1  func FwdReach (M)
2      Reached := M
3      New := M
4      repeat
5          Old := Reached
6          Reached := Reached ∪ Img(New)
7          New := Reached \ Old
8      until New = ∅
9      return Reached
10 end
```

```
1  func Img (M)
2      Res := ∅
3      forall t ∈ T do
4          Res := Res ∪ Fire (M, t)
5      od
6      return Res
7  end
```



FwdReach(M) BwdReach(M)

Figure 4.1: Forward and backward reachability analysis

reached in the previous iterations. The set *New* is computed as a difference between the sets *Reached* and *Old*. The process ends when the set *New* is found to be empty. It is easy to see that if d is a diameter[3] of the reachability graph $\mathcal{RG}_N$, then the algorithm terminates after making maximally $d + 1$ iterations.

Note that if we are only interested whether some markings in a set $M' \in \mathcal{M}$ are reachable from markings in M, we can modify the algorithm to work "*on-the-fly*". Every time after the computation of a set Img(*New*), we can check if this set intersects the set M' and finish the process with a positive answer if so. The functions PreImg and BwdReach can be implemented analogously to the functions Img and FwdReach.

Table 4.3 provides statistics[4] gathered during the computation of state spaces of a number of Petri net models[5] using Algorithm 8. The first columns provide an information about the model: a number of places, transitions, and a maximal number of tokens a place can contain (*bnd*). Next comes the size of the state space and the number of ROIDD nodes (N_G) and arcs (A_G) needed to encode it. Finally, the last columns show the number of iterations (*it*) made in Algorithm 8, the total number of ROIDD nodes

[3] See the definitions in the Appendix.
[4] The benchmark was done on a PC with Intel Pentium 4, 2.8GHz, 512MB RAM running Linux
[5] Short descriptions of the models can be found in the Appendix.

Model	Net			$\mathcal{R}_N$			$\mathcal{R}_N$ generation		
	$\|P\|$	$\|T\|$	*bnd*	$\|\mathcal{R}_N\|$	N_G	A_G	*it*	Σ	τ
FMS20	22	20	20	$6.0 \cdot 10^{12}$	$1.7 \cdot 10^3$	$8.4 \cdot 10^3$	281	$2.7 \cdot 10^5$	8
FMS40	22	20	40	$2.6 \cdot 10^{16}$	$5.9 \cdot 10^3$	$3.6 \cdot 10^4$	561	$1.5 \cdot 10^6$	435
HAL	29	46	44	$3.0 \cdot 10^6$	$3.7 \cdot 10^2$	$1.3 \cdot 10^3$	112	$5.8 \cdot 10^4$	0.8
RW500	14	13	500	$3.0 \cdot 10^3$	$9.1 \cdot 10^3$	$2.4 \cdot 10^4$	1506	$4.8 \cdot 10^4$	1.2
RW$\leq$500	14	13	500	$3.8 \cdot 10^8$	$9.1 \cdot 10^3$	$2.4 \cdot 10^4$	1506	$5.2 \cdot 10^4$	1.3
MUL66	15	14	36	$2.8 \cdot 10^6$	$2.1 \cdot 10^3$	$1.1 \cdot 10^4$	206	$3.8 \cdot 10^5$	3
MUL810	15	14	80	$1.0 \cdot 10^8$	$5.6 \cdot 10^3$	$4.2 \cdot 10^4$	434	$1.9 \cdot 10^6$	31
ACK33	23	24	61	$1.3 \cdot 10^9$	$8.4 \cdot 10^3$	$8.2 \cdot 10^4$	826	$5.6 \cdot 10^6$	150
SLOT9	90	90	1	$3.8 \cdot 10^{11}$	$1.6 \cdot 10^3$	$4.1 \cdot 10^3$	160	$1.3 \cdot 10^6$	28
PUSH9	168	157	1	$1.7 \cdot 10^8$	$6.4 \cdot 10^2$	$1.6 \cdot 10^3$	952	$1.7 \cdot 10^7$	155
CS1	231	202	1	$2.5 \cdot 10^4$	$4.1 \cdot 10^3$	$9.0 \cdot 10^3$	293	$3.2 \cdot 10^5$	11
CS5	231	202	1	$1.7 \cdot 10^6$	$2.0 \cdot 10^4$	$4.5 \cdot 10^4$	315	$5.6 \cdot 10^6$	225
OS	198	176	1	$2.8 \cdot 10^6$	$4.9 \cdot 10^3$	$1.1 \cdot 10^4$	656	$3.7 \cdot 10^6$	120

Table 4.3: State space generation using symbolic BFS

(Σ), and the time in seconds (τ) needed to compute $\mathcal{R}_N(m_0)$. Notice that though ROIDDs provide quite compact representation for the state spaces, the total number of used nodes and the computation time can be relatively high.

4.3 Improving the Reachability Analysis

4.3.1 Transition Chaining

It was noticed that the breath-first order strategy used in Algorithm 8 is not well suited for asynchronous systems. Often, sizes of decision diagrams encoding the working sets of the algorithm are much larger during the state space exploration than upon termination. As efficiency of the operations on decision diagrams depends on their sizes, the performance decreases as big diagrams start to be generated. A number of techniques have been proposed to combat the problem [BCL$^+$94, PRCB94, GV01, CLS01, SP02].

Straying from the breath-first strategy in the exploration of state spaces can (heuristically) improve the reachability analysis. In the technique denoted as *transition chaining* [PRCB94], states reached after firing a transition are immediately added to the set of states to be explored in the next step. The underlying idea of the heuristic is that such a technique can help to discover new states faster, and the faster states can be discovered, the faster the algorithm terminates. Actually, the overall performance can be also improved due to the fact that intermediate sets of states are represented by smaller decision diagrams. The line *6* in Algorithm 8 is replaced with

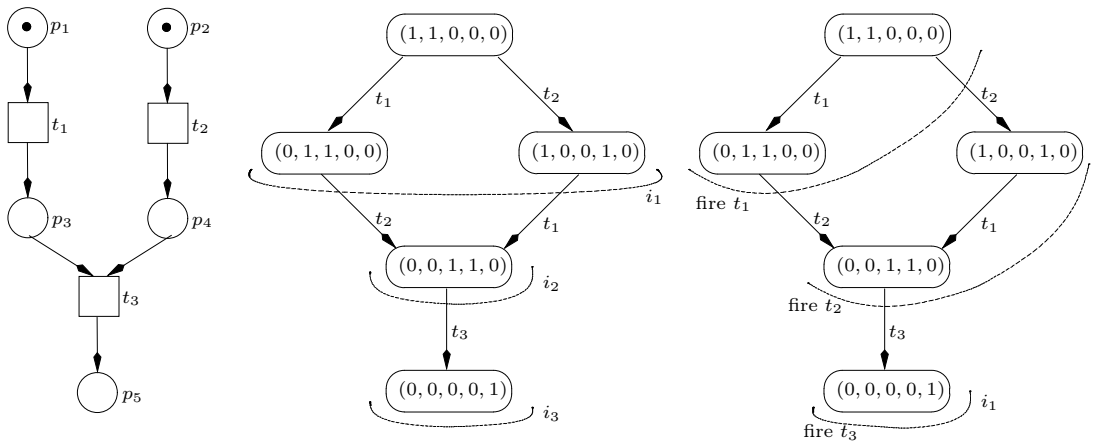


Figure 4.2: An illustration of the transition chaining technique

- forall $t \in T$ do
- $\quad New := New \cup \text{Fire}\ (New,\ t)$
- od
- $Reached := Reached \cup New$

This slight modification of the algorithm can produce the "domino effect" accelerating the state space exploration. When the Algorithm 8 is used to compute a reachability set $\mathcal{R}_N(m_0)$, exactly $r(\mathcal{RG}_N)$[6] iterations must be made to generate all states. With transition chaining, the number of iterations at most equals to $r(\mathcal{RG}_N)$, but is usually smaller and can, in principle, be reduced by a factor of up to $|T|$.

Example 14
Consider a P/T net in Fig. 4.2(a). Algorithm 8 requires three iterations to generate all reachable states. With transition chaining, all states are generated during a single iteration, compare the figures 4.2(b) and 4.2(c).

Obviously, when transition chaining is used, the order in which transitions are fired can have an influence on the speed of the new states generation and, consequently, on the number of iterations needed to end up the process. A simple heuristic algorithm to define the transition ordering for 1-bounded Petri nets was suggested in [Noa99]. First, all transitions enabled in the initial marking are added to the ordering. In the next step, tokens are put in post-places of these transitions (without the usual deletion of tokens from pre-places). All newly enabled transitions are added to the ordering and

[6] $r(\mathcal{RG}_N)$ denotes a radius of the reachability graph, see the definitions in the Appendix.

the procedure is repeated. The process terminates when no more new transitions can be added to the ordering. During the state space exploration, transitions chaining is used and transitions are always applied in this computed order. Several techniques to schedule the application of transitions were studied in [SP02]. In the *token traverse* (TOK) technique, a static analysis employing *causality* between transition is used to build the transition application scheme. This scheme does not imply a static application order of transition, a feedback from the traversal is used to adapt the order dynamically.

For the further reduction of the number of intermediate diagrams a function

$$\text{FireUnion}(M_1, M_2, t) = \text{Fire}(M_1, t) \cup M_2.$$

was implemented in [Noa99] as a dedicated ZBDD operation and several variations of Algorithm 8 were considered. A variation, which resulted in the best performance (the lowest number of ZBDDs nodes used in the computations and, consequently, the fastest times), is presented in Algorithm 9. We have adopted this technique and implemented the functions FireUnion and RevFireUnion as special ROIDD operations. The functions employ a supplementary function AuxFireUnion, which is implemented as a combination of AuxApply from Algorithm 3 and AuxFire from Algorithm 7. We forbear from the discussion on quite technical details of the implementation.

Table 4.4 provides statistics gathered during the generation of state spaces with and without transition chaining.

- BFS denotes the original Algorithm 8.
- MBFS_1 denotes Algorithm 9 with a random order of transitions.
- MBFS_2 denotes Algorithm 9 with the transitions ordering computed after [Noa99].

Algorithm 9 (Forward reachability with transition chaining and FireUnion)

```
1  func FwdReach (M)
2     Reached := M
3     repeat
4        Old := Reached
5        forall t ∈ T do
6           Reached := FireUnion(Reached, Reached, t)
7        od
8     until Reached = Old
9     return Reached
10 end
```

Model	BFS			$MBFS_1$			$MBFS_2$			TOK		
	it	Σ	τ	*it*	Σ	τ	*it*	Σ	τ	*it*	Σ	τ
FMS20	281	$2.7 \cdot 10^5$	8	25	$6.4 \cdot 10^4$	0.3	22	$7.3 \cdot 10^4$	0.3	53	$6.3 \cdot 10^4$	0.2
FMS40	561	$1.5 \cdot 10^6$	435	45	$4.2 \cdot 10^5$	1.7	42	$4.4 \cdot 10^5$	1.8	105	$3.2 \cdot 10^5$	1.3
HAL	112	$5.8 \cdot 10^4$	0.8	95	$3.2 \cdot 10^4$	0.6	93	$3.4 \cdot 10^4$	0.7	118	$5.2 \cdot 10^4$	15
RW500	1506	$4.8 \cdot 10^4$	1.2	502	$3.0 \cdot 10^4$	2.3	501	$3.0 \cdot 10^4$	2.4	293	$2.7 \cdot 10^4$	1.3
RW$\leq$500	1506	$5.2 \cdot 10^4$	1.3	502	$3.0 \cdot 10^4$	2.5	501	$3.0 \cdot 10^4$	2.5	293	$3.1 \cdot 10^4$	1.3
MUL66	206	$3.8 \cdot 10^5$	3	157	$1.8 \cdot 10^5$	2.9	157	$1.8 \cdot 10^5$	2.6	145	$6.1 \cdot 10^5$	16
MUL810	434	$1.9 \cdot 10^6$	31	337	$8.9 \cdot 10^5$	32	320	$8.9 \cdot 10^5$	31	304	$3.5 \cdot 10^6$	600
ACK33	826	$5.6 \cdot 10^6$	150	598	$5.3 \cdot 10^6$	220	430	$4.0 \cdot 10^6$	160			>3600
SLOT9	160	$1.3 \cdot 10^6$	28	19	$2.1 \cdot 10^5$	1.7	12	$6.1 \cdot 10^5$	2.5	37	$1.9 \cdot 10^5$	1.5
PUSH9	952	$1.7 \cdot 10^7$	155	175	$1.1 \cdot 10^5$	3.1	51	$1.2 \cdot 10^5$	1.1	118	$1.8 \cdot 10^5$	1.1
CS1	293	$3.2 \cdot 10^5$	11	154	$5.2 \cdot 10^4$	3.4	4	$5.2 \cdot 10^4$	0.7	73	$2.0 \cdot 10^5$	2.3
CS5	315	$5.6 \cdot 10^6$	225	96	$4.6 \cdot 10^5$	25	10	$4.8 \cdot 10^5$	5.8	109	$6.5 \cdot 10^5$	11
OS	656	$3.7 \cdot 10^6$	120	221	$4.3 \cdot 10^5$	18	11	$2.9 \cdot 10^5$	2.8	209	$3.7 \cdot 10^5$	6.6

Table 4.4: State space generation statistics

- TOK denotes the token traverse algorithm of [SP02]. For this algorithm the variable *it* denotes the number of calls to FireUnion divided by the number of transitions in the net.

Application of the transition chaining technique resulted in the decreased number of iterations, reduced the number of used nodes and the computation times. An appropriate transitions ordering can significantly improve the performance of Algorithm 9 on 1-bounded models with many transitions. Notice that application of the static order led to more stable results and outperformed the dynamic token traverse technique on many models. A variation of this technique denoted as *weighted token traverse* [SP02] can potentially accelerate the state space exploration, it involves, however, computation of the number of new states explored by firing a transition. In our case this operation requires *large integer arithmetic* and is very expensive.

We have also studied the influence of the transitions ordering on the performance of the backward search. A new simple heuristic (to our knowledge, not yet described in the literature) for the case when the static application order of transitions is used consists in changing of the computed ordering on the reversed one. Improvements in the efficiency achieved with this heuristic are usually comparable to the ones achieved by switching from $MBFS_1$ to $MBFS_2$.

In k-bounded Petri nets, it is often a case that pre-places of a transition t contain enough tokens for t to fire several times. We have noticed that firing t until fixpoint is reached in Algorithm 9 can help to discover new states faster and can produce sets of states which are encoded by smaller ROIDDs. We replace the line *6* with

```
· repeat
·     Old2 := Reached
·     Reached := FireUnion(Reached, Reached, t)
· until Reached = Old2
```

The only somewhat similar heuristic that we have met in the literature was suggested in [BCL$^+$94] for the analysis of asynchronous digital circuits with ROBDDs and partitioned transition relations. Local fixpoints of subcircuits were repeatedly computed until a global fixpoint was reached.

To reduce a number of intermediate ROIDDs we implement as a special ROIDD operation a function FireFixp(M, t), which returns a set of states that represents a fixpoint of the set M with respect to the firing of the transition t. Recall that firing of t is a local event, affecting only its neighboring places and leaving all others places of the net without changes. Loosely speaking, we can exploit this fact and the structure of ROIDDs to compute fixpoints in the "middle" of an ROIDD. Compare Algorithms 10, 7 and 3. For the sake of simplicity, we assume that the function FireFixp will be used only with finite sets M. As usual, analogously to FireFixp, we implement also the complementary function RevFireFixp(M, t).

Table 4.4 provides statistics on the effects of the application of the heuristic.

- MBFS_2 denotes Algorithm 9 with the computed transitions ordering.
- MBFS_3 uses chaining and the same order of transitions application as MBFS_2, but, as discussed above, every transition is fired until fixpoint is reached. For this

Model	MBFS_2			MBFS_3			MBFS_4		
	it	Σ	τ	it	Σ	τ	it	Σ	τ
FMS20	22	$7.3 \cdot 10^4$	0.3	37	$2.9 \cdot 10^4$	0.2	21	$1.9 \cdot 10^4$	0.1
FMS40	42	$4.4 \cdot 10^5$	1.8	73	$1.4 \cdot 10^5$	0.9	41	$8.2 \cdot 10^4$	0.7
HAL	93	$3.4 \cdot 10^4$	0.7	12	$8.2 \cdot 10^3$	0.1	7	$8.2 \cdot 10^3$	0.1
RW500	501	$3.0 \cdot 10^4$	2.4	732	$3.0 \cdot 10^4$	2.4	501	$3.0 \cdot 10^4$	2.4
RW$\leq$500	501	$3.0 \cdot 10^4$	2.5	732	$3.0 \cdot 10^4$	2.4	501	$3.0 \cdot 10^4$	2.4
MUL66	157	$1.8 \cdot 10^5$	2.6	269	$1.6 \cdot 10^5$	2.0	63	$9.7 \cdot 10^4$	1.0
MUL810	320	$8.9 \cdot 10^5$	31	886	$1.0 \cdot 10^6$	35	115	$5.2 \cdot 10^5$	10
ACK33	430	$4.0 \cdot 10^6$	160	441	$1.3 \cdot 10^6$	15	66	$8.7 \cdot 10^5$	11
SLOT9	12	$6.1 \cdot 10^5$	2.5	19	$6.1 \cdot 10^5$	3.1	12	$6.1 \cdot 10^5$	2.5
PUSH9	51	$1.2 \cdot 10^5$	1.1	57	$1.2 \cdot 10^5$	1.2	51	$1.2 \cdot 10^5$	1.1
CS1	4	$5.2 \cdot 10^4$	0.7	5	$5.2 \cdot 10^4$	0.7	4	$5.2 \cdot 10^4$	0.7
CS5	10	$4.8 \cdot 10^5$	5.8	14	$4.8 \cdot 10^5$	6.3	10	$4.8 \cdot 10^5$	5.8
OS	11	$2.9 \cdot 10^5$	2.8	17	$2.9 \cdot 10^5$	3.1	11	$2.9 \cdot 10^5$	3.0

Table 4.5: State space generation statistics

Algorithm 10 (FireFixp as a special ROIDD operation)

```
1  func iFireFixp (G_M, t)
2     func AuxFireFixp (r, al)
3        if al = ∅ ∨ r = 0 then return r fi
4        a := head(al)
5        if ResultTable[r, a] ≠ ∅ then return ResultTable[r, a] fi
6        if var(r) < a.var then
7           NewPart := part(r)
8           forall I_j ∈ NewPart do
9              NewChild_j := AuxFireFixp(child_j(r), al)
10          od
11          res := MakeNode(var(r), NewPart, NewChild)
12       elseif var(r) = a.var then
13          newRoot := r
14          repeat
15             oldRoot := newRoot
16             newRoot := AuxFireUnion(newRoot, newRoot, al)
17          until oldRoot = newRoot
18          res := newRoot
19       else  /* var(r) > a.var */
20          error "refuse to handle infinite sets"
21       fi
22       ResultTable[r, a] := res
23       return res
24    end
25    begin
26       G_M'.root := AuxFireFixp(G_M.root, t.al)
27       return G_M'
28    end
```

algorithm the variable *it* denotes the number of calls to FireUnion divided by the number of transitions in the net.

- MBFS_4 is identical to MBFS_3, but employs the function FireFixp, *it* denotes here the number of calls to FireFixp divided by the number of transitions.

Of course, a modification that only fires every transition until fixpoint is reached can not improve performance on 1-bounded models. However, it can indeed decrease the number and sizes of intermediate ROIDDs when applied to k-bounded nets, especially when the function FireFixp is employed. As we shall see later, this effect becomes even more noticeable when we scale-up the models. Of course, this modification can be combined not only with MBFS but with any traversal strategy. In the next section we propose a strategy that exploits both the structure of Petri nets and the structure of ROIDDs.

4.3.2 Saturation Algorithm

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs. We assume that a shared ROIDD is used to store sets of marking of N and that the ROIDD variable ordering π is defined as $x_1 <_\pi x_2 <_\pi \ldots <_\pi x_n$. We define the following functions for transitions of the net.

1. $\text{Bot}(t)$ returns an index of the lowest level in the ROIDD on which the transition t *depends:*
$$\text{Bot}(t) = \max\{j \mid \text{Pl}(x_j) \in {\bullet t} \cup t{\bullet}\}.$$
2. $\text{Top}(t)$ returns an index of the highest level on which t depends:
$$\text{Top}(t) = \min\{j \mid \text{Pl}(x_j) \in {\bullet t} \cup t{\bullet}\}.$$

We define now a linear order σ for the transitions of the net as follows:

1. $t_j <_\sigma t_k$ if $\text{Bot}(t_j) > \text{Bot}(t_k)$,
2. if $\text{Bot}(t_j) = \text{Bot}(t_k)$ then $t_j <_\sigma t_k$ if $\text{Top}(t_j) > \text{Top}(t_k)$,
3. if $\text{Bot}(t_j) = \text{Bot}(t_k)$ and $\text{Top}(t_j) = \text{Top}(t_k)$ then $t_j <_\sigma t_k$ if $j < k$.

For convenience, we assume that transitions are enumerated accordingly to this newly defined order: $t_{\sigma 1} <_\sigma t_{\sigma 2} <_\sigma \ldots <_\sigma t_{\sigma|T|}$. A function $\text{FirstDep}(t_{\sigma k})$ returns an index of the first with respect to the order σ transition that has common pre- or post-places with $t_{\sigma k}$
$$\text{FirstDep}(t_{\sigma k}) = \min\{j \mid ({\bullet t_{\sigma k}} \cup t_{\sigma k}{\bullet}) \cap ({\bullet t_{\sigma j}} \cup t_{\sigma j}{\bullet}) \neq \emptyset\}.$$
Notice that we do not exclude the case $\text{FirstDep}(t_{\sigma k}) = k$, which occurs when there exists no other transitions which precede t in the order σ and share places with it.

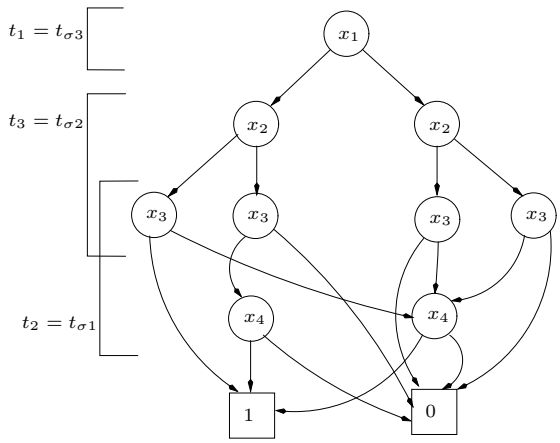


Figure 4.3: An illustration for the Example 15

Example 15
Suppose that we have a P/T net with four places and three transitions and that an ROIDD in Fig. 4.3 encodes some set of its markings (for the sake of simplicity, we omit labels on arcs). Let us assume that

$$\bullet t_1 \cup t_1 \bullet = \{\mathrm{Pl}(x_1)\}, \quad \bullet t_2 \cup t_2 \bullet = \{\mathrm{Pl}(x_3), \mathrm{Pl}(x_4)\}, \quad \bullet t_3 \cup t_3 \bullet = \{\mathrm{Pl}(x_2), \mathrm{Pl}(x_3)\}.$$

In this case

1. $\mathrm{Bot}(t_1) = 1,\ \mathrm{Top}(t_1) = 1,\ \mathrm{Bot}(t_2) = 4,\ \mathrm{Top}(t_2) = 3,\ \mathrm{Bot}(t_3) = 3,\ \mathrm{Top}(t_3) = 2.$
2. The order σ is defined as $t_2 <_\sigma t_3 <_\sigma t_1$ and transitions are enumerated accordingly: $t_{\sigma 1} = t_2$, $t_{\sigma 2} = t_3$, $t_{\sigma 3} = t_1$.
3. $\mathrm{FirstDep}(t_{\sigma 1}) = 1, \quad \mathrm{FirstDep}(t_{\sigma 2}) = 1, \quad \mathrm{FirstDep}(t_{\sigma 3}) = 3.$

We say that a transition t is *saturated* in the set of markings M if M represents a fixpoint with respect to firing of t and any transition $t_{\sigma j}$ such that $t_{\sigma j} <_\sigma t$.

In Algorithm 11, which computes a set of all markings reachable from markings in a set M, we saturate transitions accordingly to the order σ. To saturate a transition t, we compute a fixpoint of the working set *Reached* with respect to firing of this transition. If this adds new markings to the working set, then we must saturate again all transitions $t_{\sigma k} <_\sigma t$ that can fire in these markings and, potentially, add further markings to the working set. Due to the locality of Petri nets, we do not have to consider transitions that have no common places with t, thus, we can proceed to the transition $t_{\sigma\, \mathrm{FirstDep}(t)}$. In the case when there are no transitions that precede t in the order σ and share places

with it or when the set *Reached* already represented a fixpoint with respect to firing of t, we proceed to saturate the next transition in the order σ.

The algorithm terminates when the transition $t_{\sigma|T|}$ is found to be saturated in the set *Reached*. It is easy to see that the termination is guaranteed for any bounded net N and any set $M \subseteq \mathcal{R}_N(m_0)$, as the working set *Reached* is a monotonically increasing subset of $\mathcal{R}_N(m_0)$. Obviously, the order in which transitions are fired and states are added to the working set has no influence on the resulting set, unless some transition that can add states to the set *Reached* is ignored forever during the iterations. A trivial proof that this can not happen is done by contradiction. Of course, only states that are reachable from states in M can be added to the working set. Thus, for a set of states M the algorithm indeed computes the set

$$M' = \{\, m' \in M_N \mid \exists m \in M : m \xrightarrow{*} m' \,\}.$$

Intuitively, we want to achieve an effect that an ROIDD encoding the working set *Reached* grows in breadth from bottom to the top during the state space exploration. According to the order σ, transitions that affect lower levels of the ROIDD are saturated before transitions affecting higher levels. We compute fixpoints of the working set with respect to firing of every transition, hoping that it helps to discover new states faster and produces more regular sets of states which can be encoded by smaller ROIDDs. Obviously, the efficiency of the saturation strategy depends on the structure of the net and on a good ROIDD variable ordering. Fortunately, the ordering needed to get a compact representation of sets of markings, is in most cases also a good ordering for the saturation algorithm.

Algorithm 11 (Forward reachability using saturation)

```
1  func FwdReach (M)
2     Reached := M
3     i := 1
4     repeat
5        Old := Reached
6        repeat
7           Old2 := Reached
8           Reached := FireUnion(Reached, Reached, t_σi)
9        until Reached = Old2
10       if Reached = Old then
11          i := i + 1
12       else
13          j := FirstDep(t_σi)
14          if j = i then i := i + 1 else i := j fi
15       fi
16    until i = |T| + 1
17    return Reached
18 end
```

Model	Net			SAT_1			SAT_2			SAT_3		
	$\|P\|$	$\|T\|$	bnd	it	Σ	τ	it	Σ	τ	it	Σ	τ
FMS150	22	20	150	181	$1.4 \cdot 10^5$	5	61	$1.3 \cdot 10^5$	5	383	$2.7 \cdot 10^5$	7.2
FMS250	22	20	250	301	$3.7 \cdot 10^5$	28	100	$3.4 \cdot 10^5$	25	638	$8.0 \cdot 10^5$	51
HAL	29	46	44	16	$6.8 \cdot 10^3$	0.1	10	$6.8 \cdot 10^3$	0.1	1437	$3.9 \cdot 10^4$	1.8
RW500	14	13	500	390	$2.1 \cdot 10^4$	0.9	274	$2.1 \cdot 10^4$	0.7	274	$2.1 \cdot 10^4$	0.7
RW$\leq$500	14	13	500	390	$2.1 \cdot 10^4$	0.9	274	$2.1 \cdot 10^4$	0.7	274	$2.1 \cdot 10^4$	0.7
MUL2010	15	14	200	970	$1.4 \cdot 10^5$	7.4	547	$7.7 \cdot 10^4$	1.9	9854	$2.5 \cdot 10^5$	140
MUL3010	15	14	300	1453	$2.1 \cdot 10^5$	16	819	$1.2 \cdot 10^5$	4.4	15014	$3.9 \cdot 10^5$	345
ACK33	23	24	61	200	$1.7 \cdot 10^5$	2.6	150	$1.6 \cdot 10^5$	2.3	269	$1.9 \cdot 10^5$	2.6
ACK34	23	24	125	429	$5.3 \cdot 10^5$	14	322	$4.7 \cdot 10^5$	12.5	580	$5.9 \cdot 10^5$	14.5
JAN14	6	6	4096	1448	$5.1 \cdot 10^4$	42	20	$4.9 \cdot 10^4$	42	2427	$4.3 \cdot 10^4$	19
POTATO	18	14	117	192	$1.6 \cdot 10^6$	63	98	$1.3 \cdot 10^6$	55	412	$1.3 \cdot 10^6$	59
KAN25	16	16	25	32	$2.0 \cdot 10^5$	0.7	5	$4.4 \cdot 10^4$	0.3	107	$1.3 \cdot 10^4$	0.3
KAN50	16	16	50	74	$6.2 \cdot 10^6$	45	5	$5.8 \cdot 10^5$	15	213	$4.6 \cdot 10^4$	2.5
SLOT9	90	90	1	36	$6.5 \cdot 10^4$	0.5	31	$6.5 \cdot 10^4$	0.5	31	$6.5 \cdot 10^4$	0.5
PUSH9	168	157	1	41	$5.6 \cdot 10^4$	0.3	35	$5.6 \cdot 10^4$	0.3	35	$5.6 \cdot 10^4$	0.3
CS5	231	202	1	44	$1.8 \cdot 10^5$	5.6	41	$1.8 \cdot 10^5$	5.3	41	$1.8 \cdot 10^5$	5.3
OS	198	176	1	104	$1.5 \cdot 10^5$	3.5	98	$1.5 \cdot 10^5$	3.0	98	$1.5 \cdot 10^5$	3.0

Model	$\mathcal{R}_N$		BFS			$MBFS_2$			$MBFS_4$		
	$\|\mathcal{R}_N\|$	$\|N_G\|$	it	Σ	τ	it	Σ	τ	it	Σ	τ
FMS150	$4.8 \cdot 10^{23}$	$8.3 \cdot 10^4$		size		152	$4.1 \cdot 10^7$	760	151	$5.1 \cdot 10^6$	270
FMS250	$3.4 \cdot 10^{26}$	$2.3 \cdot 10^5$		size			time			time	
HAL	$3.0 \cdot 10^6$	$3.7 \cdot 10^2$	112	$5.8 \cdot 10^4$	0.8	93	$3.4 \cdot 10^4$	0.7	7	$8.2 \cdot 10^3$	0.1
RW500	$3.0 \cdot 10^3$	$9.1 \cdot 10^3$	1506	$4.8 \cdot 10^4$	1.2	501	$3.0 \cdot 10^4$	2.4	501	$3.0 \cdot 10^4$	2.4
RW$\leq$500	$3.8 \cdot 10^8$	$9.1 \cdot 10^3$	1506	$5.2 \cdot 10^4$	1.3	501	$3.0 \cdot 10^4$	2.5	501	$3.0 \cdot 10^4$	2.4
MUL2010	$7.4 \cdot 10^9$	$1.3 \cdot 10^4$	1082	$5.0 \cdot 10^6$	570	800	$4.1 \cdot 10^6$	710	242	$1.6 \cdot 10^6$	110
MUL3010	$5.2 \cdot 10^{10}$	$1.9 \cdot 10^4$		time			time		362	$3.6 \cdot 10^6$	430
ACK33	$1.3 \cdot 10^9$	$8.4 \cdot 10^3$	826	$5.6 \cdot 10^6$	150	430	$4.0 \cdot 10^6$	160	66	$8.7 \cdot 10^5$	11
ACK34	$1.4 \cdot 10^{11}$	$1.8 \cdot 10^4$		time			time		488	$9.8 \cdot 10^6$	300
JAN14	$1.8 \cdot 10^7$	$5.2 \cdot 10^3$		time		6825	$4.3 \cdot 10^5$	630	12	$5.3 \cdot 10^4$	52
POTATO	$3.3 \cdot 10^{10}$	$8.5 \cdot 10^5$		size		131	$4.2 \cdot 10^6$	3000	19	$1.6 \cdot 10^6$	210
KAN25	$7.6 \cdot 10^{12}$	$2.0 \cdot 10^3$	351	$1.4 \cdot 10^6$	39	39	$1.3 \cdot 10^6$	7	5	$2.5 \cdot 10^5$	2.2
KAN50	$1.0 \cdot 10^{16}$	$7.0 \cdot 10^3$		time		73	$2.8 \cdot 10^7$	310		size	
SLOT9	$3.8 \cdot 10^{11}$	$1.6 \cdot 10^3$	160	$1.3 \cdot 10^6$	28	12	$6.1 \cdot 10^5$	2.5	12	$6.1 \cdot 10^5$	2.5
PUSH9	$1.7 \cdot 10^8$	$6.4 \cdot 10^2$	952	$1.7 \cdot 10^7$	155	51	$1.2 \cdot 10^5$	1.1	51	$1.2 \cdot 10^5$	1.1
CS5	$1.7 \cdot 10^6$	$2.0 \cdot 10^4$	315	$5.6 \cdot 10^6$	225	10	$4.8 \cdot 10^5$	5.8	10	$4.8 \cdot 10^5$	5.8
OS	$2.8 \cdot 10^6$	$4.9 \cdot 10^3$	656	$3.7 \cdot 10^6$	120	11	$2.9 \cdot 10^5$	3.0	11	$2.9 \cdot 10^5$	3.0

Table 4.6: State space generation statistics

Table 4.6 provides statistics on the effectiveness of the new saturation strategy.

- SAT_1 denotes the saturation Algorithm 11.
- SAT_2 denotes a modification of the saturation algorithm in which fixpoints with respect to firing of transitions are computed using the function FireFixp.
- SAT_3 denotes a modification of the saturation algorithm in which transitions fire only once and not until a fixpoint is reached. The lines *6–9* of Algorithm 11 are replaced with the line *8*.
- BFS denotes the traditional BFS Algorithm 8.
- MBFS_2 denotes the transition chaining Algorithm 9 with the transitions ordering computed after [Noa99].
- MBFS_4 employs transition chaining and the same computed transition ordering as MBFS_2, but the function FireFixp is used to compute fixpoints with respect to firing of transitions.
- `size` means that peak sizes of ROIDDs encoding working sets of an algorithm exceeded the limit of $2 \cdot 10^6$ nodes or the memory limit of 300 MB.
- `time` means the computation exceeded the time limit of one hour.

The saturation technique consistently outperformed all others on all of the considered models. It indeed manages to keep sizes of intermediate diagrams smaller than other approaches, this results also in the reduced computation times. Consider plots in Fig. 4.4 which illustrate how sizes of ROIDDs encoding the working set *Reached* changed during the iterations of Algorithms 8, 9, and 11 when generating reachability sets of several nets. It is not surprisingly that ROIDDs encoding the set *Reached* are significantly larger than the ROIDD encoding $\mathcal{R}_N(m_0)$ when the traditional breath-first search Algorithm 8 is employed. One could expect that ROIDDs encoding the smaller set *New* can be also small, in fact, this is not the case. In some cases ROIDDs encoding *New* were even larger than ROIDDs encoding *Reached*. For all the models, the plots for diagrams encoding *New* had the same shape as plots for the set *Reached*. Transition chaining allows to reduce sizes of ROIDDs encoding the working set. With the saturation strategy, intermediate diagrams are kept even smaller, there are more cases when the peak size of ROIDDs encoding the set *Reached* is close to the size of the diagram encoding the reachability set. The improvements can be drastic, especially on k-bounded models. Firing every transition until fixpoint is reached allows usually to reduce the number of intermediate ROIDDs and their sizes on such models, especially when the function FireFixp is used. Only on one model (KAN) the variation of the

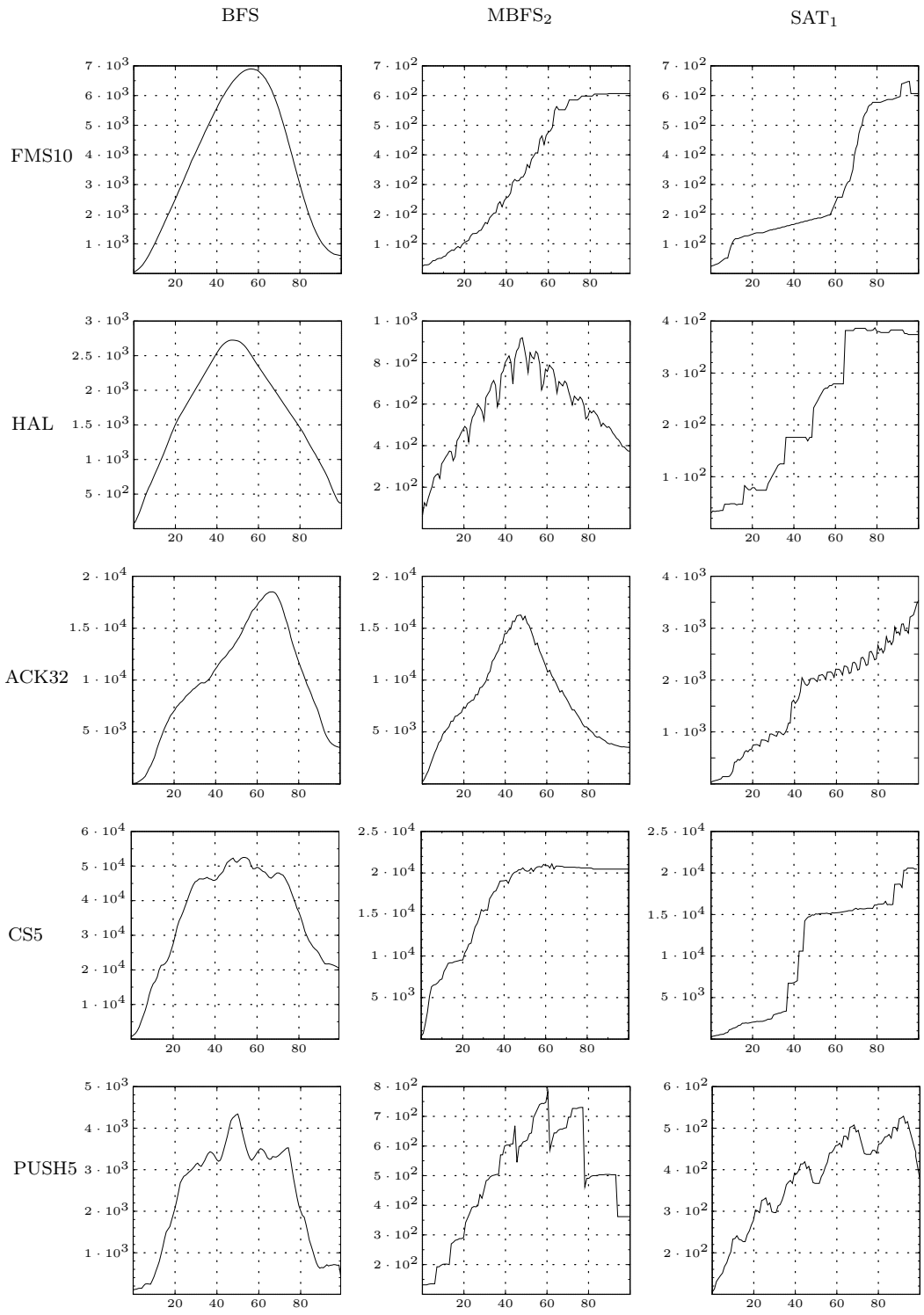


Figure 4.4: A number of ROIDD nodes in the diagram encoding the working set *Reached* during the state space generation

Algorithm 12 (Computation of all direct successors states)

