Analysis of
well-formedness and soundness
by reduction techniques
and their implementation

By
J.M.E.M. van der Werf B.Sc.

Supervisor: Prof. dr. K.M van Hee
Eindhoven, 21th August 2006
Preface

This thesis is part of my master’s project for the master Business Information Systems, at the Technische Universiteit Eindhoven. I graduated at the section Architecture of Information Systems, headed by Prof. dr. K.M. van Hee.

My first real introduction with Petri nets, and the use of analysis techniques was at the course of “Systeemmodelleren 1”, and “OGO 2.2”. At the course of Systeemmodelleren, prof. dr. van Hee asked for student assistants for a project named Petriweb[12, 15]. From Petriweb I got involved at LaQuSo, where I helped developing the LaQuSo toolset, and where I participated in several projects. Especially the ProRail case to analyze UML activity diagrams on correctness, gave a great help to write this thesis.

I would like to thank Kees van Hee, Olivia Oanea, Alexander Serebrenik, and Natalia Sidorova for their readings and comments on my thesis, Marc Voorhoeve and Reinier Post for the nice and fruitful discussions about Yasper, Petriweb and my thesis, Marija Petkovic and Natalia Sidorova for the joined work on the ProRail case, and Ivo Raedts and all other employees of LaQuSo, for the wonderful time at LaQuSo.
Summary

As the reachability graph of a Petri net can grow exponentially in the size of the net, many analysis techniques based on this graph are time and space consuming. Two important properties to analyze are well-formedness and soundness, but both are related to the reachability graph. For different classes of Petri nets, different techniques can be used to analyze these properties. Since most of these analysis methods rely on the reachability graph, we search for analysis based on the structure of a Petri net. Many reduction techniques are based on the structure of a Petri net.

In this thesis we present different reduction techniques on subclasses of Petri nets to reduce the Petri net and the reachability graph. The presented reduction techniques are from Desel and Esparza, Murata and Berthelot. We further introduce the workflow abstraction rule, and show that the order in which the rule is applied on a net does not matter. These reduction rules are implemented as libraries for the tool Yasper.

The discussed class checks and reduction techniques are combined into a procedure to analyze Petri nets on well-formedness and soundness, for different subclasses of Petri nets. The presented procedure is used in a case study.

As we use a reduced net to analyze, the results are based on this reduced net. We therefore search for methods to translate the results of the reduced net to the results of the original net. One of these methods is trace refinement. With trace refinement, we transform a trace in the reduced net to a trace in the original net. The reduction rules of Desel and Esparza on Free Choice Petri nets, preserve well-formedness, but do not take the initial marking in account. We extended the rules of Desel and Esparza with marking, and show that these reductions preserve liveness and boundedness. For this set of reduction techniques we present a method for trace refinement.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Extended abstraction rule</td>
<td>58</td>
</tr>
<tr>
<td>6.2</td>
<td>Extended transition rule</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Extended place rule</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>Case study: ProRail</td>
<td>65</td>
</tr>
<tr>
<td>7.1</td>
<td>Translation to Petri nets</td>
<td>65</td>
</tr>
<tr>
<td>7.2</td>
<td>Analysis results</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>Petriweb</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions</td>
<td>71</td>
</tr>
<tr>
<td>A</td>
<td>Algorithms</td>
<td>73</td>
</tr>
<tr>
<td>B</td>
<td>Yasper: a tool for workflow modeling and analysis</td>
<td>89</td>
</tr>
<tr>
<td>B.1</td>
<td>Introduction</td>
<td>89</td>
</tr>
<tr>
<td>B.2</td>
<td>Features</td>
<td>90</td>
</tr>
<tr>
<td>B.3</td>
<td>Combining Yasper with other tools</td>
<td>91</td>
</tr>
<tr>
<td>B.4</td>
<td>Conclusions and future work</td>
<td>92</td>
</tr>
<tr>
<td>C</td>
<td>Petriweb: A Repository for Petri Nets</td>
<td>93</td>
</tr>
<tr>
<td>C.1</td>
<td>Needs for Model Repositories</td>
<td>93</td>
</tr>
<tr>
<td>C.2</td>
<td>Petriweb</td>
<td>95</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Model retrieval</td>
<td>95</td>
</tr>
<tr>
<td>C.2.2</td>
<td>Model management</td>
<td>97</td>
</tr>
<tr>
<td>C.2.3</td>
<td>Users and communities</td>
<td>97</td>
</tr>
<tr>
<td>C.2.4</td>
<td>Architecture</td>
<td>98</td>
</tr>
<tr>
<td>C.3</td>
<td>PNML: the Petri Net Markup Language</td>
<td>100</td>
</tr>
<tr>
<td>C.3.1</td>
<td>PNML Viewer</td>
<td>100</td>
</tr>
</tbody>
</table>
C.3.2 PNML Checker ........................................ 100
C.3.3 Issues with PNML ..................................... 101
C.4 Future Work ............................................ 102
C.5 Conclusions ............................................. 103
Chapter 1

Introduction

Models are used often to support business processes within an organization, e.g. with workflow management tools, such as Staffware, and COSA. Tools for orchestration, such as BizTalk and Oracle BPEL (the Business Process Execution Language), use process models to control the communication between objects and applications.

Models also play an important part in the development of applications. The use of models starts early in the design phase. Throughout the development, models are used to specify, build and test applications and systems. The used models have to fulfill the properties formulated in the requirements. With analysis, these properties can be verified. If a property does not hold, the model contains an error, and we would like to show this error to the designer. Many errors can be shown to the designer by replaying an erroneous sequence of steps, called a trace.

In this thesis, we focus on Petri nets as process modeling language. Petri nets have a formal semantics and a graphical representation. For many years, there has been research on the analysis of Petri nets and its properties. Many of the properties are related to the reachability graph of the Petri net. In this thesis, we focus on well-formedness and soundness. If a Petri net is well-formed, then the reachability graph is finite, and transitions can always eventually fire. Soundness is only applicable on Workflow Petri nets, with an initial place and final place. A
Workflow Petri net is sound if in any case, it will terminate eventually, and at the moment it terminates, there is a token in the final place and all the other places are empty[35]. Soundness is closely related to the problem of well-formedness. A Workflow Petri net is sound if and only if its closure (i.e. the Petri net obtained by adding a transition with the final place as input place, and the initial place as output place) is well-formed.

Unfortunately, the reachability graph can grow exponentially in size of the places and transitions of the Petri net. Due to this, many analysis techniques and methods are also exponential, either in time or in space. Therefore we would like to reduce the net, while the properties to analyze are preserved, i.e. the reduced Petri net has the same properties as the original Petri net. Different reduction techniques already exist. In this thesis, we discuss three reduction techniques given in literature, and introduce a fourth reduction rule, together with the implementation of these rules. These rules reduce a Petri net into a smaller Petri net preserving the well-formedness and soundness properties. Further, we show the relation between the different reduction rules.

We would like to present the analysis results of the reduced net to the modeller (see Figure 1.1). However, the rules do not give a method to transform the analysis results on the reduced net to the original Petri net. In this thesis we also focus on this transformation.

Chapter 2 gives an introduction to Petri nets, and presents different subclasses of Petri nets, e.g. State Machine Petri nets, Marked Graph Petri nets, Free Choice Petri nets, and Workflow Petri nets, and extensions of Petri nets: Reset nets and Inhibitor nets. We prove that if the net obtained by removing the extensions is bounded, the original net with extensions is also bounded. We further discuss some Workflow Petri net classes: ST-nets and Batch Workflow Petri nets.

In Chapter 3, we show three different reduction techniques: the reduction techniques of Desel and Esparza on Free Choice Petri nets, and some reduction rules of Murata and Berthelot. Further, we introduce a fourth reduction rule, the workflow reduction rule. We show how these rules are related with each other. In Chapter 4, a procedure is introduced on how the different techniques and checks presented can be combined to analyze Petri nets. Chapter 5 discusses how these techniques are implemented as libraries for Yasper.

In Chapter 6, we provide a method to transform traces of the reduced net to traces of the original Petri net. This method is called trace refinement. Further, we extend the reduction rules of Desel and Esparza, such that they do take the marking into account. For these extended rules, we show that they preserve liveness and boundedness, and how the traces can be refined to the original net.

In Chapter 7, we present a case study for a client of LaQuSo. In this case study, we transformed a state diagram and some activity diagrams into Petri nets and performed a soundness analysis.

In Chapter 8, we discuss how the discussed methods and techniques can be used in the Petriweb project. Finally, in Chapter 9, we conclude the thesis.
In Appendix A, different derived algorithms and their explanation are given. The next appendices are the two articles about respectively Yasper (Appendix B, [16]), and Petriweb (Appendix C, [15]).
Chapter 2

Petri nets

This chapter gives an introduction to Petri nets. A Petri net is a mathematical model describing concurrent systems, with a graphical representation. A Petri net can be represented as a directed, bipartite graph with places (drawn as circles) and transitions (drawn as squares) as nodes. The state of the net, called the marking, is given by drawing black tokens in places. A transition is enabled if it has enough tokens in all its incoming places. If a transition fires, it consumes the tokens needed, and produces tokens in its outgoing places.

Figure 2.1: A Petri net model modelling two traffic lights on a crossway

In Figure 2.1(a) a Petri net is given modelling two traffic lights on a crossway. The marking of the Petri net, shows that both traffic lights are red, and that it is safe for one of the traffic lights to become green. In Figure 2.1(b), the transition $rg_2$ has fired, and the second traffic light has become green. The marking now shows that the first traffic light cannot become green, since there is no token in the place safety. One of the properties of this net that we would like to prove, is that it is safe: the state in which both traffic lights are green, can never occur.
In the next section we present the basic mathematical definitions of Petri nets, and properties of Petri nets in a formal way. Different classes of Petri nets exist. In this thesis we introduce State Machine Petri nets (Section 2.3), Marked Graph Petri nets (Section 2.4), Free Choice Petri nets (Section 2.5), Workflow Petri nets (Section 2.6), ST-nets (Section 2.7), and Batch Workflow Petri nets (Section 2.8). In Section 2.9, extensions of Petri nets with inhibitor arcs and reset arcs are discussed.

2.1 Preliminaries

Here we introduce the notions of graphs and of bags.

Definition 1 (directed graph, undirected graph, directed bipartite graph). A directed graph is a pair $G = \langle V, A \rangle$ with $V$ a set of vertices, also called nodes, and $A \subseteq (V \times V)$ a set of arcs. An arc $(u, v) \in A$ is directed from the tail $u$ to the head $v$. If the relation $A$ is symmetric, the graph is called undirected. A directed graph $G = (V, A)$ is called a directed bipartite graph iff $V$ has a partition into two subsets $V_1$ and $V_2$, such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and $\forall (u, v) \in A : u \in V_1 \land v \in V_2$.

In a graph, we can traverse the arcs to the different vertices, thus following a path.

Definition 2 (directed path). A directed path in a directed graph $G = (V, A)$ from a node $n_1 \in V$ to a node $n_k \in V$ is a sequence $(n_1, n_2, \ldots, n_k)$ with $\forall i : 0 < i \leq k : (n_{i-1}, n_i) \in A$. The length of a path is defined as the number of arcs in the path.

Definition 3 (distance). Let $G = (V, E)$ be a graph. The distance between two nodes $n_1, n_2 \in V$, is 0 if $n_1 = n_2$, otherwise it is the length of the shortest path between $n_1$ and $n_2$.

If we can return to the first node of the path from the final node in the path, it is a circuit.

Definition 4 (circuit). A circuit in a graph $G = (V, E)$ is a path from a node $n_1 \in V$ to $n_2 \in V$ such that each node occurs no more than once in it, and $(n_2, n_1) \in E$.

If we can reach all nodes starting at a single node, the graph is connected.

Definition 5 (connected, strongly connected). A graph is connected if for every two nodes $n_1$ and $n_2$ there is an undirected path from $n_1$ to $n_2$. A graph is strongly connected if for every two nodes $n_1$ and $n_2$ there is a directed path from $n_1$ to $n_2$.

Definition 6 (labeled graph). Let $\Sigma_V$ and $\Sigma_A$ be two alphabets. A labeled graph is a tuple $G = (V, A, \mapsto_V)$, with:

- $V$ a set of vertices or nodes;
- $A \subseteq (V \times \Sigma_A \times V)$ a set of arcs with labels. An arc $(u, t, v)$ is directed from the tail $u$ to the head $v$, and has label $t$;
• $\mathbb{V} : V \rightarrow \Sigma_V$ a function giving a label to a node.

In a bag or multi-set, elements can occur multiple times.

**Definition 7 (bag).** Let $\mathbb{N}$ denote the set of natural numbers. A bag $B$ over some set $S$ is a mapping $B : S \rightarrow \mathbb{N}$. For $s \in S$, $B(s)$ denotes the number of occurrences of $s$ in the bag $B$. We write $\mathbb{N}^S$ for the set of bags over $S$. We denote bags by listing the occurring elements between square brackets and we use superscripts for the multiplicity of the occurrences.

A bag $B = [a^2, b^3, c]$ consists of two times an $a$, three times a $b$ and a single $c$. Let $X, Y \in \mathbb{N}^S$, and $s \in S$. We define the following operations on bags:

- $s \in X \Leftrightarrow X(s) > 0$;
- $(X + Y)(s) = X(s) + Y(s)$;
- $(X - Y)(s) = \max(0, X(s) - Y(s))$;
- $(X \leq Y) \Leftrightarrow \forall s \in S : X(s) \leq Y(s)$;
- $(X < Y) \Leftrightarrow X \leq Y \land X \neq Y$.

Sets are considered to be bags, with all elements mapped to 1, thus making all bag operations applicable.

**Definition 8 (sequence[25]).** Let $A$ be a set, and $n \in \mathbb{N}$. A sequence over $A$ of length $n$ is a function $\sigma : \{i \in \mathbb{N} | 1 \leq i \leq n\} \rightarrow A$. If $n > 0$, and $\sigma(1) = a_1, \ldots, \sigma(n) = a_n$, then we write the sequence $\sigma$ as $a_1 \ldots a_n$. The sequence of length 0 is called the empty sequence, and is denoted by $\epsilon$. The set of sequences over $A$ is denoted by $A^*$, and the set of all nonempty sequences over $A$ is denoted by $A^+$.

### 2.2 Petri nets, definitions and properties

All definitions, theorems in the remainder of this section, unless specified otherwise, are taken from [9, 18].

**Definition 9 (Petri net[18]).** A Petri net is a tuple $N = (S, T, F)$, where

- $S$ is a finite set, which elements are called places.
- $T$ is a finite set, which elements are called transitions.
- $F \subseteq (S \times T) \cup (T \times S)$ is the flow relation. Elements in this relation are called arcs.

An element $n \in (S \cup T)$ is a node of $N$. Often we call a Petri net just a net. A Petri net is called empty, if $S = T = F = \emptyset$. The Petri net $N = ([s], \{t\}, \{(s, t), (t, s)\})$ is called the atomic net.
Some authors consider $F$ as a bag, rather than a set, allowing, thus, certain arcs to appear multiple times. We restrict our attention to $F$ being a set, i.e., all arcs have multiplicity 1.

**Definition 10 (subnet\[9\]).** A Petri net $N' = (S', T', F')$ is a subnet of a Petri net $N = (S, T, F)$ iff

- $S' \subseteq S$
- $T' \subseteq T$
- $F' = F \cap ((S' \times T') \cup (T' \times S'))$

We write $N' \subseteq N$.

**Definition 11 (preset and postset of a node\[9\]).** The preset of a node $x \in N$ is a bag containing all incoming nodes of $x$:

$$\cdot x = \{ y | (y, x) \in F \}$$

The preset of a set of nodes $R \subseteq N$ is defined as follows:

$$\cdot R = \bigcup_{s \in R} \cdot s$$

The postset of a node $x \in N$ is a bag containing all outgoing nodes from $x$:

$$x^* = \{ y | (x, y) \in F \}$$

The postset of a set of nodes $R \subseteq N$ is defined as follows:

$$R^* = \bigcup_{s \in R} s^*$$

The notation $\cdot n$ and $n^*$ is used both for the set and for its characteristic function. The interpretation follows from the context. If the dot-notation is used with respect to some Petri net $N'$, we write $\cdot_{N'} n$ and $n^*_{N'}$.

**Definition 12 (source and sink places\[9\]).** A source place is a place $i \in S$ such that $\cdot i = \emptyset$. Thus there are no incoming arcs into place $i$. A sink place is a place $f \in S$ such that $f^* = \emptyset$. Thus there are no outgoing arcs from $f$.

**Definition 13 (marking\[18\]).** Markings are configurations of a Petri net. A marking $M$ of $N$ is a bag over $S$, where $M(s)$ denotes the number of tokens in $s \in S$ in marking $M$. A place $s \in S$ is marked in a marking $M$ if $M(s) > 0$. We write $\mathcal{M}$ for the marking that only has a single token in $s$. The total number of tokens on a set of places $R$ in a marking $M$ is denoted by $M(R)$:

$$M(R) = \sum_{s \in R} M(s).$$
Definition 14 (system[9]). A Petri net $N$ with a marking $M$ is written as $(N, M)$ and is called a system.

We now define the firing rule for transitions.

Definition 15 (enabled transitions, firings and firing sequences[9, 18]). A transition $t \in T$ is enabled in marking $M$ iff $\bullet t \subseteq M$. An enabled transition $t$ may fire; by this we mean perform action $t$. A firing is a transformation $M \rightarrow M'$, written $M \xrightarrow{t} M'$, where $M' = M - \bullet t + \bullet \ast$. A sequence of transitions $\sigma$ is called a firing sequence in $(N, M)$ iff it is either the empty sequence, or it is $(t_1, \ldots, t_n)$ and there exists markings $M_1, \ldots, M_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$. The set of possible firing sequences is written as $T^{*}$. We write $M \xrightarrow{\sigma} M'$ iff there exists a sequence $\sigma$, such that $M \xrightarrow{\sigma} M'$.

Definition 16 (Parikh vector[9]). Let $N = (S, T, F)$ be a Petri net and $\sigma$ a firing sequence. The Parikh vector $\vec{\sigma} : T \rightarrow \mathbb{N}$ of $\sigma$ maps every transition $t \in T$ to the number of occurrences of $t$ in $\sigma$.

Definition 17 (incidence matrix, flow functions[9, 18]). Let $N = (S, T, F)$ be a Petri net. The incidence matrix $N : (S \times T) \rightarrow \{-1, 0, 1\}$ of $N$ is defined by

$$
N(s, t) = \begin{cases} 
0 & \text{if } \{(s, t) \not\in F \land (t, s) \not\in F\} \text{ or } \{(s, t), (t, s)\} \subseteq F \\
-1 & \text{if } (s, t) \in F \land (t, s) \not\in F \\
1 & \text{if } (t, s) \in F \land (s, t) \not\in F
\end{cases}
$$

The matrix $F^{-} : (S \times T) \rightarrow \mathbb{N}$ is the input flow function, defined by $F^{-}(s, t) = \bullet t(s)$. The matrix $F^{+} : (S \times T) \rightarrow \mathbb{N}$ is the output flow function, defined by $F^{+}(s, t) = \bullet \ast(s)$. Note that $N = F^{+} - F^{-}$.

Lemma 1 (marking equation[9]). If $M \xrightarrow{\sigma} M'$ then $M' = M + N \cdot \vec{\sigma}$.

Note that the reverse does not hold.

Definition 18 (linear dependent place[9]). Let $N = (S, T, F)$ be a Petri net. A place $s \in S$ is linear dependent if there exists a vector $\Lambda : S \rightarrow \mathbb{Q}$ such that $\Lambda(s) = 0$ and $\Lambda \cdot N = 0$.

Definition 19 (linear dependent transition[9]). Let $N = (S, T, F)$ be a Petri net. A transition $t \in T$ is linear dependent if there exists a vector $\Lambda : T \rightarrow \mathbb{Q}$ such that $\Lambda(t) = 0$ and $N \cdot \Lambda = 0$.

Definition 20 (reachable, reachability problem[18]). A marking $M'$ is reachable in $(N, M)$ iff $M \xrightarrow{\sigma} M'$. The set of reachable markings from $M$ is denoted as $[M>]$. The reachability problem is a problem of deciding whether a given marking $M'$ is reachable from a given marking $M$.

Definition 21 (liveness, deadlock-free, structurally live, dead, deadlock[9, 18]). A transition $t$ in a system $(N, M_0)$ is live iff for every reachable marking $M$, there is a marking $M'$ reachable from $M$ that enables $t$. A system $(N, M_0)$ is live iff all its transitions are live. If $(N, M_0)$ is live, the marking $M_0$ is a live marking. A system $(N, M_0)$ is deadlock-free iff every reachable marking $M$ enables at least one transition. A Petri net $N$ is structurally live iff there exists a marking $M_0$ of $N$, such
that $(N, M_0)$ is live. A set of markings $R$ is a livelock in the system $(N, M_0)$ iff: 
$\forall M \in R : M_0 \xrightarrow{*} M \implies [M'] = R$. A transition $t$ of a system $(N, M_0)$ is dead iff there is no marking $M'$ reachable from $M_0$ which enables $t$. A marking $M$ is a deadlock iff there are no enabled transitions in $M$.

Note that the definition of deadlock implies that all transitions are dead.

**Definition 22 (boundedness, structurally bounded, safe) [9].** A place $s$ is bounded in a system $(N, M_0)$ iff there is a natural number $k \in \mathbb{N}$ such that $M(s) \leq k$ for every reachable marking $M$. We say that $s$ is $k$-bounded. A system $(N, M_0)$ is bounded iff all its places are bounded. It is called $k$-bounded iff no place has a bound greater than $k$. The marking $M_0$ is called a bounded marking of $N$. A Petri net $N$ is structurally bounded if for every initial marking $M_0$, the system $(N, M_0)$ is bounded. A system $(N, M_0)$ is safe iff it is 1-bounded.

**Lemma 2.** Elementary properties of bounded systems [9]

- Every bounded system is $k$-bounded for some $k \in \mathbb{N}$.
- Every bounded system has a finite set of reachable markings.

**Definition 23 (well-formed net) [9].** A Petri net is well-formed if there exists a marking $M_0$ of $N$ such that $(N, M_0)$ is a live and bounded system.

**Theorem 1 (Strongly Connectedness Theorem) [9].** Well-formed nets are strongly connected.

**Definition 24 (reachability graph) [18].** For a system $(N, M_0)$, the reachability graph is a labeled graph $(V, E)$ where $V = [M_0]$ and $E \subseteq (V \times T \times V)$, such that $(m_1, t, m_2) \in E$ iff $m_1 \xrightarrow{t} m_2$.

We further define operations on Petri nets.

**Definition 25 (union).** The union of two Petri nets $N_1 = (S_1, T_1, F_1)$ and $N_2 = (S_2, T_2, F_2)$ is defined as

$$(N_1 \cup N_2) = (S_1 \cup S_2, T_1 \cup T_2, F_1 \cup F_2) \quad (2.1)$$

**Definition 26 (intersection).** The intersection of two Petri nets $N_1 = (S_1, T_1, F_1)$ and $N_2 = (S_2, T_2, F_2)$ is defined as

$$(N_1 \cap N_2) = (S_1 \cap S_2, T_1 \cap T_2, F_1 \cap F_2) \quad (2.2)$$

**Definition 27 (difference).** The difference of two Petri nets $N_1 = (S_1, T_1, F_1)$ and $N_2 = (S_2, T_2, F_2)$ is defined as

$$(N_1 \setminus N_2) = (S_1 \setminus S_2, T_1 \setminus T_2, F_1 \setminus F_2) \quad (2.3)$$

**Definition 28 (place difference).** Given a Petri net $N = (S, T, F)$ and a place $s \in S$, we define the Petri net $N' = (S', T', F') = N \setminus \{s\}$ as follows: $S' = S \setminus \{s\}$, $T' = T$, and $F' = F \setminus ((s \times \{s\}) \cup \{s \times s^*\})$.

**Definition 29 (transition difference).** Given a Petri net $N = (S, T, F)$ and a transition $t \in T$, we define the Petri net $N' = (S', T', F') = N \setminus \{t\}$ as follows: $S' = S$, $T' = T \setminus \{t\}$, and $F' = F \setminus ((\{t\} \times \{t\}) \cup \{\{t\} \times t^*\})$.  


2.3 State Machine Petri nets

Definition 30 (State Machine Petri net\cite{9}). A Petri net \( N = (S, T, F) \) is a State Machine Petri net iff:

\[
\forall t \in T : |\bullet t| \leq 1 \land |\circ t| \leq 1
\]  
(2.4)

Let \( N = (S, T, F) \) be a State Machine Petri net and \( M_0 \) a marking of \( N \).

Theorem 2 (Liveness Theorem\cite{9}). The system \((N, M_0)\) is live iff \( N \) is strongly connected and \( M_0 \) marks at least one place.

Theorem 3 (Boundedness Theorem\cite{9}). The system \((N, M_0)\) is \( k \)-bounded iff \( M_0(S) \leq k \).

2.4 Marked Graph Petri nets

Definition 31 (Marked Graph Petri net\cite{9}). A Petri net \( N = (S, T, F) \) is a Marked Graph Petri net iff:

\[
\forall s \in S : |\bullet^* s| \leq 1 \land |s^*| \leq 1
\]  
(2.5)

Let \( N = (S, T, F) \) be a Marked Graph Petri net, and \( M_0 \) a marking of \( N \).

Lemma 3 (Token count in circuits\cite{9}). Let \( \gamma \) be a circuit of \( N \). Let \( R \) be the set of places of \( \gamma \), and define \( M(\gamma) \overset{\text{def}}{=} M(R) \) on a marking \( M \). For every reachable marking \( M \), \( M(\gamma) = M_0(\gamma) \) holds.

Theorem 4 (Liveness Theorem\cite{9}). A Marked Graph Petri net is live iff every circuit is initially marked.

Theorem 5 (Boundedness Theorem\cite{9}). A live Marked Graph Petri net is \( k \)-bounded iff for every place \( s \in S \) there exists a circuit \( \gamma \), which contains \( s \) and satisfies \( M_0(\gamma) \leq k \).

2.5 Free Choice Petri nets

Definition 32 (Free Choice Petri net\cite{9}). A Petri Net \( N = (S, T, F) \) is a Free Choice Petri net iff:

\[
\forall t_1, t_2 \in T : \bullet t_1 \cap \bullet t_2 \neq \emptyset \implies \bullet t_1 = \bullet t_2
\]  
(2.6)

Lemma 4. Formula 2.6 is equivalent to:

\[
\{(s, t), (r, t), (s, u)\} \subseteq F \implies (r, u) \in F
\]  
(2.7)

and

\[
\forall t \in T : \forall s \in \bullet t : \forall u \in s^* : \forall r \in \circ u : r \in \circ t
\]  
(2.8)
Proof. The equivalence of Formula 2.7 is proven in [9], here we prove both (2.7) ⇒ (2.8) and (2.8) ⇒ (2.7).

(⇒) Let \( t \in T \), \( s \in \bullet t \). We then have \((s, t) \in F\). Let \( u \in s^* \), thus having \((s, u) \in F\). Let \( r \in \bullet u \), thus \((r, u) \in F\). By 2.7, also \((r, t) \in F\), thus having \( r \in \bullet t \).

(⇐) Let \( t, u \in T \), and \( s, r \in S \), such that \{\((s, t), (r, t), (s, u)\)\} ⊆ \( F \). We thus have \( s \in \bullet t \), \( r \in \bullet t \) and \( s \in \bullet u \). By 2.8, we also have \( r \in \bullet t \), thus \((r, t) \in F\).

Before we can give some important properties of Free Choice Petri nets, we need to define siphons, traps and their properties.

**Definition 33 (siphon, proper siphon, minimal siphon[9]).** A set of places \( R \) of a net is a siphon iff \( R \subseteq \bullet R \). A siphon is called proper if it is not the empty set. A siphon is called minimal if it is proper and does not include any other proper siphon.

Three important properties of siphons are[9]:

- If \( R \) is a siphon, then every marking \( M' \) reachable from some marking \( M \) such that \( M(R) = 0 \), satisfies \( M'(R) = 0 \).
- Every proper siphon of a live system is marked at the initial state.
- Let \((N, M_0)\) be a deadlocked system, i.e. \( M_0 \) is a dead marking of \( N \). Then the set \( R \) of places of \( N \) unmarked at \( M_0 \) is a proper siphon.

**Definition 34 (trap, proper trap[9]).** A set of places \( R \) of a net is a trap iff \( \bullet R \subseteq R^* \). A trap is called proper if it is not the empty set.

Two important properties of traps are[9]:

- If \( R \) is a trap then every marking \( M' \) reachable from some marking \( M \) with \( M(R) > 0 \) satisfies \( M'(R) > 0 \).
- If every proper siphon of a system includes an initially marked trap, then the system is deadlock-free.

Furthermore, both siphons and traps have some structural properties[9]:

- The union of siphons (traps) is a siphon (trap).
- Every siphon includes a unique maximal trap with respect to set inclusion.
- A siphon includes a marked trap iff its maximal marked trap is marked.

With the definitions of siphons and traps, we now can derive the following theorems:

**Theorem 6 (Commoner’s Theorem[9]).** A Free Choice system is live if and only if every proper siphon includes an initially marked trap.
Theorem 7 (relationship between liveness and deadlock-freedom[9]). A bounded and strongly connected Free Choice system is live iff it is deadlock-free.

The following lemmas establish interesting properties of singleton siphons and traps in well-formed Free Choice Petri nets.

Lemma 5. Let \( N = (S, T, F) \) be a well-formed Free Choice Petri net, \( m \) a marking such that the system \((N, m)\) is live and bounded, and \( s \in S \). If \( \{s\} \subseteq s^* \) or \( s^* \subseteq \{s\} \), then \( s = s^* \), and \( m(s) > 0 \).

Proof. Let \( N = (S, T, F) \) be a well-formed Free Choice Petri net, \( m \) a marking such that the system \((N, m)\) is live and bounded, and be \( s \in S \).

Suppose \( \{s\} \subseteq s^* \), then the set \( R = \{s\} \) is a siphon, moreover, it is a proper siphon. Since \( N \) is live, by Theorem 6, \( R \) includes an initially marked trap. The set \( R \) contains a single element, \( s \). Thus \( R \) is also a trap, and \( s \) is marked. Since \( R \) is both a trap and a siphon, we have \( s = s^* \).

Suppose \( s^* \subseteq \{s\} \), then the set \( R = \{s\} \) is a trap. Suppose \( s^* \neq \{s\} \). Then there is a transition \( t \in T \), such that \( s \in t^* \), but \( t \not\in s^* \). Since \( R \) is a trap, any token put in \( R \) remains in \( s \). The system \((N, m)\) is live. Therefore, from any marking reachable from \( m \), a marking \( m' \) with \( t \leq m' \) is reachable. All markings \( m'' \) with \( m' \xrightarrow{t} m'' \) satisfy \( m'(s) < m''(s) \). There is no transition \( u \) removing tokens from \( R \), since it is a trap. Thus place \( s \) is unbounded, which contradicts the fact that the system is bounded. Hence \( s = s^* \) holds, thus the set \( R \) is also a siphon. All siphons include a initially marked trap, thus \( m(s) > 0 \).

Lemma 6. Let \( N = (S, T, F) \) be a well-formed Free Choice Petri net, \( m \) a marking such that the system \((N, m)\) is live and bounded, \( s \in S \), such that \( \{s\} \cap s^* \neq \emptyset \), and \( R = \{s\} \) which is neither a trap nor a siphon. Then there exist transitions \( t, u, v \in T \), \( t \neq u \neq v \), such that \( t \in \{s\} \cap s^* \), \( u \in s^* \setminus \{s\} \), and \( v \in \{s\} \cap s^* \).

Proof. Let \( N = (S, T, F) \) be a well-formed Free Choice Petri net, \( m \) a marking such that the system \((N, m)\) is live and bounded, \( s \in S \), such that \( \{s\} \cap s^* \neq \emptyset \), and \( R = \{s\} \) which is neither a trap nor a siphon. Since \( R \) is not a siphon, there is a transition \( t \in T \), such that \( t \in \{s\} \), but \( t \not\in s^* \). The transition \( t \) can fire infinite times, but since \((N, m)\) is bounded, there has to be a transition that has \( s \) as input place, but not as output. Since \( R \) is not a trap, there is a transition \( u \in T \), such that \( u \in s^* \), but \( u \not\in \{s\} \), thus \( u \in s^* \setminus \{s\} \), and \( u \) removes tokens from place \( s \). Since \( \{s\} \cap s^* \neq \emptyset \), there has to be at least one transition \( v \in \{s\} \cap s^* \).

2.6 Workflow Petri nets

Definition 35 (Workflow Petri net[35]). A Petri net \( N = (S, T, F) \) is a Workflow Petri net iff:
1. $N$ has an initial node $i \in S$, which is the only source node.

2. $N$ has a final node $f \in S$, which is the only sink node.

3. For any node $n \in (S \cup T)$, there exists a path from $i$ to $n$ and from $n$ to $f$. (The path-property of Workflow Petri nets)

If both the nodes $i$ and $f$ are places, the Petri net is called a $p$Workflow Petri net. If both the nodes $i$ and $f$ are transitions, the Petri net is called a $t$Workflow Petri net. The closure of a Workflow Petri net is the Petri net obtained by adding a transition with the final place as input place, and the initial place as output place.

**Definition 36 (Subworkflow Petri net).** Let $N = (S, T, F)$ be a Petri net. The net $N_1 = (S_1, T_1, F_1) \subseteq N$ is a Subworkflow Petri net of $N$ iff

1. $N_1$ is a Workflow Petri net, call the source $i'$ and the sink $f'$;
2. for each node $n \in (S \cup T) \setminus \{i', f'\}; (\bullet n \cup n \bullet) \subseteq (S_1 \cup T_1);
3. $(\bullet f' \cup i' \bullet) \subseteq (S_1 \cup T_1)$.

An important notion in Workflow theory is the notion of soundness, introduced in [35].