```
1  func Img (M)
2     Res := ∅
3     for i := 1 to T do
4        Res := Res ∪ Fire (M, t_σi)
5     od
6     return Res
7  end
```

saturation algorithm in which transitions are fired only once resulted in significantly smaller sizes of intermediate diagrams.

We have noticed that adjusting the transition order σ can sometimes improve efficiency of the saturation algorithm. For example, postponing firing of transitions that only consume tokens without producing them can lead to more regular sets of markings encoded by smaller ROIDDs. Experiments with different orders σ have shown that an order which exploits both the structure of decision diagrams and the structure of the net leads to the best results.

As usually, analogously to FwdReach we implement the complementary saturation-based function BwdReach. A heuristic that transitions affecting lower levels of ROIDDs must be fired before transitions affecting higher levels can also improve efficiency of the algorithm implementing the function Img. In Algorithm 12 transitions are fired accordingly to the order σ. We modify the implementation of the function PreImg in the same way.

Related techniques that exploit the locality of Petri nets and the structure of decision diagrams have been developed elaborately in [CLS01, CMS03, CS03]. There are some similarities between their and our approaches, however, there are even more differences. In their approach, a net is partitioned into K subnets. Local state spaces are enumerated using explicit techniques. Every local marking is assigned to a natural number, a global marking corresponds to a K-tuple of natural numbers. Multi-valued decision diagrams (MDDs) are used to encode sets of global markings, Kronecker operators encode transitions. An MDD node at a level k is called saturated if it represents a fixed point with respect to the firing of any transition that does not affect any level above k. During the state space exploration, MDD nodes are saturated level per level, starting from the lowest one. To saturate a node at level k, all transitions affecting this and lower levels are fired. If this creates nodes at lower levels, they are saturated immediately upon creation. The approach is very promising for large-sized nets that exhibit a highly asynchronous behavior and can be easily partitioned into a number of small-sized loosely coupled subnets having small local state spaces. Our approach aims

primary at the analysis of relatively small-sized k-bounded nets which do not allow such a partitioning. Algorithm 11 can be easily adapted to implement a *limited reachability analysis*, which, as we shall see later, is very often required. Implementation of the limited reachability analysis in [CS03] is more complicated and combines both the saturation and the traditional breath-first search.

Of course, we have not considered here all the techniques that were suggested to reduce sizes of intermediate decision diagrams. We have concentrated on those that can be adopted to the analysis of Petri nets and are not ROBDDs-specific. For a survey on techniques for the reduction of intermediate decision diagrams we refer the interested reader to [GV01].

4.3.3 Limited Reachability Analysis

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs. It is easy to see that for any set $M \subseteq \mathcal{R}_N(m_0)$ a result of the function FwdReach(M) is also a subset of $\mathcal{R}_N(m_0)$. By contrast, a set $M' = \text{BwdReach}(M)$ can also contain states not present in $\mathcal{R}_N(m_0)$. Hence, the backward reachability analysis can become very inefficient, as too many states not reachable from m_0 can be explored in the algorithm computing BwdReach(M).

Example 16
Consider a P/T net in Fig. 4.5. It is easy to see that its reachability set contains only two markings $m_0 = (n, 0, 0, 0)$ and $m_1 = (0, n, 0, 0)$. A set $M' = \text{BwdReach}(\{m_1\})$ contains $\frac{(n+2)\cdot(n+1)}{2} + 1$ markings. In addition to m_0 it includes also all markings where n tokens are distributed over the places p_2, p_3, and p_4.

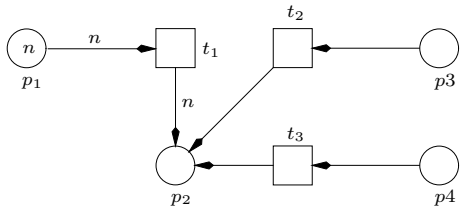


Figure 4.5: A P/T net

Let us consider now some P/T net N which contains a transition without post-places. It is easy to see that a set $M' = \text{BwdReach}(M)$ is infinite even if N is bounded and M is a finite set of markings.

To speedup convergence of the algorithms implementing the functions BwdReach(M) and FwdReach(M), or just to make it possible that the algorithm computing BwdReach

terminates, we can often use some *care set* C to limit states the algorithms have to explore. We define the set $M' = \text{LimFwdReach}(M, C)$ inductively:

1. The set M' contains all markings $m' \in (M \cap C)$.
2. If M' contains a marking m', then it also contains all markings

$$\{m'' \in C \mid \exists t \in T : m' \xrightarrow{t} m''\}.$$

Notice that we have defined this set in such a way that it contains only markings reachable from the set $M \cap C$ over paths in which every marking belongs to the set C. Later, we shall need the limited reachability analysis in the implementation of algorithms for the symbolic decomposition of sets of states into strongly connected components and in model checking algorithms.

Algorithms 8, 9 and 11 can be easily adapted to implement the limited reachability analysis. The set *Reached* must be intersected with the set C at the beginning of the computation, hereafter, every set returned by the function Fire or FireUnion must be intersected with C as well. It is easy to see that this strategy can not be applied to the variation of Algorithm 11 in which we employ the function FireFixp. A naive intersection of the set returned by FireFixp with C can not guaranty that we do not stray from paths on which every marking belongs to C. Nevertheless, we can still use the function FireFixp for the computation of fixpoints with respect to *forward-safe* transitions. We refine the classification of transitions suggested in [CS03].

Definition 36
Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs, $t \in T$ be some transition, and let M be some set of markings of N. The transition t can be classified as follows:

1. t is *forward-safe* with respect to M if its firing can not lead from a state in M to a state not in in M: $\text{Fire}(M) \subseteq M$.
2. t is *forward-unsafe* with respect to M if its firing can lead from a state in M to a state not in M: $\text{Fire}(M) \setminus M \neq \emptyset$.
3. t is *forward-dead* with respect to M if there is no state in M from which its firing leads to a set in M: $\text{Fire}(M) \cap M = \emptyset$.

The notions of *backward-safe*, *backward-unsafe*, and *backward-dead* transitions are defined analogously.

Consider Algorithm 13 which implements the limited reachability analysis. First, all transitions of the net are classified, initializing boolean arrays *safe* and *dead*. Transitions that are forward-dead with respect to the set C can be skipped. The function

Algorithm 13 (Limited forward reachability with saturation)

```
1  func LimFwdReach (M, C)
2
3     forall t ∈ T do     /* Classifying transitions */
4        dead[t] := (Fire(C) ∩ S = ∅)
5        safe[t] := (Fire(C) \ S = ∅)
6     od
7
8     Reached := M ∩ C
9     i := 1
10    repeat
11       Old := Reached
12       if ¬dead[t_σi] then
13          if safe[t_σi] then
14             Reached := FireFixp(Reached, t_σi)
15          else
16             repeat
17                Old2 := Reached
18                Reached := FireUnion(Reached, Reached, t) ∩ C
19             until Reached = Old2
20          fi
21       fi
22       if Reached = Old then
23          i := i + 1
24       else
25          j := FirstDep(t_σi)
26          if j = i then i := i + 1 else i := j fi
27       fi
28    until i = |T| + 1
29    return Reached
30 end
```

FireFixp is used to compute a fixpoint of the set *Reached* with respect to t when t is a forward-safe transition. Note here that all states generated by forward-safe transitions are guaranteed to belong to the set C. For forward-unsafe transitions, fixpoints are computed using the function FireUnion, in this case sets returned by this function must be intersected with C. As usually, analogously to LimFwdReach, we implement also the complementary function LimBwdReach.

If we want to make a backward reachability analysis, usually $\mathcal{R}_N(m_0)$ is the first choice to be used as a care set, as in many cases $\mathcal{R}_N(m_0)$ can be computed very efficiently using the saturation algorithm. If computation of $\mathcal{R}_N(m_0)$ must be unconditionally avoided, we can try to employ the structural methods of the Petri nets theory. Traps and P-invariants can be used to construct an *approximation* of $\mathcal{R}_N(m_0)$, which can be used then as a care set for the backward reachability analysis. Such techniques have been discussed in [LR95, Cor98, PCP99].

Example 17

Consider a P/T net in Fig. 4.6. From the existence of two minimal semipositive P-invariants $\vec{y_1} = (1, 1, 0, 1)$ and $\vec{y_2} = (2, 0, 1, 4)$ follows that for all reachable markings m holds

$$\begin{aligned} m(p_1) + m(p_2) + m(p_4) &= 3, \\ 2 \cdot m(p_1) + m(p_3) + 4 \cdot m(p_4) &= 8. \end{aligned}$$

Moreover, from the two traps $H_1 = \{p_1, p_2\}$ and $H_2 = \{p_1, p_4\}$ follows

$$m(p_1) > 0 \vee m(p_2) > 0, \quad m(p_1) > 0 \vee m(p_4) > 0.$$

Using this information we can cheaply construct a *coarse* approximation A, a set described by the characteristic function

$$\begin{aligned} \chi_A = \; & (p_1 \in [0, 4) \wedge p_2 \in [0, 3) \wedge p_3 \in [0, 7) \wedge p_4 \in [0, 3)) \wedge \\ & (p_1 > 0 \vee p_2 > 0) \wedge (p_1 > 0 \vee p_4 > 0). \end{aligned}$$

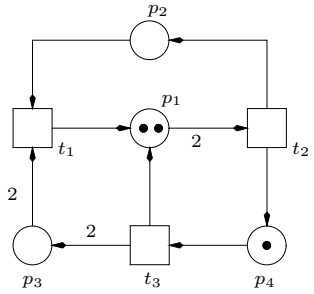


Figure 4.6: A P/T net covered by semipositive P-Invariants

Of course, P-Invariants can be used to construct much finer approximations. Unfortunately, interval logic functions are badly suited for the representation of arithmetic constraints, thus, in general, construction of such approximations can not be done efficiently.

Provided that the set A is finite, we can refine the approximation using a *backward state elimination*:

```
1  repeat
2     Old := A
3     A := Img(A) ∪ {m_0}
4  until A = Old
```

At each step of the recurrence, all those states that are not reachable from any of the states in A are eliminated from A, except the initial state. Obviously, this procedure can not guaranty that the refined approximation will be the exact state space, as it can not remove states not belonging to the set $\mathcal{R}_N(m_0)$ that build non-trivial strongly connected components. Note also that the procedure can be very expensive when there are many states in A that must be eliminated, but only few states can be removed from A at each iteration.

4.3.4 Heuristics for Variable Ordering

Recall that sizes of decision diagrams depend heavily on the used variable ordering. We use a static variable ordering technique in our ROIDD package, hence, we aspire to get a good variable ordering using the structural information of a Petri net.

It was noticed that decision diagrams tend to be smaller if related variables are close together in the ordering. We can presume that pre- and post-places of a transition depend on each other, our saturation algorithm also suggests that variables assigned to adjacent places of a transition should lie close to each other in the ordering. This idea is exploited in a simple greedy algorithm proposed in [Noa99] for the computation of ZBDD variable ordering. The ordering π is constructed in the bottom-up manner. Assuming that $x_1 <_\pi x_2 <_\pi \ldots <_\pi x_{|P|}$, we assign places to the variables starting from the variable $x_{|P|}$. To select a place for a variable x_i, we compute weights $\mathrm{W}(p)$ for all places $p \in P \setminus S$, where S denotes a set of places already assigned to some variable

$$S = \bigcup_{i<j\leq|P|} \mathrm{Pl}(x_j), \qquad \mathrm{W}(p) := \frac{\sum\limits_{t\in\bullet p} \frac{|\bullet t\cap S|}{|\bullet t|} + \sum\limits_{t\in p\bullet} \frac{|t\bullet\cap S|}{|t\bullet|}}{|\bullet p \cup p\bullet|}.$$

A place p with the highest weight $\mathrm{W}(p)$ is assigned then to the variable x_i.

We notice also that moving variables that have a large number of different values down in the ordering can decrease a breadth of an ROIDD and reduce its size. P-invariants and traps can help finding places that potentially induce such variables

(recall Example 17). Moreover, we can assume that places belonging to one P-invariant depend on each other. These facts can be used to adjust the weighting function W.

The described heuristics allow us to compute automatically quite good ROIDD variable orderings for most of the nets we face. For the exceptional cases, we provide in our tool a possibility to fine tune manually the computed ordering.

4.4 Symbolic SCC Decomposition

Decomposing a graph into its strongly connected components (SCCs) is a fundamental graph problem and has many applications in the analysis of different properties. For example, recall that liveness and reversibility of a Petri net can be decided by analyzing terminal SCCs of its reachability graph. The classic algorithm for the SCC decomposition is the Tarjan's algorithm [Tar72]. It is an *explicit* algorithm that has to consider every node of a graph individually. Hence, it is not feasible for large graphs.

The first symbolic algorithms for SCC decomposition were based on the computation of the transitive closure (TC) of the transition relation [MMB93]. This operation is often very expensive and algorithms based on the reachability analysis [XB98, XB99, BGS00] were shown to be superior over the TC-algorithms.

In this section we shall discuss efficient reachability-analysis-based SCC decomposition algorithms for Petri nets. First, we have to introduce several notations and discuss properties of SCCs.

4.4.1 Properties of SCCs

Definition 37 (SCC-closed set)
Let $G = [V, E]$ be a directed graph. We denote with $\mathcal{C}(G)$ a set of all SCCs of G. A set of nodes $U \subseteq V$ is called *SCC-closed* if no SCC intersects both U and its complement $\overline{U}$

$$\nexists\, U' \in \mathcal{C}(G) : U' \cap U \neq \emptyset \ \wedge \ U' \setminus U \neq \emptyset.$$

Lemma 5
Let $G = [V, E]$ be a directed graph, and let U and U' be two SCC-closed sets. A set $D = U \setminus U'$ is SCC-closed.

Proof: If $U \cap U' = \emptyset$, then $D = U$ and D is SCC-closed. If $U \cap U' \neq \emptyset$, then either $U \subseteq U'$ or $U' \subseteq U$. If $U \subseteq U'$, then $D = \emptyset$ and D is obviously SCC-closed. Let us assume that $U' \subset U$. Suppose that D is not SCC-closed, then there must exist an SCC $C \in \mathcal{C}(G) : C \cap D \neq \emptyset \ \wedge \ C \setminus D \neq \emptyset$. Since U is SCC-closed we have $C \subseteq U$ and $C \setminus D \subseteq U$. Since $U = D \cup U'$ then $C \setminus D \subseteq U'$. This implies $C \cap U' \neq \emptyset$, which contradicts to that U' is SCC-closed. □

Definition 38 (Forward and Backward Sets)
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

- A set $\mathcal{F}(v) = \{ v' \in V \mid v \xrightarrow{*} v' \}$ is denoted as a *forward set* of v.
- A set $\mathcal{B}(v) = \{ v' \in V \mid v' \xrightarrow{*} v \}$ is denoted a *backward set* of v.

Notice that by the definition a node v belongs both to its forward and its backward sets. The following corollary follows directly from the definition.

Corollary 11
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

- *For all nodes $v' \in \mathcal{F}(v)$ holds $\mathcal{F}(v') \subseteq \mathcal{F}(v)$.*
- *For all nodes $v' \in \mathcal{B}(v)$ holds $\mathcal{B}(v') \subseteq \mathcal{B}(v)$.*

Lemma 6
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G. The sets $\mathcal{F}(v)$ and $\mathcal{B}(v)$ are SCC-closed.

Proof: Suppose that $\mathcal{F}(v)$ is not SCC-closed, then there exists an SCC

$$C \in \mathcal{C}(G) : C \cap \mathcal{F}(v) \neq \emptyset \ \wedge \ C \setminus \mathcal{F}(v) \neq \emptyset.$$

Let a node $v' \in V$ belong to the set $C \cap \mathcal{F}(v)$ and $v'' \in V$ belong to the set $C \setminus \mathcal{F}(v)$. Since C is an SCC we have $v' \xrightarrow{*} v''$ and $v'' \in \mathcal{F}(v')$. From $v' \in \mathcal{F}(v)$ and Corollary 11 follows $v'' \in \mathcal{F}(v)$, which contradicts to the assumption $v'' \in C \setminus \mathcal{F}(v)$. Using similar considerations we can show that $\mathcal{B}(v)$ is also SCC-closed. □

From Lemmas 5 and 6 follows the following corollary.

Corollary 12
Let $G = [V, E]$ be a directed graph, and let $v, v' \in V$ be some nodes of G. The sets $\mathcal{F}(v) \setminus \mathcal{F}(v')$ and $\mathcal{B}(v) \setminus \mathcal{B}(v')$ are SCC-closed.

Theorem 2
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G. A set $C = \mathcal{F}(v) \cap \mathcal{B}(v)$ is an SCC of G.

Proof: Suppose that C is not an SCC of G. We have to consider two cases then.

1. There exist nodes $v', v'' \in C : v' \not\xrightarrow{*} v''$ or $v'' \not\xrightarrow{*} v'$. If $v' \not\xrightarrow{*} v''$, then $v'' \notin \mathcal{F}(v')$ which by Corollary 11 implies that $v'' \notin \mathcal{F}(v)$. This contradicts to the definition $C = \mathcal{F}(v) \cap \mathcal{B}(v)$. Analogously, if $v'' \not\xrightarrow{*} v'$, then $v'' \notin \mathcal{B}(v')$, which implies that $v'' \notin \mathcal{B}(v)$ and contradicts again to the definition of C.

2. There exist at least one node $u \in \overline{C} : \forall u' \in C\ u' \xrightarrow{*} u$ and $u \xrightarrow{*} u'$. As $v \in C$ then $v \xrightarrow{*} u$ and $u \xrightarrow{*} v$, which contradicts to the assumption $u \in \overline{\mathcal{F}(v) \cap \mathcal{B}(v)}$.

□

Notice that if $\mathcal{F}(v) \cap \mathcal{B}(v) = \{ v \}$, then the set $\{ v \}$ is a trivial SCC when $v \not\longrightarrow v$, nontrivial otherwise.

Definition 39 (Recurrent and transient nodes)
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

1. v is denoted as a *recurrent node* if and only if $\forall v' \in V : v \xrightarrow{*} v'$ holds also $v' \xrightarrow{*} v$.

2. v is denoted as *transient* if and only if $\exists v' \in V : v \xrightarrow{*} v'$, but $v' \not\xrightarrow{*} v$.

Obviously, every recurrent node belongs to some *terminal* SCC of G. Transient nodes are those, not belonging to any terminal SCC of G. The following corollary follows directly from the definition.

Corollary 13
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

1. *v is recurrent if and only if $\mathcal{F}(v) \subseteq \mathcal{B}(v)$.*

2. *v is transient if and only if $\mathcal{F}(v) \not\subseteq \mathcal{B}(v)$.*

Theorem 3
Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G. If v is transient, then all nodes in $\mathcal{B}(v)$ are transient. If v is recurrent, then $\mathcal{F}(v)$ is a terminal SCC and a set $\mathcal{B}(v) \setminus \mathcal{F}(v)$ (if not empty) contains only transient nodes.

Proof: Suppose v is transient. By Corollary 13 there exists $v' \in \mathcal{F}(v) : v' \notin \mathcal{B}(v)$. Suppose $v'' \in \mathcal{B}(v)$, then $v' \in \mathcal{F}(v'')$ as $\mathcal{F}(v) \subseteq \mathcal{F}(v'')$ by Corollary 11. On other hand $\mathcal{B}(v'') \subseteq \mathcal{B}(v)$ since $v'' \in \mathcal{B}(v)$. Therefore, we have a node $v' \in \mathcal{F}(v'')$, but $v' \notin \mathcal{B}(v'')$. This implies $\mathcal{F}(v'') \not\subseteq \mathcal{B}(v'')$ and v'' is transient by Corollary 13.

Suppose now v is recurrent. By Corollary 13 $\mathcal{F}(v) \subseteq \mathcal{B}(v)$, so $\mathcal{F}(v) \cap \mathcal{B}(v) = \mathcal{F}(v)$ and from Theorem 2 follows that $\mathcal{F}(v)$ is an SCC. We have to show now that $\mathcal{F}(v)$ is a terminal SCC. By Corollary 11 for all nodes $v' \in \mathcal{F}(v)$ holds $\mathcal{F}(v') \subseteq \mathcal{F}(v)$, which implies that $\mathcal{F}(v)$ is terminal.

Finally, suppose that v is recurrent, $\mathcal{F}(v)$ is a terminal SCC and $\mathcal{B}(v) \setminus \mathcal{F}(v) \neq \emptyset$. Let $v' \in \mathcal{B}(v) \setminus \mathcal{F}(v)$, then v' is transient by the definition, as $v' \xrightarrow{*} v$, but $v \not\xrightarrow{*} v'$. □

Now we can proceed to discuss SCC decomposition algorithms. An algorithm enumerating terminal SCCs is worth a special consideration, as different analysis techniques rely only on terminal SCCs.

4.4.2 Computation of Terminal SCCs

Consider Algorithm 14, it is an adapted (to our needs and notations) version of the algorithm introduced in [XB98] for the state classification of finite-state Markov chains. The algorithm aims at the enumeration of terminal SCCs in a directed graph $G = [V, E]$. The set V' contains nodes, not yet considered by the decomposition procedure. At the beginning of each iteration we take some random node v from V' and compute its backward and forward sets in the graph induced by the nodes of V'. Due to Corollary 13 and Theorem 3 the set F is a terminal SCC if $F \subseteq B$. Found terminal SCCs are reported using the function ReportTerminalSCC. Nodes in the set B do not need to be considered any more, as they either belong to the found terminal SCC or are transient due to Theorem 3. Hence, the set B is removed from V'. The iteration terminates when there are no more nodes in V' to be considered. The termination is guaranteed, as the set V' is initially finite and at least one node is removed from V' during each iteration.

The worst case for the algorithm occurs if at every iteration the backward set of the taken node v contains only this node while its forward set contains all other nodes in V'. In this case only v is removed from V' and exactly $|V|$ iterations must be made. Assuming that computation of the sets $\mathcal{F}(v)$ and $\mathcal{B}(v)$ requires linear number of steps, we conclude that the complexity of the algorithm is $\mathcal{O}(|V|^2)$.

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs and let S be some finite set of its markings. Algorithm 15 aims at the efficient enumeration of terminal SCCs in S. We notice first that the set containing trivial terminal SCCs of S can be easily computed as $D = S \setminus \mathrm{PreImg}(S)$. Due to Theorem 3 we can conclude that for every state $s \in D$, states in the set $\mathcal{B}(s) \setminus \{s\}$ are all transient. Suppose that a reachability graph of a net has a form of the graph in Fig. 4.7 and we want to find all terminal SCCs in $\mathcal{R}_N(m_0)$. We compute first the set

$$D = S \setminus \mathrm{PreImg}(S) = \{m_0, \ldots, m_{12}\} \setminus \{m_0, \ldots, m_9, m_{11}, m_{12}\} = \{m_{10}\}.$$

Markings $m_0, \ldots, m_8$ that belong to the set $\mathcal{B}(m_{10})$ are all transient and must not be considered any more.

To compute the sets B and F we use efficient saturation-based reachability functions LimFwdReach and LimBwdReach. The set S can be used to limit the backward reachability analysis. As we are interested only in such sets F that $F \subseteq B$, we can use the set B to limit the forward reachability analysis. As we do not compute the whole set $\mathcal{F}(s)$, we have to check if the found set is really a terminal SCC, i.e. $\mathrm{Img}(F) \setminus F = \emptyset$.

The worst case for the algorithm still occurs if at every iteration the backward set of the taken state s contains only this state. Though we avoid the quadratic complexity of the original algorithm due to the usage of the limited analysis, eliminating only one trivial SCC per iteration is still inefficient. We can prune trivial SCCs more efficiently using the *forward trimming*:

Algorithm 14 (Enumeration of terminal SCCs in a set of nodes V)

```
1  proc TerminalSCCs (V)
2     V' := V
3     while V' ≠ ∅ do
4        v := oneof (V')
5        F := F(v)
6        B := B(v)
7        if F \ B = ∅ then ReportTerminalSCC(F) fi
8        V' := V' \ B
9     od
10 end
```

Algorithm 15 (Enumeration of terminal SCCs in a set of markings S)

```
1  proc TerminalSCCs (S)
2     D := S \ PreImg(S)
3     ReportTrivialTerminalSCCs(D)
4     B := LimBwdReach(D, S)
5     S := S \ B
6     if S = ∅ then return fi
7     while S ≠ ∅ do
8        s := Pick(S)
9        B := LimBwdReach({s}, S)
10       F := LimFwdReach({s}, B)
11       if Img(F) \ F = ∅ then
12          ReportTerminalSCC(F)
13       fi
14       S := S \ B
15    od
16 end
```

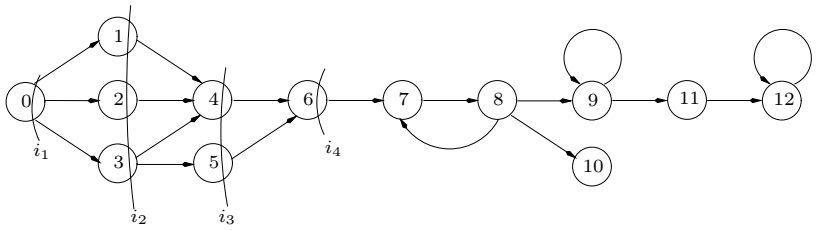


Figure 4.7: A graph used in the explanation of Algorithm 15

· repeat
· *Old* := *S*
· *S* := *S* ∩ Img(*S*)
· until *S* = *Old*

This procedure deletes all states that can not be reached from a state in some non-trivial SCC. For example, consider again Fig. 4.7, but suppose now that the marking m_{10} does not belong to the graph. In this case the markings $m_0, \ldots, m_8$ would not be deleted as predecessors of a trivial SCC. The forward trimming would delete the markings $m_0, \ldots, m_6$ in four iterations.

Computation of Img is usually denoted in symbolic algorithms as a step. It is easy to see that the number of steps in Algorithm 15 is limited by $\mathcal{O}(|S|)$ due to the usage of the functions for the limited reachability analysis.

4.4.3 Lockstep Algorithm

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs and let S be some finite set of its markings. Algorithm 16 can be used to decompose S into strongly connected components. It is an adapted to our needs version of the algorithm proposed in [BGS00]. The algorithm is based on the facts that the forward set and the backward set of a state are SCC-closed (recall Corollary 12) and that intersection of these sets forms an SCC (due to Theorem 2).
First, a random state s is taken from S. Then the sets F and B are computed simultaneously (that's why the algorithm is named Lockstep). It is possible, that computation of one of the sets requires less iterations. Let us assume, for example, that after the termination of the first cycle $F = \mathcal{F}(s)$, but $B \neq \mathcal{B}(s)$. As an SCC containing s is $C = \mathcal{F}(s) \cap \mathcal{B}(s)$, we can limit the computation of the set B by F. When *bFront* $= \emptyset$ the set B includes all states of the SCC containing s. We report the found SCC $C = F \cap B$ and then recur. For the recursion we split the set S into three subsets: $S \setminus F$, $F \setminus C$, and C and recur on the first two. The worst case complexity bound of the algorithm is $\mathcal{O}(\log(|S|) \cdot |S|)$ [BGS00].

Obviously, symbolic enumeration of trivial SCCs can not be efficient. When we are interested in non-trivial SCCs (which is quite often the case), then before picking a state on the line *4*, we prune all states that can not be reached from a state in some non-trivial SCC of S and can not reach a non-trivial SCC in S. In addition to the forward trimming defined above, we apply also the *backward trimming*:

· repeat
· *Old* := *S*
· *S* := *S* ∩ PreImg(*S*)
· until *S* = *Old*

Algorithm 16 (Lockstep)