**Definition 37 (soundness[35]).** A Workflow Petri net $N = (S, T, F)$ with initial place $i \in S$ and final place $f \in S$ is sound iff:

- for every marking $M$ reachable from $[i]$, there exists a firing sequence leading from $M$ to $[f]$,
- marking $[f]$ is the only marking reachable from $[i]$ with at least one token in place $f$, and
- There are no dead transitions in the system $(N, [i])$.

This definition of soundness ensures that a single token in the initial place, leads eventually to a single token in the final place, and no tokens are left in the Petri net. In [3], the authors show that the second condition of soundness is superficial. The notion of soundness is generalized in [3].

**Definition 38 ($k$-soundness and generalized soundness[3]).** A Workflow Petri net $N = (S, T, F)$ with initial place $i \in S$ and final place $f \in S$ is $k$-sound for $k \in \mathbb{N}$ iff $[f^k]$ is reachable from all markings $M$ of $N$ from $([i]^k]$. A Workflow Petri net is generalized sound iff it is $k$-sound for every natural $k$.

Further, we introduce and prove some lemmas about operations on Workflow Petri nets and Subworkflow Petri nets.

**Lemma 7.** Let $N = (S, T, F)$ be a $p$Workflow Petri net, and $N_1 = (S_1, T_1, F_1) \subseteq N$ a Subworkflow Petri net of $N$. For each node $n \in (S \cup T) \setminus (S_1 \cup T_1)$ and node $n' \in (S_1 \cup T_1)$ holds:
1. for all paths from \( n \) to \( n' \), the path passes through the source \( i \) of \( N_1 \), and

2. for all paths from \( n' \) to \( n \), the path passes through the sink \( f \) of \( N_1 \).

\textbf{Proof.} Let \( n \in (S \cup T) \setminus (S_1 \cup T_1) \) and node \( n' \in (S_1 \cup T_1) \). Suppose there is a path from \( n \) to \( n' \) that does not pass the source \( i \). Then there are two nodes \( n_1 \) and \( n_2 \) on this path such that \( n_1 \in (S \cup T) \setminus (S_1 \cup T_1) \) and \( n_2 \in (S_1 \cup T_1) \), and \( n_2 \in n_1 \), thus \( \langle n_1, n_2 \rangle \in F \). But then we have \( n_2 \notin N_1 \), thus \( N_1 \) cannot be a Subworkflow Petri net. Thus there is no path from \( n \) to \( n' \) that does not pass source \( i \) of \( N' \). The proof that all paths from \( n' \) in \( N' \) to \( n \) in \( N \) has to pass the sink \( f \) of \( N' \) is analogous.

\textbf{Lemma 8.} Let \( N = (S, T, F) \) be a Petri net, and \( N_1 = (S_1, T_1, F_1) \) with source place \( i \in S_1 \) and sink place \( i' \in S_1 \) and \( N_2 = (S_2, T_2, F_2) \) with source place \( i' \in S_2 \) and sink place \( f \in S_2 \) be two Subworkflow Petri nets of \( N \), such that \( N_1 \cap N_2 \) is not an empty Petri net, \( N_1 \not\subseteq N_2 \), and \( N_2 \not\subseteq N_1 \). Then \( N_1 \cup N_2 \) is a Workflow Petri net, with source node \( i \) and sink node \( f \).

\textbf{Proof.} We prove that \( N_1 \cup N_2 \) is a Subworkflow Petri net of \( N \). First we show that \( i' \in S_1 \) and \( f' \in S_2 \). Since \( N_1 \cap N_2 \) is a non-empty Petri net, there is a node \( n \), such that \( n \in S_1 \cup S_2 \cup T_1 \cup T_2 \). Since it is in \( N_1 \), we have \( n \in N_1 \) and \( n \in N_2 \). Suppose \( i \notin S_1 \), then, by Lemma 7, there is a path from \( i \) to \( n \), not passing through \( i' \). But as \( n \in S_2 \cup T_2 \), by the same lemma, all paths to \( n \) pass through \( i' \). This contradicts, hence \( i' \in S_1 \), and \( i' \in S_1 \cup S_2 \). As \( i' \in S_1 \), we have \( i' \subseteq T_1 \), and \( i' \subseteq T_1 \cup T_2 \). But as \( n \in S_2 \), we obtain \( f' \subseteq S_2 \) and \( f' \subseteq (S_1 \cup S_2 \cup T_1 \cup T_2) \). Hence we have also \( n \in S_1 \cup S_2 \cup T_1 \cup T_2 \). Since \( f \in S_2 \), and \( i \in S_1 \), we have directly \( f \subseteq S_2 \cup T_1 \cup T_2 \).

As we can reach from each node in \( N_1 \) the final place \( f' \), and since \( f' \) is also a node in \( N_2 \), these nodes can reach \( f \). As \( i' \) is a node in \( N_1 \), we can reach nodes in \( N_2 \) from the source place \( i \), and is \( N_1 \cup N_2 \) a Workflow Petri net with source \( i \) and sink \( f \). Hence \( N_1 \cup N_2 \) is a Subworkflow Petri net of \( N \).

\textbf{Lemma 9.} Let \( N = (S, T, F) \) be a pWorkflow Petri net, \( N_1 = (S_1, T_1, F_1) \subseteq N \) a subWorkflow Petri net, with source \( i \in S_1 \) and sink \( f' \in S_1 \), and \( N_2 = (S_2, T_2, F_2) \subseteq N \) a subWorkflow Petri net, with source \( i' \in S_2 \) and sink \( f \in S_2 \), such that \( (N_1 \cup N_2) = N \). If \( |S_1 \cap S_2| > 1 \), \( N_1 \not\subseteq N_2 \) and \( N_2 \not\subseteq N_1 \), then the net \( N \) contains three subWorkflow Petri nets:

1. \( (N_1 \cap N_2) \), with source \( i' \) and sink \( f' \);
2. \( (N_1 \setminus N_2) \cup \{i'\} \), with source \( i \) and sink \( i' \);
3. \( (N_2 \setminus N_1) \cup \{f'\} \), with source \( f' \) and sink \( f \).

\textbf{Proof.} Let \( N = (S, T, F) \) be a pWorkflow Petri net, \( N_1 = (S_1, T_1, F_1) \subseteq N \) a subWorkflow Petri net, with source \( i \) and sink \( f' \), and \( N_2 = (S_2, T_2, F_2) \subseteq N \) a subWorkflow Petri net, with source \( i' \) and sink \( f \), such that \( (N_1 \cup N_2) = N \).
1. Consider \((N_1 \cap N_2)\). Since \(N_1, N_2 \subseteq N\), we also have \((N_1 \cap N_2) \subseteq N\), thus it is a subnet of \(N\). Let \(n_1 \in (S_1 \cup T_1) \setminus (S_2 \cup T_2)\), \(n_2 \in (S_2 \cup T_2) \setminus (S_1 \cup T_1)\) and \(\hat{n} \in (S_1 \cup T_1) \cap (S_2 \cup T_2)\). By lemma \(\gamma\) any path to \(\hat{n}\) has to go through \(i'\), since \(\hat{n} \in N_2\), also because of the same lemma, since \(\hat{n} \in N_1\), all paths from \(\hat{n}\) to the final node, has to pass \(f'\). Thus \(i'\) is the source of, and \(f'\) is the sink of net \((N_1 \cap N_2)\). All nodes \(n \in \left((S_1 \cup T_1) \cap (S_2 \cup T_2)\right) \setminus \{i', f'\}\) satisfy \((\bullet n \cup n_1^\bullet) \subseteq N_1 \land (\bullet n \cup n_2^\bullet) \subseteq N_2\), thus also \((\bullet n \cup n_1^\bullet) \subseteq (N_1 \cap N_2)\).

2. Consider \((N_1 \setminus N_2) \cup \{i'\}\). It is a subnet of \(N\), since \(N_1 \subseteq N\). Also, the sink of \(N\), place \(f\), is in \(N_2\), and thus not in \(N_1 \setminus N_2\), and \(N\) is a Workflow Petri net. Therefore, all paths from \(n_1\) needs to go through \(N_2\), and thus, by lemma \(\gamma\), passing \(i'\). Thus \(i'\) is the source of, and \(f'\) is the sink of net \((N_1 \setminus N_2) \cup \{i'\}\). All nodes \(n \in \left((S_1 \cup T_1) \setminus (S_2 \cup T_2)\right) \setminus \{i, i'\}\) satisfy \((\bullet n \cup n_1^\bullet) \subseteq (S_1 \cup T_1)\). Since \(i'\) is the only node connected with \(N_2\), we have \((\bullet n \cup n_1^\bullet) \subseteq \left((S_1 \cup T_1) \setminus (S_2 \cup T_1)\right) \cup \{i'\}\).

3. Consider \((N_2 \setminus N_1) \cap \{f'\}\). It is a subnet of \(N\), since \(N_2 \subseteq N\). Also, the source of \(N\), place \(f\), in \(N_1\), and thus not in \(N_2 \setminus N_1\), and \(N\) is a Workflow Petri net. Therefore, all paths to \(n_2\) needs to go through \(N_1\), and thus, by lemma \(\gamma\), passing \(f'\). Thus \(f'\) is the source of, and \(f\) is the sink of net \((N_2 \setminus N_1) \cup \{f'\}\). All nodes \(n \in \left(S_2 \cup T_2\right) \setminus \left(S_1 \cup T_2\right) \setminus \{f, f'\}\) satisfy \((\bullet n \cup n_2^\bullet) \subseteq (S_2 \cup T_2)\). Since \(f'\) is the only node connected with \(N_1\), we have \((\bullet n \cup n_2^\bullet) \subseteq \left((S_2 \cup T_2) \setminus (S_1 \cup T_1)\right) \cup \{f'\}\).

2.7 ST-nets

A special class of Workflow Petri net that is general sound is the class of ST-nets. An ST-net is constructed from state machines and marked graphs. The definition and theorems from this section come from [37]. Before we can define the class of ST-nets, we need two definitions of refinement.

**Definition 39 (place refinement, transition refinement[37])**. Given two Workflow Petri net \(N_1 = (S_1, T_1, F_1)\) and \(N_2 = (S_2, T_2, F_2)\). Place refinement of a place \(s \in S_1\) with \(p\) Workflow Petri net \(N_2\) yields a Workflow Petri net \(N = N_1 \otimes_s N_2\), built as follows: \(s \in S_1\) is replaced in \(N_1\) by \(N_2\); the transitions in \(s\) become input transitions of the source of \(N_2\), the transitions in \(s^*\) become output transitions of the sink of \(N_2\).

Transition Refinement of a transition \(t \in T_1\) with \(t\) Workflow Petri net \(N_2\) yields a Workflow Petri net \(N = N_1 \otimes_t N_2\), built as follows: \(t \in T_1\) is replaced in \(N_1\) by \(N_2\); the places in \(t\) become input places of the source of \(N_2\), the places in \(t^*\) become output places of the sink of \(N_2\).

For any Petri net \(N = L \otimes_n M\), where \(n\) is a node in \(L\), the Petri net \(M\) is called a factor, and \(L\) is called the quotient.

With these definitions, we can define the class of ST-nets as follows:
Definition 40 (ST-nets[37]). The set $\mathcal{N}$ of ST-nets is the smallest set of nets $\mathcal{N}$ defined as follows:

- if $N$ is an acyclic Marked Graph Workflow Petri net, then $N \in \mathcal{N}$;
- if $N$ is a State Machine Workflow Petri net, then $N \in \mathcal{N}$;
- if $N = (S, T, F) \in \mathcal{N}$, $s \in S$ and $M \in \mathcal{N}$ is a Workflow Petri net, then then $(N \otimes_s M) \in \mathcal{N}$;
- if $N = (S, T, F) \in \mathcal{N}$, $t \in T$ and $M \in \mathcal{N}$ is a Workflow Petri net, then $(N \otimes_t M) \in \mathcal{N}$;

The main theorem about ST-nets used in this thesis is about generalized soundness:

Theorem 8. [37] All ST-nets are generalized sound.

2.8 Batch Workflow Petri nets

Batch Workflow Petri nets are introduced in [38], and form a subclass of the Workflow Petri nets. Before defining the class of Batch Workflow Petri nets, we first need to introduce the notions of redundancy and persistency. A place is redundant if it cannot be reached in the Workflow Petri net, a transition is redundant if it cannot fire. A place is persistent if the net cannot terminate properly, once that place becomes marked. Formally:

Definition 41 (non-redundant, non-persistent[38]). Let $N = (S, T, F)$ be a Workflow Petri net with source $i$ and sink $f$. A place $s \in S$ is non-redundant iff there exist $k \in \mathbb{N}$ and $m \in \mathbb{N}^S$ such that $[i^k] \xrightarrow{m} m \land m(s) > 0$. A place $s \in S$ is non-persistent iff there exist $k \in \mathbb{N}$ and $m \in \mathbb{N}^S$ such that $m(s) > 0 \land m \xrightarrow{f^k} [f^k]$. A transition $t \in T$ is non-redundant iff there exist $k \in \mathbb{N}$ and $m \in \mathbb{N}^S$ such that $[i^k] \xrightarrow{m} m \xrightarrow{t}$.

Lemma 10. [38]A Workflow Petri net has no redundant places iff it has no redundant transitions.

Theorem 9. [38]Let $N = (S, T, F)$ be a Workflow Petri net. The the following holds:

1. $N$ has no redundant places iff $S \setminus \{i\}$ contains no proper siphon, and
2. $N$ has no persistent places iff $S \setminus \{f\}$ contains no proper trap.

Lemma 11. [38]Let $N = (S, T, F)$ be a Petri net with a single source place $i$ and a single sink place $f$, and every transition of $N$ has at least one input and one output place. Moreover, $S \setminus \{i\}$ contains no proper siphon and $S \setminus \{f\}$ contains no trap. Then $N$ is a Workflow Petri net.

The last lemma gives an characterization of Workflow Petri nets. Using this lemma, we define a Batch Workflow Petri net as:
Definition 42 (Batch Workflow Petri net[38]). A Batch Workflow Petri net $N$ is a Petri net that has the following properties:

1. $N$ has a single source place $i$ and a single sink place $f$;
2. every transition of $N$ has at least one input and one output place;
3. every siphon of $N$ contains $i$;
4. every trap of $N$ contains $f$.

In [38], the authors show that Batch Workflow Petri nets form a subclass of the Workflow Petri nets.

2.9 Petri net extensions

2.9.1 Reset nets

One of the known extensions of Petri nets includes Reset arcs, allowing to empty places in a single firing. Nets with reset arcs are called Reset nets[3].

Definition 43 (Reset net). A Reset net $N$ is a tuple $\langle S, T, F, R \rangle$, where

- $S$ is a finite set, and its elements are called places.
- $T$ is a finite set, and its elements are called transitions.
- $F \subseteq (S \times T) \cup (T \times S)$ is the flow relation. Elements in this relation are called arcs.
- $R \subseteq (T \times S)$ is the set of reset arcs. We write $R(t)$ for the set of places reset by transition $t \in T$.

If $N$ is a Reset net, we write $\overline{N}$ for the net without reset arcs.

With Reset nets, the firing of transitions change.

Definition 44 (firing rule for Reset nets). Let $N = (S, T, F)$ be a Reset net, and $m$ a marking of $N$, with $\bullet t \leq m$. If $t$ fires, the new marking $m'$ becomes:

$$\forall s \in S : m'(s) = \begin{cases} 
m(s) - t(s) & s \in \bullet t \\
m(s) + \bullet t(s) & s \in \bullet^* t \\
0 & (t, s) \in R \\
m(s) & \text{otherwise} \end{cases}$$

Note that the incidence matrix of a Reset net is the same as the net without reset arcs. However, the marking equation does not hold for Reset nets.
2.9.2 Inhibitor nets

Another known extension of Petri nets allows to test emptiness of places, by means of inhibitor arcs. Nets with inhibitor arcs are called Inhibitor nets[30].

**Definition 45 (Inhibitor net).** An Inhibitor net $N$ is a tuple $(S, T, F, I)$, where

- $S$ is a finite set, and its elements are called places.
- $T$ is a finite set, and its elements are called transitions.
- $F \subseteq (S \times T) \cup (T \times S)$ is the flow Relation. Elements in this relation are called arcs.
- $I \subseteq (T \times S)$ is the set of inhibitor arcs. We write $I(t)$ for the set of places tested by transition $t \in T$.

If $N$ is a Inhibitor net, we write $\overline{N}$ for the net without inhibitor arcs.

With Inhibitor nets, the firing rule does not change, only the rule when a transition is enabled.

**Definition 46 (enabled transition in Inhibitor nets).** Let $N = (S, T, F, I)$ be a Inhibitor net, and $m$ a marking of $N$. A transition $t$ is enabled in $(N, m)$ iff:

1. $^*t \leq m$, and
2. $\forall s \in I(t) : m(s) = 0$.

Note that the incidence matrix of an Inhibitor net is the same as the net without inhibitor arcs. Further, the marking equation holds. However, not all transitions enabled in a marking in the net without inhibitor arcs are enabled in the Inhibitor net.