```
1  proc Lockstep(S)
2     if S = ∅ then return fi
3
4     s := Pick(S)
5     F := {s}; fFront := {s}
6     B := {s}; bFront := {s}
7
8     while fFront ≠ ∅ ∧ bFront ≠ ∅ do
9        fFront := (Img(fFront) ∩ S) \ F
10       F := F ∪ fFront
11       bFront := (PreImg(bFront) ∩ S) \ B
12       B := B ∪ bFront
13    od
14
15    if fFront = ∅ then
16       Converged := F
17       while bFront ≠ ∅ do
18          bFront := (PreImg(bFront) ∩ F) \ B
19          B := B ∪ bFront
20       od
21    else
22       Converged := B
23       while fFront ≠ ∅ do
24          fFront := (Img(fFront) ∩ B) \ F
25          F := F ∪ fFront
26       od
27    fi
28
29    C := F ∩ B
30    reportSCC(C)
31    Lockstep(Converged \ C)
32    Lockstep(S \ Converged)
33 end
```

The Lockstep algorithm is based on the breath-first search, but, obviously, its correctness does not depend on the exploration strategy. To improve the efficiency of the algorithm, we rewrite it using our saturation techniques.

4.5 Analysis of Basic Petri Nets Properties

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs, and let $\mathcal{R}_N(m_0)$ be a set of reachable from m_0 markings. Recall that with χ_{E_t} we have denoted a characteristic function for a set of markings in which a transition $t \in T$ can be enabled. Correspondingly, χ_{D_t} is a characteristic function for markings in which t can not be enabled

$$\chi_{E_t} = \bigwedge_{p_i \in {}^\bullet t} (p_i \geq t^-(p_i) \ \wedge \ p_i < t_I^-(p_i) \ \wedge \ p_i \geq t_R^-(p_i)), \quad \chi_{D_t} = \neg\chi_{E_t}.$$

Let $\mathcal{D}_N$ be a set of all *potentially* dead markings of N, its characteristic function can be defined as $\chi_{\mathcal{D}_N} = \bigwedge_{t\in T} \chi_{D_t}$. If we are only interested whether dead markings are reachable from m_0, then, as mentioned in section 4.2.3, we can use the modified versions of FwdReach or BwdReach to check the reachability on-the-fly. Assuming that the set $\mathcal{R}_N(m_0)$ is already computed, a set of *reachable from* m_0 dead markings can be computed as $\mathcal{D}_N(m_0) = \mathcal{R}_N(m_0) \cap \mathcal{D}_N$. Alternatively, we can compute this set as $\mathcal{D}_N(m_0) = \mathcal{R}_N(m_0) \setminus \mathrm{PreImg}(\mathcal{R}_N(m_0))$, avoiding the construction of the set $\mathcal{D}_N$.

The set of markings from which m_0 is reachable can be computed as $\mathrm{BwdReach}(m_0)$. Hence, to check reversibility of N we can check whether this set contains all reachable from m_0 markings or, equally, if

$$\mathcal{R}_N(m_0) \equiv \mathrm{LimBwdReach}(m_0, \mathcal{R}_N(m_0)).$$

Analogously, to check liveness of a transition t we can check if

$$\mathcal{R}_N(m_0) \equiv \mathrm{LimBwdReach}(E_t, \mathcal{R}_N(m_0)).$$

Recall that liveness of transitions can be also decided using terminal SCCs of $\mathcal{RG}_N$, thus, we can employ Algorithm 15 which enumerates terminal SCCs. In the function ReportTerminalSCC we check if the found terminal SCC C contains markings in which t can fire: $C \cap E_t \neq \emptyset$, t is not live if we meet some SCC C such that $C \cap E_t = \emptyset$. This approach is much more efficient when liveness of many transitions must be decided (for example, when we are deciding liveness of the whole net N) and $\mathcal{RG}_N$ contains few terminal SCCs. This is very often a case in Petri net models of *reactive systems* [MP92] which are usually designed not to terminate. Of course, reversibility can be also decided using SCC decomposition.

Surely, before checking reversibility or liveness, we first make a cheaper test if the net has reachable dead markings. If $\mathcal{D}_N(m_0) \neq \emptyset$, then N can not be reversible and has no live transitions.

4.6 Closing Remark

The key idea underlying symbolic methods is to represent sets of states concisely using their characteristic functions and to manipulate them as if they were in bulk. For sets of markings describable by interval logic functions, reasoning in terms of sets and set operations is isomorphic to reasoning in terms of interval logic functions and logic operations on them. As ROIDDs provide a canonical form representation for interval logic functions, they can be used as an efficient data structure for storage and manipulations of large sets of markings. Moreover, ROIDDs allow a quite natural implementation of the special operations needed in symbolic algorithms.

Though small decision diagrams can encode large sets of states, not every large set of states can be encoded by a small decision diagram. Straying from the traditional breath-first strategy in the exploration of state spaces can improve efficiency of the reachability analysis. We have studied a number of iteration strategies and techniques to reduce sizes of intermediate diagrams and proposed a new saturation-based approach, which exploits the structure of ROIDDs and the structure of k-bounded Petri nets. It manages to keep sizes of intermediate diagrams smaller than other approaches and can significantly improve efficiency of the reachability analysis.

Algorithms for the symbolic SCC decomposition are based on the observation that an SCC containing some state can be computed as an intersection of a set of states that can reach this state with a set of states that are reachable from it. An algorithm introduced in [XB98] for the state classification of finite-state Markov chains can be adapted to the enumeration of terminal SCCs in sets of markings of Petri nets. Moreover, it benefits also from the new saturation strategy.

An efficient implementation of the reachability analysis allows an efficient check of the basic Petri net properties. Usually, not only basic net properties are an object of interest. In the next chapter we shall discuss how a wide class of properties can be specified with *temporal logics* and verified using *model checking*.

5 Temporal Logic and Model Checking

Temporal logics have proven to be useful for specifying properties of concurrent systems as they can describe the ordering of events in time without introducing the time explicitely. Originally, temporal logics were developed by philosophers to study the way that time is used in natural language arguments. They were first suggested for the specification of properties and verification of concurrent programs in [Pnu77]. The introduction of temporal-logic *model checking* algorithms [CE81] allowed this type of reasoning to be automated.

Model checking was developed as a technique for the formal verification of hardware and software systems. It has been proven to be a successful method, frequently used to uncover well-hidden bugs in complex sequential circuit designs and communication protocols. Model checking provides means to check whether a *finite state model* of a system satisfies a given specification. The benefit of this restriction is that the verification can be performed fully automatically. The procedure uses normally an exhaustive exploration of all possible states of the model to determine whether it satisfies a property expressed in a temporal logic. An important feature of a model checker is that in case of a negative result, the user is often provided with a trace which can be used as a counterexample for the checked property. Counterexamples can help to determine whether the negative result was caused by an error in the system or comes from incorrect modeling or an incorrect specification. Today, applications of model checking are not limited to the hardware and software verification, it is often used to study properties of models arising in different, not necessary technical areas [CF03]. Perfect monographs on model checking are [CGP01] and [BBF$^+$01].

There are two possible views regarding the nature of time, which induce two types of temporal logics [Lam80]. In *linear* temporal logics, time is treated as if each moment has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and can be regarded as formulas describing a behavior of a single computation of a system. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logics are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of the system. The discussion of relative merits of linear versus branching time has a long history [Pnu77, Lam80, EL85, EH86, Var01].

In the *Linear Temporal Logic* (*LTL*) [Pnu80], formulas are composed of *atomic propositions*, usual boolean operators, and *temporal operators*: **X** ("next"), **U** ("until"), **F** ("fi-

nally"), **G** ("globally"), and **R** ("release"). The branching temporal logic CTL* [EH86] augments LTL by the *path quantifiers* **E** ("there exists a computation") and **A** ("for all computations"). The branching *Computation Tree Logic* (*CTL*) [CE81] is a fragment of CTL* in which every temporal operator must be preceded by a path quantifier. CTL and LTL are expressively incomparable [Lam80], for example, the LTL formula $\mathbf{FG}\varphi$ can not be expressed in CTL, while the CTL formula $\mathbf{AGEF}\varphi$ is not expressible in LTL. The expressiveness of CTL can be increased if its semantics is modified to handle *fairness*. A *fairness constraint* can be an arbitrary set of states. Given a set of fairness constraints $\mathcal{F}$, a *fair path* must contain an element of each fairness constraint infinitely often. The path quantifiers of CTL are restricted then to fair paths.

Given a *labeled state transition graph* M and a CTL formula φ, the model checking problem is to decide whether φ holds in all initial states of M. The classic algorithm to solve the problem is a bottom-up labeling procedure [CES86]. It labels all states of M by subformulas of φ, starting from the innermost formulas and proceeding such that when labeling by some formula, all its subformulas are already processed. Depending on a formula, labeling can involve a backward reachability analysis and SCC decomposition. The procedure requires time $\mathcal{O}(|M| \cdot |\varphi|)$. The complexity of the model checking algorithm for CTL with fairness constraints is $\mathcal{O}(|M| \cdot |\varphi| \cdot |\mathcal{F}|)$.

Given a labeled state transition graph M and an LTL formula φ, the model checking problem is to decide whether φ holds in all computations of M. The dominant approach to LTL model checking is the *automata theoretic approach* [VW86, Var96]. The key idea of this approach is that, given an LTL formula, it is possible to construct a *Büchi automaton* [Büc60] that accepts all computations that satisfy this formula. This allows to reduce the model checking problem to an automata theoretic problem as follows: (1) construct the automaton $A_{\neg\varphi}$ that corresponds to the negation of the formula φ, (2) take the product of M and $A_{\neg\varphi}$ to obtain the automaton $A_{M,\varphi}$, and (3) check if the automaton $A_{M,\varphi}$ accepts some input. If it does not, then φ holds in M. The problem can be solved in time $\mathcal{O}(|M| \cdot 2^{|\varphi|})$ [LP85]. This is considered acceptable since the size of the specification is typically significantly smaller than the size of M.

The main drawback of model checking is the *state explosion problem*. A number of techniques have been proposed to combat the problem. The most successful approaches are partial-order reductions [Val91, God91, Pel94] and symbolic methods [BCM$^+$90, McM92]. The original CTL model checking algorithm [CES86] can be adapted to the case when the labeled state transition graph is represented symbolically. The symbolic CTL model checking algorithm [BCM$^+$90] is based on computing fixpoints of *predicate transformers*. Symbolic LTL model checking can be reduced to symbolic CTL model checking with fairness constraints [CGH94].

In this chapter (based on [CGP01]) we introduce the temporal logics CTL and LTL and discuss corresponding model checking algorithms.

5.1 Temporal Logics

5.1.1 Computation Tree Logic CTL*

Formulas of the *Computation Tree Logic CTL** [EH86] are composed of *atomic propositions*, usual boolean operators, *temporal operators*, and *path quantifiers*. Atomic propositions are used to make statements about states of a system. These propositions are elementary statements which have well-defined truth values in every given state. The meaning of temporal logic formulas is defined with respect to a *labeled state transition graph* or a *Kripke structure* [Kri63].

Definition 40 (Kripke structure)
Let $\mathcal{AP}$ be a set of atomic propositions. A *Kripke structure* is a tuple $M = [S, R, L, S_0]$ where:

- S is a finite set of states.
- $R \subseteq S \times S$ is a total transition relation, hence $\forall s \in S \;\; \exists s' \in S : (s, s') \in R$.
- $L : S \to 2^{AP}$ is a function labeling each state with a set of atomic propositions true in that state.
- $S_0 \subseteq S$ is a set of initial states.

A *computation* or a *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $(s_i, s_{i+1}) \in R \; \forall i \geq 0$. $\pi(i)$ denotes a state s_i in π, π^i denotes a *suffix* of π starting at $\pi(i)$, $\mathcal{P}(s)$ denotes a set of all paths starting at a state $s \in S$.

Conceptually, CTL* formulas can describe properties of *computation trees*. The tree is formed by designating a state in a Kripke structure as the initial state and then unwinding the structure into an infinite tree with the designated state as the root, as illustrated in Fig. 5.1.

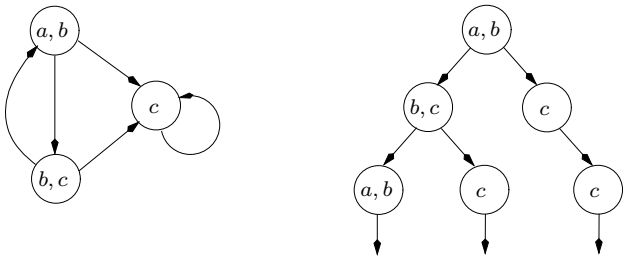


Figure 5.1: Unwinding Kripke structure to get a computation tree

Temporal operators describe properties of a path through the computation tree. There are five basic so-called *future-tense* temporal operators:

- $\mathbf{X}\phi$ ("next time ϕ") requires that the property ϕ holds in the next state of the path.
- $\mathbf{F}\phi$ ("finally" ϕ or "eventually ϕ") operator is used to assert that the property ϕ will hold at some future state of the path.
- $\mathbf{G}\phi$ ("globally ϕ") specifies that the property ϕ holds in the current and all future states of the path.
- $\phi_1\mathbf{U}\phi_2$ ("ϕ_1 until ϕ_2") holds if there is a state in the path where ϕ_2 holds and in every preceding state holds ϕ_1.
- $\phi_1\mathbf{R}\phi_2$ ("ϕ_1 releases ϕ_2") is the logical dual of the $\mathbf{U}$ operator. It requires that the ϕ_2 holds along the path up to and including the first state where ϕ_1 holds. However, ϕ_1 is not required to hold eventually.

Definition 41 (Syntax of CTL*)
There are two types of formulas in CTL*: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let $\mathcal{AP}$ be the set of atomic propositions. The syntax of state formulas is given by the following rules:

- Every atomic propositions $p \in \mathcal{AP}$ is a state formula.
- If ϕ_1 and ϕ_2 are state formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\neg\phi_1$ are state formulas.
- If ϕ is a path formula, then $\mathbf{E}\phi$ and $\mathbf{A}\phi$ are state formulas.

The syntax of path formulas is defined by the following rules:

- If ϕ is a state formula, then ϕ is also a path formula.
- If ϕ_1 and ϕ_2 are path formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\neg\phi_1$ are path formulas.
- If ϕ_1 and ϕ_2 are path formulas, then $\mathbf{X}\phi_1, \mathbf{F}\phi_1, \phi_1\mathbf{U}\phi_2, \phi_1\mathbf{R}\phi_2$, and $\mathbf{G}\phi_1$ are path formulas.

We introduce the constants *True* and *False* as abbreviations for $p \vee \neg p$ and $p \wedge \neg p$.

Definition 42 (Semantics of CTL*)
The semantics of CTL* is defined with respect to a Kripke structure M. If ϕ is a state formula, then $M, s \models \phi$ means that ϕ holds in the state s. If γ is a path formula, then $M, \pi \models \gamma$ means that γ holds along the path π. Assuming that p is an atomic proposition, ϕ, ϕ_1, and ϕ_2 are state formulas and γ, γ_1, and γ_2 are path formulas, the relation $\models$ is defined inductively as follows:

1. $M, s \models p$ if $p \in L(s)$.

2. $M, s \models \neg\phi$ if $M, s \not\models \phi$.
3. $M, s \models \phi_1 \vee \phi_2$ if $M, s \models \phi_1$ or $M, s \models \phi_2$.
4. $M, s \models \phi_1 \wedge \phi_2$ if $M, s \models \phi_1$ and $M, s \models \phi_2$.
5. $M, s \models \mathbf{E}\gamma$ if $\exists\pi \in \mathcal{P}(s)$ such that $M, \pi \models \gamma$.
6. $M, s \models \mathbf{A}\gamma$ if $\forall\pi \in \mathcal{P}(s)$ $M, \pi \models \gamma$.
7. $M, \pi \models \phi$ if $M, \pi(0) \models \phi$.
8. $M, \pi \models \neg\gamma$ if $M, \pi \not\models \gamma$.
9. $M, \pi \models \gamma_1 \vee \gamma_2$ if $M, \pi \models \gamma_1$ or $M, \pi \models \gamma_2$.
10. $M, \pi \models \gamma_1 \wedge \gamma_2$ if $M, \pi \models \gamma_1$ and $M, \pi \models \gamma_2$.
11. $M, \pi \models \mathbf{X}\gamma$ if $M, \pi^1 \models \gamma$.
12. $M, \pi \models \mathbf{F}\gamma$ if $\exists k \geq 0$ such that $M, \pi^k \models \gamma$.
13. $M, \pi \models \mathbf{G}\gamma$ if $\forall i \geq 0$ $M, \pi^i \models \gamma$.
14. $M, \pi \models \gamma_1 \mathbf{U} \gamma_2$ if $\exists k \geq 0$ such that $M, \pi^k \models \gamma_2$ and $0 \leq j < k$ $M, \pi^j \models \gamma_1$.
15. $M, \pi \models \gamma_1 \mathbf{R} \gamma_2$ if $\forall j \geq 0$, if for every $i < j$ $M, \pi^i \not\models \gamma_1$, then $M, \pi^j \models \gamma_2$.

Any CTL* operator can be expressed with help of $\mathbf{X}, \mathbf{U}$, and $\mathbf{E}$:

- $\mathbf{F}\phi \equiv \mathit{True}\mathbf{U}\phi$
- $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$
- $\phi_1 \mathbf{R} \phi_2 \equiv \neg(\neg\phi_1 \mathbf{U} \neg\phi_2)$
- $\mathbf{A}(\phi) \equiv \neg\mathbf{E}(\neg\phi)$.

5.1.2 CTL and LTL

Linear Temporal Logic (*LTL*) [Pnu80] and *Computation Tree Logic* (*CTL*) [CE81] are the two most commonly supported temporal logics in model checking tools. In the *branching-time* logic CTL, the temporal operators quantify over the paths that are possible from a given state. In the *linear-time* logic LTL, operators are provided for describing events along a single computation.

Definition 43 (Syntax of LTL)
Let $\mathcal{AP}$ be the set of atomic propositions. The syntax of LTL formulas is given by the following rules:

- Every atomic proposition $p \in \mathcal{AP}$ is an LTL formula.
- If ϕ_1 and ϕ_2 are LTL formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\neg\phi_1$ are LTL formulas.
- If ϕ_1 and ϕ_2 are LTL formulas, then $\mathbf{X}\phi_1, \mathbf{F}\phi_1, \phi_1\mathbf{U}\phi_2, \phi_1\mathbf{R}\phi_2$, and $\mathbf{G}\phi_1$ are LTL formulas.

Each LTL formula ϕ corresponds to a CTL* formula $\mathbf{A}\phi$, where ϕ is a path formula in which state subformulas can be build only from atomic propositions connected by boolean operators.

CTL can be seen a restricted fragment of CTL* in which each of the temporal operators **X**, **F**, **U**, **G**, and **R** must be directly preceded by a path quantifier. Formally, CTL is the subset of CTL* that is obtained by defining the syntax of path formulas as follows:

- If ϕ_1 and ϕ_2 are *state* formulas, then $\mathbf{X}\phi_1, \mathbf{F}\phi_1, \phi_1\mathbf{U}\phi_2, \phi_1\mathbf{R}\phi_2$, and $\mathbf{G}\phi_1$ are path formulas.

The need to quantify always over the possible futures noticeably limits the expressivity of CTL. CTL formulas are *state* formulas, the truth of any formula ϕ depends only on the current state and does not depend on the current path[1]. For example, it is not possible to express the property "for all computations, if ϕ_1 holds infinitely often in a computation, then ϕ_2 also holds in this computation", $\mathbf{GF}\phi_1 \rightarrow \phi_2$ in LTL[2], as well as the property "ϕ_2 holds in all those computations in which ϕ_1 holds invariantly", $\mathbf{G}\phi_1 \rightarrow \phi_2$ in LTL. This limitation carries also its benefits, allowing to implement efficient CTL model checking algorithms. Correspondingly, LTL deals only with a set of computations and not in the way these are organized into a tree, hence LTL can not express that at some state it is possible to extend the computation in this or that way. Characteristically, the property "p is always potentially reachable", $\mathbf{AGEF}p$ in CTL, can not be expressed in LTL.

Each of the basic CTL operators can be expressed in terms of $\mathbf{EX}, \mathbf{EU}$, and $\mathbf{EG}$.

- $\mathbf{AX}\phi \equiv \neg\mathbf{EX}(\neg\phi)$
- $\mathbf{EF}\phi \equiv \mathbf{E}[\mathit{True}\mathbf{U}\phi]$
- $\mathbf{AG}\phi \equiv \neg\mathbf{EF}(\neg\phi)$

[1] Defined formally, ϕ is a state formula if for any two paths π_1, π_2, from $\pi_1(i) = \pi_2(j)$ follows that $M, \pi_1(i) \models \phi$ if and only if $M, \pi_2(j) \models \phi$.
[2] We use here the usual abbreviation $a \rightarrow b \equiv \neg a \vee b$.

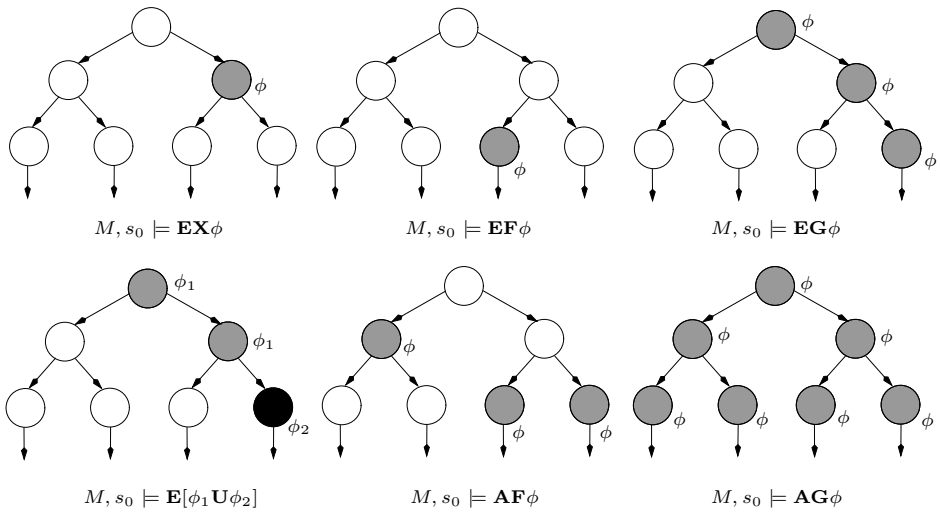


Figure 5.2: Basic CTL operators

- $\mathbf{AF}\phi \equiv \neg\mathbf{EG}(\neg\phi)$
- $\mathbf{A}[\phi_1 \mathbf{R} \phi_2] \equiv \neg\mathbf{E}[\neg\phi_1 \mathbf{U} \neg\phi_2]$
- $\mathbf{A}[\phi_1 \mathbf{U} \phi_2] \equiv \neg\mathbf{E}[\neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)] \wedge \neg\mathbf{EG}(\neg\phi_2)$
- $\mathbf{E}[\phi_1 \mathbf{R} \phi_2] \equiv \neg\mathbf{A}[\neg\phi_1 \mathbf{U} \neg\phi_2]$

Example 18

Some of the basic CTL operators are illustrated in Fig. 5.2. The operators are easiest to understand in terms of the computation tree obtained by unwinding the Kripke structure M with the state s_0 as a root. Several typical CTL and LTL formulas that might arise in verifying concurrent systems are given below. Notice that some properties can be expressed only in one of the temporal logics.

- $\phi_{CTL} = \neg\mathbf{EF}\mathit{Bad},\ \phi_{LTL} = \mathbf{G}(\neg\mathit{Bad})$: The system never reaches the *Bad* state.
- $\phi_{CTL} = \mathbf{AG}(\mathit{Req} \rightarrow \mathbf{AF}\mathit{Ack}),\ \phi_{LTL} = \mathbf{G}(\mathit{Req} \rightarrow \mathbf{F}\mathit{Ack})$: Every time when a request occurs it is eventually acknowledged.
- $\phi_{LTL} = \mathbf{GF}(\neg\mathit{Lost}) \rightarrow \mathbf{G}(\mathit{Sent} \rightarrow \mathbf{F}\mathit{Received})$: Every sent message that is not systematically lost, is finally received.
- $\phi_{CTL} = \mathbf{AGAF}(\mathit{InCS}_i),\ \phi_{LTL} = \mathbf{GF}(\mathit{InCS})$: A process i has a possibility to enter its critical section infinitely often.

- $\phi_{LTL} = \mathbf{GF}(ReqCS_i) \rightarrow \mathbf{GF}(InCS_i)$: If a process i tries to enter its critical section infinitely often, it will also succeed infinitely often.
- $\phi_{CTL} = \mathbf{AGEF}(Start)$: From any state it is possible to get to the *Start* state.

5.1.3 Fairness

As could be seen in the examples above, in many cases we are only interested in the correctness along *fair* paths. For example, when considering communication protocols that operate over channels that may occasionally lose message, we do not wish to consider the case when all sent messages are systematically lost. Alternatively, when verifying some mutual exclusion algorithm, we may wish to consider only those executions in which a process scheduler does not ignore one of the processes forever. Such properties can not be expressed in CTL. In order to deal with fairness in CTL, its semantics must be slightly modified [EL85].

A *fairness constraint* can be an arbitrary set of states, usually described by a formula of the logic. A *fair path* must contain an element of each fairness constraint infinitely often. The path quantifiers of the logic are restricted to fair paths.

Definition 44 (Fair Kripke structure)
Let $\mathcal{AP}$ be a set of atomic propositions. A fair *Kripke structure* is a tuple $M = [S, R, L, S_0, \mathcal{F}]$ where:

- $[S, R, L, S_0]$ is a Kripke structure.
- $\mathcal{F} \subseteq 2^S$ is a set of fairness constraints.

Let $\pi = s_0 s_1 s_2 \ldots$ be a path in M, we define a set $\inf(\pi) = \{s \mid \forall k \geq 0 \; \exists i > k : s_i = s\}$. A path π is called *fair* if $\inf(\pi) \cap F \neq \emptyset$ for every $F \in \mathcal{F}$. We shall write $M, s \models_{\mathcal{F}} \phi$ if ϕ holds in s when the path quantifiers in ϕ are restricted to fair paths.

Example 19
In CTL with fairness constraints, we can express properties that could not be expressed in CTL in the previous example.

- $\phi_{FCTL} = \mathbf{AG}(Sent \rightarrow \mathbf{AF}Received)$, $\mathcal{F} = \{\neg Lost\}$: Every sent message that is not systematically lost, is finally received.
- $\phi_{FCTL} = \mathbf{AGAF}(InCS_i)$, $\mathcal{F} = \{ReqCS_i\}$: If a process i tries to enter its critical section infinitely often, it will also succeed infinitely often.

5.2 Model Checking CTL

Given a Kripke structure $M = [S, R, L, S_0]$ and a CTL formula φ, the model checking problem is to decide whether φ holds in all initial states of M: $M, s \models \varphi \; \forall s \in S_0$.

5.2.1 Explicit Algorithm

We consider now the classic *global* algorithm to solve the CTL model checking problem [CES86]. Let $M = [S, R, L, S_0]$ be a Kripke structure, we shall determine which states in S satisfy the CTL formula φ. The algorithm labels each state s with the set $\mathit{labels}(s)$ of subformulas of φ which are true in s. Initially $\mathit{labels}(s) = L(s)$. The algorithm goes through a series of stages. During the ith stage, subformulas with $i-1$ nested CTL operators are processed. When a subformula is processed, it is added to the labels of each state in which it is true. Once the algorithm terminates, we will have $M, s \models \phi$ if and only if $\phi \in \mathit{labels}(s)$.

Recall that any CTL formula can be expressed using only atomic propositions, boolean operators and temporal operators $\mathbf{EX}, \mathbf{EU}$, and $\mathbf{EG}$. Hence, for the intermediate stages of the algorithm it is sufficient to be able to handle the cases when a formula has one of following forms: $\neg\phi$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\mathbf{EX}\phi$, $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, or $\mathbf{EG}\phi$.

1. For formulas of the form $\neg\phi$, we label those states that are not labeled by ϕ. For $\phi_1 \vee \phi_2$, we label any state that is labeled either by ϕ_1 or ϕ_2. For $\phi_1 \wedge \phi_2$, we label any state that is labeled by ϕ_1 and ϕ_2.

2. For $\mathbf{EX}\phi$, we label every state that has some successor labeled by ϕ.

3. For formulas of the form $\phi = \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ we first find all states that are labeled with ϕ_2 and label them with ϕ. We work then backwards using the converse of R and find all states that can reach states labeled by ϕ_2 along a path in which each state is labeled with ϕ_1. All such states must be labeled by ϕ. This behavior is implemented in Algorithm 17.

4. For formulas of the form $\mathbf{EG}\phi$, let a graph $[S', R']$ be obtained from M by deleting from S those states in which ϕ does not hold and restricting R accordingly. Notice that R' may not be total. Algorithm 18 depends on the observation $M, s \models \mathbf{EG}\phi$ if and only if $s \in S'$ and there exists a path in $[S', R']$ that leads from s to some node t in a nontrivial SCC of the graph $[S', R']$.

In order to handle an arbitrary CTL formula φ, we apply the state-labeling algorithm to the subformulas of φ, starting from the shortest, most deeply nested and work outward to include all subformulas of φ. By proceeding in this manner we can guarantee that whenever we process a subformula of φ all its subformulas have already been processed. Finally we can check if all initial states of M are labeled by φ. The algorithm requires time $\mathcal{O}(|\varphi| \cdot (|S| + |R|))$.

Algorithm 17 (Labeling states satisfying $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$)

```
1  proc CheckEU (φ1, φ2)
2     T := {s | φ2 ∈ labels(s)}
3     forall s ∈ T do labels(s) ∪ {E[φ1 U φ2]} od
4
5     while T ≠ ∅ do
6        s := oneof (T)
7        T := T\{s}
8        forall t : R(t, s) do
9           if E[φ1 U φ2] ∉ labels(t) ∧ φ1 ∈ labels(t) then
10             labels(t) := labels(t) ∪ {E[φ1 U φ2]}
11             T := T ∪ {t}
12          fi
13       od
14    od
15 end
```

Algorithm 18 (Labeling states satisfying $\mathbf{EG}\phi_1$)

```
1  proc CheckEG (φ1)
2     S' := {s | φ1 ∈ labels(s)}
3     T := {s | s ∈ nontrivial SCC of S'}
4     forall s ∈ T do labels(s) ∪ {EGφ1} od
5
6     while T ≠ ∅ do
7        s := oneof (T)
8        T := T\{s}
9        forall t : t ∈ S' ∧ R(t, s) do
10          if EGφ1 ∉ labels(t) then
11             labels(t) := labels(t) ∪ {EGφ1}
12             T := T ∪ {t}
13          fi
14       od
15    od
16 end
```

5.2.2 Handling Fairness in Explicit Algorithm

Let $M = [S, R, L, S_0, \mathcal{F}]$ be a fair Kripke structure. An SCC C of M is called *fair* if it intersects all fair sets: $C \cap F \neq \emptyset \; \forall F \in \mathcal{F}$. We consider first how to check formulas of the form $\mathbf{EG}_{fair}\phi$[3]. Let a graph $[S', R']$ be obtained from M by deleting from S those states in which ϕ does not *fairly* hold and restricting R accordingly. A state s satisfies the formula $\mathbf{EG}_{fair}\phi$ if and only if $s \in S'$ and there exists a path in $[S', R']$ that leads from s to some node t in a *fair* nontrivial SCC of the graph $[S', R']$. An algorithm that implements the function CheckFairEG replaces computation of the set T on the line *3* in Algorithm 18 with

$$T := \{s \mid s \in \text{nontrivial } \textit{fair} \text{ SCC of } S'\}.$$

In order to check other CTL formulas with respect to the fair Kripke structure we introduce first an additional atomic proposition *fair*, which is true in a state if and only if there is a fair path starting from this state, $\textit{fair} = \mathbf{EG}_{fair}\, \textit{True}$. A call to the function CheckFairEG(*True*) can be used to label states with the new atomic proposition.

1. To determine if $M, s \models_{\mathcal{F}} p$ for $p \in \mathcal{AP}$ we check $M, s \models p \wedge \textit{fair}$ using the ordinary model checking procedure.
2. To determine if $M, s \models_{\mathcal{F}} \mathbf{EX}\phi$ we check $M, s \models \mathbf{EX}(\phi \wedge \textit{fair})$.
3. To determine if $M, s \models_{\mathcal{F}} \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ we check $M, s \models \mathbf{E}(\phi_1 \mathbf{U}(\phi_2 \wedge \textit{fair}))$.

The algorithm requires time $\mathcal{O}(|\varphi| \cdot (|S| + |R|) \cdot |\mathcal{F}|)$.

5.2.3 Symbolic Algorithm

Recall that in symbolic algorithms operations are performed not on individual states, but on sets of states. The symbolic CTL model checking algorithm employs a *fixpoint characterization* of the temporal logic operators. We shall discuss symbolic algorithms in terms of sets of states of a Kripke structure.

Definition 45

Let $M = [S, R, L, S_0]$ be a Kripke structure, the set 2^S of all subsets of S forms a lattice[4] $\mathcal{L}$ under the set inclusion ordering: $\mathcal{L} = [2^S, \subseteq]$.

- Each element $S' \in \mathcal{L}$ can be thought as a *predicate* on S, where the predicate is viewed as being true for exactly the states in S'.

[3] We shall use this notation to stress the point that the formula $\mathbf{EG}\phi$ must be considered with respect to the fair Kripke structure.

[4] Lattices are introduced in the Appendix.

- A function $\tau : 2^S \to 2^S$ is called a *predicate transformer*.
- A predicate transformer τ is called *monotonic* if $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$.
- A set $S' \subseteq S$ is a *fixpoint* of a predicate transformer τ if $\tau(S') = S'$.

Theorem 4 ([Tar55])
A monotonic predicate transformer τ always has a least fixpoint $\mu Z.\tau(Z)$ and a greatest fixpoint $\nu Z.\tau(Z)$

$$\mu Z.\tau(Z) = \cap\{Z \mid \tau(Z) \subseteq Z\}, \quad \nu Z.\tau(Z) = \cup\{Z \mid \tau(Z) \supseteq Z\}.$$

Proposition 3
It can be shown that for a monotonic predicate transformer $\tau : 2^S \to 2^S$ there always exist integers i_0, j_0 such that $\mu Z.\tau(Z) = \tau^{i_0}(\emptyset)$ and $\nu Z.\tau(Z) = \tau^{j_0}(S)$, where $\tau^i(Z)$ denotes i applications of τ to Z

$$\tau^0(Z) = Z, \quad \tau^{i+1}(Z) = \tau(\tau^i(Z)).$$

If we identify each CTL formula ϕ with the predicate $\{s \mid M, s \models \phi\}$, then each of the basic CTL operators may be characterized as a least or a greatest fixpoint of an appropriate monotonic predicate transformer [CE81]

$$\begin{array}{lcllcl} \mathbf{EF}\phi & = & \mu Z.\, \phi \vee \mathbf{EX}\ Z & \mathbf{E}[\phi_1 \mathbf{U} \phi_2] & = & \mu Z.\, \phi_2 \vee (\phi_1 \wedge \mathbf{EX}\ Z) \\ \mathbf{AF}\phi & = & \mu Z.\, \phi \vee \mathbf{AX}\ Z & \mathbf{A}[\phi_1 \mathbf{U} \phi_2] & = & \mu Z.\, \phi_2 \vee (\phi_1 \wedge \mathbf{AX}\ Z) \\ \mathbf{EG}\phi & = & \nu Z.\, \phi \wedge \mathbf{EX}\ Z & \mathbf{E}[\phi_1 \mathbf{R} \phi_2] & = & \nu Z.\, \phi_2 \wedge (\phi_1 \vee \mathbf{EX}\ Z) \\ \mathbf{AG}\phi & = & \nu Z.\, \phi \wedge \mathbf{AX}\ Z & \mathbf{A}[\phi_1 \mathbf{R} \phi_2] & = & \nu Z.\, \phi_2 \wedge (\phi_1 \vee \mathbf{AX}\ Z). \end{array}$$

The symbolic CTL model checking algorithm is implemented using a function Check that gets a CTL formula ϕ to be checked as an argument and returns a set of states that satisfy ϕ. We define Check inductively over the structure of CTL formulas. Recall that any CTL formula can be expressed using only atomic propositions, boolean operators and temporal operators $\mathbf{EX}, \mathbf{EU}$, and $\mathbf{EG}$. Hence, it is enough to consider only the following cases.

1. If $\phi = p$ where $p \in \mathcal{AP}$, then $\text{Check}(\phi) = \text{EvalAtomic}(p)$.
2. If $\phi = \phi_1 \wedge \phi_2$, then $\text{Check}(\phi) = \text{Check}(\phi_1) \cap \text{Check}(\phi_2)$.
3. If $\phi = \phi_1 \vee \phi_2$, then $\text{Check}(\phi) = \text{Check}(\phi_1) \cup \text{Check}(\phi_2)$.
4. If $\phi = \neg\phi_1$, then $\text{Check}(\phi) = S \setminus \text{Check}(\phi_1)$.
5. If $\phi = \mathbf{EX}\phi_1$, then $\text{Check}(\phi) = \text{EvalEX}(\text{Check}(\phi_1))$.

6. If $\phi = \mathbf{EG}\phi_1$, then $\text{Check}(\phi) = \text{EvalEG}(\text{Check}(\phi_1))$.

7. If $\phi = \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, then $\text{Check}(\phi) = \text{EvalEU}(\text{Check}(\phi_1), \text{Check}(\phi_2))$.

Notice that the functions EvalEX, EvalEG, and EvalEU get sets of states as arguments, EvalAtomic gets an atomic proposition and the function Check gets a CTL formula as an argument. In order to check if the Kripke structure M satisfies an arbitrary CTL formula φ, we have to check if all initial states of M belong to the set returned by the function $\text{Check}(\varphi)$.

The function $\text{EvalAtomic}(p)$ returns the set $\{s \in S \mid p \in L(s)\}$. The function $\text{EvalEX}(S')$ can be implemented using the standard symbolic operation $\text{PreImg}(S')$, which returns a set of all predecessors states of the states in the set S':

$$\text{EvalEX}(S') = \text{PreImg}(S') = \{s \in S \mid \exists s' \in S' : (s, s') \in R\}.$$

The function $\text{EvalEU}(S'_1, S'_2)$ gets two sets of states as arguments, the formula ϕ_1 is satisfied in the set S'_1, ϕ_2 is satisfied in S'_2. The function is implemented as suggested by the Proposition 3 and fixpoint characterization of the operators **EU**. Consider Algorithm 19, in EvalEU we compute a sequence of approximations $Q_0, Q_1, \ldots Q_i, \ldots$ that converges to a set of states that satisfy $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ in a finite number of steps. The function $\text{EvalEG}(S')$ is implemented analogously.

Algorithm 19 (Computing states satisfying $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$)

```
1  func EvalEU (S'_1, S'_2 )
2      Q := ∅
3      Q' := S'_2
4      while (Q ≠ Q') do
5          Q := Q'
6          Q' := S'_2 ∪ (S'_1 ∩ EvalEX(Q'))
7      od
8      return Q
9  end
```

Algorithm 20 (Computing states satisfying $\mathbf{EG}\phi_1$)

```
1  func EvalEG (S')
2      Q := S
3      Q' := S'
4      while (Q ≠ Q') do
5          Q := Q'
6          Q' := S' ∩ EvalEX(Q')
7      od
8      return Q
9  end
```

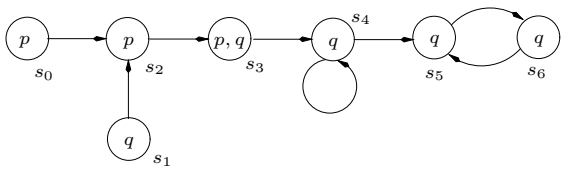


Figure 5.3: A Kripke structure used in the explanation of Algorithms 19 and 20

Example 20

Consider a Kripke structure M in Fig. 5.3. Suppose that we want to check a CTL formula $\varphi = \mathbf{E}[p\mathbf{U}q]$. A formula $\phi_1 = p$ holds in the set $S_1' = \{s_0, s_2, s_3\}$, a formula $\phi_2 = q$ holds in the set $S_2' = \{s_1, s_3, s_4, s_5, s_6\}$. In the functions EvalEU we compute a sequence of approximations $Q_0 \subset Q_1 \subset Q_2 \subset \ldots$

1. $Q_0 = S_2' = \{s_1, s_3, s_4, s_5, s_6\}$.
2. $Q_1 = S_2' \cup (S_1' \cap \text{EvalEX}(Q_0)) = \{s_1, s_2, s_3, s_4, s_5, s_6\}$.
3. $Q_2 = S_2' \cup (S_1' \cap \text{EvalEX}(Q_1)) = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$.
4. $Q_3 = S_2' \cup (S_1' \cap \text{EvalEX}(Q_2)) = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\} = Q_2$.

The fixpoint is reached, therefore the algorithm terminates and returns the set Q_3.

Suppose now that we want to check a CTL formula $\mathbf{EG}q$. In the functions EvalEG we compute a sequence of approximations $Q_0 \supset Q_1 \supset Q_2 \supset \ldots$

1. $Q_0 = S_2' = \{s_1, s_3, s_4, s_5, s_6\}$.
2. $Q_1 = S_2' \cap \text{EvalEX}(Q_0) = \{s_3, s_4, s_5, s_6\}$.
3. $Q_2 = S_2' \cap \text{EvalEX}(Q_1) = \{s_3, s_4, s_5, s_6\} = Q_1$.

5.2.4 Handling Fairness in Symbolic Algorithm

Let $M = [S, R, L, S_0, \mathcal{F}]$ be a fair Kripke structure. Consider a formula $\mathbf{EG}\phi$ under the fairness constraints $\mathcal{F}$. The formula means that there is a path beginning in the current state on which ϕ holds globally and each formula in $\mathcal{F}$ holds infinitely often on this path. The set of such states Z is the largest set in which all states both satisfy ϕ and have a path on which every state satisfies ϕ to a non-trivial fair SCC. This characterization can be expressed by means of a fixpoint as follows:

$$\mathbf{EG}_{fair}\phi = \nu Z.\ \phi \wedge \bigwedge_{\forall F \in \mathcal{F}} \mathbf{EX}(\mathbf{E}[\phi\ \mathbf{U}\ (Z \wedge F)]).$$

Algorithm 21 (Computing states satisfying $\mathbf{EG}_{fair}\phi_1$)

```
1  func EvalFairEG(S')
2     Q' := S
3     repeat
4        Q := Q'
5        forall F ∈ F do
6           D := EvalEU(S', Q' ∩ F)
7           Q' := Q' ∩ EvalEX(D)
8        od
9     until (Q = Q')
10    return Q
11 end
```

Implementing the function EvalFairEG using this representation results in a variation of the Emerson-Lei algorithm [EL86]. Consider Algorithm 21, it maintains a set of states that *might* lead to a non-trivial fair SCC. This approximation is repeatedly refined by removing states that can not lead to any non-trivial fair SCC. The procedure requires time $\mathcal{O}(|\mathcal{F}| \cdot |S|^2)$ as the algorithm employs computation of nested fixpoints.

A set *fair* contains all states that are beginnings of some fair computation paths: $fair = \text{EvalFairEG}(S)$. Analogously to the function Check, we define now the function CheckFair inductively over the structure of CTL formulas.

1. If $\phi = p$ where $p \in \mathcal{AP}$, then $\text{CheckFair}(\phi) = \text{EvalAtomic}(p) \cap fair$.
2. If $\phi = \phi_1 \wedge \phi_2$, then $\text{CheckFair}(\phi) = \text{CheckFair}(\phi_1) \cap \text{CheckFair}(\phi_2)$.
3. If $\phi = \phi_1 \vee \phi_2$, then $\text{CheckFair}(\phi) = \text{CheckFair}(\phi_1) \cup \text{CheckFair}(\phi_2)$.
4. If $\phi = \neg\phi_1$, then $\text{CheckFair}(\phi) = S \setminus \text{CheckFair}(\phi_1)$.
5. If $\phi = \mathbf{EX}\phi_1$, then $\text{CheckFair}(\phi) = \text{EvalEX}(\text{CheckFair}(\phi_1) \cap fair)$.
6. If $\phi = \mathbf{EG}\phi_1$, then $\text{CheckFair}(\phi) = \text{EvalFairEG}(\text{CheckFair}(\phi_1))$.
7. If $\phi = \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, then $\text{CheckFair}(\phi) = \text{EvalEU}(\text{CheckFair}(\phi_1), \text{CheckFair}(\phi_2) \cap fair)$.

Example 21

Consider a Kripke structure M in Fig. 5.4. Suppose that we want to check the CTL formula $\varphi = \mathbf{EG}q$ under fairness constraints $\mathcal{F} = \{p\}$. A formula $\phi_1 = p$ holds in the set $F = \{s_0, s_5, s_6\}$, $\phi_2 = q$ holds in the set $S' = \{s_0, s_2, s_3, s_4, s_5, s_6\}$. In the functions EvalFairEG we compute a sequence of approximations $Q_0 \supset Q_1 \supset Q_2 \supset \ldots$

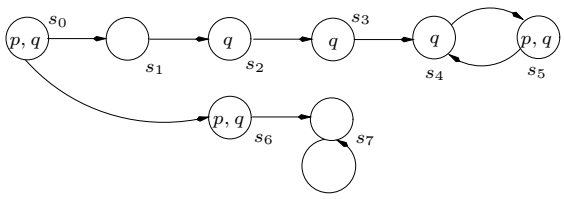


Figure 5.4: A Kripke structure used in the explanation of Algorithm 21

1. $Q_0 = S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$.
2. - $Q_0 \cap F = \{s_0, s_5, s_6\}$
   - $D = \text{EvalEU}(S', Q_0 \cap F) = \{s_0, s_2, s_3, s_4, s_5, s_6\}$
   - $Q_1 := Q_0 \cap \text{EvalEX}(D) = \{s_0, s_2, s_3, s_4, s_5\}$.
3. - $Q_1 \cap F = \{s_0, s_5\}$
   - $D = \text{EvalEU}(S', Q_1 \cap F) = \{s_0, s_2, s_3, s_4, s_5\}$
   - $Q_2 := Q_1 \cap \text{EvalEX}(D) = \{s_2, s_3, s_4, s_5\}$.
4. - $Q_2 \cap F = \{s_5\}$
   - $D = \text{EvalEU}(S', Q_2 \cap F) = \{s_2, s_3, s_4, s_5\}$
   - $Q_3 := Q_2 \cap \text{EvalEX}(D) = \{s_2, s_3, s_4, s_5\} = Q_2$.

The fixpoint is reached, therefore the algorithm terminates and returns the set Q_3.

5.3 Model Checking LTL

Given a Kripke structure $M = [S, R, L, S_0]$ and an LTL formula φ, the model checking problem is to decide whether φ holds in all computations of M:

$$M, \pi \models \varphi \; \forall \pi \in \mathcal{P}(s) \; \forall s \in S_0.$$

Today, the dominant approach to LTL model checking is the *automata theoretic approach* [VW86, Var96]. The approach is based on the fact that given an LTL formula, it is possible to construct a *Büchi automaton* [Büc60] that accepts all computations that satisfy this formula. This allows to reduce the model checking problem to an automata theoretic problem.

5.3.1 Büchi automata

Definition 46 (Büchi automaton)
A *Büchi automaton* is a tuple $A = [\Sigma, Q, \Delta, q_0, F]$ where:

- Σ is a finite *alphabet*.
- Q is a finite set of states.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $q_0 \in Q$ is an initial state.
- $F \subseteq Q$ is a set of *accepting states*.

A run of A over an infinite word ζ is an infinite sequence of states $\sigma = q'_0 q'_1 \ldots$ such that $q_0 = q'_0$ and $(q'_i, \zeta(i), q'_{i+1}) \in \Delta \ \forall i \geq 0$. We define a set

$$\inf(\sigma) = \{q \in Q \mid \forall k \geq 0 \ \exists i > k : q'_i = q\}.$$

A run σ over ζ is called *accepting* if $\inf(\sigma) \cap F \neq \emptyset$, we say in this case that A accepts the infinite word ζ. We define the language $\mathcal{L}(A)$ as a set of infinite words accepted by A. The automaton A is called *empty* if its language $\mathcal{L}(A) = \emptyset$. We shall denote F as a *fair set* and call an SCC of A *fair* if it is non-trivial and intersects F.

Notice that we allow the transition relation Δ to be nondeterministic, that is, there can be transitions $(q, a, q'), (q, a, q'') \in \Delta$ where $q' \neq q''$. Every nondeterministic automaton over *finite words* one can be translated into an equivalent deterministic automaton that accepts the same language using the *subset construction*. This is not the case with Büchi automata, not every Büchi automaton has an equivalent deterministic Büchi automaton.

Example 22
Two Büchi automata are shown in Fig. 5.5. The automaton A_1 accepts infinite words with a finite number of a's, the automaton A_2 accepts words with an infinite number of a's. The accepting states are denoted by doubled circles, initial states by small incoming arrows. Note that A_2 is deterministic, but it is not possible to construct an equivalent deterministic automaton that accepts the language $\mathcal{L}(A_1)$.

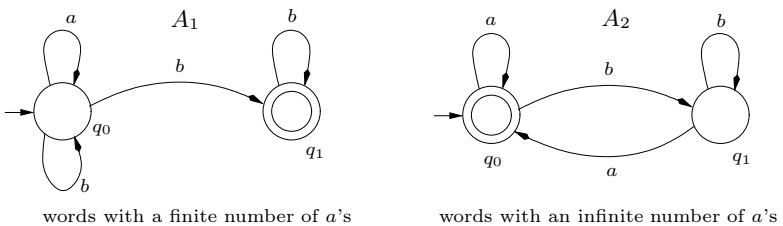


Figure 5.5: Büchi automata

Sometimes it is convenient to work with Büchi automata with several accepting sets of states, although this does not extend the set of languages that can be expressed.

Definition 47 (Generalized Büchi automaton)
A *generalized Büchi automaton* is a tuple $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ where:

- Σ is a finite *alphabet*.
- Q is a finite set of states.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $q_0 \in Q$ is an initial state.
- $\mathcal{F} \subseteq 2^Q$ is a set of fair sets.

A generalized Büchi automaton A accepts an infinite word ζ if there exists a run σ over ζ such that $\inf(\sigma) \cap F \neq \emptyset \; \forall F \in \mathcal{F}$. An SCC of A is called *fair* if it is non-trivial and intersects all fair sets.

It is often defined that every infinite run of a generalized Büchi automaton A is accepting when $\mathcal{F} = \emptyset$, we renounce such a definition to avoid ambiguities. There exists a simple translation from a generalized Büchi automaton to a Büchi automaton.

Lemma 7 ([Tho90])
Let $A = [\Sigma, Q, \Delta, q_0, \{F_1, \ldots, F_n\}]$ be a generalized Büchi automaton, and let $A' = [\Sigma, Q', \Delta', q_0', F']$ be a Büchi automaton. $\mathcal{L}(A) = \mathcal{L}(A')$ if

- $Q' = Q \times \{1, \ldots, n\}$
- $q_0' = (q_0, 1)$
- $\Delta' \subseteq Q' \times \Sigma \times Q'$ *is defined as*

$$\begin{aligned} \Delta' \quad = \quad & \{\, ((q_1, i), a, (q_2, i)) \mid (q_1, a, q_2) \in \Delta \wedge q_1 \notin F_i \,\} \cup \\ & \{\, ((q_1, i), a, (q_2, (i+1) \bmod n)) \mid (q_1, a, q_2) \in \Delta \wedge q_1 \in F_i \,\} \end{aligned}$$

- $F' = F_1 \times \{1\}$.

The translation expands the size of the automaton by a factor of $n + 1$. However, for *weak* automata we can avoid this translation to reduce the number of fair sets to one. We introduce a notion of a *strength* of generalized Büchi automata after [SB00].

Definition 48
Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F} = \{F_1, \ldots, F_n\}]$ be a generalized Büchi automaton, and let Θ_A be the union of all fair SCCs of A.

- A is called *weak* if and only if for every SCC C of A, either for each fair set $F_i \supseteq C$, or there exists a fair set F_k such that $F_k \cap C = \emptyset$.

- A is called *terminal* if and only if it is weak, there is no arcs from a fair SCC to a non-fair SCC: $\nexists q \in \Theta_A,\ q' \in Q \setminus \Theta_A,\ a \in \Sigma\ :\ (q, a, q') \in \Delta$, and for every state $s \in \Theta_A$ holds $\Sigma = \bigcup\{a \in \Sigma \mid (s, a, s') \in \Delta\}$.
- A is called *strong* if it is not weak.

Lemma 8 ([SB00])
Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ be a weak generalized Büchi automaton. Let A' be a Büchi automaton defined as $A' = [\Sigma, Q, \Delta, q_0, \{\Theta_A\}]$. Then $\mathcal{L}(A) = \mathcal{L}(A')$.

5.3.2 Automata Theoretic Approach

Let $M = [S, R, L, S_0]$ be a Kripke structure. A computation of M can be considered as an infinite word over the alphabet $\Sigma = 2^{\mathcal{AP}}$. Thus, the set of all computations of M builds a language $\mathcal{L}(M) \subseteq \Sigma^w$. Analogously, every LTL formula φ describes a language $\mathcal{L}(\varphi) \subseteq \Sigma^w$, defined as a set of all infinite words that satisfy this formula.

Theorem 5 ([VW94, Var96])
Let ϕ be an LTL formula. There exists a Büchi automaton $A_\phi = [\Sigma, Q_\phi, \Delta_\phi, q_{0_\phi}, F_\phi]$ such that $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$ and $|Q_\phi| \leq 2^{\mathcal{O}(|\phi|)}$.

For every Kripke structure M we can also construct a Büchi automaton A_M such that $\mathcal{L}(A_M) = \mathcal{L}(M)$ as follows: $A_M = [\Sigma,\ S \cup \{s^i\},\ \Delta_M,\ s^i,\ S \cup \{s^i\}]$ where:

- $(s, p, s') \in \Delta_M$ for $s, s' \in S$ if and only if $(s, s') \in R$ and $p = L(s')$.
- $(s^i, p, s) \in \Delta_M$ if and only if $s \in S_0$ and $p = L(s)$.

Notice that all the states of the constructed automaton are accepting.

An LTL formula φ holds in all computations of M if $\mathcal{L}(A_M) \subseteq \mathcal{L}(A_\varphi)$. This can be rewritten as $\mathcal{L}(A_M) \cap \overline{\mathcal{L}(A_\varphi)} = \emptyset$, which means, there is no computation in M that is disallowed by φ. If the intersection is not empty, any computation in it corresponds to a counterexample. Büchi automata are closed under intersection and complementation [Büc60]. This means that there exists an automaton that accepts exactly the intersection of the languages of two automata and an automaton that recognizes exactly the complement of the language of a given automaton. Computation of the complement for a Büchi automaton is a complicated procedure, which is, in general, exponential. Fortunately, instead of translating an LTL formula φ into a Büchi automaton A_φ and then complementing it, we can simply translate $\neg\varphi$, which immediately provides an automaton $A_{\neg\varphi}$ for the complement language $\overline{\mathcal{L}(A_\varphi)}$.
Moreover, the fact that all states of the automaton A_M are accepting simplifies also the construction of the intersection of these Büchi automata. It is defined then as $A_M \cap A_{\neg\varphi} = A_{M,\varphi} = [\Sigma, Q, \Delta, q_0, F]$ where:

- $\Sigma = 2^{\mathcal{AP}}$
- $Q = (S \cup \{s^i\}) \times Q_{\neg\varphi}$
- $((s_i, q_j),\ a,\ (s_m, q_n)) \in \Delta$ if and only if $(s_i, a, s_m) \in \Delta_M$ and $(q_j, a, q_n) \in \Delta_{\neg\varphi}$
- $q_0 = (s^i, q_{0_{\neg\varphi}})$
- $F = (S \cup \{s^i\}) \times F_{\neg\varphi}$.

The constructed automaton $A_{M,\varphi}$ is often denoted as a *product automaton*. To check if the LTL formula φ holds in all computations of M, it is enough to check whether the product automaton $A_{M,\varphi}$ is empty.

A usual translation procedure produces from an LTL formula $\neg\varphi$ a generalized Büchi automaton $A_{\neg\varphi} = [\Sigma, Q_{\neg\varphi}, \Delta_{\neg\varphi}, q_{0_{\neg\varphi}}, \{F_{\neg\varphi 1}, \ldots, F_{\neg\varphi n}\}]$. We can also define the product automaton $A_{M,\varphi}$ as a generalized automaton $A_{M,\varphi} = [\Sigma, Q, \Delta, q_0, \{F_1, \ldots, F_n\}]$ where Q, Δ, and q_0 are defined as above and $F_i = (S \cup \{s^i\}) \times F_{\neg\varphi i})$. Notice that the strength of the product automaton depends only on the strength of $A_{\neg\varphi}$. It easy to see that $A_{M,\varphi}$ can not be stronger than $A_{\neg\varphi}$.

Checking emptiness of a Büchi automaton is simple. Let σ be an accepting run of a Büchi automaton $A = [\Sigma, Q, \Delta, q_0, F]$. Then σ contains infinitely many accepting states from F. Since Q is finite, there is some suffix σ' of σ such that every state on it appears infinitely many times. Each state on σ' is reachable from any other state on σ'. Hence, the states in σ' are included in an SCC that is reachable from the initial state and contains an accepting state. Conversely, any fair SCC that is reachable from the initial state generates an accepting run of the automaton. Thus, the Büchi automaton is not empty if there is a reachable accepting state with a cycle back to itself or, equivalently, there is a reachable fair SCC.

Analogously, a generalized Büchi automaton is not empty if there is a reachable state lying on a cycle that intersects all fair sets, or a equivalently, there is a reachable fair SCC. If the automaton is not empty, then there is a counterexample, which can be presented in a finitary manner. The counterexample is a run, constructed from a finite prefix followed by a periodic sequence of states.

Tarjan's algorithm [Tar72] for finding SCCs can be used to decide emptiness of a Büchi automaton in time $\mathcal{O}(|Q| + |\Delta|)$. Consider Algorithm 22. For every state q of an automaton we keep two integers: $q.num$ and $q.lowlink$, $q.num$ stores the deep-first search (DFS) number of the state. $q.lowlink$ is the lowest DFS number of a state t in the same SCC as q such that t was reachable from q via states that were not yet explored when the search reached q. If after visiting all adjacent states of q its $q.lowlink = q.num$, then q is a *root* of a found SCC. When an SCC is found, we check if it is fair, in this case the automaton is not empty.

A number of algorithms that can be more efficient in practice and/or require less memory have been proposed by many authors, see [SE05] for the latest survey.

Algorithm 22 (Emptiness check)