2.9.3 Combining extensions

The Reachability problem for Petri nets with more than two special arcs (either reset or inhibitor arcs) is undecidable[10, 3, 30, 26].

However, if the net without the special arcs is bounded, some analysis can be performed. Here we prove that in case the net without special arcs is bounded, the original net with special arcs is bounded as well.

**Lemma 12.** Let $N = (S, T, F, R, I)$ be a Petri net with both reset arcs and inhibitor arcs, $m_0$ a marking in $N$, and $\overline{N} = (S, T, F)$ the net without special arcs. Every firing sequence in $(N, m_0)$ is a firing sequence in $\overline{N}$, and for every reachable marking $m$ in $(N, m_0)$, there is a reachable marking $\overline{m}$ in $(\overline{N}, m_0)$, such that $m \leq \overline{m}$.
Proof. Let $N = (S, T, F, R, I)$ be a Petri net with both reset arcs and inhibitor arcs, and $m_0$ a marking in $N$. We prove the lemma by induction on the length of $\sigma$.

Assume $\sigma = \epsilon$. This sequence is both a firing sequence in $N$ and in $\overline{N}$, choose $\overline{m} = m_0$.

Assume $\sigma = \sigma't$. By the induction hypothesis, we have $m, \overline{m} \in \mathbb{N}^S$, such that $m_0 \xrightarrow{\sigma'} m$ in $N$, $m_0 \xrightarrow{\sigma'} \overline{m}$ in $\overline{N}$, and $m \leq \overline{m}$. Since transition $t$ is enabled, we have $t \in S$, such that $\sigma' = \epsilon$. Thus, $\overline{m}$ is also enabled in $\overline{N}$. Let $m_1, \overline{m}_1 \in \mathbb{N}^S$, such that $m \xrightarrow{t} m_1$ in $N$, and $\overline{m} \xrightarrow{t} \overline{m}_1$ in $\overline{N}$.

Hence, we have for all $s \in S$ that $m_1(s) \leq \overline{m}_1(s)$, and $m_1 \leq \overline{m}_1$.

Using Lemma 12, we prove the following well-known result.

**Theorem 10.** Let $N = (S, T, F, R, I)$ be a Petri net with both reset arcs and inhibitor arcs. Let $\overline{N} = (S, T, F)$ be the net without reset arcs and inhibitor arcs. Let $m_0$ be a marking in $N$. If $(\overline{N}, m_0)$ is $k$-bounded, the system $(N, m_0)$ is also $k$-bounded.

Proof. By Lemma 12, for every marking $m$ in $N$, there is a reachable marking $\overline{m}$ in $(\overline{N}, m_0)$, such that $m \leq \overline{m}$. As $(\overline{N}, m_0)$ is $k$-bounded, we have for all places $s \in S : m(s) \leq k$. Hence, $\forall s \in S : m(s) \leq \overline{m}(s) \leq k$. Thus, $(N, m_0)$ is also $k$-bounded.

This theorem shows that, although the reachability problem in general is undecidable for both extensions of Petri nets, we still can do analysis. If the Petri net without the extensions is bounded, we can create a reachability graph of the net with extensions, since it is bounded.
Chapter 3

Reduction techniques

3.1 Overview

Many techniques and methods exist to analyze Petri nets. Model checking is one of the most successful and used technique. However, applying can be quite time and space consuming, especially in large models. To be able to do the same analysis in less time, smaller nets are needed. A way to make large nets smaller, is to reduce them. However, the reductions should preserve the properties to be verified. The reductions discussed in this thesis all preserve specific properties: liveness, boundedness and soundness\(^1\). The reduction rules defined on Free Choice Petri nets are discussed and explained in more detail in Section 3.2. In Section 3.3 a general reduction rule is introduced and proven to be a correct reduction: the workflow reduction rule, and Section 3.4 discusses general reduction rules applicable on any Petri net class.

3.2 Reduction techniques for Free Choice Petri nets

In [9], Desel and Esparza define three reduction rules on Free Choice Petri nets. These rules are proven to reduce any well-formed Free Choice Petri net into the smallest well-formed Free Choice Petri net, the atomic net, as shown in Figure 3.1. They proved that this set of reduction rules is sound and complete, i.e., every well-formed Free Choice Petri net can be reduced to the atomic net, and if a Free Choice Petri net can be reduced to the atomic net, it is a well-formed Free Choice Petri net.

The three rules defined are:

1. Abstraction rule

\(^1\)Soundness is closely related to both liveness and boundedness
2. Linear dependent places
3. Linear dependent transitions

### 3.2.1 Abstraction Rule

Desel and Esparza define the abstraction rule as:

**Definition 47 (Abstraction rule).** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets. Then \( (N, \tilde{N}) \in \varphi_A \) iff there exist a place \( s \in S \) and a transition \( t \in T \), such that:

1. \( \bullet s \neq \emptyset, \bullet = \{t\} \)
2. \( t^* \neq \emptyset, \bullet t = \{s\} \)
3. \( (\bullet s \times t^*) \cap F = \emptyset \)

and the net \( \tilde{N} \) is:

1. \( \tilde{S} = S \setminus \{s\} \)
2. \( \tilde{T} = T \setminus \{t\} \)
3. \( \tilde{F} = (F \cap ((\tilde{S} \times \tilde{T}) \cup (\tilde{T} \times \tilde{S})) \cup (\bullet s \times t^*) \)

This rule does not use the Free Choice property, and can therefore be applied to general Petri nets.

In the next subsection, we present the algorithm we developed to apply this rule on a given (Free Choice) Petri net.

**Algorithm**

In the algorithm we traverse all transitions, and check if it has outgoing arcs and a single incoming place (condition 2). If this is the case, the combination is tested on the other conditions.
The third condition can be rewritten into:

\[ \forall x \in s^* : \neg(\exists y \in t^* : (x, y) \in F) \]

Thus, if we calculate the pre and post set of respectively \( s \) and \( t \), we can check whether this condition holds or not. If it holds, we remove \( s \) and \( t \), and add arcs between the preset and postset of \( s \) and \( t \). The algorithm can be found in Appendix A.

### 3.2.2 Linear Dependent Places

The second reduction rule of Desel and Esparza is based on the incidence matrix of a Petri net. It reduces places that are positive linear dependent. A linear dependent place \( p \) is positive linear dependent iff \( \Lambda \geq 0 \). For Free Choice nets, the rule preserves well-formedness.

**Definition 48 (Linear dependent place rule).** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets. We say \((N, \tilde{N}) \in \phi_S\) iff the following conditions hold:

1. \( |S| > 1 \)
2. \( s \) is a positive linear dependent place of \( N \)
3. \( s^* \cup s^* \neq \emptyset \)

and the net \( \tilde{N} \) satisfies:

1. \( \tilde{N} = N \setminus \{s\} \)

The next section discusses the algorithm we developed to apply the rule on a Free Choice Petri net.
Algorithm

In the algorithm the incidence matrix is brought to the echelon form by using gaussian elimination, while logging the steps taken. If the row of a place in the incidence matrix only consists of 0s, we can see in the log how the place is constructed in terms of the other places. If this is a positive combination of other places, the place is positive linear dependent and can be removed from the Petri net. The algorithm can be found in Appendix A.

Note that the algorithm presented here is sound. If it finds a positive linear dependent place, it can be removed. However, the algorithm is not complete, since a negative linear dependent place can be positive linear dependent when choosing a different set of independent places. To solve this, the following linear programming problem has to be solved for each place $s \in S$:

$$\Lambda \cdot N = \overrightarrow{s} | \Lambda \geq 0, \Lambda(s) = 0$$

3.2.3 Linear Dependent Transitions

The third reduction rule Desel and Esparza define, is the dual of the second rule, and is called the linear dependent transition rule. The rule is similar to the linear dependent place rule, but now focuses on positive linear dependent transitions.

Definition 49. Let $N = (S, T, F)$ and $\bar{N} = (\bar{S}, \bar{T}, \bar{F})$ be two Free Choice Petri nets. We have $(N, \bar{N}) \in \phi_T$ iff the following conditions hold:

1. $|T| > 1$

2. $t$ is a positive linear dependent transition of $N$

3. $(t \cup t^*) \neq \emptyset$
and the net $\tilde{N}$ satisfies:

1. $\tilde{N} = N \setminus \{t\}$

Figure 3.4: Positive linear dependent transition $b$ is removed

To calculate whether a transition is a positive linear dependent transition, we can use the same algorithms used for the linear dependent places, by transposing the incidence matrix of the Petri net.

### 3.2.4 Test for well-formed Free Choice Petri Nets

If a Petri Net is a well-formed Free Choice Petri Net, it reduces to an atomic net. If it does not, it was not well-formed. As the algorithm for the abstraction rule is less expensive than the linear dependent places and transitions, we suggest to start with the abstraction rule.

### 3.3 Reduction techniques with Workflow Petri nets

Generalized sound Workflow Petri nets have the property that every token put in the initial place, is eventually, under the fairness assumption, in the final place of the Workflow Petri net, and there are no tokens left in the net. A generalized sound Workflow Petri net thus can be seen as a refined place. Models are often created by stepwise refinement: first an abstract model is created, then the different places and transitions are in each step more refined. Analyzing Petri nets, we would like to use this same principle, only the other way around: reduce a Subworkflow Petri net that is generalized sound into a single place. We call this reduction rule the Workflow reduction rule.

**Definition 50 (Workflow reduction rule $\phi_W$).** Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets. We have $(N, \tilde{N}) \in \phi_W$ iff
• there exists a Subworkflow Petri net $N' = (S', T', F') \subseteq N$, being a pWorkflow Petri net with source $i' \in S'$ and sink $f' \in S'$, and

• $N'$ is generalized sound.

and $\tilde{N}$ satisfies, where $s$ is a new place, $s \not\in S$:

• $\tilde{S} = (S \setminus S') \cup \{s\}$

• $\tilde{T} = T \setminus T'$

• $\tilde{F} = (F \setminus F') \cup (\hat{f}^{-1}(i') \times \{s\}) \cup (\{s\} \times (f')^*_{\tilde{N}})$

We now show that if a Petri net is live and bounded, then the reduced net is also live and bounded, thus that the Workflow reduction rule preserves liveness and boundedness. We also show that if a Petri net is not live and bounded, then the reduced net is also not live and bounded.

Lemma 13. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri net, such that $(N, \tilde{N}) \in \phi_W$, and $N_1 = (S_1, T_1, F_1)$ a generalized sound Subworkflow Petri net of $N$, with source place $i$ and sink place $f$. If $N$ is live and bounded, then $\tilde{N}$ is also live and bounded.

Proof. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri net, such that $(N, \tilde{N}) \in \phi_W$, and $N_1 = (S_1, T_1, F_1)$ a generalized sound Subworkflow Petri net of $N$, with source place $i$ and sink place $f$. Suppose $N$ is live and bounded.

As $N_1$ is generalized sound, it is bounded[35], thus all places in $S_1$ are bounded. Since all places in $S$ are bounded, also all places in $S \setminus S_1$ are bounded. Let $s \not\in S$ be the place added by the reduction rule. Then we have $s^* = i$ and $s^* = f^*$. As both $i$ and $f$ are bounded, $s$ is also bounded. Thus $(S \setminus S_1) \cup \{s\}$ is also bounded. Hence, $\tilde{N}$ is bounded.

As $N_1$ is generalized sound, any token in $i$ always eventually leads to a token in $f$. Thus any firing sequence that puts a token in $i$, always eventually fire a transition in $f^*$. As $s^* = i$ and $s^* = f^*$, any token produced by $s$ leads directly to a firing of a transition in $f^*$, via place $s$. Hence if $N$ is live, $\tilde{N}$ is also live. \qed

Lemma 14. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri net, such that $(N, \tilde{N}) \in \phi_W$, and $N_1 = (S_1, T_1, F_1)$ a generalized sound Subworkflow Petri net of $N$, with source place $i$ and sink place $f$. If $N$ is not live nor bounded, then $\tilde{N}$ is also not live nor bounded.

Proof. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri net, such that $(N, \tilde{N}) \in \phi_W$, and $N_1 = (S_1, T_1, F_1)$ a generalized sound Subworkflow Petri net of $N$, with source place $i$ and sink place $f$. Suppose $N$ is not live nor bounded.

As $N_1$ is generalized sound, it is bounded, thus all places in $S_1$ are bounded. Hence, the unbounded places are in $S \setminus S_1$. Let $s \not\in S$ be the place added by
the reduction rule. Then \( s \) is also bounded. Thus \( (S \setminus S_1) \cup \{s\} \) is unbounded. Hence \( \tilde{N} \) is unbounded.

As \( N_1 \) is generalized sound, any token in \( i \) always eventually leads to a token in \( f \). \( s \star s = \star i \) and \( s \star = f \star \), any token produced by \( \star i \) leads directly to a firing of a transition in \( f \star \), via place \( s \). Hence, the transitions in \( T_1 \) do not influence liveness, and \( \tilde{N} \) remains not live.

If we search through a Petri net we can find many Workflow Petri nets. These subnets can overlap each other, or one can contain another. The reduction rule has to ensure that the order of reduction does not influence the reduction results. In other words, for any Subworkflow Petri nets \( N_1 \) and \( N_2 \), if first \( N_1 \) has been reduced, and then \( N_2 \), the result would be the same as if we first reduced \( N_2 \) and then \( N_1 \). This property is known as commutation[4].

**Lemma 15.** The reduction rule \( \phi_W \) is commutative.

**Proof.** Let \( N = (S, T, F) \) be a Petri net, \( N' = (S', T', F') \subseteq N \) be a pWorkflow Petri net with source \( i' \in S' \) and sink \( f' \in S' \), and let \( N'' = (S'', T'', F'') \subseteq N \) be a pWorkflow Petri net with source \( i'' \in S'' \) and sink \( f'' \in S'' \). Let \( s \) and \( r \) be new places: \( s, r \notin S \).

Analyzing the different reduction possibilities, we have four cases to proof:

1. \( N' \cap N'' = \emptyset \), thus no overlapping of both nets. We can first reduce \( N' \) and then \( N'' \) or the other way around. Reducing \( N' \) gives a Petri net \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) with:

\[
\tilde{S} = (S \setminus S') \cup \{s\} \\
\tilde{T} = T \setminus T' \\
\tilde{F} = (F \setminus F') \cup (\star(i') \times \{s\}) \cup (\{s\} \times (f')\star)
\]

We now reduce \( N'' \) in \( \tilde{N} \) resulting in the net \( \tilde{\tilde{N}} = (\tilde{\tilde{S}}, \tilde{\tilde{T}}, \tilde{\tilde{F}}) \):

\[
\tilde{\tilde{S}} = ((S \setminus S') \cup \{s\}) \setminus S'' \cup \{r\} \\
\tilde{\tilde{T}} = (T \setminus T') \setminus T'' \\
\tilde{\tilde{F}} = ((F \setminus F') \cup (\star(i') \times \{s\}) \cup (\{s\} \times (f')\star)) \setminus F'' \\
\cup (\star(i'\star) \times \{r\}) \cup (\{r\} \times (f''\star))
\]

The net \( \tilde{\tilde{N}} \) can be rewritten to:

\[
\tilde{\tilde{S}} = ((S \setminus S') \setminus S'') \cup \{s, r\} \\
\tilde{\tilde{T}} = (T \setminus T') \setminus T'' \\
\tilde{\tilde{F}} = ((F \setminus F') \setminus F'') \cup (\star(i') \times \{s\}) \cup (\{s\} \times (f')\star) \\
\cup (\star(i'\star) \times \{r\}) \cup (\{r\} \times (f''\star))
\]
Reducing first $N''$ and then $N'$ is analogous, and results in a net $\tilde{N} = (\tilde{\mathcal{S}}, \tilde{T}, \tilde{\mathcal{F}})$ with:

\[
\begin{align*}
\tilde{\mathcal{S}} &= ((S \setminus S') \setminus S'') \cup \{r, s\} \\
\tilde{T} &= T \setminus T'' \\
\tilde{\mathcal{F}} &= ((F \setminus F') \setminus F'') \cup (\{i''\} \times \{r\}) \cup \{r\} \times (f''(i'')) \\
&\quad \cup (\{i'\} \times \{s\}) \cup \{\{s\} \times (f'(i'))
\end{align*}
\]

Because $S' \cap S'' = T' \cap T'' = F' \cap F'' = \emptyset$, we have $\tilde{N} = \tilde{N}$.

2. Next, assume that $N' \cap N''$ is not the empty Petri net, and $|S_1 \cap S_2| = 1$. The sink of $N'$ equals the source of $N''$. We have: $S' \cap S'' = \{i''\}$ and $T' \cap T'' = F' \cap F'' = \emptyset$. Note that $f' = i'' \subseteq S'$ and $i' = f'(i') \subseteq S''$, since $N'$ and $N''$ are sub Workflow Petri nets. We first reduce $N'$. We then get the net $\tilde{N} = (\tilde{\mathcal{S}}, \tilde{T}, \tilde{\mathcal{F}})$ with:

\[
\begin{align*}
\tilde{\mathcal{S}} &= (S \setminus S') \cup \{s\} \\
\tilde{T} &= T \setminus T' \\
\tilde{\mathcal{F}} &= (F \setminus F') \cup (i' \times \{s\}) \cup \{\{s\} \times (f'(i'))
\end{align*}
\]

We cannot reduce $N''$, since $f' \not\subseteq \tilde{\mathcal{S}}$, and thus $i''$ also not. As $i'' = f'$, the place $i''$ is mapped on $s$. Now alter $N''$ such that we can reduce it: $S'' = (S'' \setminus \{i''\}) \cup \{s\}$. Reducing with $\phi_W$ now gives $\tilde{N} = (\tilde{\mathcal{S}}, \tilde{T}, \tilde{\mathcal{F}})$:

\[
\begin{align*}
\tilde{\mathcal{S}} &= (S \setminus S' \cup \{s\}) \setminus S'' \cap \{s\} \\
\tilde{T} &= (T \setminus T') \setminus T'' \\
\tilde{\mathcal{F}} &= ((F \setminus F') \setminus F'') \cup (i' \times \{s\}) \cup \{\{s\} \times (f'(i'))
\end{align*}
\]

The $\text{r}_N s$ is the same as saying the $\text{r}(i')$, because the node $i' \in N$ is mapped onto $s \in \tilde{N}$. Because the $f''$ is not changed in $\tilde{N}$, we have: $f''_{\tilde{N}} = f''_{\tilde{N}}$.

Therefore we can rewrite $\tilde{N}$ to:

\[
\begin{align*}
\tilde{\mathcal{S}} &= ((S \setminus S') \setminus S'') \cup \{r\} \\
\tilde{T} &= (T \setminus T') \setminus T'' \\
\tilde{\mathcal{F}} &= ((F \setminus F') \setminus F'') \cup (i' \times \{r\}) \cup \{\{r\} \times (f'(i'))
\end{align*}
\]

Now we first reduce $N''$ before $N'$. This results in the net $\tilde{N} = (\tilde{\mathcal{S}}, \tilde{T}, \tilde{\mathcal{F}})$:

\[
\begin{align*}
\tilde{\mathcal{S}} &= (S \setminus S'') \cup \{s\} \\
\tilde{T} &= T \setminus T'' \\
\tilde{\mathcal{F}} &= (F \setminus F'') \cup (i'' \times \{s\}) \cup \{\{s\} \times (f'(i''))
\end{align*}
\]

Because $i'' = f'$, $N'$ is not reducible in this net. Since $f' = i''$, $f'$ is mapped on $s$. Now alter the net $N'$ by replacing the place $f'$ by $r$. Now we can reduce $N'$ in the net $\tilde{N}$, resulting in the net $\tilde{N} = (\tilde{\mathcal{S}}, \tilde{T}, \tilde{\mathcal{F}})$ with:

\[
\begin{align*}
\tilde{\mathcal{S}} &= (S \setminus S'' \cup \{s\}) \setminus S' \cup \{r\} \\
\tilde{T} &= T \setminus T' \setminus T'' \\
\tilde{\mathcal{F}} &= (F \setminus F'') \cup (i'' \times \{s\}) \cup \{\{s\} \times (f'(i'')) \setminus F' \cup (\text{r}_N i' \times \{r\}) \cup \{\{r\} \times (s)\}
\end{align*}
\]
In all four cases we get the same result if we reduce first $N_\phi$. Suppose that $(s)_N^* = (\beta^{-1}(s))^\bullet \backslash T'' = (S'' \backslash T'') \backslash T'' = (T'' \cup (f''^u))^\bullet \backslash T'' = (f''^u)^\bullet$.

Thus we can simplify $N_\phi$ to:

$\tilde{S} = (S \setminus S'^u \cup \{i''\}, T' \setminus T'', F' \setminus F'')$ with source $i'$ and sink $i''$.

Now have that $N_\phi$, substituting the new place $r$ into the new place $s$, and $N_\phi$ are equal.

3. Suppose that $N' \cap N''$ is not the empty Petri net, $N' \not\subseteq N''$, and $N'' \not\subseteq N'$. According to lemma 8, we have that $N_1 \cup N_2$ is again a Subworkflow Petri net. By lemma 9, the Subworkflow Petri net $(N' \cup N'')$ consists of three Workflow Petri nets, namely:

- $N_1 = (S' / S'^u \cup \{i''\}, T' / T'' / F' \setminus F'')$ with source $i'$ and sink $i''$.
- $N_2 = (S'' / S'^u \cup \{i''\}, T'' \setminus T', F'' \setminus F'')$ with source $f'$ and sink $f''$.
- $N_3 = (S' / S'^u \cup \{i''\}, T / T'' / F' \setminus F'')$ with source $i''$ and sink $f'$.

The reduction of $N'$ and $N''$ is thus a reduction of $N_1$ and $N_2$, where the sink of $N_1$ equals the source of $N_2$, solved in case 2, and a reduction of $N_2$ and $N_3$, where the sink of $N_2$ equals the source of $N_3$, solved in case 2. Thus the case $N' \cap N'' \neq \emptyset$ also gives a correct result.

4. $N'' \subseteq N'$. First reducing $N'$, results in the net: $\tilde{N} = (S \setminus S' \cup \{s\}, T \setminus T', F' \setminus F'' \cup (\{i''\} \times \{s\}) \cup (\{s\} \times (f''^u)^\bullet))$. In this net, all elements of $N''$ are already reduced, thus this is the result of first reducing $N'$, then $N''$. If we first reduce $N''$, we get the net $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$:

$\tilde{S} = (S \setminus S' \cup \{r\}) \cup \{r\}$

$\tilde{T} = T \setminus T'' \cup (\{r\} \times \{s\}) \cup (\{s\} \times (f''^u)^\bullet)$

All elements of $N''$ are mapped on the place $r$, as $N'' \subseteq N'$, $N''$ is also reduced in $N'$ to place $r$. Reducing this net, results in $\hat{N} = (\hat{S}, \hat{T}, \hat{F})$, with:

$\hat{S} = (S \setminus S' \cup \{r\} \setminus S' \cup \{s\}) = S \setminus S' \cup \{s\}$

$\hat{T} = T \setminus T'' \cup T' \setminus T'$

$\hat{F} = (F \setminus F'') \cup (\{r\} \times (f''^u)^\bullet) \cup (\{s\} \times (f''^u)^\bullet)$

$\hat{F} = (F'' \setminus F'') \cup (\{i''\} \times \{s\}) \cup (\{s\} \times (f''^u)^\bullet)

Thus $\tilde{N} = \hat{N}$.

In all four cases we get the same result if we reduce first $N'$ and then $N''$ or we do it vice versa. Thus the rule $\phi_W$ is commutative.
3.3.1 Algorithm

To be able to reduce a Subworkflow Petri net, we need an algorithm to find a Subworkflow Petri net in a general Petri net $N$. A main property is that the closure of a Workflow Petri net is a strongly connected component. This property we use in the algorithm. If we take a candidate source $i$ and a candidate sink $f$, remove the incoming arcs of $i$ and the outgoing arcs of $f$, and shortcut it, we should obtain a strongly connected component in the net. By using a standard algorithm for finding strongly connected components, we can obtain the component of $i$ and $f$. Removing the shortcut transition from the component, we have a Subworkflow Petri net, with source $i$, and sink $f$.

**Algorithm 3.1: Find Subworkflow Petri net**

<table>
<thead>
<tr>
<th>Input: Workflow Petri net $N = (S, T, F)$, source candidate $i \in S$, sink candidate $f \in S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: If $N$ contains a Subworkflow Petri net $N_1$, $N_1$ is returned, otherwise the empty set</td>
</tr>
<tr>
<td>foreach $a \in (\bullet i \times i) \cap F$ do</td>
</tr>
<tr>
<td>$F := F \setminus {a};$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>foreach $a \in (f \times f^*) \cap F$ do</td>
</tr>
<tr>
<td>$F := F \setminus {a};$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>$t^* :=$ new transition;</td>
</tr>
<tr>
<td>$T := T \cup {t^*};$</td>
</tr>
<tr>
<td>$F := F \cup {(f, t^<em>), (t^</em>, i)};$</td>
</tr>
<tr>
<td>$WF :=$ giveStronglyConnectedComponent($N, i$);</td>
</tr>
<tr>
<td>if $t^* \in WF$ then</td>
</tr>
<tr>
<td>if $(\exists n \in \bullet i : n \in WF) \lor (\exists n \in f_N : n \in WF)$ then</td>
</tr>
<tr>
<td>return $\emptyset;$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $WF \setminus {t^*};$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $\emptyset;$</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

To calculate the strongly connected component of a node $n$ in the net $N$, we use a standard algorithm from [8], page 552 – 557. The algorithm uses the Depth First Search (DFS) to calculate the strongly connected components in a graph $G$.

The next step is to check generalized soundness. If this sub net is an ST net, it is clear, otherwise we use the algorithm for batch Workflow Petri nets, to check whether it is a batch Workflow Petri net, and call the generalized soundness checker[20, 28]. If the result yields generalized sound, we can reduce it to a single place, and connect it with $\bullet i$ and $f^*$. The resulting net $\tilde{N}$ is the reduced net of $N$, having: $(N, \tilde{N}) \in \phi_W$. 

30
3.4 General reduction techniques

In this section two general reduction techniques, applicable on any class of Petri nets, are discussed. First we discuss the rules of Murata, defined in [27], and secondly a set of reduction rules of Berthelot [5].

3.4.1 The reduction rules of Murata

In [27] Murata defines six different reduction rules, preserving liveness, safeness and boundedness. In this section we introduce the rules of Murata, as he has introduced them, and define them mathematically.

1. Fusion of Series Places. This rule removes a place and a transition not influencing the process. Formally:

**Definition 51 (Fusion of Series Places).** Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets. We say $(N, \tilde{N}) \in \phi_{M1}$ iff there exists places $s, r \in S$, $s \neq r$ and a transition $t \in T$ such that:

   - (a) $\cdot t = \{s\}, \cdot t = \{r\}$
   - (b) $\cdot s = \{t\}, \cdot s \neq \emptyset$
   - (c) $\not\subseteq \cdot r$

   and the net $\tilde{N}$ satisfies:

![Figure 3.5: The six reduction rules, defined by Murata](image-url)
Lemma 16. Fusion of Series Places is a special case of the abstraction rule.

Proof. The first condition of the abstraction rule, is fulfilled, since \( \bullet s \neq \emptyset \) and \( s^* = \{t\} \), by the first condition of \( \phi_{M1} \). The second condition of the abstraction rule is also fulfilled, since \( \bullet t = \{s\} \), and \( t^* = \{r\} \neq \emptyset \), by the second condition of the Fusion of Series Places. The third condition of the abstraction rule is fulfilled by the third condition of the Fusion of Series Places. Therefore, also the abstraction rule is applicable. The construction of the reduced net of both rules are identical, and therefore, the result of both rules is the same. Thus Fusion of Series Places is a special case of the abstraction rule. \( \square \)

2. Fusion of Series Transitions. This rule removes a transition and a place not influencing the process.

Definition 52 (Fusion of Series Transitions). Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Petri nets. We say \( (N, \tilde{N}) \in \phi_{M2} \) iff there exists a place \( s \in S \) and transitions \( t, u \in T \), \( t \neq u \) such that:

(a) \( \bullet s = \{u\}, s^* = \{t\} \)
(b) \( \bullet t = \{s\}, t^* \neq \emptyset \)
(c) \( u^* \not\subseteq t^* \)

and the net \( \tilde{N} \) satisfies:

(a) \( \tilde{S} = S \setminus \{s\} \)
(b) \( \tilde{T} = T \setminus \{t\} \)
(c) \( \tilde{F} = (F \cap ((\tilde{S} \times \tilde{T}) \cup (\tilde{T} \times \tilde{S}))) \cup (\bullet s \times t^*_N) \)

Lemma 17. Fusion of Series Transitions is a special case of the abstraction rule.

Proof. The proof is identical to the proof of lemma 16. \( \square \)

3. Fusion of Parallel Places. If two places have the same singleton preset and the same singleton postset, the places are identical and can be removed.

Definition 53 (Fusion of Parallel Places). Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Petri nets. We say \( (N, \tilde{N}) \in \phi_{M3} \) iff there exist places \( s, r \in S, s \neq r \) and transitions \( t, u \in T, t \neq u \), such that:

(a) \( \bullet s = \bullet r = \{t\} \)
(b) \( s^* = r^* = \{u\} \)

and the net \( \tilde{N} \) satisfies:

(a) \( \tilde{N} = N \setminus \{s\} \)
Lemma 18. Fusion of Parallel Places applied on Free Choice Petri nets is a special case of the linear dependent places rule.

Proof. Let $N$ be a Free Choice Petri net. As $s, r \in S$, and $s \neq r$, we have $|S| > 1$. Further, we have, $s^* \neq \emptyset$ and $r^* \neq \emptyset$, thus the third condition of the linear dependent places rule is fulfilled. We only have to show that $s$ is a positive linear dependent place. Since $s^* = r^*$, we have $F^-(s) = F^-(r)$, and $F^+(s) = F^+(r)$. Therefore $N(s) = F^+(s) - F^-(s) = F^+(r) - F^-(r) = N(r)$. Thus $s$ is a linear dependent place. It is a positive linear combination of the place $r \neq s$. Therefore we can also apply the positive linear dependent places rule. From the construction of the reduced net of both reduction rules can directly be seen that they give the same results. 

4. Fusion of Parallel Transitions. This rule is the dual of the fusion of parallel places, working on transitions.

Definition 54 (Fusion of Parallel Transitions). Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets. We say $(N, \tilde{N}) \in \phi_{M5}$ iff there exist places $s, r \in S$, $s \neq r$ and transitions $t, u \in T$, $t \neq u$, such that:

(a) $t^* = u^* = \{r\}$
(b) $t^* = u^* = \{s\}$

and the net $\tilde{N}$ satisfies:

(a) $\tilde{N} = N \setminus \{t\}$

Lemma 19. Fusion of Parallel Transitions applied on Free Choice Petri nets is a special case of the linear dependent transitions rule.

Proof. Let $N$ be a Free Choice Petri net. As $t, u \in T$, and $t \neq u$, we have $|T| > 1$. Further, we have $t^* \neq \emptyset$ and $t^* \neq \emptyset$, thus the third condition of the linear dependent transitions rule is fulfilled. We have $F^-(t) = F^-(u)$, $F^+(t) = F^+(u)$ by the two conditions of the Fusion of Parallel Transitions, and therefore: $N(t) = F^+(t) - F^-(t) = F^+(u) - F^-(u) = N(u)$, thus $t$ is a positive linear dependent place. 

5. Elimination of Self-loop Places. If a transition $t$ consumes and produces in the same place $s$ with one or more tokens, and $s^* = s^* = \{t\}$, the place can be removed. This place is an implicit place.

Definition 55 (Elimination of Self-loop Places). Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets, and $m$ the initial marking in $N$. We say $(N, \tilde{N}) \in \phi_{M5}$ iff there exists a place $s \in S$ and a transition $t \in T$, such that:

(a) $m(s) > 0$
(b) $s^* = \{t\}$
(c) $s^* = \{t\}$

and the net $\tilde{N}$ satisfies:

(a) $\tilde{N} = N \setminus \{s\}$

33
6. Elimination of Self-loop Transitions. If a transition \( t \) both consumes and produces in the same place \( s \) and \( t^* = t^* = \{s\} \), then this transition can be removed. It is an implicit transition.

**Definition 56 (Elimination of Self-loop Transitions).** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Petri nets. We say \( (N, \tilde{N}) \in \phi_{M6} \) iff there exists a place \( s \in S \) and a transition \( t \in T \), such that:

(a) \( t^* = \{s\} \)

(b) \( t^* = \{s\} \)

and the net \( \tilde{N} \) satisfies:

(a) \( \tilde{N} = N \setminus \{t\} \)

### 3.4.2 The reduction rules of Berthelot

Berthelot distinguishes in his thesis\[5\] three classes of reduction rules:

1. place substitution,
2. simplification of implicit places, and  
3. irrelevant transitions.

All reduction rules are applicable on general Petri nets and preserve liveness, boundedness, and termination. In this section we discuss briefly the first and third reduction rules. The first rule is a generalization of the abstraction rule \( \phi_A \), the last we also implemented.

### 3.4.3 Place substitution

The first rule Berthelot introduces, substitutes a single place by removing it, and for each reachable marking from firing a transition in the preset of \( s \), a new transition is added. In this way the number of places is reduced, only the number of possible transitions grow.

Before we define this rule, we define a homomorphism on markings. This homomorphism maps markings of the original net to markings of the reduced net.

**Definition 57 (substitution homomorphism).** Let \( N = (S, T, F) \) be a Petri net, and \( M \) a marking of \( N \). Let \( p \in S \) be a place, \( S' = S \setminus \{p\} \), and \( Q \in \mathcal{P}(\mathbb{N}^{S'}) \). Let \( D = \{M + [p^k] | M \in \mathbb{N}^{S'}, k \in \mathbb{N}\} \). The substitution homomorphism \( HS : D \to \mathcal{P}(\mathbb{N}^{S'}) \) is defined as:

- \( HS([p^m]) = Q \):
• ∀m′ ∈ N′: HS(m′) = {m′};
• ∀m₁, m₂ ∈ D: HS(m₁ + m₂) = HS(m₁) ⊕ HS(m₂);

where ⊕ is defined on two sets A and B as: A ⊕ B = \{a + b | a ∈ A, b ∈ B\}. We write HS_{[p^m]/Q}(M) for the substitution homomorphism of marking [p^m] with Q.