```
1  proc CheckEmptiness (A)
2     count := 0
3
4     proc visit(q)
5        Visited := Visited ∪ {q}
6        count := count + 1
7        num[q] := count
8        lowlink[q] := count
9        push(SCCStack, q)
10
11       forall { q' | [q, _, q'] ∈ Δ } do
12          if q' ∉ Visited then
13             visit(q')
14             lowlink[q] := min(lowlink[q'], lowlink[q])
15          elseif q' ∈ SCCStack then
16             lowlink[q] := min(num[q'], lowlink[q])
17          fi
18       od
19
20       if num[q] = lowlink[q] then
21          C := ∅
22          repeat
23             x := pop(SCCStack)
24             C := C ∪ {x}
25          until x = q
26          if C ∩ F ≠ ∅ then
27             if C ≠ {q} then
28                exit ("a fair SCC is found")
29             else
30                if (q, _, q) ∈ Δ then exit ("a fair SCC is found") fi
31             fi
32          fi
33       fi
34    end
35
36    begin
37       visit(q_0)
38       exit ("no fair SCC was found")
39    end
```

5.3.3 Translating LTL Formulas into Büchi Automata

As outlined above, the automata theoretic approach to LTL model checking consists of translating the negation of a given LTL formula φ into a Büchi automaton $A_{\neg\varphi}$, and checking the product $A_{M,\varphi}$ of the property automaton and the model for emptiness. The initial approaches to the translation of LTL formulas were not designed to yield small automata. The procedure suggested in [WVS83] always yielded to the worst case result $|Q_\phi| = 2^{\mathcal{O}(|\varphi|)}$. Over last twenty years many LTL translation algorithms have been developed. Most of them strive to produce small automata in an attempt to produce a smaller product $A_{M,\varphi}$ and hence to speedup the emptiness check. As noticed lately in [ST03], another way to reduce $A_{M,\varphi}$ is to output more deterministic automata. Most of the modern translation algorithms are based on the approach proposed in [GPVW95]. We sketch the general procedure and mention possible improvements.

Definition 49

- An LTL formula ϕ is said to be in the *negation normal form* (NNF) if the only temporal operators used in it are $\mathbf{X}, \mathbf{U}$, and $\mathbf{R}$ and negations are applied only to atomic propositions.
- For every temporal operator $op \in \{\mathbf{X}, \mathbf{U}, \mathbf{R}\}$, we say that ϕ is an *op-formula* if *op* is the root operator of ϕ.
- An *elementary formula* is either a constant $c \in \{\mathit{True}, \mathit{False}\}$, an atomic proposition $p \in \mathcal{AP}$, a negation of an atomic proposition, or an $\mathbf{X}$-formula.
- A *cover* for a set of LTL formulas $\{\phi_k\}_k$ is a set of sets of elementary formulas $\{\{\nu_{ij}\}_j\}_i$ such that $\bigwedge_k \phi_k \Leftrightarrow \bigvee_i \bigwedge_j \nu_{ij}$.
- We say that the occurrence of a subformula ϕ_1 in an LTL formula ϕ is a *top level occurrence* if it occurs only in the scope of boolean operators.

Using De Morgan's Laws and the following equalities every LTL formula can be translated into an equal LTL formula of the same length[5] in the negation normal form.

$$\begin{array}{lcllcl} \mathbf{F}\phi & \equiv & \mathit{True}\mathbf{U}\phi & \mathbf{G}\phi & \equiv & \mathit{False}\mathbf{R}\phi \\ \neg(\phi_1\mathbf{U}\phi_2) & \equiv & \neg\phi_1\mathbf{R}\neg\phi_2 & \neg\mathbf{X}\phi & \equiv & \mathbf{X}(\neg\phi). \end{array}$$

A cover for a set of LTL formulas $\{\phi_k\}_k$ is a typically obtained by computing the *disjunctive normal form* (DNF) of $\bigwedge_k \phi_k$, considering $\mathbf{X}$-subformulas as boolean propositions.

Let $\mathcal{AP}$ be a set of atomic propositions used in φ. A translation procedure that we consider creates a generalized Büchi automaton $A_\varphi = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$. However, it annotates *edges* of the automaton with boolean expressions, rather than subsets of $\mathcal{AP}$.

[5] This means with the same number of temporal operators.

Each edge may represent several *transitions* in Δ, where each transition corresponds to a truth assignment for $\mathcal{AP}$ that satisfies this boolean expression. For example, when $\mathcal{AP} = \{p_1, p_2, p_3\}$, an edge labeled with $p_1 \wedge \neg p_3$ corresponds to transitions labeled with $\{p_1, p_2\}$ and $\{p_1\}$.

The general translation schema of an LTL formula φ into a Büchi automaton works as follows. First, φ is rewritten in the negation normal form. Second, φ is expanded by applying the *tableau rewriting rules*:

$$\phi_1 \mathbf{U} \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \mathbf{U} \phi_2)) \qquad \phi_1 \mathbf{R} \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}(\phi_1 \mathbf{R} \phi_2))$$

until no **U**-formula or **R**-formula occurs a the top level. Then the resulting formula is rewritten into a cover by computing its disjunctive normal form. Each disjunct of the cover corresponds to a state of the resulting automaton. Atomic propositions and their negations define a *label* l of the state, that is, a condition that an input word must satisfy in this state. This means that in the automaton A_φ, all incoming edges of this state must be labeled by l. Remaining **X**-formulas represent the *next part* of the state, that is, the obligations that must be fulfilled to get an accepting run. They determine edges outcoming from the state as well as definition of fair sets.

The expansion process is applied recursively to the next part of each state, creating new covers until no new obligations are produced. This results in a *closed* set of covers $\mathcal{C}$, so that, for each cover $C_{\phi_i} \in \mathcal{C}$, the next part of each disjunct in C_{ϕ_i} has a cover in $\mathcal{C}$. The automaton is obtained by connecting each state to those in the cover of its next part. A special state q_0 without incoming edges is created and connected with all states induced by the elementary cover of φ. Fair sets F_i must be added to the automaton for every elementary subformula of the form $\mathbf{X}(\phi_1 \mathbf{U} \phi_2)$. The fair set F_i contains all states $q \neq q_0$ such that the label of q implies ϕ_2 or the next part of the state does not imply $\phi_1 \mathbf{U} \phi_2$. In the case when no fair sets must be added, we define $\mathcal{F} = \{Q \setminus q_0\}$.

Example 23

Let us start with a very simple translation of the formula $\varphi = \mathbf{G}\neg p$. We rewrite φ in the NNF and apply tableau expansion rules

$$\varphi = \mathit{False} \mathbf{R} \neg p = \neg p \wedge \mathbf{X}(\mathit{False} \mathbf{R} \neg p).$$

The formula φ is in the DNF, thus the cover $C_\varphi = \{\{\neg p, \mathbf{X}(\mathit{False} \mathbf{R} \neg p)\}\}$. This means, we have a state q_1 with a label $l_1 = \neg p$. The obligation in the next part of q_1 is $\phi_1 = \mathit{False} \mathbf{R} \neg p = \varphi$. Thus, the cover $C_{\phi_1} = C_\varphi$ and we have already got the closed set of covers $\mathcal{C} = \{C_\varphi\}$. We create two states q_0 and q_1. The cover in the next part of q_1 is C_φ, thus we create an edge from q_1 back to itself. As q_1 is induced by the cover C_φ, we create also an edge from q_0 to q_1. All incoming edges of q_1 must be labeled by the label $l_1 = \neg p$. We have met no subformulas of the form $\mathbf{X}(\phi_1 \mathbf{U} \phi_2)$, therefore $\mathcal{F} = \{Q \setminus q_0\} = \{\{q_1\}\}$. The resulting automaton is shown in Fig. 5.6, left.

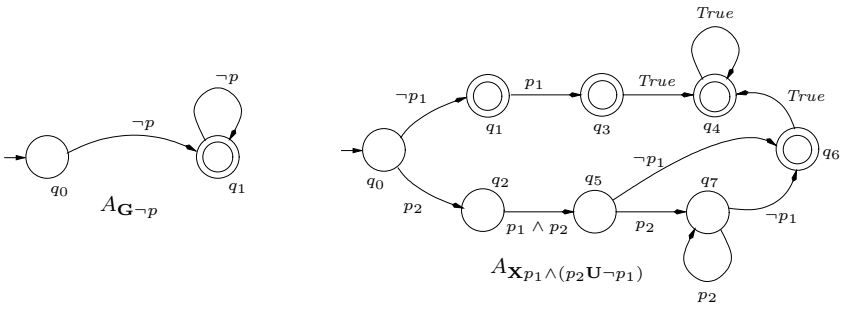


Figure 5.6: Automata produced in the Examples 23 and 24

Example 24

Let us consider now a more complicated translation of the formula $\varphi = \mathbf{X}p_1 \wedge (p_2 \mathbf{U} \neg p_1)$. We apply tableau expansion rules and rewrite φ in the DNF:

$$\begin{aligned} \varphi &= \mathbf{X}p_1 \wedge (p_2 \mathbf{U} \neg p_1) \\ &= \mathbf{X}p_1 \wedge (\neg p_1 \vee p_2 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1)) \\ &= \neg p_1 \wedge \mathbf{X}p_1 \ \vee \ p_2 \wedge \mathbf{X}p_1 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1). \end{aligned}$$

The cover for the formula φ is

$$C_\varphi = \{\{\neg p_1, \mathbf{X}p_1\}, \{p_2, \mathbf{X}p_1, \mathbf{X}(p_2 \mathbf{U} \neg p_1)\}\}.$$

This means, we have a state q_1 with a label $l_1 = \neg p_1$ and a state q_2 with a label $l_2 = p_2$. New obligations are $\phi_1 = p_1$ and $\phi_2 = p_1 \wedge (p_2 \mathbf{U} \neg p_1)$.

We notice that $\phi_1 = p_1 \wedge \mathbf{X}\mathit{True}$, thus $C_{\phi_1} = \{\{p_1, \mathbf{X}\mathit{True}\}\}$. We get a state q_3 with a label $l_3 = p_1$ and a new obligation $\phi_3 = \mathit{True}$.

$\phi_3 = \mathit{True} \wedge \mathbf{X}\mathit{True}$ and the cover $C_{\phi_3} = \{\{\mathit{True}, \mathbf{X}\mathit{True}\}\}$, a state q_4 has a label $l_4 = \mathit{True}$, no new obligations have to be considered.

We return back and compute a cover for ϕ_2

$$\begin{aligned} \phi_2 &= p_1 \wedge (p_2 \mathbf{U} \neg p_1) \\ &= p_1 \wedge (\neg p_1 \vee p_2 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1)) \\ &= p_1 \wedge \neg p_1 \ \vee \ p_1 \wedge p_2 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1) \\ &= p_1 \wedge p_2 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1). \end{aligned}$$

The cover $C_{\phi_2} = \{\{p_1, p_2, \mathbf{X}(p_2 \mathbf{U} \neg p_1)\}\}$, we get a state q_5 with a label $l_5 = p_1 \wedge p_2$ and a new obligation $\phi_5 = p_2 \mathbf{U} \neg p_1$.

Finally,

$$\phi_5 = (p_2 \mathbf{U} \neg p_1) = \neg p_1 \vee p_2 \wedge \mathbf{X}(p_2 \mathbf{U} \neg p_1).$$

The cover $C_{\phi_5} = \{\{\neg p_1, \mathbf{X}\,True\}, \{p_2, \mathbf{X}(p_2\mathbf{U}\neg p_1)\}\}$, we get a state q_6 with a label $l_6 = \neg p_1$ and a state q_7 with a label p_2. No new obligations have to be considered, we have finally got the closed set of covers $\mathcal{C} = \{C_\varphi, C_{\phi_1}, \ldots, C_{\phi_5}\}$. The resulting automaton is shown in Fig. 5.6, right. This time we have a fair set induced by the formula $\mathbf{X}(p_2\mathbf{U}\neg p_1)$, therefore $\mathcal{F} = \{\{q_1, q_3, q_4, q_6\}\}$.

We can simplify the fair set. The automaton has one non-trivial SCC $\{q_4\}$, which is also fair. According to Definition 48, we conclude that the automaton is terminal, due Lemma 8, we can define the set $\mathcal{F} = \{\{q_4\}\}$.

In practice, the procedure translating an LTL formula ϕ into a Büchi automaton proceeds in three stages:

1. application of rewrite rules to ϕ,
2. translation of ϕ into A_ϕ,
3. optimization of A_ϕ.

Rewriting is a cheap, simple and effective way to minimize the result of the translation. Usually, a family of rewrite rules is defined. The rules are applied then recursively, reducing the number of operators in the LTL formula. Let us consider, for example, reductions defined in [EH00].

Definition 50
A class of *pure eventuality* formulas is defined as a smallest set of LTL formulas (in the negation normal form) satisfying:

1. Any formula of the form $True\mathbf{U}\phi$ is a pure eventuality formula.
2. If ϕ_1 and ϕ_2 are pure eventuality formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1\mathbf{U}\phi_2$, $False\mathbf{R}\phi_1$, $\phi_1\mathbf{R}\phi_2$, and $\mathbf{X}\phi_1$ are also pure eventuality formulas.

A class of *pure universal* formulas is defined as a smallest set of LTL formulas (in the negation normal form) satisfying:

1. Any formula of the form $False\mathbf{R}\phi$ is a pure universal formula.
2. If ϕ_1 and ϕ_2 are pure universal formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1\mathbf{R}\phi_2$, $True\mathbf{U}\phi_1$, $\phi_1\mathbf{U}\phi_2$, and $\mathbf{X}\phi_1$ are also pure universal formulas.

Lemma 9 ([EH00])
For all LTL formulas ϕ_1, ϕ_2, and ϕ_3 the following equivalences hold:

1. $(\phi_1\mathbf{U}\phi_2) \wedge (\phi_3\mathbf{U}\phi_2) \equiv (\phi_1 \wedge \phi_3)\mathbf{U}\phi_2$

2. $(\phi_1 \mathbf{U} \phi_2) \vee (\phi_1 \mathbf{U} \phi_3) \equiv \phi_1 \mathbf{U} (\phi_2 \vee \phi_3)$

3. $True \mathbf{U} (\phi_1 \mathbf{U} \phi_2) \equiv True \mathbf{U} \phi_2$

4. *If ϕ is a pure eventuality formula, then* $\phi_1 \mathbf{U} \phi \equiv \phi$, *and* $True \mathbf{U} \phi \equiv \phi$

5. *If ϕ is a pure universal formula, then* $\phi_1 \mathbf{R} \phi \equiv \phi$, *and* $False \mathbf{R} \phi \equiv \phi$.

Recall that in the translation stage we have applied a procedure that produces a closed set of covers. An LTL formula has many covers, which one is chosen, affects the size of the resulting automaton by directly affecting what states are added to the automaton and by determining what covers will belong to the closed set. *Boolean optimization techniques* are applied in [SB00] striving for smaller automata, techniques employing *semantic branching* are used in [ST03] to get more deterministic automata.

A number of techniques have been proposed to optimize Büchi automata. *Simulation* and *bisimulation* are used to reduce the number of states and transitions in [EH00] and [SB00]. Simplification of fair sets also has several benefits. It may lead to a reduction in the strength of the resulting automaton, simplifying the symbolic model checking. Even when the strength of the automaton is not reduced, fewer, smaller fair sets usually lead to faster convergence of the emptiness check. Finally, it may also enable further reductions in the number of states and transitions. Techniques for simplification of fair sets based on the analysis of SCCs were suggested in [SB00].

5.4 Closing Remark

Model checking is a technique that provides means to automatically check whether a finite state model of a system satisfies a given specification. Normally, the procedure uses an exhaustive exploration of all possible states of the model to determine whether it satisfies a property expressed in a temporal logic. The main drawback of model checking is the state explosion problem. Symbolic methods, with which we deal in this thesis, are one of the most successful approaches to combat it.

Two possible views regarding the nature of time induce two types of temporal logics. In linear temporal logics, time is treated as if each moment has a unique possible future. In branching temporal logics, each moment in time may split into various possible futures. We have concentrated on the two most popular and commonly supported temporal logics: on the linear time logic LTL and on the branching time logic CTL. Being relatively easy to use, these logics allow specification of many properties of interest, as well as efficient implementation of model checking tools. A model checker for the more expressive logic CTL* can be implemented as a combination of CTL and LTL model checkers [CGP01].

CTL and LTL are expressively incomparable. CTL formulas are state formulas, the truth of a CTL formula depends only on the current state and does not depend on the current path. This limitation carries also its benefits, allowing to implement simple and efficient CTL model checking algorithms. LTL deals only with a set of computations and not in the way these are organized into a tree, hence LTL can not express that at some state it is possible to extend the computation in this or that way. The dominant approach to the LTL model checking is the automata theoretic approach.

Actually, we have only touched a tip of the iceberg "Model checking", we refer once again to [CGP01]. In the next chapters we shall discuss an implementation of symbolic CTL and LTL model checkers for Petri nets.

6 Symbolic CTL Model Checking of Bounded Petri Nets

Functions for the symbolic manipulations of Petri nets defined in chapter 4 allow a straightforward implementation of a symbolic CTL model checker. In this chapter we consider how to make this implementation efficient.

Conventional symbolic CTL algorithms introduced in the previous chapter are based on the breath-first order exploration of the state space. As shown in section 4.3, straying from this strategy can significantly improve efficiency of symbolic algorithms for Petri nets. We study how the saturation technique introduced in section 4.3.2 can be employed in CTL algorithms.

Performance of the symbolic state space exploration depends heavily on the structure of the model. Sometimes, forward state traversals are quite efficient, whereas intermediate decision diagrams created during the backward state space exploration become too large and can not be handled efficiently. A CTL model checking algorithm based mainly on forward state traversals was suggested in [INH96]. We show how this algorithm can be implemented using functions defined in chapter 4.

We also discuss how to improve efficiency of the model checking with fairness constraints and how to generate counterexamples and witnesses.

6.1 Petri Nets and CTL

Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a bounded P/T net with extended arcs. We assume that places of N are enumerated $P = \{p_1, \ldots, p_n\}$. Since states of N are its reachable markings, we should take the set of atomic propositions $\mathcal{AP}$ as an arbitrary set of propositions on the set of markings. For convenience, we take $\mathcal{AP}$ as a set of interval logic expressions over the variables $p_1, \ldots, p_n$, with the following interpretation: an atomic proposition G is satisfied in a state m if m belongs to the set of markings M_G described by an interval logic function induced by the expression G[1]. This allows quite natural specifications for Petri nets like "$\mathbf{AGEF}(p_1 = 3 \wedge p_2 \in [2, 5) \vee p_4 > 0)$".

We say that the net N satisfies a CTL formula φ if this formula is satisfied in the Kripke structure $M_N = [S, R, L, S_0]$ where:

[1]For the sake of brevity, we shall write simply "a set of markings M_G described by G".

- $S = \mathcal{R}_N(m_0)$
- $R = \{(m, m') \mid m, m' \in S \ \wedge \exists t \in T : \ m \xrightarrow{t} m'\} \cup \{(m, m) \mid m \in S \cap \mathcal{D}_N\}$
- $L(m) = \{G \in \mathcal{AP} \mid m \in M_G\}$
- $S_0 = \{m_0\}$.

Notice that by adding self-loops in dead states we guaranty that the relation R is total. Sometimes we have a set[2] of initial markings $M_0 = \{m_{0_1}, \ldots, m_{0_k}\}$ and want to check if the net satisfies a CTL formula for all of them. In this case we can solve one model checking problem instead of k if we modify the definition of M_N as

- $S = \bigcup_{m \in M_0} \mathcal{R}_N(m)$
- $S_0 = M_0$.

Thus, to check if the net N satisfies a CTL formula φ we can compute the reachability sets $\mathcal{R}_N(m_{0_i})$, construct the Kripke structure M_N and apply model checking algorithms discussed in the previous chapter. Obviously, an explicit construction of M_N suffers from the state explosion problem. We aim at symbolic model checking and reuse functions for the symbolic manipulations of Petri nets that we have defined in chapter 4. As states of the net N correspond directly to states of the Kripke structure M_N, the set S can be computed as

$$S = \bigcup_{m \in M_0} \mathcal{R}_N(m) = \text{FwdReach}(M_0).$$

Furthermore, examining symbolic CTL algorithms introduced in sections 5.2.3 and 5.2.4, we notice that the only functions that have to be adapted are EvalAtomic and EvalEX, as all algorithms are implemented using only usual set operators and these functions. As defined above, atomic propositions are interval logic expressions now, thus

$$\text{EvalAtomic}(G) = S \cap M_G,$$

where M_G is a set of markings described by G. When computing $\text{EvalEX}(S')$ we must take into account that the function PreImg was defined for Petri nets in such a way that the returned set can also contain markings that do not belong to $\mathcal{R}_N(m_0)$. Moreover, due to the definition of the Kripke structure M_N, we have to simulate a self-loop in dead states, thus

$$\text{EvalEX}(S') = (\text{PreImg}(S') \cap S) \ \cup \ (\mathcal{D}_N \cap S').$$

Implementation of a symbolic CTL model checker for P/T nets with extended arcs is now straightforward. In the next sections we shall discuss how we can improve conventional CTL model checking algorithms.

[2] Of course, we have to require that this set is finite.

6.2 Employing Saturation Strategy

Let us consider the algorithm computing a set Z of states satisfying $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ (Algorithm 19 on page 103). We denote a set of states that satisfy ϕ_1 as S'_1 and a set of states satisfying ϕ_2 as S'_2. Starting from the set S'_2, the algorithm iteratively adds all states that reach them on the paths along states in S'_1. In other words, it performs the limited backward reachability analysis starting from the states in S'_2. The exploration of the state space is done in the breath-first order. Recall that performance of the symbolic reachability analysis of Petri nets can be drastically improved if we can stray from the breath-first exploration and employ the saturation strategy. We notice that we can compute the set Z using the function LimBwdReach defined in section 4.3.3

$$Z = \text{LimBwdReach}(S'_2,\ S'_1 \cup S'_2).$$

This means, we get "for free" an efficient saturation-based implementation of the function EvalEU.

Consider now the algorithm computing a set Z of states satisfying $\mathbf{EG}\phi$ (Algorithm 20 on page 103). Let S' be a set of states that satisfy ϕ. The algorithm initializes its working set Q' with a set S' and iteratively removes states that have no successors in Q' until only non-trivial SCCs of S' and their incoming paths along states in S' are left. We have not yet defined any suitable saturation-based function that could be directly used to compute Z. Moreover, as the algorithm is based on the computation of the greatest fixpoint and its working set Q' is monotonically *decreasing*, we can not directly rewrite it to use the saturation strategy, which exploits the fact that transitions can *add* states to the working set in any desired order and independently from each other. Recall that $\mathbf{EG}\phi \equiv \neg\mathbf{AF}\neg\phi$ and $\mathbf{AF}$ has a least fixpoint characterization

$$\mathbf{AF}\phi = \mu Z.\, \phi \vee \mathbf{AX}\ Z.$$

We can not apply the saturation strategy to compute the set of states satisfying $\mathbf{AF}\neg\phi$ as well: for a state s we have to consider *all* its direct successors to decide whether s can be added to the working set or not. Nevertheless, saturation-based algorithms can be still employed to compute Z. We can adapt Algorithm 18 (page 100) used in the explicit CTL model checking. The saturation-based version of the Lockstep algorithm (Algorithm 16 on page 87) can be used to enumerate all SCCs in the set S'. Let T be a set of all non-trivial SCCs computed in this way, we can compute the set Z as

$$Z = \text{LimBwdReach}(T \cup (\mathcal{D}_N \cap S')\ ,\ S').$$

However, this approach is not promising. The need to enumerate trivial SCCs decreases drastically performance of the Lockstep algorithm. Eliminating trivial SCCs using the backward trimming corresponds directly to the elimination procedure of the original algorithm computing $\mathbf{EG}\phi$.

6.3 OWCTY Algorithm

Let us consider the algorithm computing states satisfying $\mathbf{EG}_{fair}\phi_1$ (Algorithm 21 on page 105). It maintains a set of states that may lead to a non-trivial fair SCC and repeatedly refines this approximation by removing states that can not lead to any non-trivial fair SCC. Only one pruning step is done per iteration. It was noticed that more aggressive pruning strategies can improve efficiency of the algorithm. As analyzed in [FFK+01], a total number of computations made in the algorithm can be reduced by balancing a number of computations made in the function EvalEU and a number of pruning steps. Consider Algorithm 23, proposed in [FFK+01] under the name One-Way-Catch-Them-Young (OWCTY). The difference between the pruning strategies of the original algorithm (denoted as EL, for Emerson-Lei) and OWCTY is easiest to understand by an example.

Example 25

Two Kripke structures are sketched in Fig. 6.1. We assume that all states satisfy ϕ_1, fair states are shaded. Consider the upper Kripke structure. Both algorithms eliminate the rightmost state in the first iteration and capture the remaining states in the approximation set. During the first iteration, OWCTY eliminates all but the leftmost fair state. EL eliminates only the rightmost fair set. EL requires an additional iteration to eliminate each of the five middle fair states. Each iteration involves a reachability computation that OWCTY does not perform. If the chain of fair states contained n fair states, OWCTY would perform $\mathcal{O}(n)$ calls to EvalEX while EL would make $\mathcal{O}(n^2)$ calls. Thus, EL has a quadratic overhead relative to OWCTY on such Kripke structures.

Now consider the second Kripke structure. Both algorithms eliminate the rightmost

Algorithm 23 (Computing states satisfying $\mathbf{EG}_{fair}\phi_1$)

```
1  func EvalFairEG(S')
2     Q' := S'
3     repeat
4        Q := Q'
5        forall F ∈ F do
6           Q' := EvalEU(Q', Q' ∩ EvalEX(F ∩ Q'))
7           repeat
8              Old := Q'
9              Q' := Q' ∩ EvalEX(Q')
10          until Old = Q'
11       od
12    until (Q = Q')
13    return Q
14 end
```

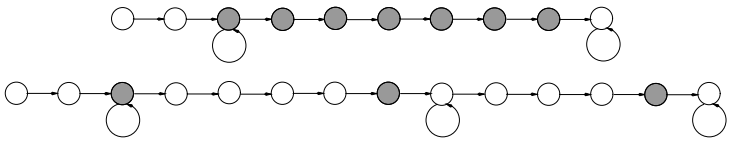


Figure 6.1: Kripke structures illustrating the difference between EL and OWCTY

state in the first iteration and retain the remaining states in the approximation set. During the first iteration EL throws away the rightmost fair state. The reachability computation in the second iteration begins at the middle fair state. Thus, EL eliminates the non-fair sets between the right two fair sets without traversing them explicitly. OWCTY, in contrast uses an additional call to EvalEX to eliminate each of these non-fair states. The Kripke structure contains two copies of a chain of states consisting of four non-fair states followed by a fair state. If it had k consecutive copies of this chain, each with m states in the initial non-fair chain, EL would perform $\mathcal{O}(k^2 \cdot m)$ calls to EvalEX as compared to OWCTY's $\mathcal{O}(k^2 \cdot m + km) = \mathcal{O}(k^2 \cdot m)$. That is, the overhead of OWCTY relative to EL is linear. As for any Kripke structure OWCTY performs no more external iterations than EL, in practice, OWCTY almost always matches or improves on EL's performance.

Notice that in our case the efficiency of the algorithms is improved due to the saturation-based implementation of the function EvalEU.

6.4 Model Checking Based on Forward Traversals

Performance of the symbolic state space exploration depends heavily on the structure of the model. Sometimes, forward state traversals are quite efficient, whereas intermediate decision diagrams created during the backward state space exploration become too large and can not be handled efficiently. A CTL model checking algorithm based mainly on forward state traversals was suggested in [INH96]. The algorithm can check many realistic CTL properties without doing backward state space exploration.

The underlying idea of the algorithm is that the model checking problem can be translated into a problem of comparing a formula with the constant *False*. Let $M = [S, R, L, S_0]$ be a Kripke structure, $s \in S_0$ be some state and ϕ be an arbitrary CTL formula. Let p_0 be a CTL formula that holds only in the state[3] s_0, and P_0 be a formula that holds only in the states of the set S_0. The notation "$\models$" can be rewritten then as follows:

$$M, s_0 \models \phi \iff p_0 \wedge \phi \neq \mathit{False}$$

[3]Equivalently, we can say that p_0 is a characteristic function of the set $\{s_0\}$.

$$M, s_0 \models \phi \iff p_0 \wedge \neg\phi = \mathit{False}$$
$$\forall s \in S_0\ M, s \models \phi \iff P_0 \wedge \neg\phi = \mathit{False}.$$

We introduce temporal operators that can be evaluated in the forward manner. **EY** and **EH** are commonly used *past-tense* CTL operators, FwdUntil and FwdGlobal were proposed in [INH96].

- **EY** is a past-tense operator dual to **EX**, $\mathbf{EY}\phi$ holds in a state $s \in S$ if ϕ holds at least in one of the predecessors states of s

$$\mathrm{EvalEY}(S') = \{s \in S \mid \exists s' \in S' : (s', s) \in R\}.$$

- **EH** is a past-tense operator dual to **EG**, $\mathbf{EH}\phi = \nu Z.\, \phi \wedge \mathbf{EY}\ Z$.
- $\mathrm{FwdUntil}(\phi_1, \phi_2) = \mu Z.\, \phi_1 \vee \mathbf{EY}(Z \wedge \phi_2)$.
- $\mathrm{FwdGlobal}(\phi_1, \phi_2) = \mathbf{EH}(\mathrm{FwdUntil}(\phi_1, \phi_2) \wedge \phi_2))$.

Using these operators, a family of conversion rules which can replace outermost evaluations of **EX**, **EG**, and **EU** can be defined [INH96]

$$\begin{aligned}
\phi_1 \wedge \mathbf{EX}\phi_2 \neq \mathit{False} &\iff \mathbf{EY}\phi_1 \wedge \phi_2 \neq \mathit{False} \\
\phi_1 \wedge \mathbf{EX}\phi_2 = \mathit{False} &\iff \mathbf{EY}\phi_1 \wedge \phi_2 = \mathit{False} \\
\phi_1 \wedge \mathbf{E}[\phi_2 \mathbf{U} \phi_3] \neq \mathit{False} &\iff \mathrm{FwdUntil}(\phi_1, \phi_2) \wedge \phi_3 \neq \mathit{False} \\
\phi_1 \wedge \mathbf{E}[\phi_2 \mathbf{U} \phi_3] = \mathit{False} &\iff \mathrm{FwdUntil}(\phi_1, \phi_2) \wedge \phi_3 = \mathit{False} \\
\phi_1 \wedge \mathbf{EG}\phi_2 \neq \mathit{False} &\iff \mathrm{FwdGlobal}(\phi_1, \phi_2) \neq \mathit{False} \\
\phi_1 \wedge \mathbf{EG}\phi_2 = \mathit{False} &\iff \mathrm{FwdGlobal}(\phi_1, \phi_2) = \mathit{False}.
\end{aligned}$$

Notice that a problem of comparing a disjunctive expression with the constant *False*, such as "$\phi_1 \vee \phi_2 \neq \mathit{False}$", can be divided into subproblems, such as "$\phi_1 \neq \mathit{False}$" and "$\phi_2 \neq \mathit{False}$". Each term can be checked separately and if at least one of them is not the constant *False*, the entire expression is not the constant *False*. Notice also that not all CTL operators must be converted into forward traversal operators. Remaining operators can be evaluated in the usual manner, with backward state traversals.

Given a CTL formula ϕ, a conversion procedure must proceed as follows:

1. Rewrite ϕ to use only temporal operators **EX**, **EU**, and **EG**.
2. Translate the "$\models$" notation into an expression comparing a formula with the constant *False*.
3. Arrange outermost boolean operations in disjunctive normal form, and divide the problem into a set of subproblems comparing each term with *False*.

4. For each subproblem, convert an outermost operator **EX**, **EU** or **EG** using the conversion rules defined above, if applicable.

5. For each newly updated subproblem, call the procedure recursively from the step 3.

Applying this procedure, many typically used CTL formulas can be fully converted to forward state traversal problems.

Example 26
Let us consider the conversion of a quite often used CTL formula $\mathbf{AG}(p_1 \rightarrow \mathbf{AF}p_2)$

$$\begin{aligned}
M, s_0 \models \quad & \mathbf{AG}(p_1 \rightarrow \mathbf{AF}p_2) \\
\Longleftrightarrow \quad & M, s_0 \models \neg\mathbf{E}[\mathit{True}\mathbf{U}(p_1 \wedge \mathbf{EG}\neg p_2)] \\
\Longleftrightarrow \quad & p_0 \wedge \mathbf{E}[\mathit{True}\mathbf{U}(p_1 \wedge \mathbf{EG}\neg p_2) = \mathit{False} \\
\Longleftrightarrow \quad & \mathrm{FwdUntil}(p_0, \mathit{True}) \wedge (p_1 \wedge \mathbf{EG}\neg p_2) = \mathit{False} \\
\Longleftrightarrow \quad & (\mathrm{FwdUntil}(p_0, \mathit{True}) \wedge p_1) \wedge \mathbf{EG}\neg p_2) = \mathit{False} \\
\Longleftrightarrow \quad & \mathrm{FwdGlobal}((\mathrm{FwdUntil}(p_0, \mathit{True}) \wedge p_1), \neg p_2) = \mathit{False}.
\end{aligned}$$

Not every CTL formula can be converted to use only forward operators. Let us try to translate a formula $\phi = \mathbf{AGEF}p$.

$$\begin{aligned}
M, s_0 \models \quad & \mathbf{AGEF}p \\
\Longleftrightarrow \quad & M, s_0 \models \neg\mathbf{E}[\mathit{True}\mathbf{U}\neg\mathbf{E}[\mathit{True}\mathbf{U}p]] \\
\Longleftrightarrow \quad & p_0 \wedge \mathbf{E}[\mathit{True}\mathbf{U}\neg\mathbf{E}[\mathit{True}\mathbf{U}p]] = \mathit{False} \\
\Longleftrightarrow \quad & \mathrm{FwdUntil}(p_0, \mathit{True}) \wedge \neg E[\mathit{True}\mathbf{U}p] = \mathit{False}.
\end{aligned}$$

No more conversion rules are applicable, therefore both forward and backward traversals must be done to check this formula.

We aim at symbolic model checking of Petri nets, thus, to implement the described approach we must implement functions EvalEY, EvalFwdUntil, and EvalEH using operations defined in chapter 4. Implementing EvalEY is easy, we just have not to forget to simulate a self-loop in dead states.

$$\mathrm{EvalEY}(S') = \mathrm{Img}(S') \cup (\mathcal{D}_N \cap S').$$

A fixpoint characterization of the operator FwdUntil suggests the following implementation of the function $\mathrm{EvalFwdUntil}(S'_1, S'_2)$. We must compute all states reachable from the states in $S'_1 \cap S'_2$ over the paths in S'_2. We notice that it is exactly the set of states which must be returned by the function $\mathrm{LimFwdReach}(S'_1, S'_2)$ (recall the

definition in section 4.3.3). Thus, we can reuse Algorithm 13 and the forward based CTL model checking will immediately benefit from our saturation strategy. An algorithm implementing the function EvalEH replaces only the line *6* in the algorithm implementing EvalEG (Algorithm 20 on page 103) with

$$Q' := S' \cap \text{EvalEY}(Q').$$

Now, given a CTL formula ϕ to be verified, we convert it as described above and get a number of subproblems of the form $\phi_i \neq \mathit{False}$ or $\phi_i = \mathit{False}$. As usual, we evaluate ϕ_i starting from the innermost subformulas and obtain a set Z of states that satisfy ϕ_i. The set $Z = \emptyset$ if and only if $\phi_i = \mathit{False}$.

6.5 Counterexamples and Witnesses

An important feature of a model checker is the ability to generate *counterexamples* and *witnesses*. When this feature is enabled and the model checker determines that a formula with a universal path quantifier is not satisfied, it can find a computation which demonstrates why the negation of the formula is true. Likewise, when the model checker determines that a formula with an existential path quantifier is satisfied, it can find a computation that demonstrates why it is so. For example, if a formula $\mathbf{AG}p$ is not satisfied, a path to a state in which $\neg p$ holds will be generated. A counterexample for a universally quantified formula is a witness for the dual existentially quantified formula. Thus, it is enough to consider how to generate witnesses for the operators **EX**, **EU**, and **EG**. We consider how to implement a witnesses generation procedure in a symbolic CTL model checker for Petri nets. Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a bounded P/T net with extended arcs.

Suppose, we want to show that a formula $\mathbf{EX}\phi_1$ holds at least in one of the states in the set B. Let M be a set of states where ϕ_1 is satisfied. We need to find some transition $t \in T$ such that $\exists m' \in B$, $\exists m \in M$ and $m' \xrightarrow{t} m$. Another possibility is to find some dead state that belongs both to B and M. This behavior is implemented in the function ShowEX in Algorithm 24.

Suppose that we want to show that a formula $\phi = \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ holds at least in one of the states in the set $M \subseteq S$. Let S'_1 be a set of states satisfying ϕ_1 and S'_2 be a set of states satisfying ϕ_2. If $M \cap S'_2 \neq \emptyset$, then the witness is trivial, so let us assume that $M \cap S'_2 \neq \emptyset$. Starting with a set $Q_0 = M \cap S'_1$, we generate a sequence of *"onion-rings"* $Q_1, \ldots, Q_n$ as follows:

$$Q_i = (\text{Img}(Q_{i-1}) \cap S'_1) \setminus Q_\Sigma \quad \text{where } Q_\Sigma = \bigcup_{0 \leq j \leq i-1} Q_j.$$

The generation is stopped when either $\text{Img}(Q_n) \cap S'_2 \neq \emptyset$ or $Q_n = \emptyset$. Actually, we just do the limited forward breath-first search and save intermediate frontier sets. As Q_Σ is

Algorithm 24 (Witness generation for EX)

```
1  func ShowEX (M, B)
2     M' := M ∩ D_N ∩ B
3     if M' ≠ ∅ then
4        say("dead")
5        return M'
6     fi
7     forall t ∈ T do
8        M' := RevFire(M, t) ∩ B
9        if M' ≠ ∅ then
10          say(t)
11          return M'
12       fi
13    od
14     /* EX does not hold */
15    return ∅
16 end
```

Algorithm 25 (Auxiliary functions for the witness generation procedure)

```
1  func FindEU (M, S'_1, S'_2)
2     M' := M ∩ S'_2
3     if M' ≠ ∅ then return (M', ∅) fi
4     Q := M ∩ S'_1
5     Q_Σ := ∅
6     Stack := ∅
7     while Q ≠ ∅ do
8        push(Stack, Q)
9        Q_Σ := Q_Σ ∪ Q
10       M' := Img(Q) ∩ S'_2
11       if M' ≠ ∅ then break fi
12       Q := (Img(Q) ∩ S'_1) \ Q_Σ
13    od
14    if M' ≠ ∅ then return (M', Stack) fi
15    return (∅, ∅)      /* EU does not hold */
16 end
```