In this definition, the original marking with p is mapped on a mapping without p. The tokens in p are mapped on the markings defined by Q. The set D is the set of markings in the original net.

As an example, let N = (S, T, F) be a Petri net, p ∈ S, S′ = S \ {p}, and M₁', M₂', M₃' markings over S'. Take Q = \{M₂', M₃'\}. Then

\[
HS_{[p^m]/Q}(M₁') = \{M₁'\}
\]
\[
HS_{[p^m]/Q}(p) = \{M₂', M₃'\}
\]
\[
HS_{[p^m]/Q}(M₁' + p) = \{M₁' + M₂', M₁' + M₃'\}
\]
\[
HS_{[p^m]/Q}(M₁' + p²) = \{M₁' + 2 · M₂', M₁' + 2 · M₃', M₁' + M₂' + M₃'\}
\]

We can now define the first reduction rule of Berthelot:

Definition 58 (place substitution R1). Let N = (S, T, F) and \(\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})\) be two Petri nets, with markings m and \(\tilde{m}\). The relation ((N, m), (\(\tilde{N}, \tilde{m}\)) ∈ \(\phi_{R1}\) exists if and only if there exists places s ∈ S, n ∈ N and sets G, H ⊆ T such that:

1. only transitions in H produce in s: *s = H;
2. only transitions in $G$ consume from $s$: $s^* = G$;
3. the only input place of $G$ is $s$;
4. the postset of $G$ is not empty: $G^* \neq \emptyset$;
5. the postset of $G$ does not contain $s$: $s \not\in G^*$;
6. the input transitions of $s$ do not consume from $s$: $s \not\in \cdot H$;

Let $h \in H$. For each element $W \in H S[[s^n]/Q](h^*)$, we add a transition $h_W$, such that $h_W^* = h$, and $h_W^* = W$. We denote the set of these transitions with $H S(h)$. We write for a set $H$: $H S(H) = \cup_{h \in H} H S(h)$. The system $(\tilde{N}, \tilde{m})$ is constructed as:

$$
\tilde{m} = H S[[s^n]/Q](m);
\tilde{S} = S \setminus \{s\};
\tilde{T} = H S(H) \cup (T \setminus (G \cup H));
\tilde{F} = (F \cap ((\tilde{S} \times \tilde{T}) \cup (\tilde{T} \times \tilde{S}))) \cup
\bigcup_{h \in H, W \in H S[[s^n]/Q](h^*)} ((s \times \{h_W\}) \cup (\{h_W\} \times W)).
$$

Figure 3.6 gives an example of this rule. In this example we have: $H = \{h_1, h_2\}$, $G = \{f_1, f_2\}$, and $Q = \{\overrightarrow{e}, \overrightarrow{f} + \overrightarrow{g}\}$, $H S[[s^n]/Q](h_1^*) = H S[[s^n]/Q](\overrightarrow{p}) = \{\overrightarrow{e}, \overrightarrow{f} + \overrightarrow{g}\}$, and $H S[[s^n]/Q](h_2^*) = H S[[s^n]/Q](\overrightarrow{s} + \overrightarrow{d}) = \{\overrightarrow{e} + \overrightarrow{d}, \overrightarrow{f} + \overrightarrow{g} + \overrightarrow{d}\}$. The elements of the homomorphism define the four resulting transitions $h_1-1$, $h_1-2$, $h_2-1$, and $h_2-2$.

### 3.4.4 Irrelevant transitions

Irrelevant transitions are transitions that do not influence the marking, either by having the same preset and postset, or by having an identical preset and postset. This two cases are split in two separate reduction rules: the identical transition rule RO, for the first, and the identity transition rule R3 for the latter.

**Definition 59 (identity transition rule R3).** Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets. We say $(N, \tilde{N}) \in \phi_{R3}$ if there is a transition $t \in T$, such that:

1. $t^* = t^*$.

and the net $\tilde{N}$ satisfies:

1. $\tilde{N} = N \setminus \{t\}$

36
Lemma 20. The Elimination of Self-loop Transitions is a special case of the identity transition rule.

Proof. From the conditions of the Elimination of Self-loop Transitions, we have \( t = \{ s \} = t^\ast \). Thus we have \( t = t^\ast \), and thus the identity transition rule can also be applied. From the construction of the reduced net of both rules, we can directly conclude that both yield the same result.

Definition 60 (identical transition rule \( \text{Ro} \)). Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Petri nets. We say \( (N, \tilde{N}) \in \phi_{\text{Ro}} \) iff there are two transition \( t_1, t_2 \in T \), \( t_1 \neq t_2 \), such that:

1. \( \cdot t_1 = \cdot t_2 \);
2. \( t_1^\ast = t_2^\ast \).

and the net \( \tilde{N} \) satisfies:

1. \( \tilde{N} = N \setminus \{ t_1 \} \)

Lemma 21. The identical transition rule applied on Free Choice Petri nets is a special case of the linear dependent transitions rule.

Proof. We know \( |T| > 1 \), since \( t_1, t_2 \in T \) and \( t_1 \neq t_2 \). Since \( \cdot t_1 = \cdot t_2 \) and \( t_1^\ast = t_2^\ast \), we have \( F^+ (t_1) = F^+ (t_2) \), and \( F^- (t_1) = F^- (t_2) \) respectively. Thus we have \( N(t_1) = F^+ (t_1) - F^- (t_1) = F^+ (t_2) - F^- (t_2) = N(t_2) \), thus \( t_1 \) is a positive linear dependent transition. Both rules use the same construction for the reduced net, and therefore yield the same result.

3.5 Classification

In this section we classify the reduction rules in two ways, according to how the reduction influences the reachability graph, and on relationships between the dif-
3.5.1 Classification based on marking

Many analysis problems are related to the reachability graph. The reachability graph of a system can be reduced in two ways: on the one hand by reducing the size of the markings (the number of places becomes smaller), and on the other hand by reducing the number of transitions.

The following reduction rules reduce the marking:

- Berthelot's place substitution, $\phi_R^1$;
- Murata's fusion of Series places, $\phi_M^1$;
- Murata's fusion of Series transitions, $\phi_M^2$;
- Berthelot's simplification implicit places, $\phi_R^2$;
- Murata's fusion of parallel places, $\phi_M^3$;
- Murata's elimination of self-loop places, $\phi_M^5$;
- Desel and Esparza's abstraction rule, $\phi_A$;
- Desel and Esparza's positive linear dependent places, $\phi_S$;
- Workflow reduction rule, $\phi_W$;

Note that Berthelot's simplification of implicit places ($\phi_R^3$) is a special case. It does not reduce the size of the marking, it only makes the marking smaller, and reduces the number of arcs between the simplified place and its input and output transitions.

The following reduction rules reduce the number of transitions:

- Berthelot's identical transitions, $\phi_R^0$;
• Berthelot’s identity transition, $\phi_{R3}$;
• Murata’s fusion of parallel transitions, $\phi_{M4}$;
• Murata’s elimination of self-loop transitions, $\phi_{M6}$;
• Desel and Esparza’s positive linear dependent transitions, $\phi_{T}$;

3.5.2 Relationship between reduction rules

Next, we classify the reduction rules according to their relationship between each other.

• The abstraction rule $\phi_{A}$ is a generalization of Murata’s fusion of series places, $\phi_{M1}$, and of Murata’s fusion of series transitions, $\phi_{M2}$.

• Berthelot’s place substitution, $\phi_{R1}$ is a generalization of the abstraction rule $\phi_{A}$, since $\phi_{A}$ requires a single input place of a transition, whereas $\phi_{R1}$ does not. The proof can be found in Section 6.1.

• For Free Choice Petri nets, Desel and Esparza’s positive linear dependent place rule, $\phi_{S}$, removes places that are positive linear dependent. In Murata’s simplification of parallel places, $\phi_{M3}$, a positive linear dependent place is removed.

• Desel and Esparza’s positive linear dependent transition rule, $\phi_{T}$, operate on Free Choice Petri nets and remove a positive linear dependent transition. This is a more general approach than Berthelot’s identical transitions rule, $\phi_{R0}$, and identity transitions rule, $\phi_{R3}$, the removed transitions are positive linear dependent. However, Berthelot’s rules can be applied on any class of Petri nets.

• Murata’s fusion of parallel transitions, $\phi_{M4}$, can be generalized by the identical transitions rule, $\phi_{R0}$. Murata gives an extra condition that there is a single input place for both transitions, whereas this condition is left out by Berthelot.

• Murata’s elimination of self-loop transitions, $\phi_{M6}$, can be generalized by Berthelot’s identity transitions rule, $\phi_{R3}$. Berthelot only requires that input places and output places have to be equal, Murata also requires that this is a single place.

The relationships are depicted in Figure 3.9.
Figure 3.9: Classification of the reduction rules based on generalization
Chapter 4

Analysis procedure

In this chapter, we combine the checks of Chapter 2 and Chapter 3 into a procedure to analyze Petri nets. Figure 4.1 shows the procedure on the highest level. As generalized soundness only works on Workflow Petri nets, we first check whether the given Petri net is a Workflow Petri net. If this is the case, the sub procedure for generalized soundness is started, otherwise the sub procedure for well-formedness is started.

Figure 4.1: Procedure for analyzing Petri nets, highest level

4.1 Sub procedure for soundness analysis

The sub procedure for soundness analysis is depicted in Figure 4.2. We first try to reduce the Workflow Petri net. We use here the reduction rules of Murata (Algorithms A.20, A.21, A.22, A.23, A.24, and A.25), Berthelot (Algorithms A.26
and A.27), and the Workflow reduction rule (Algorithm 3.1), until we obtain a smallest, not reducible Petri net. After the different reductions, we check whether the net is an ST-net (Algorithm A.8). If the net is an ST-net, we directly know that the Petri net is generalized sound. If the net was not an ST-net, we continue the procedure. We first perform a check whether the net is a Free Choice Petri net (Algorithm A.3). This check could also be done before the ST-net check. However, not all ST-nets are Free Choice Petri nets, and vice versa, and therefore you need to do the ST-net check in both cases. Therefore we chose to perform the ST-net check first, although this algorithm is more expensive.

If the net is a Free Choice Petri net, we can use the property that the closure of a sound Workflow Petri net is well-formed, and use the reduction rules of Desel and Esparza (algorithms A.15, A.17, and the dual of Algorithm A.17) to reduce the closed net to a smallest net. If this net is an atomic net (Algorithm A.4), the closed net is well-formed. However, more analysis is needed here to check if the net with as marking a single token in the initial place is live and bounded.

Otherwise, if the net is not an ST-net, nor a Free Choice Petri net, we perform the next check, for Batch Workflow Petri nets. First we try to transform it into a Batch Workflow Petri net (Algorithm A.14). If the transformation succeeds, and the result is a Batch Workflow Petri net (Algorithm A.13), we can apply analysis tools on it, to check generalized soundness (see [28]). If the transformation did not succeed, the Petri net could not be generalized sound.

### 4.2 Sub procedure for well-formedness analysis

The procedure for well-formedness analysis is performed, if the given Petri net is no Workflow Petri net. We first try to reduce the Petri net. We use the reduction rules of Murata (Algorithms A.20, A.21, A.22, A.23, A.24, and the Workflow reduction rule (Algorithm 3.1), until we obtain a smallest, not reducible Petri net. Then, we check to which class the Petri net belongs (algorithms A.1, A.2, and A.3). In the case it is a State Machine Petri net, the net is reduced using the abstraction rule (Algorithm A.15) and the linear dependent transitions rule (dual of Algorithm A.17), until no reductions can be performed anymore. If the resulting net is the atomic net, it is well-formed. If it is a Marked Graph Petri net, we reduce it using the abstraction rule and the linear dependent places rule (Algorithm A.17). In the case the net is a Free Choice Petri net, we use all reduction techniques of Desel and Esparza.

Otherwise, it does not belong to any of these classes, and we check whether it is an atomic net. If not, we need to use analysis tools to do further analyze.
Figure 4.2: Sub procedure for analyzing Petri nets on soundness
Figure 4.3: Sub procedure for analyzing Petri nets on well-formedness
Chapter 5

Implementation

In this chapter, we discuss the implementation of the algorithms in this thesis. Most of the described algorithms are implemented as libraries for Yasper, a tool for editing and simulation of Petri nets. After a brief introduction to Yasper, we discuss how the different discussed algorithms are implemented and integrated with Yasper.

5.1 Yasper

Yasper [36, 16], “Yet Another Smart Process Editor”, is a Petri net editor, used and developed at the AIS group of this university, to facilitate research activities. The project started in 2003, as a part of a graduation project[24], and is still in development. It does not only offer easy editing facilities, it also provides analysis and simulation methods. It also supports the new Petri net exchange format, PNML[11, 21, 7, 42]. The code base of Yasper is written in VB.NET and is re-used in various other research projects.

In Yasper, Petri nets are extended with hierarchy, reset arcs and inhibitor arcs. Further, it offers special support for workflow design.

Yasper currently consists of three different modes:

**Edit** Edit mode provides facilities to create, and design a Petri net. Some synthesis algorithms are implemented in this mode, such as the addition of self-loops on places. Further, the different reduction rules implemented are currently available in the edit mode.

**Run manually** In this mode, the user can select enabled transitions to fire. This mode helps to verify whether the model behaves as desired. It enables the user to quickly find and correct errors.
**Run automatically** In this mode, the user can perform automated simulation on Petri nets.

More information about the different modes can be found in [16]. The article of [16] can also been found in Appendix B.

When editing a Petri net model, one wants to be able to work on specific parts of the Petri net, e.g. to extend it with hierarchy, or to remove it. A Petri net consists of nodes (e.g. places, transitions, subnets) and arcs (also more specialized arcs). Figure 5.1 gives a simplified UML class model of this design.

![Diagram of Petri nets, selections and elements in a UML class diagram](image)

To work on subsets of elements in the Petri net, a selection is added to the design of Yasper. A Selection is basically a collection of elements in the same Petri net, on which one can perform certain actions. These actions can vary from informing the user about certain properties to special operations as reduction or synthesis.

### 5.2 Extended Composite Pattern

The *extended composite pattern* is a generalized pattern of the *composite pattern* described in [14]. The composite patterns gives a pattern to build a hierarchy of operations, and to access these. This pattern has an abstract component class, from which leaves and a composite are derived. It has as advantage that both leaves and composite components are treated in the same way. However, the composite pattern has a single composite: perform the action for all its children. Therefore we extended the pattern with different types of composites. Figure 5.2 depicts the structure of this extended pattern.

In the pattern the following objects participate:

**Client** The Client calls a component via the public interface of component.

**Component** The component defines the public interface of the objects. Furthermore, if needed all default, common behavior is implemented in this class.

**Leaf** A leaf defines the behavior of a basic component. A leaf does not have any children.
Composite  The composite defines the common behavior for components having children.

Disjunctive  The disjunctive defines that only one of its children is executed. On certain criteria, that can be set, a single child is picked and executed.

Conjunctive  The conjunctive defines that the operation is executed for all its children.

The main advantage of this pattern is that the client does not need to have information about each specific component, but the interface of the class component is enough. In this way clients and components can be developed independently, and be combined in many ways. Note that this pattern supports the addition of different kinds of composite classes.

5.3 Properties of Petri nets

Properties give information about the Petri net. They typically do not change the Petri net. Properties either hold or not. In case of the latter, often a counterexample can be given, which shows that the property does not hold.

We use the extended composite pattern for a design supporting properties. Figure 5.3 presents the design as an UML class model.

There is a general interface, called IPNMLProperty, with a single method holdsOn. This method calculates whether the property holds on the given selection of a Petri net. The class Property implements this interface, and can be used to inherit from.

Secondly, there is an interface for properties with a counterexample, named IPNMLLocalProperty. This interface inherits from IPNMLProperty and has an extra
Method to implement, named *CounterExample*. This method returns the counterexample of the property as a selection, or nothing if there is no counterexample. The class *LocalProperty* implements this method. The *holdsOn* method checks whether there is a counterexample. If the counterexample method finds no counterexample, it returns true.

Many properties can be composed out of different properties. To support this, three extra classes are introduced. First of all a container is defined to hold different properties. This class is called *CompositeProperty*, and can have member properties. Furthermore, two default classes, derived from *CompositeProperty* are implemented: a class to calculate the conjunction of the member properties, *ConjunctiveProperty*, and a class to calculate the disjunction of the member properties, called *DisjunctiveProperty*.

### 5.4 Transformations on Petri nets

A second class of operations on Petri nets are transformations: operations that change the Petri net. Transformations can be applied on selections, if the selection fulfills certain criteria, and can be combined in many ways. Therefore, we use the extended composite pattern to implement the different kinds of transformations. In our design, transformations are called modifiers. Figure 5.4 depicts the UML class diagram of the design.