```
1  func Show (M, Stack)
2     M' := M
3     while Stack ≠ ∅ do
4        Q := pop(Stack)
5        M' := ShowEX(M', Q)
6     od
7     return M'
8  end
```

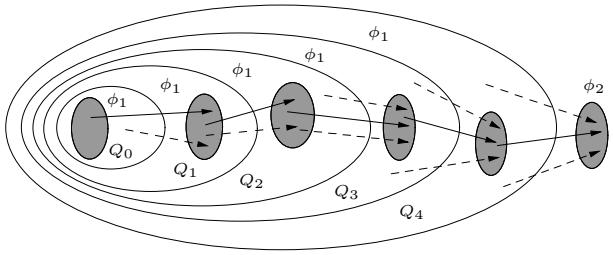


Figure 6.2: Generation of a witness for $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$

a monotonically increasing subset of S, the search terminates after a finite number of steps. $Q_n = \emptyset$ if ϕ is not satisfied in all states in M. Suppose $M'_n = \mathrm{Img}(Q_n) \cap S'_2$ and $M'_n \neq \emptyset$. In this case the witness (presented in the reversed order) can be generated by n calls $M'_i = \mathrm{ShowEX}(M'_{i+1}, Q_i)$ for $i = n-1, \ldots, 0$. The correctness of the procedure follows from the fact that

$$\text{for } 0 \leq i \leq n-1 \quad M'_i \subseteq Q_i \text{ and } Q_{i+1} \subseteq \mathrm{Img}\, Q_i.$$

Notice that due to the usage of the breath-first exploration we have generated the shortest possible witness. The described procedure is illustrated in Fig. 6.2. The sets M'_i are shaded, transitions chosen in ShowEX are drawn with solid, possible transitions that were not chosen, with dashed lines. Consider Algorithm 25. Generation of onion-rings is implemented in the function FindEU, it returns the *Stack* with onion-rings and the set M'_n. Demonstration of the witness is implemented in the function Show, it returns the set M'_0.

Let us consider the case when we must generate a witness for a set $M \subseteq S$ and a formula $\phi = \mathbf{EG}\phi_1$. Let Z be a set of states satisfying ϕ. The problem is easy if there exists some dead state of N that belongs to the set Z. Let G_D be an interval logic expression that describes the set $Z \cap \mathcal{D}_N$. We just need to generate a witness for the formula $\mathbf{E}[\phi_1 \mathbf{U} G_D]$. Let us assume that $Z \cap \mathcal{D}_N = \emptyset$, a witness has a form of a finite stem followed by a finite cycle in this case. We shall employ the symbolic algorithm which enumerates SCCs, recall section 4.4. Let C be the first non-trivial SCC found in the set Z. With a call $(M'_1, \mathit{Stack}_1) = \mathrm{FindEU}(M, Z, C)$ we find the first states in C that are reachable from states in M. We can search for a cycle now, so we pick some state $s \in M'_1$ and make a call $(M'_2, \mathit{Stack}_2) = \mathrm{FindEU}(\mathrm{Img}(\{s\}), C, \{s\})$. Since C is a non-trivial SCC, the cycle is guaranteed to be found. We announce the end of the cycle and make a call $M'_3 = \mathrm{Show}(\{s\}, \mathit{Stack}_2)$ to demonstrate (in the reversed order) a path from states in $\mathrm{Img}(\{s\})$ to s. The demonstration of the cycle is completed with a call $\mathrm{ShowEX}(M'_3, \{s\})$. Hereafter, the stem of the witness is demonstrated using onion-rings stored in Stack_1.

Generation of a witness for a formula $\phi = \mathbf{EG}_{fair}\phi_1$ is similar. We just need to find a fair non-trivial SCC and demonstrate a cycle that intersects all fair sets.

In procedures generating witnesses for $\mathbf{EG}\phi_1$ and $\mathbf{EG}_{fair}\phi_1$, the found SCC may be far away from the initial states and can induce a long witness. To generate witnesses with shorter stems we can adopt an approach proposed in [RBS00]. The SCC enumeration algorithm must be modified so, that sets to be decomposed are prioritized by the distance from the initial states (computed using onion-rings produced by the breath-first order traversal of the set Z).

We have to notice that due to the fact that generation of witnesses is based on the breath-first order exploration of the state space, this procedure becomes expensive when long witnesses must be generated and intermediate sets of states are not represented concisely by decision diagrams. Adopting the conventional algorithm [CGMZ95, CGP01] (only generation of witnesses for formulas $\mathbf{EG}_{fair}\phi$ is considered there) would result in the procedure that can require even more breath-first order traversals.

6.6 Closing Remark

Functions for the symbolic manipulations of Petri nets defined in chapter 4 allow a straightforward implementation of a symbolic CTL model checker. We have discussed how to make this implementation efficient.

Conventional symbolic CTL algorithms are based on the breath-first order exploration of the state space. Recall that any CTL formula can be expressed using only temporal operators $\mathbf{EX}, \mathbf{EU}$, and $\mathbf{EG}$. Correspondingly, the model checking algorithm employs functions EvalEX, EvalEU, and EvalEG. A single computation of EvalEX can be considered as a cheap operation. Our saturation-based implementation allows to improve significantly the efficiency of EvalEU. Unfortunately, we could not employ the saturation technique to improve the computation of EvalEG.

Performance of the symbolic state space exploration depends heavily on the structure of the model. Sometimes, forward state traversals are quite efficient, whereas intermediate decision diagrams created during the backward state space exploration become too large and can not be handled efficiently. A CTL model checking algorithm based mainly on forward state traversals [INH96] can be easily implemented using functions defined in chapter 4, moreover, it benefits also from the saturation technique.

More aggressive pruning strategies can improve efficiency of the Emerson-Lei algorithm, which is used in the CTL model checking with fairness constraints. We shall return to them in the next chapter where we shall discuss implementation of a symbolic LTL model checker for Petri nets.

7 Symbolic LTL Model Checking of Bounded Petri Nets

Implementing a symbolic LTL model checker for Petri nets is a challenging task. Implementation for 1-bounded P/T nets was considered in [Spr01]. We are not aware of any attempts to implement a symbolic LTL model checker for k-bounded nets.

The automata theoretic approach to LTL model checking consists of translating a model M and a negation of an LTL formula φ into Büchi automata A_M and $A_{\neg\varphi}$ and checking the product automaton $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ for emptiness. Construction of the automaton A_M for a Petri net N_M involves computation of its reachability set and suffers obviously from the state explosion problem. An approach based on the *Büchi net formalism* [EM97] was employed in [Spr01]. The underlying idea of the approach is to construct a *product Büchi net* instead of the product Büchi automaton and reduce then the model checking problem to a certain *net emptiness* problem.

When the net N_M has no reachable dead states, the approach employing Büchi nets is very elegant and has a number of advantages. However, it becomes unnatural when nets which can have reachable dead states must be verified. We forbear from the adaptation of the Büchi net formalism. The underlying idea of the used approach is to construct a product net in such a way that sets of its reachable markings can represent sets of states of the product automaton $A_{M,\varphi}$. Operations defined in chapter 4 are used to implement functions for the symbolic exploration of $A_{M,\varphi}$. These functions are employed then in the implementation of algorithms for the emptiness check of $A_{M,\varphi}$.

7.1 LTL and Petri Nets

Definition 51
Let $N = [P, T, F, I, R, Z, V, V_I, V_R, m_0]$ be a P/T net with extended arcs. A *run* of the net N is an infinite sequence of markings $m'_0, m'_1, m'_2, \ldots$ such that $m'_0 = m_0$ and for every $i \geq 0$ either $\exists t \in T \ : \ m'_i \xrightarrow{t} m'_{i+1}$ or $m'_i \in \mathcal{D}_N$ and $m'_{i+1} = m'_i$. A *language* $\mathcal{L}(N(m_0))$ of the net N is a set of all its runs.

For every bounded P/T net with extended arcs N we can construct a Büchi automaton $A_M = [\Sigma, Q_M, \Delta_M, q_{0_M}, Q_M]$ such that $\mathcal{L}(A_M) = \mathcal{L}(N(m_0))$ as follows:

- $\Sigma = M_N$

- $Q_M = \mathcal{R}_N(m_0) \cup \{q_{0_M}\}$
- $(m, m', m') \in \Delta_M$ for $m, m' \in (Q_M \setminus \{q_{0_M}\})$ if and only if $\exists t \in T : m \xrightarrow{t} m'$ or $m \in \mathcal{D}_N$ and $m' = m$
- $(q_{0_M}, m, m) \in \Delta_M$ if and only if $m = m_0$.

Given a set of initial markings $I_0 = \{m_{0_1}, \ldots, m_{0_k}\}$, we can construct a Büchi automaton A_{MI_0} such that $\mathcal{L}(A_{MI_0}) = \bigcup_{m \in I_0} \mathcal{L}(N(m))$ if we modify the definition as follows:

- $Q_M = \bigcup_{m \in I_0} \mathcal{R}_N(m) \cup \{q_{0_M}\}$
- $(q_{0_M}, m, m) \in \Delta_M$ if and only if $m \in I_0$.

We assume that places of N are enumerated $P = \{p_1, \ldots, p_n\}$. As a set of atomic propositions for LTL formulas we can define a set containing interval logic expressions over the variables $p_1, \ldots, p_n$. We shall use Büchi automata produced by the procedure described in section 5.3.3. Recall that it creates automata with *edges* labeled by boolean expressions over atomic propositions. Let $A_{\neg\varphi} = [\Sigma, Q_{\neg\varphi}, \Delta_{\neg\varphi}, q_{0_{\neg\varphi}}, \mathcal{F}_{\neg\varphi}]$ be such an automaton, $\mathcal{F}_{\neg\varphi} = \{F_{\neg\varphi 1}, \ldots, F_{\neg\varphi n}\}$. We can also treat $\Delta_{\neg\varphi}$ directly as a set of edges, notice that due to Definition 25, labels on the edges are again interval logic expressions. The product automaton $A_{M,\varphi} = A_M \cap A_{\neg\varphi} = [\Sigma, Q_{M,\varphi}, \Delta_{M,\varphi}, q_{0_{M,\varphi}}, \mathcal{F}_{M,\varphi}]$ is defined then as follows:

- $\Sigma = M_N$
- $Q_{M,\varphi} = Q_M \times Q_{\neg\varphi}$
- $((m, q), m', (m', q')) \in \Delta_{M,\varphi}$ if and only if $(m, m', m') \in \Delta_M$ and there exists $(q, G, q') \in \Delta_{\neg\varphi}$ such that $m' \in M_G$ (M_G denotes a set of markings described by the expression G).
- $q_{0_{M,\varphi}} = (q_{0_M}, q_{0_{\neg\varphi}})$
- $\mathcal{F}_{M,\varphi} = \{F_1, \ldots, F_n\}$ where $F_i = (Q_M \times F_{\neg\varphi i})$.

7.2 Product Net

An explicit construction and exploration of the product automaton $A_{M,\varphi}$ defined in the previous section suffers obviously from the state explosion problem. We can try to adopt an approach based on the *Büchi net formalism*. The approach was introduced in [EM97] to implement a *semidecision* test whether a 1-bounded P/T net satisfies an LTL formula φ (a procedure based on the T-invariants analysis which might answer

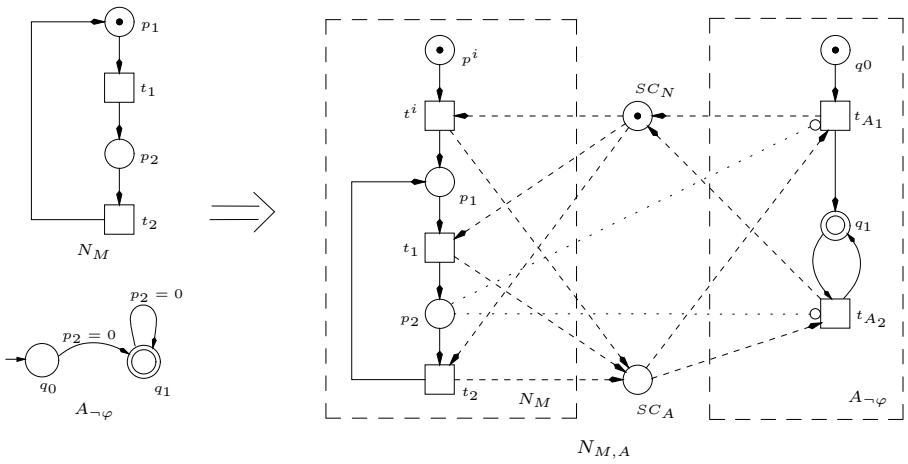


Figure 7.1: Example construction of a product Büchi net

either "don't know", or "yes", in which case φ was satisfied). The underlying idea of the approach was to construct a *product Büchi net* instead of the product Büchi automaton. The model checking problem was reduced then to a certain *net emptiness* problem, very similar to the emptiness problem of Büchi automata. This approach was employed in [Spr01] for the symbolic LTL model checking of 1-bounded P/T nets. The following example should give an intuition how we can uplift the technique to k-bounded P/T nets with extended arcs.

A net N_M, a Büchi automaton $A_{\neg\varphi}$ (obtained from the LTL formula $\neg\varphi = \mathbf{G}(p_2 = 0)$), and a corresponding product Büchi net $N_{M,A}$ are shown in Fig. 7.1. The places SC_N and SC_A are used to guaranty that the net counterparts corresponding to N_M and $A_{\neg\varphi}$ alternate their steps. The $A_{\neg\varphi}$ counterpart can be seen as an "observer", which monitors markings of N_M and allows to recognize its accepting runs. Every edge $(q, G, q') \in \Delta_{\neg\varphi}$ is represented by a transition, which is connected with *contextual* arcs with places of N_M and can fire only in markings of N_M which belong to the set described by the expression G. So, inhibitor arcs connect the place p_2 with transitions t_{A_1} and t_{A_2}. The Büchi net formalism defines a notion of accepting places and a notion of the net emptiness. A Büchi net is not empty if some marking in which an accepting place contains a token lies on a cycle in its reachability graph. In our example, the place q_1 is accepting and the net $N_{M,A}$ is empty (it is easy to see that its reachability graph contains no cycles).

The main problem of the sketched construction of the product Büchi net $N_{M,A}$ is that it can correctly represent the behavior of the product automaton $A_{M,\varphi}$ (i.e. $N_{M,A}$ is empty if and only if $A_{M,\varphi}$ is empty) only when the net N_M has no reachable dead states. Suppose that the transition t_1 does not exist in N_M, obviously, $A_{M,\varphi}$ is not

empty then. There exits an accepting run

$$(q_{0_M}, q_0),\ (m_0, q_1),\ (m_0, q_1),\ (m_0, q_1), \ldots$$

However, the net $N_{M,A}$ is empty in this case: when t_1 is missing, the reachability graph of $N_{M,A}$ contains only three markings and no cycles. The construction of $N_{M,A}$ requires also that contextual arcs can be used to allow transitions corresponding to the edges of $A_{\neg\varphi}$ fire only in markings which belong to sets described by expressions on these edges. We can deal with both of these problems.

It is easy to see that for any interval logic expression G which is a conjunction of *atomic* interval logic expressions we can create a number of contextual arcs that allow a transition t to fire exactly in markings described by G. For every atomic expression $p_i \in [a, b)$ we create two arcs connecting the place p_i and the transition t: a read arc with a weight a and an inhibitor arc with a weight b. We do not need to create the read arc when $a = 0$ and the inhibitor arc when $b = \infty$.

Thus, if it is not known beforehand that the net N_M has no reachable dead markings, then dummy transitions which can fire in dead markings can be added to the net. To obtain them we convert the characteristic function $\chi_{\mathcal{D}_{N_M}}$ into the disjunctive normal form and create a transition for each term in it. Analogously, if an expression on some edge of $A_{\neg\varphi}$ is not a conjunction of atomic interval logic expressions, we can split the edge into a number of edges which satisfy this requirement. Thus, from the theoretical viewpoint, we can always construct a product Büchi net $N_{M,A}$ which represents correctly the behavior of a product automaton $A_{M,\varphi}$.

However, from the practical viewpoint, due to the explosion of transitions the construction of $N_{M,A}$ has more drawbacks than advantages. We construct instead a "not synchronized" product net $N_\times$ in such a way that sets of its reachable markings can represent sets of states of the product automaton $A_{M,\varphi}$. We shall use operations defined in chapter 4 and implement functions for the symbolic exploration of $A_{M,\varphi}$. We forbear from the formal introduction of the Büchi net formalism and continue to reason in terms of Büchi automata.

When constructing $N_\times$ for the net N_M and the automaton $A_{\neg\varphi}$ in Fig. 7.1, we create neither places SC_A and SC_N nor all dashed arcs present in the net $N_{M,A}$. Formally, the construction is defined as follows.

Definition 52 (Product net)
Let $N_M = [P_M, T_M, F_M, I_M, R_M, Z_M, V_M, V_{I_M}, V_{R_M}, m_{0_M}]$ be a P/T net with extended arcs and let $A_{\neg\varphi} = [\Sigma, Q_{\neg\varphi}, \Delta_{\neg\varphi}, q_{0_{\neg\varphi}}]$ be a Büchi automaton obtained from an LTL formula $\neg\varphi$. Let us assume that edges and fair sets of $A_{\neg\varphi}$ are enumerated:

$$\Delta_{\neg\varphi} = \{(q'_1, G_1, q''_1), \ldots, (q'_n, G_n, q''_n)\}, \qquad \mathcal{F}_{\neg\varphi} = \{F_{\neg\varphi 1}, \ldots, F_{\neg\varphi m}\}.$$

We construct the product net $N_\times = [P, T, F, I_M, R_M, Z_M, V, V_{I_M}, V_{R_M}, \mathbf{m_0}]$ as follows:

- $P = \{p^i\} \cup P_M \cup Q_{\neg\varphi}$
- $T = \{t^i\} \cup T_M \cup T_A$, where $T_A = \{t_{A_1}, \ldots, t_{A_n}\}$
- $F = F_M \cup$
  $\{(p^i, t^i)\} \cup$
  $\{(t^i, p) \mid \exists p \in P_M : m_{0_M}(p) > 0\} \cup$
  $\{(q'_k, t_{A_k}), (t_{A_k}, q''_k) \mid \exists (q'_k, G_k, q''_k) \in \Delta_{\neg\varphi}\}$
- $V(f) = V_M(f)$ for all $f \in F_M$,
  $V((p^i, t^i)) = 1$,
  $V((t^i, p)) = m_{0_M}(p)$ for all $(t^i, p) \in F$,
  $V((q'_k, t_{A_k})) = 1$ for all $(q'_k, t_{A_k}) \in F$,
  $V((t_{Ak}, q''_k)) = 1$ for all $(t_{A_k}, q'_k) \in F$
- $\mathbf{m_0}(p) = 1$ if $p = p^i$ or $p = q_{0_{\neg\varphi}}$ otherwise $\mathbf{m_0}(p) = 0$.

We shall use the following notations:

1. $\mathcal{M}_\times$ denotes a set of all possible markings of $N_\times$.
2. $\mathcal{R}_{N_\times}$ denotes a set of markings $\mathcal{R}_{N_\times}(\mathbf{m_0})$.
3. χ_{D_t} denotes a characteristic function for markings in which a transition t can not be enabled (recall section 4.2.2).
4. $\mathcal{D}_M \in 2^{\mathcal{M}_\times}$ denotes a set of markings described by a characteristic function $\chi_{\mathcal{D}_M} = \bigwedge_{t \in T_M} \chi_{D_t}$.
5. A function Cnd : $T_A \to 2^{\mathcal{M}_\times}$ takes a transition t_{A_i} as argument and returns a set of markings described by the expression G_i on the edge $(q'_i, G_i, q''_i) \in \Delta_{\neg\varphi}$.
6. $\mathcal{F}_\times = \{F_1, \ldots, F_m\}$ denotes a set of sets of markings, every $F_i \in 2^{\mathcal{M}_\times}$ is described by a characteristic function $\chi_{F_i} = \bigvee_{q \in F_{\neg\varphi i}} (q = 1)$.

Lemma 10
Let $N_\times$ be a product net constructed for N_M and $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. Then $|\mathcal{R}_{N_\times}| = |Q_{M,\varphi}|$.

Proof: Due to the construction, $N_\times$ consists of two subnets: N_1, which is induced by places in $\{p^i\} \cup P_M$ and transitions in $\{t^i\} \cup T_M$, and N_2, which is induced by places in $Q_{\neg\varphi}$ and transitions in T_A. In its turn, N_1 consists of the net N_M, the place p^i, and the transition t^i. t^i can fire only once, its firing produces the initial marking of N_M, thus N_1 has $|\mathcal{R}_{N_M}(m_0)| + 1$ markings. N_2 is a *state machine*: every transition has exactly one pre-place and one post-place. There exists one token in the place q_0 in the

initial marking of N_2, thus in all reachable markings of N_2 only one of the places in $Q_{\neg\varphi}$ contains a token. All states of $A_{\neg\varphi}$ are reachable from its initial state, hence N_2 contains $|Q_{\neg\varphi}|$ markings. The subnets N_1 and N_2 are not connected, therefore the state space of $N_\times$ is a product of the state spaces of these subnets. Due to the construction of A_M and $A_{M,\varphi}$

$$|Q_{M,\varphi}| = (|\mathcal{R}_{N_M}(m_0)| + 1) \cdot |Q_{\neg\varphi}|. \qquad \square$$

We can say that every marking $\mathbf{m} \in \mathcal{R}_{N_\times}$ *represents* some state of the product automaton $A_{M,\varphi}$. We define a one-to-one function ϱ mapping states of $A_{M,\varphi}$ to markings in $\mathcal{R}_{N_\times}$ as follows: $\varrho(m, q_i) = \mathbf{m}$ if $\mathbf{m}(q_i) = 1$, $\mathbf{m}(q_k) = 0\ \forall k \neq i$ and

- either $m = q_{0_M}$ and $\mathbf{m}(p^i) = 1$, $\mathbf{m}(p) = 0\ \forall p \in P_M$
- or $m \neq q_{0_M}$ and $\mathbf{m}(p^i) = 0$, $\mathbf{m}(p) = m(p)\ \forall p \in P_M$.

A marking $\mathbf{m} \in \mathcal{R}_{N_\times}$ represents a state (m, q_i) of the product automaton $A_{M,\varphi}$ if $\varrho(m, q_i) = \mathbf{m}$. A set of markings $K \in 2^{\mathcal{R}_{N_\times}}$ represents a set of states K' of $A_{M,\varphi}$ if $|K| = |K'|$ and every marking in K represents some state in K'. It is easy to see that $\mathbf{m_0}$ represents the initial state of $A_{M,\varphi}$, and that fair sets of $A_{M,\varphi}$ are represented by sets in $\mathcal{F}_\times$ intersected with $\mathcal{R}_{N_\times}$.

As we can represent sets of states of Petri nets symbolically, we have also got a possibility to represent symbolically sets of states of the product automaton $A_{M,\varphi}$. Now we shall implement functions for the symbolic exploration of $A_{M,\varphi}$ using functions defined in chapter 4.

We define functions $\mathrm{Post_M} : 2^{\mathcal{R}_{N_\times}} \to 2^{\mathcal{R}_{N_\times}}$ and $\mathrm{Post_A} : 2^{\mathcal{R}_{N_\times}} \to 2^{\mathcal{R}_{N_\times}}$ as follows:

$$\begin{aligned}\mathrm{Post_M}(K) = \{\mathbf{m}' \in \mathcal{R}_{N_\times} \mid\ & \exists \mathbf{m} \in K,\ \exists t \in T_M\ : \\ & \varrho(m, q) = \mathbf{m}, \varrho(m', q) = \mathbf{m}', m \xrightarrow{t} m'\}\end{aligned}$$

$$\begin{aligned}\mathrm{Post_A}(K) = \{\mathbf{m}' \in \mathcal{R}_{N_\times} \mid\ & \exists \mathbf{m} \in K,\ \exists (q, G_C, q') \in \Delta_{\neg\varphi}\ : \\ & \varrho(m, q) = \mathbf{m}, \varrho(m, q') = \mathbf{m}', m \in C\}.\end{aligned}$$

Implementation of these functions is trivial due to the construction of the net $N_\times$

$$\mathrm{Post_M}(K) = \bigcup_{t \in T_M} \mathrm{Fire}(K, t), \qquad \mathrm{Post_A}(K) = \bigcup_{t \in T_A} \mathrm{Fire}(K \cap \mathrm{Cnd}(t), t).$$

Due to the construction of A_M, $A_{\neg\varphi}$, and $A_{M,\varphi}$, the product automaton $A_{M,\varphi}$ can change its state from (m, q) to (m', q') if and only if

1. a) There exists $t \in T_M$ such that $m \xrightarrow{t} m'$ or

b) $m \in \mathcal{D}_N$ and $m' = m$ or
c) $m = q_{0_M}$ and $m' = m_{0_M}$.

2. There exists $(q, G, q') \in \Delta_{\neg\varphi}$ and m' belongs to the set described by G.

This behavior is implemented in the function $\mathrm{Img}_\times : 2^{\mathcal{R}_{N_\times}} \to 2^{\mathcal{R}_{N_\times}}$

$$\mathrm{Img}_\times(K) = \mathrm{Post_A}(\mathrm{Post_M}(K) \ \cup \ K \cap \mathcal{D}_M \ \cup \ \mathrm{Fire}(K, t^i)).$$

Lemma 11
Let K_A be a set of states of the product automaton $A_{M,\varphi}$ represented by a set of markings $K \in 2^{\mathcal{R}_{N_\times}}$. $\mathrm{Img}_\times(K)$ returns a set of markings which represents the set of all direct successors states of K_A in $A_{M,\varphi}$

$$\{(m', q') \in Q_{M,\varphi} \mid \exists (m, q) \in K_A, \ \exists ((m, q), m', (m', q')) \in \Delta_{M,\varphi}\}.$$

We implement a function $\mathrm{FwdReach}_\times : 2^{\mathcal{R}_{N_\times}} \to 2^{\mathcal{R}_{N_\times}}$ replacing a call to the function Img with a call to the function $\mathrm{Img}_\times$ in Algorithm 8 (page 62). Now a set of markings representing a set of states of the product automaton $A_{M,\varphi}$ reachable from $q_{0_{M,\varphi}}$ can be computed as $Q'_{M,\varphi} = \mathrm{FwdReach}_\times(\{\mathbf{m_0}\})$. To avoid problems with large intermediate diagrams that appear during the breath-first exploration made in Algorithm 8, we can try to adopt our saturation strategy, recall section 4.3.2.
First, we implement a function $\mathrm{Fire}_\times : 2^{\mathcal{R}_{N_\times}} \times (T_M \cup \{t^i\}) \to 2^{\mathcal{R}_{N_\times}}$

$$\mathrm{Fire}_\times(K, t) = \mathrm{Post_A}(\mathrm{Fire}(K, t)).$$

It is easy to see that $\mathrm{Fire}_\times(K, t) \subseteq \mathrm{Img}_\times(K)$ and that

$$\mathrm{Img}_\times(K) = \bigcup_{t \in T_M \cup \{t^i\}} (\mathrm{Fire}_\times(K, t)) \ \cup \ \mathrm{Post_A}(K \cap \mathcal{D}_M).$$

For the net $N_\times$ we define the ROIDD variable ordering π as follows: we assign first variables in the ordering to places in $Q_{\neg\varphi}$ and p^i. The remaining variables, assigned to the places in P_M, are ordered exactly as variables assigned to places of the net N_M. We define the linear order σ for transitions in T_M exactly as described in section 4.3.2, and assume that they are enumerated accordingly to this order.

Consider Algorithm 26 which computes a set of markings representing a set of all states of $A_{M,\varphi}$ reachable from states represented by markings in the set K. It differs only slightly from the original Algorithm 11 (page 71). We compute now fixpoints with respect to the function $\mathrm{Fire}_\times$ and take finally care that states corresponding to dead markings of the net N_M are handled correctly. Of course, more computations are done in the new algorithm and more intermediate diagrams are created. As firing of every

Algorithm 26 (Forward reachability analysis with saturation)

```
1  func FwdReach_× (K)
2      Reached := K ∪ Fire_×(K, t^i)
3      i := 1
4      repeat
5          Old := Reached
6          repeat
7              Old2 := Reached
8              Reached := Reached ∪ Fire_×(Reached, t_σi)
9          until Reached = Old2
10         if Reached = Old then
11             i := i + 1
12         else
13             j := FirstDep(t_σi)
14             if j = i then i := i + 1 else i := j fi
15         fi
16     until i = |T_M| + 1
17
18     D := Reached ∩ D_M
19     if D ≠ ∅ then
20         repeat
21             Old := Reached
22             Reached := Reached ∪ Post_A(Reached)
23         until Reached = Old
24     fi
25     return Reached
26 end
```

transition in T_M is followed by firing of all transitions in T_A, the approach is promising when the automaton $A_{\neg\varphi}$ is relatively small. Automata for many typically used LTL formulas are indeed small (up to several tens of states and edges) and improvements in efficiency achieved by switching to Algorithm 26 are comparable to the ones achieved by switching from the BFS Algorithm 8 to the saturation Algorithm 11.

Analogously[1] to the functions $\mathrm{Post_M}$, $\mathrm{Post_A}$, $\mathrm{Img}_\times$, and $\mathrm{Fire}_\times$, we implement the functions $\mathrm{Prev_M}$, $\mathrm{Prev_A}$, $\mathrm{PreImg}_\times$, and $\mathrm{RevFire}_\times$. We implement also the functions for the limited reachability analysis $\mathrm{LimFwdReach}_\times$ and $\mathrm{LimBwdReach}_\times$. Consider, for example, the function $\mathrm{LimFwdReach}_\times(K,\ C)$. Its implementation modifies slightly Algorithm 26: the initial set *Reached*, as well as every set returned by the functions $\mathrm{Fire}_\times$ and $\mathrm{Post_A}$ is intersected with C. Equipped with the functions $\mathrm{Img}_\times$, $\mathrm{PreImg}_\times$, $\mathrm{LimFwdReach}_\times$, and $\mathrm{LimBwdReach}_\times$, which allow the exploration of the product automaton $A_{M,\varphi}$, we can proceed to implement algorithms for the symbolic emptiness check. For the sake of simplicity, we shall abstract from the implementation details and reason so, as if these functions manipulate directly on sets of states of $A_{M,\varphi}$.

7.3 Computing Emptiness

7.3.1 SCC-hull Algorithms

Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ be a generalized Büchi automaton. The classic algorithm to decide emptiness of A is the Emerson-Lei algorithm [EL86], which computes a set K of all states that have a path to some fair SCC by evaluating the nested fixpoint formula

$$K = \nu Y.\ \bigcap_{\forall F \in \mathcal{F}} \mathrm{PreImg}(\mu Z.\, Y \cap (F \cup \mathrm{PreImg}(Z))).$$

Evaluation of this formula requires two nested loops. The inner loop identifies the states from which there exists a path that is contained in Y and reaches a state in F. It finds these states by starting with states in $Y \cap F$ and extending the paths backward. Hereafter, the predecessors of these states are computed using PreImg to eliminate states that are not on cycles. The procedure is repeated for each fair set. Every *terminal* SCC in the computed set K is fair. The Büchi automaton A is empty if its initial state q_0 does not belong to the computed set K. The obvious complexity bound of the algorithm is $\mathcal{O}(|\mathcal{F}| \cdot |Q|^2)$. It can be sharpen [BGS00] to $\mathcal{O}(|\mathcal{F}| \cdot d \cdot h)$, where d is the diameter of A and h is the diameter of its SCC quotient graph.

A variation of the algorithm proposed in [HTKB92] is based on forward traversals and replaces PreImg computations made in the algorithm with Img computations. It computes a set K containing all fair SCCs and states reachable from them. Every *initial* SCC in the computed set K is fair. The sets computed by the algorithms are illustrated

[1] Of course, taking care of the right sequence of the application of transitions in T_M and T_A.

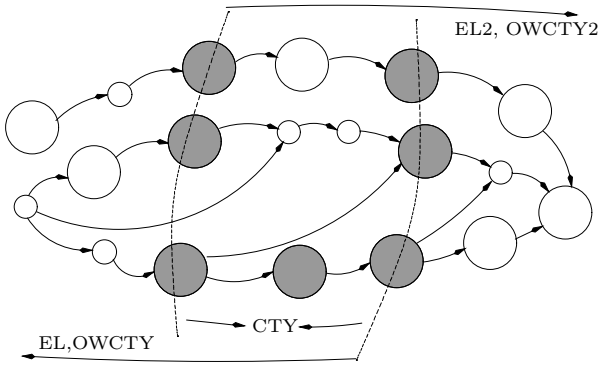


Figure 7.2: An SCC quotient graph of Q. Shaded circles are fair SCCs, big circles are non-trivial SCCs, small circles are trivial SCCs.

in Fig. 7.2, the original algorithm is denoted as EL, the forward variation as EL2. The automaton A is empty if no states in the set $K \cap \bigcap_{\forall F \in \mathcal{F}} F$ are reachable from q_0.

Let $N_\times$ be a product net constructed for N_M and $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. A procedure CheckOWCTY in Algorithm 27 allows to check emptiness of $A_{M,\varphi}$. It uses functions defined in the previous section and employs the pruning strategy of the OWCTY algorithm introduced in section 6.3. $Q'_{M,\varphi}$ is a set of states reachable from the initial state of $A_{M,\varphi}$. We limit the search for fair SCCs and states that have a path to them to the set $Q'_{M,\varphi}$. Hence, it is enough to check whether the computed set K is empty to decide emptiness of $A_{M,\varphi}$. We implement the forward variation of the algorithm analogously: in CheckOWCTY2, the calls to the functions $\text{LimBwdReach}_\times$ and $\text{PreImg}_\times$ are replaced with the calls to $\text{LimFwdReach}_\times$ and $\text{Img}_\times$.

A Catch-Them-Young (CTY) algorithm proposed in [HKSV97] aggressively prunes the set of states potentially lying on accepting cycles, a set of states which it computes is also illustrated in Fig. 7.2. In the adapted to our needs implementation a computation on the line *7* of Algorithm 27 is replaced with

$$K := \text{LimBwdReach}_\times(F \cap K, K) \cap \text{LimFwdReach}_\times(F \cap K, K),$$

and a computation on the line *10* is replaced with

$$K := K \cap \text{PreImg}_\times(K) \cap \text{Img}_\times(K).$$

Experiments made in [FFK$^+$01] showed that usually too many computations are made in the CTY algorithm and it is significantly outperformed by the OWCTY. We notice also that simultaneous backward and forward pruning of the set K results usually in intermediate sets of states that are encoded by larger decision diagrams.

Algorithm 27 (Emptiness check)

```
1  proc CheckOWCTY(N_×)
2     Q'_{M,φ} := FwdReach_×(m_0)
3     K := Q'_{M,φ}
4     repeat
5        Old := K
6        forall F ∈ F_× do
7           K := LimBwdReach_×(F ∩ K, K)
8           repeat
9              Old2 := K
10             K := K ∩ PreImg_×(K)
11          until Old2 = K
12       od
13    until (Old = K)
14    if K ≠ ∅ then
15       exit ("a fair SCC is found")
16    fi
17 end
```

The EL algorithm and its variations are often denoted as *SCC-hull* algorithms [RBS00]. They compute an SCC-hull, that is, a set of states that contains all fair SCCs without enumeration of these SCCs. Symbolic SCC-enumeration algorithms can be also employed to decide emptiness of Büchi automata.

7.3.2 Algorithm Based on SCC-enumeration

Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ be a generalized Büchi automaton. A is not empty if there exists a fair SCC reachable from q_0.

Let $N_\times$ be a product net constructed for N_M and $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. We use functions defined in section 7.2 for the exploration of $A_{M,\varphi}$ to implement an algorithm that enumerates SCCs in sets of states of $A_{M,\varphi}$ (recall section 4.4.3). The procedure is adapted as follows. Before decomposing a set of states S, we check whether it intersects all sets in $\mathcal{F}_\times$, S can be skipped if it is not the case as it can not contain any fair SCCs then. Instead of taking a random state $s \in S$ we take some state belonging both to S and some set in $\mathcal{F}_\times$. When a Büchi automaton is empty, it often contains many accepting states forming trivial SCCs. Thus, to avoid the expensive enumeration of trivial SCCs, the trimming routines must be enabled in the decomposition procedure. To decide emptiness of $A_{M,\varphi}$ we compute the set of all its reachable states $Q'_{M,\varphi} = \mathrm{FwdReach}_\times(\mathbf{m_0})$ and decompose it into SCCs. Once an SCC is found, we check if it intersects all sets in $\mathcal{F}_\times$ and stop the

process announcing non-emptiness if so. Though the SCC-enumeration algorithm has a better worst case complexity $\mathcal{O}(\log(|Q'_{M,\varphi}|) \cdot |Q'_{M,\varphi}|)$ than SCC-hull algorithms, in practice its performance is often inferior to that of SCC-hull algorithms. Experiments made in [RBS00] corroborate our observations.

7.3.3 Algorithms for Weak and Terminal Automata

The emptiness check is easy for weak and terminal Büchi automata [BRS99] (recall Definition 48 on page 108).

Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ be a weak Büchi automaton. Any SCC of A contains either only accepting states or only non-accepting states. Thus, the only way for a run to visit an accepting state infinitely often is to eventually be confined inside one non-trivial SCC that contains only accepting states.

Let $N_\times$ be a product net constructed for N_M and a weak Büchi automaton $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. Recall that $A_{M,\varphi}$ can not be stronger than $A_{\neg\varphi}$. We assume that $A_{\neg\varphi}$ was simplified according to Lemma 8 and the set $\mathcal{F}_\times$ contains only one set F_1. To decide emptiness of $A_{M,\varphi}$ we compute first the set of all its reachable states $Q'_{M,\varphi} = \mathrm{FwdReach}_\times(\mathbf{m_0})$. Backward or forward trimming can be used to check if the set $Q'_{M,\varphi} \cap F_1$ contains at least one non-trivial SCC. Forward trimming is employed in Algorithm 28.

Let $A = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ be a terminal Büchi automaton simplified according to Lemma 8. A run of A which reaches some accepting state is guaranteed to be confined inside a terminal SCC containing accepting states.

Let $N_\times$ be a product net constructed for N_M and a terminal Büchi automaton $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. We assume that $A_{\neg\varphi}$ was simplified according to Lemma 8 and the set $\mathcal{F}_\times$ contains only one set F_1. We can decide emptiness of the product automaton $A_{M,\varphi}$ "on-the-fly" during the computation of $Q'_{M,\varphi}$. The automaton $A_{M,\varphi}$ is not empty if we meet some state

Algorithm 28 (Emptiness check for weak automata)

```
1  proc CheckWeak2(N_×)
2     Q'_{M,φ} := FwdReach_×(m_0)
3     K := Q'_{M,φ} ∩ F_1
4     repeat
5        Old := K
6        K := K ∩ Img_×(K)
7     until Old = K
8     if K ≠ ∅ then
9        exit ("accepting cycle found")
10    fi
11 end
```

belonging to the set F_1.

Obviously, the discussed algorithms perform only linear number of steps. Fortunately, translation of many typically used LTL formulas produces weak and terminal Büchi automata.

7.3.4 On-the-fly Algorithm

Explicit-state LTL model checkers typically check emptiness of the product automaton $A_{M,\varphi}$ "on-the-fly", i.e. while constructing it. Thus, the model checker may be able to find an accepting cycle without ever constructing the complete state space of $A_{M,\varphi}$. The best known on-the-fly algorithms use deep-first search (DFS) strategies to explore the state space. Among the discussed above symbolic algorithms for the emptiness check only the algorithm for terminal Büchi automata can decide emptiness of $A_{M,\varphi}$ on-the-fly. An algorithm which combines the approach of explicit-state model checkers with the symbolic representation of sets of states and is capable of deciding emptiness of strong automata on-the-fly was suggested in [Spr01]. Actually, the proposed *Symbolic DFS* algorithm was designed to decide emptiness of Büchi nets, but it can be straightforwardly adopted for checking emptiness of not generalized Büchi automata.

Let $A = [\Sigma, Q, \Delta, q_0, F]$ be a Büchi automaton. The underlying idea of the approach is to construct and explore a so called *M-graph* $\mathcal{MG}(A)$, whose nodes consist of sets of states of the automaton A. The M-graph $\mathcal{MG}(A)$ is used to reason about accepting cycles of A. As decision diagrams can be used to encode sets of states of A, its M-graph can, in principle, be represented more concisely than A.

We shall use a notation $q \xrightarrow{A+} q'$ to denote that the node q' is reachable from a node q in the automaton A. Consider Algorithm 29. A function MakeMG$_1$ gets a Büchi automaton A as an argument and constructs an M-graph $\mathcal{MG}(A) = [V, E]$ using a simple DFS and the function Img(K), which returns a set of all direct successors states of K. The construction is illustrated in Fig. 7.3 (we omit labels on arcs of the automaton as they are irrelevant for the example).

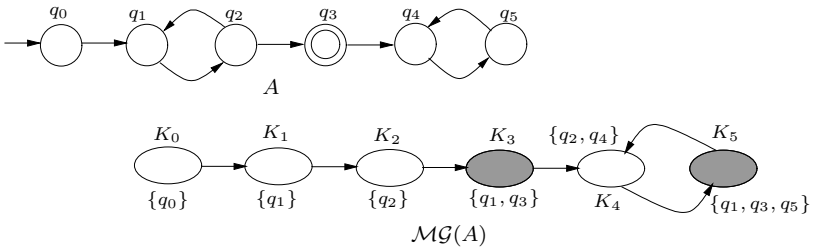


Figure 7.3: A Büchi automaton and its M-graph

Algorithm 29 (M-graph construction)