To use a transformation, it needs to implement the interface *IPNMLModifier*, or inherit from the base class, *Modifier*. This interface has two methods, a method *applies*, which returns whether the modification on the selection is possible, and a method *apply*, which performs the modification on the selection. This apply method does not necessarily check whether it is applicable, due to performance.

We defined three leaves for modification: a normal modifier, a conditional modifier, and a property modifier. A conditional modifier first calls a boolean function on the selection. If it returns true, it executes the given modifier. The property modifier
applies the modification until the property holds.

For the modifiers, we designed multiple possible composition classes, since the modifications can be done in more than two ways. We defined the following composite classes:

- **ConditionalOrModifier** This class is the implementation the disjunctive class in the pattern. If a member can be applied, it will be, and after the member is applied, the selection is returned and the algorithm stops. The first child possible is chosen.

- **RandomModifier** This class checks which member modifiers can be executed, and if there are, it randomly takes a single modifier and executes it.

- **SequentialModifier** This class implements the conjunctive class in the pattern. It executes a number of modifiers in a sequence on a net selection. Members that cannot be executed, are skipped.

- **FixedPointModifier** This class performs its member modifiers on the given selection, until the modifiers cannot be executed on the selection anymore.

- **UntilSameSelectionModifier** This class performs its member modifiers on a given selection, until the selection is stabilized. If no changes on the selection have been made, this composite stops executing.

### 5.5 Implementation of the check algorithms

The following algorithms are implemented as counterexample properties:

- **State Machine Check.** If a transition does not conform to the state machine property, it is added to the set of counterexamples.

- **Marked Graph Check.** If a place does not conform to the marked graph property, it is added to the set of counterexamples.
• Free Choice Check. If a place or transition does not conform to the free choice property, it is added to the set of counterexamples.

The workflow check is written as a conjunctive property of a check for single source and sink, and a check for the path property. The algorithms for source and sink are implemented as counterexample properties, although they return the set of source and sink places, respectively. The path property is implemented as a normal property.

The algorithm to check for ST-nets is implemented as a normal property, working on a copy of the original selection. The algorithm for the Batch Workflow Petri net check is implemented in the same way.

5.6 Desel/Esparza toolkit

The reduction rules on Free Choice Petri nets, are designed as a library for Yasper. In Figure 5.5 the UML Class Diagram of the implementation is given.

![UML Class Diagram of the Free Choice Reduction rules](image)

Figure 5.5: Class diagram of the Free Choice Reduction rules

The main class of the toolkit is the *WellFormedFreeChoiceReduction*, which reduces the given selection to the smallest net possible by executing *WellFormedFreeChoiceReductionSteps* until the selection does not change anymore. Therefore the *WellFormedFreeChoiceReduction* class inherits from the *UntilSameSelectionModifier*.

The *WellFormedFreeChoiceReductionStep* class has three children:

• *AbstractionRule*, which implements the algorithm for the abstraction rule. The *abstractionRule* is implemented as a *FixedPointModifier*, and executes
the abstractionRuleStep, until the selection does not change anymore. This abstractionRuleStep is implemented as a normal modifier, reducing a single place and transition.

- ReduceImplicitPlaces, which implements the algorithm for the positive linear dependent places rule. It calculates the incidence matrix of the given selection, and determines whether there is a positive linear dependent place, and reduces it.

- ReduceImplicitTransitions implements the algorithm for the positive linear dependent transitions rule. It is implemented in the same way as ReduceImplicitPlaces.

5.7 General reduction rules

In this section we show the design of the reduction rules on general Petri nets.

Murata reduction rules

The Murata reduction rules are implemented using the modifier classes. Each of the different rules has its own class, and is derived from the modifier class. Figure 5.6 shows the UML class diagram of the library. The class MurataRules is a class derived from the class UntilSameModifier, and has the following child modifiers:

- EliminationOfSelfLoopPlaces. This class is derived from the FixedPointModifier, and removes in each step the removal of a single self-loop place (by the class EliminationOfSelfLoopPlacesStep).

- EliminationOfSelfLoopTransitions. This class is the dual of the EliminationOfSelfLoopPlaces class.

- FusionOfParallelPlaces. This class implements the rule Fusion of Parallel Places.

- FusionOfParallelTransitions. This class implements the rule Fusion of Parallel Transitions.

- FusionOfSeriesPlaces. This class implements the rule Fusion of Series Places.

- FusionOfSeriesTransitions This class implements the rule Fusion of Series Transitions.
Berthelot reduction rules

From Berthelot, only the irrelevant transition rules are implemented, as they are special cases of the linear dependent transition rule. Figure 5.7 depicts the UML class diagram of the Berthelot reduction rules implemented. The class *BerthelotRules* is derived from the class *UntilSameSelection*, and executes the two rules until the selection does not change. It has the following child modifiers:

- **IdentityTransition**. This class is derived from *FixedPointModifier* and executes the class *IdentityTransitionStep* until it cannot be applied anymore. This latter class removes a single transition with identical preset and postset.

- **IdenticalTransition**. This class implements the Identical Transition reduction rule.
5.8 Workflow reduction rule

The workflow reduction rule is implemented according the modifier design. The main class in this design is the WorkflowReduceAll. The workflow reduction stops, if there are no sound Workflow Petri nets found in the Petri net. By inheriting the UntilSameSelection modifier, the WorkflowReduceAll class has this behavior. The WorkflowReduceAll is a composite and consists of the member WorkflowReduceStep. This class implements algorithm ?? in the method giveWorkflowOf(). If the class finds a subWorkflow Petri net, it is reduced by the class WorkflowReduction. This class first checks if the selection is an ST Workflow Petri net, and otherwise, the batch algorithm and an external tool[28] are used to determine generalized soundness. If the sub Workflow Petri net is generalized sound, it is reduced to a single place.

Figure 5.8: Class diagram of the Workflow reduction rule

The class StronglyConnected is used to calculate strongly connected components in a selection. It has two methods in the public interface:

- **giveStronglyConnectedComponent**: It calculates the strongly connected component in the given selection sel around the given node start.

- **findAllStronglyConnectedComponents**: This method calculates all strongly connected components in the selection, and returns them, all in a single selection.

The other methods are protected, and calculate the Depth First Search (DFS) both forward and backward. The difference between both methods is the direction of the arcs. In the backward methods, the nodes are visited in the opposite direction, further both implementations are implemented in the same way. The methods
visitDFSForward and visitDFSBackward return a stack with final nodes. Each node finished in the DFS is put on this stack. The parameter visitedNodes is a reference variable, and contains all nodes visited by the search.

5.9 Yasper and modifiers

Modifiers have an applies method to validate whether they can be executed or not. The menu structure in Yasper uses this to calculate whether a modifier can be shown to the user. By the general interface of modifiers, this menu structure is easy to extend with different modifiers. Most operations on Petri nets that were already implemented, are re-implemented in the modifier design. In this way, the modifier design improved the design and implementation of Yasper.

![Figure 5.9: Yasper showing a Workflow Petri net, and the reduction rules possible. The screenshot on the right shows the result](image)

5.10 Petri net properties

We implemented a prototype which incorporates all libraries, called Petri net properties, and shows, given a Petri net, the different properties and reductions of the Petri net. Figure 5.10 gives the user interface of the program.

This program implements the Workflow Petri net given in Chapter 1, to analyze soundness, boundedness and liveness. Given a Petri net, it either returns if the net is sound, live and bounded (or well-formed), or it returns the reduced net, according to the different reduction techniques applicable. If a property holds, it is colored green, if it does not, it is colored red. If it is not calculated, which is possible, the color is black.

If the given Petri net is a well-formed Free Choice Petri net, further checks have to be performed to check whether it is with the current marking live and bounded. For
Figure 5.10: Petri net Properties user interface

the other checks and reductions, well-formedness means live and bounded for the current marking. Note that the well-formedness property is only set if the reduced net is an atomic net. For all other cases, the reduced net has to be analyzed further to determine liveness and boundedness.
Chapter 6

Trace refinement

To show analysis results, often traces are used. A trace is a sequence of firings in a Petri net. With such a trace, one can visualize problems in a Petri net. In Chapter 3, methods are discussed to be able to reduce a Petri net into a smaller Petri net, preserving the properties to analyze. The problem now is that many results of the analysis cannot be shown in the original net, only in the reduced. Therefore expanding a trace in this smaller reduced net to a trace in the original net is desired. We call this trace refinement. With trace refinement, it becomes possible to analyze large Petri nets by reducing them, perform analysis on it, and transform the result traces back to the original Petri net.

We therefore introduce a mapping function on firing sequences. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Petri nets, then we define the mapping function $\alpha : \tilde{T}^* \rightarrow T^*$ as:

$$
\alpha(\epsilon) = \epsilon \\
\alpha(t\tau) = \alpha(t)\alpha(\tau)
$$

With this function, we can map firing sequences of the reduced net on firing sequences in the original net, giving the possibility to show analysis results to the user.

In this chapter we extend the rules of Desel and Esparza, to reduce also the marking, and then show that for each of the rules there is a mapping function of firing sequences. Note that most of Murata rules (all, except Elimination of Self-loop places and transitions) can be expressed in terms of Desel and Esparza’s rules.

The rules of Desel and Esparza, explained in section 3.2, can be applied on Free Choice Petri net, and preserve well-formedness. However, well-formedness only says that there is a marking for which it is live and bounded, and in many cases one wants to know if it is live and bounded for the current marking. We therefore extend the rules of Desel and Esparza with markings. Further, we show that these extended reduction rules preserve liveness and boundedness.
6.1 Extended abstraction rule

**Definition 61 (Extended abstraction rule).** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and \( m_0 \) a marking in \( N \) such that \( N \) is live and bounded, and \( \tilde{m}_0 \) a marking in \( \tilde{N} \). We say \( ((N, m_0), (\tilde{N}, \tilde{m}_0)) \in \phi_A \) iff \( (N, \tilde{N}) \in \phi_A \) and the marking \( \tilde{m}_0 \) of \( \tilde{N} \) satisfies:

\[
\forall p \in \tilde{S}: \tilde{m}_0(p) = \begin{cases} 
m_0(p) & p \notin t^* \\
m_0(p) + m_0(s) & p \in t^* \end{cases}
\]

We now define how we can refine a trace in the reduced net into a trace in the original net. For the abstraction rule, we define the function \( \alpha : \tilde{T}^* \rightarrow T^* \) as:

\[
\forall u \in \tilde{T}: \alpha(u) = \begin{cases} 
u & u \notin \bullet s \\
\nu u & u \in \bullet s \end{cases}
\]

where \( s \in S \) is the reduced place.

**Theorem 11.** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, such that \( (N, \tilde{N}) \in \phi_A \), and the reduction is performed on place \( s \in S \) and transition \( t \in T \), and let \( m_0 \) be a marking such that the system \( (N, m_0) \) is live and bounded. If the sequence \( \sigma \) is a trace in the net \( \tilde{N} \), then the trace \( \alpha(\sigma) \) is a trace in the original net \( N \).

**Proof.** We proof this lemma by induction on \( \sigma \).

Suppose \( \sigma = \epsilon \), thus the empty sequence. We have: \( \alpha(\epsilon) = \epsilon \). This is trivial also a trace of the net \( N \).

Suppose \( \sigma = u\sigma' \). We have: \( \alpha(\sigma) = \alpha(u\sigma') = \alpha(u)\alpha(\sigma') \). For \( u \in \tilde{T} \) we have two possibilities.

1. **Suppose** \( u \notin \bullet s \). Then \( \alpha(u) = u \). Since we have \( \tilde{T} = T \setminus \{t\} \) and \( u \in \tilde{T} \), we also have \( u \in T \). Since \( u \) is enabled and fires in \( \tilde{N} \), it is also enabled in \( N \) (the only place removed is \( s \), and \( s \) only enables \( t \)). Thus \( u \) can fire in \( N \). Since \( u \notin \bullet s \), firing the transition \( u \) in \( N \) gives the same marking as firing in \( \tilde{N} \). From the induction hypothesis we conclude that \( \alpha(\sigma) \) is a trace of the net \( N \).

2. **Suppose** \( u \in \bullet s \). The transition \( u \) can fire, since it is enabled in both \( \tilde{N} \) and \( N \). Then there are places in the postset of \( u \) that receive a token from \( u \) in \( \tilde{N} \), but not in \( N \). These places are exactly the places in which the transition \( t \) fires, since \( u^* = u_N^* \cup t^* \). Since \( s \) is marked, transition \( t \) can fire. After firing transition \( t \), we have a new marking in which the places from \( u_N^* \) and the places in \( t^* \) are marked. Thus after firing \( t \) after \( u \), we have the same marking as in the net \( \tilde{N} \). Thus \( ut \) is a valid sequence in \( N \). Using the induction hypothesis, we can conclude that \( \alpha(\sigma) \) is a trace of the net \( N \).
We now show that the extended abstraction rule preserves boundedness and liveness.

**Lemma 22.** The extended abstraction rule preserves boundedness and liveness for Free Choice Petri nets.

**Proof.** We prove this lemma by showing that this reduction rule is a special case of the Place Substitution rule of Berthelot (see Section 3.4.2). Let \( N = (S, T, F) \) be a Free Choice Petri net, and \( m \) a marking in \( N \). Let \( (\tilde{N}, \tilde{m}) \) be a system with \( TNN \), such that \((N, m), (\tilde{N}, \tilde{m}) \in \phi_{A^+}\).

Take \( H = s^* \), \( G = s^* = \{ t \} \), and \( Q = G^* = t^* \). Let \( P = (s^*)^* \), and \( P' = P \setminus \{ s \} \). Then, \( HS(H^*) = HS((s^*)^*) = HS(P) \). By definition, we have: \( HS(P) = HS(P' + [s]) = HS(P') + t^* \). Since \( F \cap (s \times t^*) = \emptyset \), we have \( HS(P') \cap t^* = \emptyset \), and thus \( HS(H) = H \), and for each \( h \in H, t^* \subseteq h^* \). The reduced system \((\tilde{N}, \tilde{m})\), with \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) and \( \tilde{m} \) a marking of \( \tilde{N} \), is:

\[
\begin{align*}
\tilde{m} &= HS_{[p/q]}(m) \\
\tilde{S} &= S \setminus \{ s \} = \tilde{S} \\
\tilde{T} &= T \setminus G \setminus H \cup HS(H) = T \setminus G = T \setminus \{ t \} = \tilde{T} \\
\tilde{F} &= (F \cap (\tilde{S} \times \tilde{T}) \cup (\tilde{T} \times \tilde{S}) \cup \left\{ \{ h \} \times t^* \right\}_{h \in H} \\
&= (F \cap (\tilde{S} \times \tilde{T}) \cup (\tilde{T} \times \tilde{S}) \cup (s \times t^*)) = \tilde{F}
\end{align*}
\]

We now show that \( \tilde{m} = \tilde{m} \). We have:

\[
\forall p \in \tilde{S} : \tilde{m}(p) = \begin{cases} 
  m(p) + m(s) & \text{if } p \in \tilde{T}^* \\
  m(p) & \text{otherwise}
\end{cases} = \tilde{m}(p)
\]

We thus have \((\tilde{N}, \tilde{m}) = (\tilde{N}, \tilde{m})\), and therefore, the extended abstraction rule is a special case of the Place Substitution rule. Since this rule preserves both boundedness and liveness, the extended abstraction rule also preserves boundedness and liveness.

We also show that if a Free Choice Petri net is not live, the reduced net is not live, and that if a Free Choice Petri net is not bounded, the reduced net is also not bounded.

**Lemma 23.** Let \( N = (S, T, F') \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and \( m \) and \( \tilde{m} \) markings, such that \((N, m), (\tilde{N}, \tilde{m}) \in \phi_{A^+}\). If \((\tilde{N}, \tilde{m})\) is live, then also \((N, m)\) is live. If \((\tilde{N}, \tilde{m})\) is bounded, then \((N, m)\) is bounded.

**Proof.** By Lemma 22, we have that the extended abstraction rule is a special case of the place substitution rule of Berthelot. Hence if \((\tilde{N}, \tilde{m})\) is live, then also \((N, m)\) is live, and if \((\tilde{N}, \tilde{m})\) is bounded, then \((N, m)\) is bounded.
6.2 Extended transition rule

In this section we extend the linear dependent transition rule with marking. The rule only removes transitions, it does not change the number of places in the net, nor its marking.

Definition 62 (extended transition rule). Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and \( m \) a marking in \( N \) such that \( N \) is live and bounded. The relation \( ((N, m), (\tilde{N}, m)) \in \phi_{T^+} \) exists iff \( (N, \tilde{N}) \in \phi_T \).

We prove that the extended transition rule preserves liveness and boundedness, and that any firing sequence in the reduced net is a firing sequence in the original net.

Lemma 24. Let \( N = (S, T, F) \) be a well-formed Free Choice Petri net. Let \( t \in T \) be a positive linear dependent transition. If \( t \) is enabled, there is a transition \( u \in T, u \neq t \) also enabled.