```
1  func MakeMG1 (A)
2      V := ∅; E := ∅
3      proc visit (K)
4          V := V ∪ {K}
5          K' := Img(K)
6          E := E ∪ {[K, K']}
7          if K' ∉ V then visit(K') fi
8      end
9      begin
10         visit({q0})
11         return [V, E]
12     end
```

Lemma 12
Let A be a Büchi automaton, $\mathcal{MG}(A) = [V, E]$ be an M-graph constructed using the function MakeMG$_1$*, and let $K \in V$ be a node of $\mathcal{MG}(A)$ such that $K \xrightarrow{+} K$. There exists $q \in K : q \xrightarrow{A+} q$.*

Proof: Let n be a number of states in K. Due to the construction of $\mathcal{MG}(A)$, for every state q'_1 there exists $q'_2 \in K : q'_2 \xrightarrow{A+} q'_1$. Analogously, there exist $q'_3 \in K : q'_3 \xrightarrow{A+} q'_2$. We can repeat this consideration n times and get a sequence of $n+1$ states

$$q'_{n+1} \xrightarrow{A+} q'_n \xrightarrow{A+} \ldots \xrightarrow{A+} q'_1.$$

As K has only n states, at least two states in the sequence must be identical. □

Notice that Lemma 12 does not state that for *all* states $q \in K$ holds $q \xrightarrow{A+} q$. Consider Fig. 7.3, though a state q_3 belongs to the node K_5, which lies on a cycle in $\mathcal{MG}$(A), q_3 does not lie on any cycles in the automaton A. As we aim at finding accepting cycles in A, we must partition the nodes of $\mathcal{MG}(A)$ into accepting and non-accepting. Notice also that as sets of states in different nodes of $\mathcal{MG}(A)$ can overlap, a number of nodes in $\mathcal{MG}(A)$ corresponds to the number of different subsets of Q. Thus, in the worst case, a number of nodes of the constructed M-graph $\mathcal{MG}(A)$ is exponential in the number of states of the automaton A. However, it can be shown that the size of $\mathcal{MG}(A)$ can be kept linear in the size of Q. The idea is to maintain the following invariant: let K_1 and K_2 be two nodes of $\mathcal{MG}(A)$, then $K_1 \cap K_2 = \emptyset$, $K_1 \subseteq K_2$ or $K_2 \subseteq K_1$.

Lemma 13
Let $P \subseteq 2^S$ be a set of nonempty subsets of a finite set $S \neq \emptyset$. If for all $K_1, K_2 \in P$ either $K_1 \cap K_2 = \emptyset$, $K_1 \subseteq K_2$ or $K_2 \subseteq K_1$ holds, then $|P| \leq 2 \cdot |S|$.

Proof: By induction over $|S|$. □

Given a Büchi automaton A, a function MakeMG$_2$ implemented in Algorithm 30 constructs an M-graph $\mathcal{MG}(A)$ with the following properties:

1. The size of $\mathcal{MG}(A)$ is linear in the number of states of A.
2. For a successor node K' of a node K holds either $K' \subseteq K$ or $K' \cap K = \emptyset$.
3. A union of all sets of all direct successors nodes of a node K is a set returned by the function $\mathrm{Img}(K)$.
4. Every node consists either of accepting or of not accepting states.

Theorem 6
Let $A = [\Sigma, Q, \Delta, q_0, F]$ be a Büchi automaton and let $\mathcal{MG}(A) = [V, E]$ be an M-graph constructed using the function MakeMG$_2$*. There exists an accepting run σ of A if and only if there exists a node $K \in V$ such that $K \xrightarrow{+} K$ and K consists of accepting states of A.*

Proof: "⇒": Let q be some state that occurs in σ infinitely often. Unwinding the M-graph $\mathcal{MG}(A)$ with the designated node $K_0 = \{q_0\}$, we get a tree. A union of all nodes on the level i corresponds to the result of i applications of the function Img to the set $\{q_0\}$. Thus, after a finite number of steps we can reach a node K_1 such that $q \in K_1$. We can unwind the subgraph of $\mathcal{MG}(A)$ with the designated node K_1 and, applying the same argumentation, get a node K_2 such that $q \in K_2$. Due to the construction of $\mathcal{MG}(A)$, we have $K_1 \supseteq K_2$. Repeating the consideration, we can construct an infinite sequence $K_1 \supseteq K_2 \supseteq \ldots \supseteq K \supseteq \ldots$. As K_1 has only a finite number of proper subsets, there must be nodes that repeat in this sequence.
"⇐": Follows from Lemma 12. □

Tarjan's algorithm was adapted in [Spr01] to check emptiness of Büchi nets on-the-fly. Analogously, we can adapt Tarjan's algorithm presented on page 111 to explore the M-graph $\mathcal{MG}(A)$, which would be constructed by the function MakeMG$_2$, and check emptiness of the Büchi automaton A on-the-fly.

Algorithm 30 (Divide)

```
1  func Divide(S, K)
2     Res := ∅
3     forall K' ∈ S do
4        if K ⊆ K' then
5           Res := Res ∪ {K}
6        elseif K ⊇ K' then
7           Res := Res ∪ {K'}
8           K := K \ K'
9        elseif K ∩ K' ≠ ∅ then
10          Res := Res ∪ {K ∩ K'}
11          K := K \ K'
12       fi
13    od
14    if K ≠ ∅ then Res := Res ∪ {K} fi
15    return Res
16 end
```

Algorithm 31 (M-graph construction)

```
1  func MakeMG2 (A)
2     V := ∅
3     E := ∅
4     proc visit (K)
5        V := V ∪ {K}
6        Tmp := Img(K)
7        Acc := Tmp ∩ F
8        NonAcc := Tmp \ F
9        forall K' ∈ (Divide(V, Acc) ∪ Divide(V, NonAcc)) do
10          E := E ∪ {[K, K']}
11          if K' ∉ V then visit(K') fi
12       od
13    end
14    begin
15       visit({q0})
16       return [V, E]
17    end
```

We replace the line *37* of Algorithm 22 with

- visit($\{q_0\}$)

The line *11* is replaced then with the lines *6–9* of Algorithm 31

- $Tmp := \mathrm{Img}(q)$
- $Acc := Tmp \cap F$
- $NonAcc := Tmp \setminus F$
- forall $q' \in (\mathrm{Divide}(Visited, Acc) \cup \mathrm{Divide}(Visited, NonAcc))$ do

and, finally, the line *30* is replaced with

- if $\mathrm{Img}(q) \cap q \neq \emptyset$ then exit ("a fair SCC is found") fi

We notice, however, that the complexity of the obtained algorithm is not linear, as it was analyzed in [Spr01]. Consider an automaton with n states $q_0, \ldots, q_{n-1}$ such that $\mathrm{Img}(\{q_i\}) = \{q_{i+1}\}$. Every time we explore a state $q_k,\ k > 0$, we have to look through k already constructed nodes $\{q_0\}, \ldots, \{q_{k-1}\}$ in the function Divide. Thus, the algorithm makes $\mathcal{O}((n+1) \cdot \frac{n}{2}) = \mathcal{O}(n^2)$ steps during the exploration of the whole automaton.

Tarjan's algorithm is, actually, not the best choice for the on-the-fly emptiness check. It is often criticized due to its high memory requirements. Moreover, it does not check fairness of an SCC until this SCC and all SCCs reachable from it have been completely explored. In fact, it is not necessary to compute the entire SCC to decide non-emptiness, it suffices just do detect a cycle with an accepting state. Consider, for example, an automaton in Fig. 7.4. An on-the-fly algorithm could find the cycle q_0, q_1, q_0 and stop without examining the right part of the automaton, provided that the edge $(q_0, \_, q_1)$ is explored before $(q_0, \_, q_2)$. Tarjan's algorithm is bound to explore the whole automaton regardless of the order of the exploration. However, recent developments have shown that Tarjan's algorithm can be modified to eliminate this disadvantage and to reduce memory requirements.

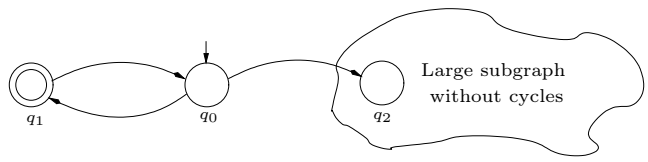


Figure 7.4: Tarjan's algorithm performs much not necessary work on such automata

Let $N_\times$ be a product net constructed for N_M and a not generalized Büchi automaton $A_{\neg\varphi}$, and let $A_{M,\varphi} = A_M \cap A_{\neg\varphi}$ be a corresponding product Büchi automaton. Instead of adapting Tarjan's algorithm as it was done in [Spr01], we adapt its variation suggested in [GV04]. Consider Algorithm 32, we just mention some differences to Tarjan's algorithm and refer the interested reader to the original publication for details.

1. An obvious minor difference is that the algorithm is iterative and not recursive.
2. Tarjan's algorithm uses an implicit procedural stack to manage the deep-first search and an explicit *SCCStack* to store partially explored SCCs. The former is a subset of the later: a new node is inserted into both stacks when it is first encountered. Once it is fully processed, it is removed from the DFS stack, but remains on the *SCCStack* until the entire SCC can be removed. This makes possible to use only a single stack and thread the DFS stack through it by means of the *pre* field and a second pointer *dftop* to the top element of the DFS stack.
3. When a transition from a node f to a node t is encountered, Tarjan's algorithm updates the *lowlink* of f either with the DFS number of t or with the *lowlink* value of t. In Algorithm 32, it is always the *lowlink* of t that is used for updates.
4. The most important addition is the *acc* field of the stack entry. It keeps track of the closest to the top of the stack accepting node on the DFS path that leads to this stack entry. Combined with the previous modification, this allows to detect an accepting cycle as soon as all transitions of this cycle have been explored. In other words, the amount of exploration is *minimal* among all DFS algorithms that follow the same search order.

Our experiments with the described in this section approach have shown that it represents primary a basis for the future research. Though Algorithm 32 can outperform symbolic emptiness check algorithms described in the previous sections on some models when the net N_M does not satisfy the LTL formula φ, it is absolutely not competitive when N_M indeed satisfies φ. Potentially, Algorithm 32 can be easier combined with partial order reductions [God91, Pel94] than algorithms considered in the previous sections.

7.4 Model Checking Procedure

Summarizing the previous sections, given a P/T net with extended arcs N_M and an LTL formula φ, the model checking procedure proceeds as follows:

1. A negation of the formula φ is translated into a generalized Büchi automaton $A_{\neg\varphi}$ as described in section 5.3.3.

Algorithm 32 (Emptiness check)

```
 1  top := -1
 2  dftop := -1
 3  violation := False

 4  proc Main (N_×)
 5  PushNode({m_0})
 6  while ¬violation ∧ dftop ≥ 0 do
 7    if Stack[dftop].succ = ∅ then
 8      PopNode()
 9    else
10      K := oneof (Stack[dftop].succ)
11      Stack[dftop].succ := Stack[dftop].succ \ K
12      if K ∉ Visited then
13        PushNode(K)
14      else
15        if K ∈ Stack then
16          m := PositionOnStack(K)
17          LowlinkUpdate(dftop, m)
18        fi
19      fi
20    fi
21  od
22  if violation then
23    exit ("an accepting cycle is found")
24  fi
25  end

26  proc PushNode (K)
27    Visited := Visited ∪ {K}
28    top := top + 1
29    Tmp := Img_×(K)
30    Stack[top].succ := Divide(Visited, Tmp ∩ F) ∪
31                       Divide(Visited, Tmp \ F)
32    Stack[top].lowlink := top
33    Stack[top].pre := dftop
34    if K ∈ F then
35      Stack[top].acc := top
36    elseif dftop ≥ 0 then
37      Stack[top].acc := Stack[dftop].acc
38    else
39      Stack[top].acc := -1
40    fi
41    dftop := top
42  end

43  proc PopNode ()
44    p := Stack[dftop].pre
45    if p ≥ 0 then
46      LowlinkUpdate(p, dftop)
47    fi
48    if Stack[dftop].lowlink = dftop then
49      top := dftop - 1
50    fi
51    dftop := p
52  end

53  proc LowlinkUpdate (f, t)
54  if Stack[t].lowlink ≤ Stack[f].lowlink then
55    if Stack[t].lowlink ≤ Stack[f].acc then
56      violation := True
57    fi
58    Stack[f].lowlink := Stack[t].lowlink
59  fi
60  end
```

2. A strength of $A_{\neg\varphi}$ is determined using the SCC-analysis. $A_{\neg\varphi}$ is simplified according to Lemma 8 if it is weak or terminal. If the Symbolic DFS algorithm is going to be used and $A_{\neg\varphi}$ has more than one fair set, $A_{\neg\varphi}$ is translated into a not generalized Büchi automaton according to Lemma 7.

3. A product net $N_\times$ is constructed for N_M and $A_{\neg\varphi}$.

4. An emptiness check procedure selected by the user is employed to decide emptiness of $A_{M,\varphi}$.

5. If $A_{M,\varphi}$ is found to be not empty, then φ is not satisfied. A counterexample is generated if requested.

Which emptiness check procedure should be selected depends on the structure of the model. Surely, specialized algorithms for terminal and weak automata are preferable when they are applicable. In our experiments, variations of the algorithms based on forward state traversals performed usually better than the ones based on backward traversals.

Notice that we can easily adapt the model checking procedure if we need to check whether N_M satisfies φ for a set of initial markings. Let χ_I be a characteristic function of this set, we define a set $M_0 \in 2^{\mathcal{M}_\times}$ as the set described by a characteristic function

$$\chi_{M_0} = \chi_I \ \wedge \ q_0 = 1 \ \wedge \bigwedge_{q \in Q_{\neg\varphi} \setminus \{q_{0\neg\varphi}\}} (q = 0).$$

Due to the construction, a state q_0 of the product automaton $A_{M,\varphi} = A_{MI_0} \cap A_{\neg\varphi}$ can not lie on accepting cycles, so we can start exploration of the state space of $A_{M,\varphi}$ directly from the set $K_1 = \mathrm{Img}_\times(\{q_0\})$ of all direct successors of q_0. It is easy to see that $K_1 = \mathrm{Post}_\mathrm{A}(M_0)$. Creation of the place p^i and the transition t^i in the net $N_\times$ can be skipped.

An important feature of a model checker is the ability to generate counterexamples. We adapt algorithms described in section 6.5 to demonstrate the non-emptiness of the product automaton $A_{M,\varphi}$. Speaking in terms of CTL, we have to generate witnesses for the following formulas:

1. $\mathbf{EF}F_1$ for terminal automata,

2. $\mathbf{EF}\,\mathbf{EG}F_1$ for weak automata,

3. $\mathbf{EG}_{fair}\,True$ for strong automata.

7.5 Closing Remark

In this chapter we have considered implementation of a symbolic LTL model checker for k-bounded P/T nets with extended arcs. The underlying idea of the used approach was to construct a product net in such a way that sets of its reachable markings can represent sets of states of the product automaton $A_{M,\varphi}$. Operations for the symbolic manipulations of Petri nets defined in chapter 4 were used then to implemented functions for the symbolic exploration of the product automaton $A_{M,\varphi}$. We have shown that it is possible to adopt the saturation strategy introduced in section 4.3.2 to improve efficiency of these functions.

We have considered implementation of a number of emptiness check algorithms that can outperform the classic Emerson-Lei algorithm [EL86], which is conventionally employed in symbolic LTL model checking. We have also shown how to adapt and improve the algorithm introduced in [Spr01] for the symbolic on-the-fly emptiness check.

8 Conclusions and Outlook

8.1 Conclusions

The research in this thesis has focused on the different techniques which can improve efficiency of the symbolic analysis of k-bounded P/T nets with extended arcs.

Instead of boolean functions and ROBDDs, traditionally used in symbolic methods, we have employed interval logic functions and ROIDDs, which allow natural, more compact, and more efficient encoding of sets of states of k-bounded nets. In chapter 3 we have studied techniques that allow an efficient implementation of an ROIDD package.

In chapter 4 we have discussed how special ROIDD operations needed in the symbolic algorithms can be implemented efficiently. We have studied how to improve efficiency of the reachability analysis and proposed a new saturation approach, which exploits the structure of ROIDDs and the structure of k-bounded P/T nets. It manages to keep sizes of intermediate diagrams smaller than other approaches and can drastically improve efficiency of the symbolic analysis of k-bounded P/T nets. Saturation techniques have been applied then in SCC enumeration algorithms and in model checking. Notice, that many of the suggested techniques can, in principle, be adopted to other net classes and other kinds of decision diagrams.
We have proposed SCC enumeration algorithms for Petri nets, adopting the algorithms introduced in [XB98] and [BGS00]. The algorithms have been used then for efficient analysis of basic net properties and have been employed in model checking algorithms.

Implementation of a symbolic CTL model checker for k-bounded P/T net with extended arcs was subject of chapter 6. We have discussed a number of techniques to improve its efficiency and considered also how to implement a non-conventional CTL model checking algorithm based on forward state space traversals.

Finally, in chapter 7 we have shown how to implement a symbolic LTL model checker for k-bounded P/T net with extended arcs. We have discussed a number of emptiness check algorithms that can outperform the Emerson-Lei algorithm [EL86], which is conventionally employed in symbolic LTL model checking. We have also considered how to adapt and improve the "on-the-fly" algorithm introduced in [Spr01].

8.2 Outlook

The considered topics provide a number of directions for the future research.

In chapter 3 we have discussed implementation of an ROIDD package. Introduction of shared lists and special techniques for memory management allowed to decrease significantly the memory requirements of the package. A more compact and efficient data structure for storage of shared lists can improve performance of the package.

Additional heuristics exploiting the structure of Petri nets can improve computation of ROIDD variable ordering and performance of the saturation algorithm introduced in chapter 4.

In chapter 6 we have discussed implementation of a symbolic CTL model checker. The saturation-based implementation allows to improve significantly the efficiency of the function EvalEU. A question how to improve computation of EvalEG remains open.

P/T net with extended arcs are expressive enough to simulate Timed P/T nets [MF76]. It is interesting to study whether the considered techniques allow to analyze such nets efficiently. Probably, the algorithms introduced in chapter 4 must be fine tuned, a special Timed CTL model checking [RK97] should be implemented.

Integration of symbolic methods with partial order techniques [Val91, God91, Pel94] for analysis of k-bounded nets is a challenging direction for the future research.

A Appendix

A.1 Notations

$\mathbb{B}$	is a set of boolean elements $\{0, 1\}$
$\mathbb{N}$	is a set of natural numbers $\{1, 2, 3, \dots\}$
$\mathbb{N}_0$	is a set $\mathbb{N} \cup 0$
$\mathbb{Z}$	is a set of integers $\{0, 1, -1, 2, -2, 3, -3, \dots\}$
$\emptyset$	is an empty set
2^S	is a power set of a set S, this means 2^S is the set of all subsets of S.

A.2 Relations

Definition 53
Let S be a set and R be a relation $R \subseteq S \times S$.

1. R is *reflexive* if $\forall x \in S$ holds $(x, x) \in R$.
2. R is *symmetric* if $\forall x, y \in S$ from $(x, y) \in R$ follows $(y, x) \in R$.
3. R is *asymmetric* if $\forall x, y \in S$ from $(x, y) \in R$ follows $(y, x) \notin R$.
4. R is *transitive* if $\forall x, y, x \in S$ from $(x, y) \in R$ and $(y, z) \in R$ follows $(x, z) \in R$.
5. R is *total* if $\forall x, y \in S$ holds $(x, y) \in R$ or $(y, x) \in R$ (or both).
6. R is a *total order* or a *linear order* if R is antisymmetric, transitive, and total. A tuple $[S, R]$ is called then a *linear ordered set*.
7. R is a *partial order* if R is reflexive, antisymmetric, and transitive. A tuple $[S, R]$ is called then a *partially ordered set* (or *poset* for short).

A.3 Lattices and Boolean Algebra

Definition 54 (Supremum)
Let $[A, \leq]$ be a partially ordered set. An element $u \in A$ is denoted as a *supremum* (or *least upper bound*) of a set $S \subseteq A$ if

1. $x \leq u$ for all $x \in S$,

2. for any $v \in A : x \leq v \ \forall x \in S$ holds that $u \leq v$.

Definition 55 (Infimum)
Let $[A, \leq]$ be a partially ordered set. An element $l \in A$ is denoted as an *infimum* (or *greatest lower bound*) of a set $S \subseteq A$ if

1. $l \leq x$ for all $x \in S$,
2. for any $v \in A : v \leq x \ \forall x \in S$ holds that $v \leq l$.

Definition 56 (Lattice as a poset)
A *lattice* is a tuple $L = [S, \leq]$ where:

1. $[S, \leq]$ is a partially ordered set,
2. each two-element set $\{x, y\}$ where $x, y \in S$ has a least upper bound $\sup\{x, y\}$ and a greatest lower bound $\inf\{x, y\}$.

A lattice $L = [S, \leq]$ is called *complete* if every subset of S has a least upper bound and a greatest lower bound.

Lattices can also be defined as algebraic structures that satisfy certain laws.

Definition 57 (Lattice as an algebraic structure)
A *lattice* is a tuple $[S, \vee, \wedge]$ where:

1. S is a nonempty set.
2. $\vee, \wedge : S \times S \to S$ are binary operations.
3. For all elements $x, y, z \in S$ and the binary operations $\vee, \wedge$ hold

Commutative laws:	$x \vee y = y \vee x,$	$x \wedge y = y \wedge x.$
Associative laws:	$x \vee (y \vee z) = (x \vee y) \vee z,$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z.$
Absorption laws:	$x \vee (x \wedge y) = x,$	$x \wedge (x \vee y) = x.$
Idempotent laws:	$x \vee x = x,$	$x \wedge x = x.$

A lattice $[S, \vee, \wedge]$ can be obtained from a lattice defined as a poset $[S, \leq]$ by defining $a \vee b = \sup\{a, b\}$ and $a \wedge b = \inf\{a, b\}$ for any $a, b \in S$. Also, from a lattice $[S, \vee, \wedge]$, one may obtain a lattice $[S, \leq]$ by setting $a \leq b$ if and only if $b = a \vee b$. Hence, these two definitions can be used interchangeably, depending on which of them is more convenient for a particular purpose.

Definition 58 (Boolean algebra)
Boolean algebra is a tuple $B = [S, \vee, \wedge, ^-, 0, 1]$ where:

1. $[S, \vee, \wedge]$ is a lattice.
2. $0 \in S$ is denoted as a *least element* of S.
3. $1 \in S$ is denoted as a *greatest element* of S.
4. For all elements $x, y, z \in S$ hold

$$\begin{aligned} \text{Distributivity laws:} \quad & x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z), \\ & x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z). \\ \text{Complement laws:} \quad & x \vee \overline{x} = 1, x \wedge \overline{x} = 0. \end{aligned}$$

Corollary 14
Let $B = [S, \vee, \wedge, ^-, 0, 1]$ be a boolean algebra. Then the following holds

$$\overline{\overline{x}} = x, \quad \overline{x \vee y} = \overline{x} \wedge \overline{y}, \quad \overline{x \wedge y} = \overline{x} \vee \overline{y}$$

Example 27
1. A tuple $B = [\mathbb{B}, \vee, \wedge, \neg, 0, 1]$ is a boolean algebra, denoted also as a *logic algebra*.
2. A tuple $B = [2^S, \cup, \cap, ^-, \emptyset, S]$ is a boolean algebra, denoted also as a *set algebra*.

Proposition 4 ([Sto36])
For every finite boolean algebra there exists an isomorphic set algebra $[2^S, \cup, \cap, ^-, \emptyset, S]$ where S is a finite set.

Definition 59 (Interval algebra)
Let $[S, <]$ be a linear ordered set with a least element. We define an *interval algebra* as a smallest boolean algebra of subsets of S containing all intervals of the form $[a, b)$ such that $a \in S$ and $b \in S$ or $b = \infty$.

Proposition 5
Every countable boolean algebra is isomorphic to an interval algebra.

A.4 Graphs

Definition 60 (Graph)
A *graph* is a tuple $G = [V, E]$ where:

1. V is a set of nodes.
2. E is a set of unordered pairs of nodes called *edges*: $E \subseteq V \times V$.

Definition 61 (Directed graph)
A *directed graph* (or *digraph*) is a tuple $G = [V, E]$ where:

1. V is a set of nodes.
2. E is a set of ordered pairs of nodes called *arcs*: $E \subseteq V \times V$.

We define the following abbreviations:

1. $v \longrightarrow v'$ means there exists $(v, v') \in E$.
2. $\xrightarrow{+}$ is a transitive closure of $\longrightarrow$.
3. $\xrightarrow{*}$ is a reflexive transitive closure of $\longrightarrow$.

Definition 62 (DAG)
Let $G = [V, E]$ be a directed graph. G is denoted as a *directed acyclic graph* (or *DAG* for short) if there exist no nodes $v \in V : v \xrightarrow{+} v$.

Definition 63 (Root)
Let $G = [V, E]$ be a directed graph. A node $v \in V$ can be denoted a *root* of G if for all nodes $v' \in V$ holds $v \xrightarrow{*} v'$.

Definition 64 (Path)
Let $G = [V, E]$ be a directed graph.

1. A sequence of nodes $v_1, \ldots, v_k$ is called a *path* of the *length* $k-1$ in the directed graph G if $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq k$.
2. A path is called *nontrivial* if its length is not 0.
3. If $v_1 = v_k$, then the path is a *cycle*.

Definition 65 (Strongly connected components)
Let $G = [V, E]$ be a directed graph.

1. A maximal set $C \subseteq V$ such that for all nodes $v, w \in C$ there exists a path from v to w is called a *strongly connected component* (*SCC*) of G.
2. An SCC C is called *nontrivial* if for all nodes $v, w \in C$ there exists a nontrivial path from v to w.
3. An SCC C is called *terminal* if for all nodes $v \in C$ there exists no $w \notin C$ such that $(v, w) \in E$.

Definition 66 (SCC quotient graph)
Let $G = [V, E]$ be a directed graph and let $\mathcal{C}$ be a set of all SCCs of G. As an *SCC quotient graph* of G we denote a DAG $G' = [\mathcal{C}, E']$ where $E' \subseteq \mathcal{C} \times \mathcal{C}$ is defined as $(C_1, C_2) \in E'$ if and only if $\exists v_1 \in C_1, v_2 \in C_2 : v_1 \longrightarrow v_2$.

Definition 67 (Distance, radius, diameter)
Let $G = [V, E]$ be a directed graph and let $v_0 \in V$ be defined as a root of G.

1. A *distance* from a node $v \in V$ to a node $v' \in V$ is defined as a length of the shortest path from v to v', if one exists.
2. A *radius* $r(G)$ of the graph G is defined as a largest distance from v_0 to other nodes $v \in V$.
3. A *diameter* $d(G)$ of the graph G is defined as a largest distance between any two nodes $v, v' \in V$, between which the distance is defined.

A.5 Binary Decision Diagrams

Binary decision diagrams (*BDDs*) were studied in [Lee59] and [Ake78] as a data structure for the representation of boolean functions. *Reduced ordered binary decision diagrams* (*ROBDDs*) defined in [Bry86] are a *canonical form* representation for boolean functions. ROBDDs are often substantially more compact than traditional normal forms, and they can be manipulated very efficiently. Hence, they have become very popular and are widely used for a variety of applications like computer aided design, verification of finite-state concurrent systems, etc.

A.5.1 Definitions

Definition 68 (Boolean expressions)
Let $B = [S, \vee, \wedge, ^-, 0, 1]$ be a boolean algebra. *Boolean expressions* consisting of symbols of variables $x_1, \ldots x_n$, symbols of operations $\wedge, \vee, ^-$, and elements of S are defined recursively.

1. The elements of S are boolean expressions.
2. The symbols of variables $x_1, \ldots x_n$ are boolean expressions.
3. If F and G are boolean expressions, then $(F \wedge G)$, $(F \vee G)$, and $\overline{F}$ are also boolean expressions.

Brackets may be omitted according to the usual operator hierarchy.

Definition 69 (Boolean functions)
Every boolean expression G induces a *boolean function* f_G

$$f_G : S^n \to S, \quad (s_1, \ldots, s_n) \mapsto f_G(s_1, \ldots s_n)$$

where $f_G(s_1, \ldots s_n)$ denotes an element of S got by replacing of variables x_i with s_i followed by the evaluation of boolean operations $\wedge, \vee$ and $^-$.

Operations on boolean functions are defined as follows:

1. $(f \vee g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \vee g(x_1, \ldots, x_n)$,
2. $(f \wedge g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \wedge g(x_1, \ldots, x_n)$,
3. $(\overline{f})(x_1, \ldots, x_n) = \overline{f(x_1, \ldots, x_n)}$.

From now on we consider only boolean expressions and functions of the *logic algebra* $B = [\mathbb{B}, \vee, \wedge, \neg, 0, 1]$.

Definition 70 (Boolean decision diagram)
A *Boolean decision diagram* (*BDD*) for variables $X = \{ x_1, \ldots, x_n \}$ is a tuple $[V, E, v_0]$ where:

1. V is a finite set of nodes.
2. $E \subseteq V \times \mathbb{B} \times V$ is finite set of arcs labeled with 0 and 1.
3. $[V, E]$ forms a DAG.
4. $v_0 \in V$ is a root of the BDD.

The following conditions must also hold

1. V contains two *terminal nodes* labeled with 0 and 1 which have no outgoing arcs. We define for these nodes a labeling function value : $V \to \mathbb{B}$, which labels one node with 0, another with 1.
2. All other nodes $v \in V$ are denoted as *nonterminal nodes*, we define for them a labeling function var : $V \to X$. Every nonterminal node v is labeled with a variable var(v) and has two successors low(v) and high(v).
3. On every path from the root to terminal nodes a variable may appear as label of a node only once.

An example of a BDD is shown in Fig. A.1. Every BDD with a root v determines a boolean function $f_v(x_1, \ldots, x_n)$ in the following manner:

1. If v is a terminal node, then $f_v = \text{value}(v)$.
2. If v is a nonterminal node with $\text{var}(v) = x_i$, then f_v is the function

$$f = (\neg x_i \wedge f_{\text{low}(v)}) \vee (x_i \wedge f_{\text{high}(v)}).$$

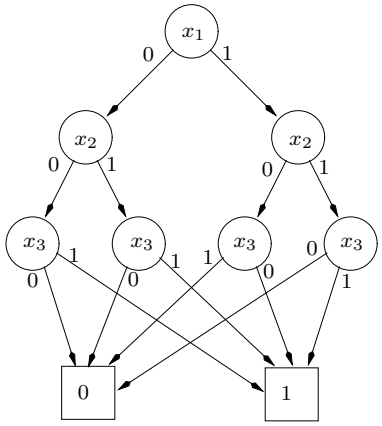


Figure A.1: A BDD for a function $f = (x_2 \wedge x_3) \vee (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_3)$

Definition 71 (Cofactors)
Let $f = f(x_1, \ldots, x_n)$ be a boolean function.

1. A function $f|_{x_i=1} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots x_n)$ is denoted as a *positive cofactor* of f with respect to the variable x_i.

2. A function $f|_{x_i=0} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots x_n)$ is denoted as a *negative cofactor* of f with respect to the variable x_i.

Every boolean function $f = f(x_1, \ldots, x_n)$ can be represented by a BDD with help of the *Bool-Shannon expansion*

$$f = (\neg x_i \wedge f|_{x_i=0}) \vee (x_i \wedge f|_{x_i=1})$$

Definition 72 (Ordered BDDs)
Let $B = [V, E, v_0]$ be a BDD. B is called *ordered* with respect to some variable ordering π if on every path from the root v_0 to terminal nodes all nodes are ordered with respect to their labels: for all non-terminal nodes v, v' if $(v, \_, v') \in E$, then $\mathrm{var}(v) <_\pi \mathrm{var}(v')$.

Definition 73 (Isomorphic BDDs)
Let $B = [V_B, E_B, v_{0_B}]$ and $F = [V_F, E_F, v_{0_F}]$ be two BDDs. B and F are called *isomorphic* if there exists a one-to-one function $\sigma : V_B \to V_F$, such that if $\sigma(v) = v'$, then

1. either v and v' are both terminal nodes labeled with the same value

2. or $\mathrm{var}(v) = \mathrm{var}(v')$, $\sigma(\mathrm{low}(v)) = \mathrm{low}(v')$, and $\sigma(\mathrm{high}(v)) = \mathrm{high}(v')$.

Definition 74 (Reduced BDD)
Let $B = [V, E, v_0]$ be a BDD. B is called *reduced* if

1. Each non-terminal node has two different children: $\nexists v \in V : \text{low}(v) = \text{high}(v)$.
2. There exist no different nodes $v, v' \in V$ such that the BDDs rooted by v and v' are isomorphic.

Lemma 14 (Canonicity of reduced ordered BDDs)
If some variable ordering π is defined, then for every boolean function $f(x_1, \ldots, x_n)$ there exists a unique reduced ordered with respect to π BDD, representing this function f.

Proof: By induction on the number of arguments of the function f [Bry86]. □

An example of a reduced ordered BDD (*ROBDD*) is shown in Fig. A.2. This ROBDD represents the same function f as the BDD in Fig. A.1. Note, that all nodes of the ROBDD on the figure are enumerated (the numbers in the upper side of nonterminal nodes, the terminal nodes get respectively the numbers 0 and 1). We shall use these numbers to address nodes. For the simplification of the algorithms we shall also assume that the function var labels terminal nodes with a special variable x such that $x >_\pi \text{var}(v)$ for all nonterminal nodes $v \in V$.

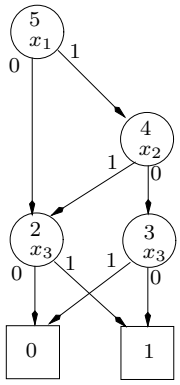


Figure A.2: An ROBDD for the function $f = (x_2 \wedge x_3) \vee (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_3)$

A.5.2 Variable Ordering

The size of an ROBDD can depend critically on the variable ordering used. In general, finding an optimal ordering is infeasible, it can be shown that even checking if a particular ordering is optimal is NP-complete [Bry92]. Moreover, there are boolean

functions that have ROBDD representations of exponential size for any variable ordering. Usually heuristics are used for finding a good variable ordering when such an ordering exists. For example, ROBDDs tend to be smaller if related variables are close together in the ordering.

Example 28
Consider a n-bit comparator function[1] $f = f(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$$f = \bigwedge_{1 \leq i \leq n} (a_i \leftrightarrow b_i).$$

The number of nodes in the ROBDD representing f will be

- $3n + 2$ if we use the variable ordering π_1 defined as

$$a_1 <_{\pi_1} b_1 <_{\pi_1} a_2 <_{\pi_1} b_2 <_{\pi_1} \ldots <_{\pi_1} a_n <_{\pi_1} b_n.$$

- $3 \cdot 2^n - 1$ if we use the variable ordering π_2 defined as

$$a_1 <_{\pi_2} a_2 <_{\pi_2} \ldots <_{\pi_2} a_n <_{\pi_2} b_1 <_{\pi_2} b_2 <_{\pi_2} \ldots <_{\pi_2} b_n.$$

Two ROBDDs representing a two-bit comparator function $f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$ are shown in Fig. A.3. The variable ordering π_1 is used for the left ROBDD, π_2 for the right.

A.5.3 Basic Operations

Consider the Apply [Bry86] algorithm (Algorithm 33) which is a uniform algorithm for computing all binary logical operations. Let $\star$ be an arbitrary two-argument logical operation, f and g be two boolean functions over the same set of variables, F and G be ROBDDs with the same variable ordering representing f and g. The algorithm calculating $f \star g$ is implemented with help of a recursive function AuxApply which gets roots r_1, r_2 of two ROBDDs as parameters. We denote with f and f' boolean functions represented by ROBDDs rooted by r_1 and r_2.
Several cases depending on the relationship between r_1 and r_2 are possible.

1. If r_1 and r_2 are both terminal nodes, then $f \star f' = r_1 \star r_2$.

2. If $\mathrm{var}(r_1) = \mathrm{var}(r_2)$, then the Bool-Shannon expansion

$$f \star f' = (\neg x_i \wedge (f|_{x_i=0} \star f'|_{x_i=0})) \vee (x_i \wedge (f|_{x_i=1} \star f'|_{x_i=1}))$$

[1] with $a \leftrightarrow b$ we denote the expression $(a \wedge b) \vee (\neg a \wedge \neg b)$

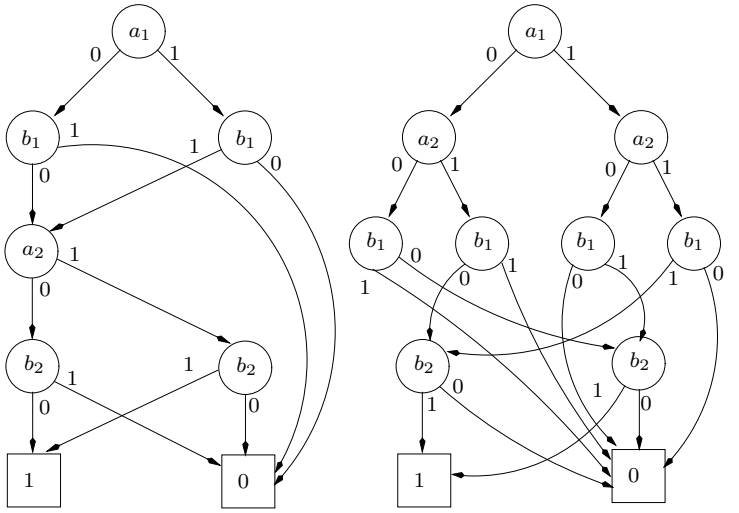


Figure A.3: ROBDDs for the two-bit comparator function

is used to break the problem into two subproblems. The subproblems are solved then recursively. The root of the resulting ROBDD will be a new node w with $\text{var}(w) = \text{var}(r_1)$, $\text{low}(w)$ will be the ROBDD for $(f|_{x_i=0} \star f'|_{x_i=0})$ and $\text{high}(w)$ the ROBDD for $(f|_{x_i=1} \star f'|_{x_i=1})$. The supplementary function MakeNode is used to insert a new node in the ROBDD. It takes care that the ROBDD is always reduced, compare Definition 74. The function checks if a node w represented by a tuple $(\text{var}(w), \text{low}(v), \text{high}(w))$ must be created.

a) First, MakeNode checks if $\text{low}(w) = \text{high}(w)$. If this is a case, then a new node should not created, as it would be redundant, the function simply returns $\text{low}(w)$.

b) Second, MakeNode uses the hashtable *UniqueTable* to check if a node w represented by a $(\text{var}(w), \text{low}(v), \text{high}(w))$ already exists in the ROBDD. *UniqueTable*$[\text{var}(v), \text{low}(v), \text{high}(v)]$ is negative if the node does not exist, otherwise it contains the number of the node. If the node is found in the hashtable, it is returned, otherwise a new ROBDD node is created. The variable *nodesInBDD* counts the number of nodes in the ROBDD. Note, that *nodesInBDD* is initialized with 2, as the values 0 and 1 are reserved for the terminal nodes.

3. If $\text{var}(r_1) < \text{var}(r_2)$, then $f'|_{x_i=0} = f'|_{x_i=1} = f'$ since f' does not depend on x_i. In this case the Bool-Shannon expansion simplifies to

$$f \star f' = (\neg x_i \wedge (f|_{x_i=0} \star f')) \vee (x_i \wedge (f|_{x_i=1} \star f'))$$

and the ROBDD for $f \star f'$ is computed recursively as in the second case.

4. If $\mathrm{var}(r_1) > \mathrm{var}(r_2)$, then the computation is similar to the previous case.

Each problem can generate two subproblems, so care must be used to prevent the algorithm from being exponential. Each subproblem corresponds to a pair of ROBDDs that are subgraphs of the F and G. The number of subgraphs in an ROBDD is limited by its size, so the number of subproblems is limited by the product of the sizes of F and G. A hashtable *ResultTable* is used to store the results of previously computed subproblems, the function AuxApply is a so-called *memory function*. Before any recursive calls are made the *ResultTable* is used to check if the subproblem has been already solved. $\mathit{ResultTable}[r_1, r_2]$ is negative if the result for the subgraphs rooted by r_1 and r_2 is not known yet, nonnegative otherwise. Usage of the memory function allows to keep the algorithm polynomial.

The equivalence check and the negation of ROBDDs (Algorithms 34, 35) are also implemented with help of memory functions and rely on the Bool-Shannon expansion. We skip discussions of these algorithms as they resemble the one we have just made for the Apply algorithm.

A.5.4 Construction of ROBDDs

We define a function Construct that takes a boolean function f as an argument and returns an ROBDD that represents f inductively.

1. If $f = 0$, $f = 1$ or $f = x_i$, then Construct(f) returns the elementary ROBDDs shown in Fig. A.4.

2. If $f = f_1 \wedge f_2$ or $f = f_1 \vee f_2$, then Construct(f) is obtained using the function Apply with the arguments Construct(f_1) and Construct(f_2).

3. If $f = \neg f_1$, then Construct(f) = Neg(Construct(f_1)).

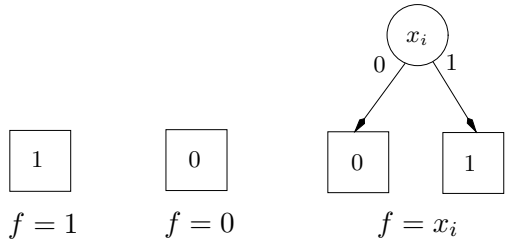


Figure A.4: Elementary ROBDDs

Algorithm 33 (Binary Operation on BDDs)

```
1  func Apply (⋆, F, G)
2     nodesInBDD := 2
3
4     func MakeNode (x, r0, r1)
5        if r0 = r1 then return r0 fi
6        res := UniqueTable[x, r0, r1]
7        if res ≥ 0 then return res fi
8        nodesInBDD := nodesInBDD + 1
9        UniqueTable[x, r0, r1] := nodesInBDD
10       return nodesInBDD
11    end
12
13    func AuxApply (r1, r2)
14       if r1 ∈ {0,1} ∧ r2 ∈ {0,1} then return r1 ⋆ r2 fi
15       if ResultTable[r1, r2] ≥ 0 then return ResultTable[r1, r2] fi
16       if var(r1) = var(r2) then
17          v0 := AuxApply(low(r1), low(r2))
18          v1 := AuxApply(high(r1), high(r2))
19          res := MakeNode(var(r1), v0, v1)
20       elseif var(r1) <π var(r2) then
21          v0 := AuxApply(low(r1), r2)
22          v1 := AuxApply(high(r1), r2)
23          res := MakeNode(var(r1), v0, v1)
24       else  /* var(r1) >π var(r2) */
25          v0 := AuxApply(r1, low(r2))
26          v1 := AuxApply(r1, high(r2))
27          res := MakeNode(var(r2), v0, v1)
28       fi
29       ResultTable[r1, r2] := res
30       return res
31    end
32
33    begin
34       B.root := AuxApply(F.root, G.root)
35       return B
36    end
```

Algorithm 34 (Equivalence check)

```
1  func Equal (F, G)
2     func AuxEqual (r1, r2)
3        if r1 ∈ {0,1} ∧ r2 ∈ {0,1} then return r1 = r2 fi
4        if var(r1) ≠ var(r2) then return 0 fi
5        if ResultTable[r1, r2] ∈ { 0,1 } then return ResultTable[r1, r2] fi
6        res := AuxEqual(low(r1), low(r2)) ∧ AuxEqual(high(r1), high(r2))
7        ResultTable[r1, r2] := res
8        return res
9     end
10    begin
11       return AuxEqual(F.root, G.root)
12    end
```

Algorithm 35 (Negation)

```
1  func Neg (F)
2     func AuxNeg (r1)
3        if r1 ∈ {0,1} then return ¬r1 fi
4        if ResultTable[r1] ≥ 0 then return ResultTable[r1] fi
5        v0 := AuxNeg(low(r1))
6        v1 := AuxNeg(high(r1))
7        res := MakeNode(var(r1), v0, v1)
8        ResultTable[r1] := res
9        return res
10    end
11    begin
12       B.root := AuxNeg(F.root)
13       return B
14    end
```

A.6 Models Used in Experiments

We provide only very short descriptions of the models used in our experiments and refer the interested reader to the original publications, where the models were presented.

ACK is a net which weakly computes the Ackermann's function [PW03].

CS is a realistic net modeling a production cell [LL95, HD95]. The production cell comprises six physical components: two conveyor belts, a rotatable robot equipped with two expendable arms, an elevating rotary table, a press, and a traveling crane. The machines are organized in a closed pipeline. Their common goal is the transport and transformation of N metal plates.

FMS is a net modeling a flexible manufacturing system with three production units where N parts of each of three different types move around on pallets [CM97].

HAL is a biochemical model, not yet published.

JAN is a net demonstrating the overexponential state explosion [Jan83], see Example 4 on page 17.

KAN is a model of the Kanban system [CM97].

MUL is a net which weakly computes $x * y$ [PW03].

POTATO is a bounded version of a biochemical model describing the complicated carbon metabolism in potato tubers [KJH05].

OS is an open version of the production cell model [LL95, HD95], see above.

PUSH is an artificial but not so simple net modeling a chain of concurrent pushers [RLH96]. Every pusher shifts a detail from its input to the output position. The net is obtained by connecting N identical subnets in a chain.

RW is a very simple model for the readers and writers protocol. RW$\leq N$ denotes a model with a set of initial markings, the number of readers and writers can vary from 1 to N.

SLOT is a simple artificial net modeling a slotted ring protocol for local area networks [PRCB94]. It is obtained by connecting N identical subnets in a circular fashion.

Bibliography

[AK77] Araki, T.; Kasami, T.: Some Decision Problems Related to the Reachability Problem for Petri Nets. In: *Theor. Computer Science 3* (1977), pp. 85–104 {10, 22}

[Ake78] Akers, S. B.: Binary Decision Diagrams. In: *IEEE Transactions on Computers* C-27 (1978), pp. 509–516 {5, 27, 157}

[BBF+01] Bérard, B.; Bidoit, M.; Finkel, A.; Laroussinie, F.; Petit, A.; Petrucci, L. ; Schnoebelen, P.: *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001 {89}

[BCL+94] Burch, J.R.; Clarke, E.M.; Long, D.E.; MacMillan, K.L. ; Dill, D.L.: Symbolic Model Checking for Sequential Circuit Verification. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13 (1994), Nr. 4, pp. 401–424 {61, 64}

[BCM+90] Burch, J.; Clarke, B.; Mcmillan, K.; Dill, D. ; Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. In: *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1990, pp. 1–33 {4, 49, 90}

[BF99] Bérard, B.; Fribourg, L.: Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic / Laboratoire Spécification et Vérification, ENS Cachan, France. 1999 (LSV-99-3). – Research Report {50}

[BGP97] Bultan, T.; Gerber, R. ; Pugh, W.: Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In: *Proceedings of the 9th International Conference on Computer-Aided Verification*, Springer-Verlag, 1997 (LNCS #1254), pp. 400–411 {50}

[BGS00] Bloem, R.; Gabow, H. N. ; Somenzi, F.: An Algorithm for Strongly Connected Component Analysis in $n \cdot \log n$ Symbolic Steps. In: *Formal Methods in Computer-Aided Design*, Springer-Verlag, 2000 (LNCS #1954), pp. 37–54 {6, 50, 79, 84, 137, 151}

[BRB90] Brace, K. S.; Rudell, R. L. ; Bryant, R. E.: Efficient Implementation of a BDD Package. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference* ACM/IEEE, IEEE Computer Society Press, 1990, pp. 40–45 {36, 44, 45, 47}

[BRS99] Bloem, R.; Ravi, K. ; Somenzi, F.: Efficient Decision Procedures for Model Checking of Linear Time Logic Properties. In: *Proceedings of the 11th International Conference on Computer-Aided Verification*, Springer-Verlag, 1999, pp. 222–235 {140}

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. C-35 (1986), Nr. 8, pp. 677–691 {5, 27, 33, 38, 49, 157, 160, 161}

[Bry92] Bryant, R. E.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In: *ACM Computing Surveys* 24 (1992), Nr. 3, pp. 293–318 {160}

[Büc60] Büchi, J. R.: On a Decision Method in Restricted Second Order Arithmetic. In: *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, Stanford University Press, 1960, pp. 1–12 {90, 104, 107}

[Bul00] Bultan, T.: BDD vs. Constraint-Based Model Checking: An Experimental Evaluation for Asynchronous Concurrent Systems. In: *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2000 (LNCS #4785), pp. 441–455 {50}

[Bus98] Busi, N.: *Petri Nets with Inhibitor and Read Arcs: Semantics, Analysis and Application to Process Calculi*, Department of Mathematics, University of Siena, Italy, PhD thesis, 1998 {24}

[CE81] Clarke, E. M.; Emerson, E. A.: Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In: *Proceedings of the Workshop on Logics of Programs*, Springer-Verlag, 1981 (LNCS #131), pp. 52–71 {4, 89, 90, 93, 100}

[CEPA+02] Couvreur, J.-M.; Encrenaz, E.; Paviot-Adet, E.; Poitrenaud, D. ; Wacrenier, P.-A.: Data Decision Diagrams for Petri Net Analysis. In: *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, Springer Verlag, 2002 (LNCS #2360), pp. 1–101 {27}

[CES86] Clarke, E. M.; Emerson, E. A. ; Sistla, A. P.: Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. In: *ACM Transactions on Programming Languages and Systems* 8 (1986), Nr. 2, pp. 244–263 {90, 97}

[CF03] Chabrier, N.; Fages, F.: Symbolic Model Checking of Biochemical Networks. In: *Computational Methods in Systems Biology*, Springer-Verlag, 2003 (LNCS #2602), pp. 149–162 {89}

[CGH94] Clarke, E. M.; Grumberg, O. ; Hamaguchi, K.: Another Look at LTL Model Checking. In: *Proceedings of the 6th International Conference on Computer-Aided Verification*, Springer–Verlag, 1994 (LNCS #818), pp. 415–427 {90}

[CGMZ95] Clarke, E. M.; Grumberg, O.; McMillan, K. L. ; Zhao, X.: Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, ACM Press, 1995, pp. 427–432 {127}

[CGP01] Clarke, E. M.; Grumberg, O. ; Peled, D.: *Model Checking*. MIT Press, 2001 {4, 89, 90, 114, 115, 127}

[CJMS01] Ciardo, G.; Jones, R. L.; Miner, A. S. ; Siminiceanu, R. I.: SMART: Stochastic Model Analyzer for Reliability and Timing. In: *Tools of Aachen 2001, International MultiConference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, 2001, pp. 29–34 {5, 49}

[CLS01] Ciardo, G.; Lüttgen, G. ; Siminiceanu, R.: Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2001 (LNCS #2031), pp. 328–342 {49, 61, 73}

[CM97] Ciardo, G.; Miner, A. S.: Storage Alternatives for Large Structured State Spaces. In: *Proceedings of the 9th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Springer-Verlag, 1997 (LNCS #1245), pp. 44–57 {166}

[CMS03] Ciardo, G.; Marmorstein, R. M. ; Siminiceanu, R.: Saturation Unbound. In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2003 (LNCS #2619), pp. 379–393 {49, 73}

[Cor98] Cortadella, J.: Combining Structural and Symbolic Methods for the Verification of Concurrent Systems. In: *Proceedings of the 1st International Conference on Application of Concurrency to System Design*, IEEE Computer Society, 1998, pp. 2–7 {77}

[CS89] Colom, J.M.; Silva, M.: Convex Geometry and Semiflows in P/T nets. A Comparative Study of Algorithms for Computation of Minimal P–semiflows. In: *Proceedings of the 10th International Conference on Application and Theory of Petri nets*, 1989 (LNCS #483), pp. 74–95 {21}

[CS03] Ciardo, G.; Siminiceanu, R.: Structural Symbolic CTL Model Checking of Asynchronous Systems. In: *Proceedings of the 15th International Conference on Computer-Aided Verification*, Springer, 2003 (LNCS #2725), pp. 40–53 {73, 74, 75}

[DE95] Desel, J.; Esparza, J.: *Cambridge Tracts in Theoretical Computer Science*. Volume 40: *Free Choice Petri Nets*. Cambridge University Press, 1995 {9, 20}

[EH86] Emerson, E. A.; Halpern, J. Y.: Sometimes and Not Never Revisited: On Branching versus Linear Time Temporal Logic. In: *Journal of the ACM* 33 (1986), pp. 151–178 {89, 90, 91}

[EH00] Etessami, K.; Holzmann, G. J.: Optimizing Büchi Automata. In: *Proceedings of the 11th International Conference on Concurrency Theory*, Springer-Verlag, 2000 (LNCS #1877), pp. 153–167 {113, 114}

[EH01] Esparza, J.; Heljanko, K.: Implementing LTL Model Checking with Net Unfoldings. In: *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, Springer-Verlag New York, Inc., 2001, pp. 37–56 {5}

[EL85] Emerson, E. A.; Lei, C.L.: Modalities for Model Checking: Branching Time Logic Strikes Back. In: *Proceedings of the 12th Symposium on Principles of Programming languages*, ACM Press, 1985, pp. 84–96 {89, 96}

[EL86] Emerson, E. A.; Lei, Chin-Laung: Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract). In: *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1986, pp. 267–278 {7, 103, 137, 149, 151}

[EM97] Esparza, J.; Melzer, S.: Model Checking LTL Using Constraint Programming. In: *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, 1997 (LNCS #1248), pp. 1–20 {129, 130}

[EN94] Esparza, J.; Nielsen, M.: Decidability Issues for Petri Nets - a Survey. In: *Bulletin of the European Association for Theoretical Computer Science* 52 (1994), pp. 245–262 {16}

[FA73] Flynn, M. J.; Agerwala, T.: Comments on Capabilities, Limitations and Correctness of Petri Nets. In: *Proceedings of the 1st Annual Symposium on Computer Architecture*, ACM Press, 1973, pp. 81–86 {10, 22}

[FEJ89] Frutos-Escrig, D.; Johnen, C.: Decidability of Home Space Property / Univ. de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique. 1989. – Technical Report {19}

[FFK+01] Fisler, K.; Fraer, R.; Kamhi, G.; Vardi, M. Y. ; Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, 2001 (LNCS #2031), pp. 420–434 {120, 138}

[Fin93] Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: *LNCS #674; Advances in Petri Nets* (1993), pp. 210–243 {18}

[God91] Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, Springer-Verlag, 1991, pp. 176–185 {4, 90, 146, 152}

[GPVW95] Gerth, R.; Peled, D.; Vardi, M. ; Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: *Protocol Specification Testing and Verification*, Chapman & Hall, 1995, pp. 3–18 {110}

[GV01] Geldenhuys, J.; Valmari, A.: Techniques for Smaller Intermediary BDDs. In: *Proceedings of the 12th International Conference on Concurrency Theory*, Springer, 2001 (LNCS #2154), pp. 233–247 {61, 74}

[GV04] Geldenhuys, J.; Valmari, A.: Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In: *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2004 (LNCS #2988), pp. 205–219 {146}

[Hac75] Hack, M.: *Decidability Questions for Petri Nets*, Cambridge, Mass.: MIT, Dept. Electrical Engineering, PhD thesis, 1975 {18, 19}

[Hac76] Hack, M.: Petri Net Language. Cambridge, MA, USA: Massachusetts Institute of Technology, 1976. – Technical Report {10, 22, 24}

[HD95] Heiner, M.; Deussen, P.: Petri Net Based Qualitative Analysis - A Case Study / Brandenburg University of Technology at Cottbus. Cottbus, Germany, 1995 (I-08/1995). – Technical Report {166}

[Hel02] Heljanko, K.: *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. Espoo, Finland, Helsinki University of Technology, Department of Computer Science and Engineering, PhD thesis, 2002 {5}

[HK04] Heiner, M.; Koch, I.: Petri Net Based System Validation in Systems Biology. In: *Proceedings of the 25th International Conference on Application and Theory of Petri Nets*, 2004 (LNCS #3099), pp. 216–237 {5, 50}

[HKSV97] Hardin, R.H.; Kurshan, R.P.; Shukla, S.K. ; Vardi, M.Y.: A New Heuristic for Bac Cycle Detection Uing BDDs. In: *Proceedings of the 9th International Conference on Computer-Aided Verification*, Springer-Verlag, 1997 (LNCS #1254), pp. 268–278 {138}

[HKW04] Heiner, M.; Koch, I. ; Will, J.: Validation of Biological Pathways Using Petri Nets - Demonstrated for Apoptosis. In: *BioSystems* 75/1-3 (2004), pp. 15–28 {5, 10, 22}

[HTKB92] Hojati, R.; Touati, H.; Kurshan, R.P. ; Brayton, R.: Efficient ω-regular Language Containment. In: *Proceedings of the 3d International Workshop on Computer-Aided Verification*, Springer-Verlag, 1992 (LNCS #663), pp. 371–382 {137}

[INH96] Iwashita, H.; Nakata, T. ; Hirose, F.: CTL Model Checking Based on Forward State Traversal. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, IEEE Computer Society, 1996, pp. 82–87 {7, 117, 121, 122, 127}

[Jan83] Jantzen, M.: The Large Markings Problem. In: *Petri Net Newsletter* 14 (1983), pp. 24–25 {15, 166}

[Jen95] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, vol. 2.* London, UK: Springer-Verlag, 1995 {9}

[Jen96] Jensen, K.: *Coloured Petri Nets (2nd ed.): Basic Concepts, Analysis Methods and Practical use: volume 1.* London, UK: Springer-Verlag, 1996 {9}

[Kam95] Kam, T.: *State Minimization of Finite State Machines Using Implicit Techniques*, University of California at Berkeley, PhD thesis, 1995 {27, 49}

[KJH05] Koch, I.; Junker, B. H. ; Heiner, M.: Application of Petri Net Theory for Modelling and Validation of the Sucrose Breakdown Pathway in the Potato Tuber. In: *Bioinformatics* 21 (2005), Nr. 7, pp. 1219–1226 {5, 50, 166}

[KM69] Karp, R. M.; Miller, R. E.: Parallel Program Schemata. In: *J. Comput. Syst. Sci.* 3 (1969), Nr. 2, pp. 147–195 {18}

[Kri63] Kripke, S. A.: Semantical Considerations on Modal Logic. In: *Acta Philosophica Fennica* 16 (1963), pp. 83–94 {91}

[Lam80] Lamport, L.: "Sometime" is Sometimes "Not Never": on the Temporal Logic of Programs. In: *Proceedings of the 7th Symposium on Principles of Programming languages*, ACM Press, 1980, pp. 174–185 {4, 89, 90}

[Lee59] Lee, C. Y.: Representation of Switching Circuits by Binary Decision Programs. In: *Bell System Technical Journal* 38 (1959), pp. 985–999 {5, 27, 157}

[LL95] Lewerentz, C. (Hrsg.); Lindner, T. (Hrsg.): *Formal Development of Reactive Systems - Case Study Production Cell*. London, UK: Springer-Verlag, 1995 {166}

[LP85] Lichtenstein, O.; Pnueli, A.: Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In: *Proceedings of the 12th Symposium on Principles of Programming Languages*, ACM Press, 1985, pp. 97–107 {90}

[LR95] Lautenbach, K.; Ridder, H.: A Completion of the S-invariance Technique by Means of Fixed Point Algorithms / Universität Koblenz-Landau. 1995 (10–95). – Technical Report {5, 6, 27, 50, 77}

[May81] Mayr, E. W.: Persistence of Vector Replacement Systems is Decidable. In: *Acta Informatica* 15 (1981), pp. 309–318 {18}

[MC99] Miner, A. S.; Ciardo, G.: Efficient Reachability Set Generation and Storage Using Decision Diagrams. In: *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, Springer, 1999 (LNCS #1639), pp. 6–25 {5}

[McM92] McMillan, K. L.: *Symbolic Model Checking: an Approach to the State Explosion Problem*. Pittsburgh, PA, USA, Carnegie Mellon University, PhD thesis, 1992 {4, 49, 90}

[MF76] Merlin, P. M.; Farber, D. J.: Recoverability of Communication Protocols - Implications of a Theoretical Study. In: *IEEE Transactions on Communications* 24 (1976), Nr. 9, pp. 1036–1043 {152}

[Min93] Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: *Proceedings of the 30th ACM/IEEE Design Automation Conference*, ACM Press, 1993, pp. 272–277 {27, 49}

[ML98] Møller, J.; Lichtenberg, J.: *Difference Decision Diagrams*. Building 344, DK-2800 Lyngby, Denmark, Department of Information Technology, Technical University of Denmark, Diplomarbeit, 1998 {27}

[MMB93] Matsunaga, Y.; McGeer, P. C. ; Brayton, R. K.: On Computing the Transitive Closure of a State Transition Relation. In: *Proceedings of the 30th International Conference on Design automation*, ACM Press, 1993, pp. 260–265 {50, 79}

[MP92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer–Verlag, 1992 {87}

[MR95] Montanari, U.; Rossi, F.: Contextual Nets. In: *Acta Informatica* 32 (1995), Nr. 6, pp. 545–596 {10, 22}

[Noa99] Noack, A.: A ZBDD Package for Efficient Model Checking of Petri Nets (in German) / Branderburgische Technische Uinversität Cottbus. 1999. – Technical Report {44, 47, 55, 62, 63, 71, 78}

[PCP99] Pastor, E.; Cortadella, J. ; Pena, M. A.: Structural Methods to Improve the Symbolic Analysis of Petri Nets. In: *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, Springer-Verlag, 1999, pp. 26–45 {49, 77}

[Pel94] Peled, D.: Combining Partial Order Reductions with On-the-fly Model-Checking. In: *Proceedings of the 6th International Conference on Computer-Aided Verification*, Springer-Verlag, 1994 (LNCS #818), pp. 377–390 {4, 90, 146, 152}

[Pet62] Petri, C. A.: *Kommunikation mit Automaten*, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, PhD thesis, 1962 {3, 9}

[Pet81] Peterson, J. L.: *Petri Net Theory and The Modeling of Systems*. Englewood Cliffs, Massachusetts,: Prentice Hall, Inc., 1981 {9}

[Pnu77] Pnueli, A.: The Temporal Logic of Programs. In: *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, IEEE Computer Society Press, 1977, pp. 46–57 {4, 89}

[Pnu80] Pnueli, A.: The Temporal Semantics of Concurrent Programs. In: *Theoretical Computer Science* 13 (1980), Nr. 1, pp. 45–60 {4, 89, 93}

[PRCB94] Pastor, E.; Roig, O.; Cortadella, J. ; Badia, R. M.: Petri Net Analysis Using Boolean Manipulation. In: *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, Springer, 1994 (LNCS #815), pp. 416–435 {5, 49, 61, 166}

[PW03] Priese, L.; Wimmel, H.: *Petri Netze*. Springer, 2003 (Theoretische Informatik) {18, 166}

[RBS00] Ravi, K.; Bloem, R. ; Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: *Proceedings of the 3d International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, 2000, pp. 143–160 {127, 139, 140}

[Rei86] Reisig, W.: *Petrinetze — Eine Einführung*. Springer–Verlag, 1986 {9}

[Rid97] Ridder, H.: *Analysis of Petri Net Models with Decision Diagrams (in German)*, Universität Koblenz-Landau, PhD thesis, 1997 {55}

[RK97] Ruf, J.; Kropf, T.: Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In: *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, Chapman & Hall, Ltd., 1997, pp. 146–163 {152}

[RLH96] Rausch, M.; Lüder, A. ; Hanisch, H.M.: Combined Synthesis of Locking and Sequential Conrollers. In: *Proceedings of the 3d International Workshop on Discrete Event Systems*, 1996, pp. 133–138 {166}

[Rud93] Rudell, R. L.: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, IEEE Computer Society Press, 1993, pp. 42–47 {46}

[SB00] Somenzi, F.; Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: *Proceedings of the 12th International Conference on Computer-Aided Verification*, Springer–Verlag, 2000 (LNCS #1855), pp. 248–263 {106, 107, 114}

[SE05] Schwoon, S.; Esparza, J.: A Note on On-The-Fly Verification Algorithms. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2005 (LNCS #3440), pp. 174–190 {108}

[SP02] Solé, M.; Pastor, E.: Traversal Techniques for Concurrent Systems. In: *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, 2002, pp. 220–237 {61, 62, 63, 64}

[Spr01] Spranger, J.: *Symbolic LTL Verification of Petri Nets (in German)*, Branderburgische Technische Uinversität Cottbus, PhD thesis, 2001 {5, 7, 129, 131, 141, 143, 145, 146, 149, 151}

[ST98] Strehl, K.; Thiele, L.: Symbolic Model Checking Using Interval Diagram Techniques / Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich. 1998. – Technical Report {5, 6, 27, 33, 55}

[ST03] Sebastiani, R.; Tonetta, S.: "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking / Informatica e Telecomunicazioni, University of Trento. 2003 (DIT-03-041). – Technical Report {110, 114}

[Sta90] Starke, P. H.: *Analyse von Petri-Netz-Modellen.* Stuttgart, Teubner, 1990 {9, 16, 20}

[Sto36] Stone, M. H.: The Theory of Representations for Boolean Algebras. In: *Transactions of the American Mathematical Society* 40 (1936), pp. 37–111 {155}

[Tar55] Tarski, A.: A Lattice-theoretical Fixpoint Theorem and its Applications. In: *Pacific Journal of Mathematics* (1955), Nr. 5, pp. 285–309 {100}

[Tar72] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. In: *SIAM Journal on Computing* 1 (1972), Nr. 2, pp. 146–160 {79, 108}

[Tho90] Thomas, W.: Automata on Infinite Objects. In: *Handbook of Theoretical Computer Science* Volume B: Formal Models and Semantics. Elsevier Science Publishers, 1990, Chapter 4, pp. 133–191 {106}

[Val91] Valmari, A.: A Stubborn Attack On State Explosion. In: *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, Springer-Verlag, 1991, pp. 156–165 {4, 90, 152}

[Var96] Vardi, M. Y.: An Automata Theoretic Approach to Linear Temporal Logic. In: *Proceedings of the 8th Banff Higher Order Workshop Conference on Logics for Concurrency : Structure versus Automata*, Springer-Verlag, 1996, pp. 238–266 {90, 104, 107}

[Var01] Vardi, M. Y.: Branching vs. Linear Time: Final Showdown. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, 2001 (LNCS #2031), pp. 1–22 {89}

[VHHP95] Varpaaniemi, K.; Halme, J.; Hiekkanen, K. ; Pyssysalo, T.: PROD Reference Manual / Helsinki University of Technology, Digital Systems Labora tory. Espoo, Finland, 1995 (B13). – Technical Report. – 56 S {5}

[VW86] Vardi, M. Y.; Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: *Proceedings of the 1st Symposium on Logic in Computer Science*, 1986, pp. 332–344 {90, 104}

[VW94] Vardi, M. Y.; Wolper, P.: Reasoning About Infinite Computations. In: *Information and Computation* 115 (1994), Nr. 1, pp. 1–37 {107}

[WVS83] Wolper, P.; Vardi, M. Y. ; Sistla, A.P.: Reasoning About Infinite Computation Paths. In: *Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science*, 1983, pp. 185–194 {110}

[XB98] Xie, A.; Beerel, P. A.: Efficient State Classification of Finite State Markov Chains. In: *Design Automation Conference*, 1998, pp. 605–610 {6, 50, 79, 82, 87, 151}

[XB99] Xie, A.; Beerel, P. A.: Implicit Enumeration of Strongly Connected Components. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, IEEE Press, 1999, pp. 37–40 {50, 79}

[YBO+98] Yang, B.; Bryant, R. E.; O'Hallaron, D. R.; Biere, A.; Coudert, O.; Janssen, G.; Ranjan, R. K. ; Somenzi, F.: A Performance Study of BDD-Based Model Checking. In: *Proceedings of the 2nd International Conference on Formal Methods in Computer-Aided Design*, 1998 (LNCS # 1522), pp. 255–289 {44, 46}

[YHTM96] Yoneda, T.; Hatori, H.; Takahara, A. ; Minato, S.: BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets. In: *LNCS #1166* (1996), pp. 435–449 {5, 55}