Proof. Since \( t \) is a positive linear dependent transition, there are transitions \( t_1, \ldots, t_n \in T \) and \( \alpha_1, \ldots, \alpha_n \in \mathbb{Q}^+ \), such that \( t = \sum_{i=1}^{n} (\alpha_i \cdot t_i) \). As these transitions have the same result as \( t \), we know there is at least one transition \( u \in \{t_i | 1 \leq i \leq n\} \) and place \( s \in S \), such that \( s \in *u \cap *t \). With the free choice property, we have \( *u = *t \). Thus \( *u = *t \leq m_1 \), thus \( u \) is also enabled. \( \square \)

Theorem 12. Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and \( m \) a marking in \( N \) such that \( N \) is live and bounded, and \( ((N, m), (\tilde{N}, m)) \in \phi_{T^+} \). The system \( (\tilde{N}, m) \) is also live and bounded. Any firing sequence \( \sigma \) in \( \tilde{N} \) is a firing sequence in \( N \).

Proof. We prove this lemma by proving that in any reachable marking \( m' \in [m] > \), there is at least one transition enabled. Let transition \( t \in T \setminus \tilde{T} \) be the linear dependent transition removed by the reduction. The proof is done by induction on the firing sequence.

Since \( (N, m) \) is live and bounded, there is at least one transition enabled. Let \( u \in \tilde{T} \) be enabled in \( (N, m) \), and \( u \neq t \), which is possible by lemma 24. The firing sequence \( \sigma = u \) leads in both nets to the same marking \( m' \), and \( \sigma \in \tilde{T}^* \).

Let \( \sigma = \sigma'u \), with \( \sigma' \in \tilde{T}^* \). By induction, we reach marking \( m' \) in both \( N \) and \( \tilde{N} \). As \( (N, m) \) is live, there is at least one transition enabled in \( N \), and if \( t \) is enabled, there is at least a second transition enabled. Thus there is a \( u \neq t \) enabled in both \( N \) and \( \tilde{N} \). Thus \( \sigma = \sigma'u \) is a firing sequence in \( N \), with \( \sigma \in \tilde{T}^* \). \( \square \)

6.3 Extended place rule

The rule for positive linear dependent places removes a place under certain conditions. The rule preserves well-formedness, but, reduces regardless the marking.
Figure 6.1 shows an example of a Free Choice Petri net that is well-formed, but is not live and bounded with the initial marking. The rule for positive linear dependent places still applies, and removes one of the two places left after two reduction steps. We therefore extend the rule with marking.

Figure 6.1: A Free Choice Petri net that is well-formed, but not for this marking. The reduced net is after applying $\phi_{A+}$

The problem of removing a place $s$ depends on the input and output transitions. If there is an input transition, that is also an output transition with respect to place $s$, there is a possibility that $s$ needs to be marked, in order to let the system be live and bounded. Then, by lemmas 5 and 6, there are two possibilities, either $s$ has to be both a trap and a siphon, and then it needs to be marked, or it is neither a trap nor a siphon, and it needs not to be marked initially. However, if $s$ is either a trap and not a siphon, or vice versa, the system cannot be live and bounded. This is expressed in the conditions of the extended place rule.

Definition 63 (extended place rule). Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Free Choice Petri nets, and $m$ a marking in $N$ such that $N$ is live and bounded, and $\tilde{m}$ a marking in $\tilde{N}$. We say $((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+}$ iff there is a place $s \in S$, and:

1. $|S| > 1$;
2. $s$ is a positive linear dependent place;
3. $s \cup s^* \neq \emptyset$;
4. if $s^* \subseteq s^*$, or $s^* \subseteq s^*$, then $s^* = s^*$ and $m(s) > 0$.

and the system $(\tilde{N}, \tilde{m})$ satisfies:

1. $\tilde{S} = S \setminus \{s\}$;
2. $\tilde{T} = T$;
3. $\tilde{F} = F \setminus ((s^* \times \{s\}) \cup (\{s\} \times s^*))$;
4. $\forall p \in \tilde{S} : \tilde{m}(p) = m(p)$. 

61
Lemma 25. Let $N = (S, T, F)$ be a Free Choice Petri net, $s \in S$, such that there exists a vector $\Lambda : S \rightarrow \mathbb{Q}^+$, with $\Lambda(s) = 0$ and $\Lambda \cdot N = \overline{S}$. Then there exists a $p \in S, p \neq s$, such that $p^* = s^*$.

Proof. Since $\Lambda(s) = 0$, and $\Lambda \cdot N = \overline{S}$, there exists a place $p \in S$, with $\Lambda(p) > 0$. Thus there is a transition $t \in s^*$, such that also $t \in p^*$. By the free choice property, this implies $s^* = p^*$.

Lemma 26. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Free Choice Petri net, and $m$ a marking in $N$, $\tilde{m}$ a marking in $\tilde{N}$, such that $((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+}$. Any firing sequence $\sigma$ in $(N, m)$ is a firing sequence in $(\tilde{N}, \tilde{m})$.

Proof. Let $s$ be the positive linear dependent place that is removed in the reduction. We proof this lemma by induction on the length of $\sigma$. Let $\sigma = \epsilon$. Then $m \xrightarrow{\epsilon} m$, and $\tilde{m} \xrightarrow{\epsilon} \tilde{m}$, thus $\sigma$ is a firing sequence of both nets.

Let $\sigma = \sigma't$. By induction, we have $m \xrightarrow{\sigma'} m_1$ and $\tilde{m} \xrightarrow{\sigma'} \tilde{m}_1$, such that $\bullet t \leq m_1$. Thus $t$ is enabled in $m_1$. Suppose $s \in \bullet t$, then it is not in $\tilde{N}$. Since $s$ is a positive linear dependent place, there is a $\Lambda$, such that $\Lambda(s) = 0$ and $\Lambda > 0$. By lemma 25, there is a place $p \in \bullet t$, which is marked in $m_1$, otherwise $t$ cannot fire. Thus removal of $s$ does not influence the firing of $t$, and $\bullet t \leq \tilde{m}_1$. Suppose $s \notin \bullet t$, then also $\bullet t \leq \tilde{m}_1$. Hence $t$ can fire.

Lemma 27. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Free Choice Petri net, and $m$ a marking in $N$, $\tilde{m}$ a marking in $\tilde{N}$, such that $((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+}$. Any firing sequence $\sigma$ in $\tilde{N}$ is a firing sequence in $N$.

Proof. Let $s$ be the positive linear dependent place that is removed in the reduction. We proof this lemma by induction on the length of $\sigma$. Let $\sigma = \epsilon$. Then $\tilde{m} \xrightarrow{\epsilon} \tilde{m}$, and $m \xrightarrow{\epsilon} m$. Thus $\sigma$ is a firing sequence of both nets.

Let $\sigma = \sigma't$. By the induction hypothesis, we have $\tilde{m} \xrightarrow{\sigma'} \tilde{m}_1$, and $m \xrightarrow{\sigma'} m_1$. We have $\bullet t \leq \tilde{m}_1$. Suppose $s \notin \bullet t$. Then also $\bullet t \leq m_1$, and $t$ is enabled in $m_1$. Suppose $s \in \bullet t$. Since there exists a $\Lambda : S \rightarrow \mathbb{Q}^+$, with $\Lambda(s) = 0$, $\Lambda \cdot N = \overline{S}$, and $\Lambda(p) > 0$, $s$ is also marked if $p$ is marked. Thus $\bullet t \leq m_1$, and $t$ is enabled. Thus $t$ can fire.

Lemma 28. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Free Choice Petri nets, and let $m$ and $\tilde{m}$ be markings, such that $((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+}$. If $(\tilde{N}, \tilde{m})$ is bounded, then $(N, m)$ is also bounded.

Proof. Let $N = (S, T, F)$ and $\tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F})$ be two Free Choice Petri nets, and let $m$ and $\tilde{m}$ be markings, such that $((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+}$. Let $s \in S$ be the place removed by the reduction rule. Suppose $(\tilde{N}, \tilde{m})$ is live and bounded.

By Lemma 2, there is a $k \in \mathbb{N}$, such that for all places $p \in S$ and markings $m'$ reachable from $m$, we have $m(p) \leq k$. There exists a vector $\Lambda$, such that $\Lambda(s) = 0$. 62
and \( \Lambda \cdot N = s' \). Thus \( s \) is the sum of \( n \) places: \( s = \sum_{i=1}^{n} \Lambda(s_i) \cdot s_i \). As each place is bounded, we have for any reachable marking \( m' : m'(s) = \sum_{i=1}^{n} \Lambda(s_i) \cdot m(s_i) \leq |S| \cdot k \). Thus \( s \) is bounded. As \( S = S \cup \{ s \} \), \( S \) is also bounded. 

**Lemma 29.** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and let \( m \) and \( \tilde{m} \) be markings, such that \( ((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+} \). If \( (\tilde{N}, \tilde{m}) \) is live, then \( (N, m) \) is also live.

**Proof.** By Lemmas 26 and 27, we automatically have that if \( (\tilde{N}, \tilde{m}) \) is live, \( (N, m) \) is also live. □

**Lemma 30.** The extended place rule preserves liveness and boundedness.

**Proof.** Let \( N = (S, T, F) \) and \( \tilde{N} = (\tilde{S}, \tilde{T}, \tilde{F}) \) be two Free Choice Petri nets, and \( m \) a marking in \( N \), \( \tilde{m} \) a marking in \( \tilde{N} \), such that \( ((N, m), (\tilde{N}, \tilde{m})) \in \phi_{S^+} \). The system \( (N, m) \) is live and bounded. As any firing sequence in \( N \) is a firing sequence in \( \tilde{N} \) and any firing sequence in \( \tilde{N} \) is a firing sequence in \( N \), \( (\tilde{N}, \tilde{m}) \) is also live and bounded. □
Chapter 7

Case study: ProRail

At LaQuSo we did a case study for ProRail, a Dutch company maintaining the railways and its control. ProRail is designing a new system to control railway tracks. This design is made in UML activity Diagrams, and we were asked to check it for correctness. This assignment was done in two stages. In April we made a quick scan, transforming the model into Petri nets, and in May and June we validated the model in more detail. In this chapter, the quick scan is discussed, which made use of parts of the theory and implementation of this thesis.

ProRail provided a UML model consisting of a State Diagram and for each state in the state diagram an Activity Diagram. They further combined the Activity Diagrams into an overall Activity Diagram. In the quick scan we checked on two properties:

- The Activity diagrams of the different states do not violate the state diagram, e.g., all transitions in State diagram are also present in the Activity Diagram and vice versa.
- It is not possible that we can be in multiple states of the State Diagram at the same time.

7.1 Translation to Petri nets

We translated the different diagrams into a single Petri net. We started with the State Diagram. Each state was translated in an XOR transition\(^1\), and each transition between states as a a place. Note that this transformation is slightly different from the traditional transformations, but by transforming in this way, it is easy to refine the different states via hierarchy.

\(^1\)An XOR transition can be seen as a shorthand for a transition for each combination of input and output places
Activities in the different diagrams where either interruptible or not. Each activity was therefore modeled as a start transition, a place, and a termination transition (See Figure 7.2). If the activity is interruptable, an extra transition was added (see Figure 7.3), and the tokens in it, before it, and after it, are emptied by reset arcs.

The transformation resulted in a subnet for each of the different states.
7.2 Analysis results

The first check, whether the activity diagrams violated the state diagram, was checked while the Petri net was constructed by checking the input and output connections of the state and the activity diagram. Activities can only be followed by activities either in the same state, or in a state that can be reached by a single state transition. We discovered three errors, two state transitions that existed in the Activity Diagrams, but not in the State Diagram and one vice versa.

To do the second check, the resulting Petri net was too large to do it with model checking or analysis tools like mCRL2 or INA. The Petri net contained 232 places and 234 transitions. We therefore applied the reduction techniques available: the abstraction rule of Desel and Esparza, and the workflow reduction rule. We were able to reduce each of the subnets into a fraction of the original net. The reduced Petri net contained 45 places and 47 transitions.

On this reduced nets we did a reachability analysis, and it turned out to be that there were two real deadlocks in the system. Further, it was possible to reach two or more states of the state diagram at the same time, thus violating the state diagram. We discovered that the subnets did not behave as proper XOR transitions, but produce multiple tokens, and thus enabling different states in the same time.
Chapter 8

Petriweb

Petriweb[12, 15] is an online repository for Petri nets. Besides the possibility to share Petri nets, it also offers properties to search and categorize Petri nets. These properties cannot only be filled in by hand, but also be calculated automatically. Properties can be of arbitrary type. Petriweb currently uses this by offering search support on boolean, integer, enumeration and string types. Petriweb also contains an animator that, given a Petri net and a trace, perform a stepwise animation of the sequence in the net.

The present work can be used to extend Petriweb in various ways. The check algorithms can be added to Petriweb as boolean properties. By offering the algorithms as web services, the algorithms can be used in Petriweb without changing the Petriweb software.

Trace refinement can be added to Petriweb as properties that produce traces in the format the animator accepts. It would be a small change to the Petriweb software to automatically start the animator on such properties.

The reduction algorithms can be incorporated that, given a Petri net produce a sequence of nets. By generalizing the animator, we can support the replay of arbitrary sequences of nets. In this way we can show how reductions are performed.

The present work is currently applied and used in Yasper. With small changes we can easily add this work to Petriweb. In this way we can offer our work to the Petri net community, without requiring them to install Yasper. Also the extensions serve the goals of Petriweb, by enhancing the use of properties, and improving the quality of Petriweb, without large changes in the Petriweb software.
Chapter 9

Conclusions

In this thesis, we discussed the use of reduction techniques to analyze Petri nets for well-formedness and soundness. We presented for different subclasses of Petri nets check and transformation algorithms, and implemented these as libraries in Yasper. These algorithms are used in a procedure to analyze a Petri net for well-formedness and soundness. We applied this procedure in a case study for ProRail.

Further, we presented different reduction techniques on subclasses of Petri nets that preserve the properties of well-formedness and soundness. For State Machine Petri net, Marked Graph Petri net, and Free Choice Petri net we presented the reduction rules of Desel and Esparza, and made an implementation directly usable in Yasper. Further, we showed an extension of this set of rules that take the marking into account, preserving both liveness and boundedness for the given marking, together with a method for trace refinement. Completeness of the extended set of rules remain to be proven.

We introduced a reduction rule for subWorkflow Petri nets in general Petri nets, and showed that the order in which this reduction rule is performed, does not matter. This rule is also implemented as a library for Yasper.

As a prototype, the analysis procedure given in the introduction, is partly implemented in a stand-alone tool called Petri net Properties, which is based on the code base of Yasper. The implemented algorithms still have to be integrated in Yasper, within a new analysis mode. In this way, the presented procedure can directly be used during the design of the Petri net. To show errors directly to the modeler via trace refinement still has to be done for other classes than Free Choice Petri nets.

Petriweb can easily be extended with the present work to enhance both the usability of the different algorithms and reduction rules, and the use of Petriweb itself.

If a Petri net is reduced with the workflow abstraction rule, and it contains an error, we know the error is not in the reduced sub Workflow Petri nets, since these subnets were error-free. To be able to show the error in the original Petri net, the
trace has to be refined. As soon as a place representing a sub Workflow Petri net is marked, a trace through the sub Workflow Petri net is needed. Currently, there is no algorithm for computing a firing sequence from $i$ to $f$ of a Workflow Petri net. However, model checkers have this functionality, and can be used to find a sequence through the sub Workflow Petri net. It is future work to find an algorithm, that calculates, given a generalized sound Workflow Petri net, the shortest path through it.
Appendix A

Algorithms

Check algorithms

State Machine Petri net check

**Algorithm A.1**: State Machine check

<table>
<thead>
<tr>
<th>Input: $N = (S, T, F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: True if $N$ is a state machine, False otherwise</td>
</tr>
</tbody>
</table>

```plaintext
foreach $t \in T$ do
  if $|\cdot t| > 1 \lor |t\cdot| > 1$ then
    return False;
  end
end
return True;
```

Marked Graph Petri net check

**Algorithm A.2**: Marked Graph check

<table>
<thead>
<tr>
<th>Input: $N = (S, T, F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: True if $N$ is a marked graph, False otherwise</td>
</tr>
</tbody>
</table>

```plaintext
foreach $s \in S$ do
  if $|\cdot s| > 1 \lor |s\cdot| > 1$ then
    return False;
  end
end
return True;
```
Free Choice Petri net check

**Algorithm A.3**: Free Choice check

| Input: \( N = (S, T, F) \) |
| Output: True if \( N \) is a Free Choice Petri net, False otherwise |

\[
\text{foreach } t \in T \text{ do} \\
\quad \text{foreach } s \in \bullet t \text{ do} \\
\qquad \text{foreach } u \in s^* \text{ do} \\
\qquad\quad \text{foreach } r \in \bullet u \text{ do} \\
\qquad\quad\quad \text{if } r \notin \bullet t \text{ then} \\
\qquad\quad\quad\quad \text{return False;} \\
\qquad\quad\quad \text{end} \\
\qquad\quad \text{end} \\
\quad \text{end} \\
\text{return True;} 
\]

Atomic net check

**Algorithm A.4**: Atomic net check

| Input: \( N = (S, T, F) \) |
| Output: True if \( N \) is an atomic net, false otherwise |

\[
\text{if } |T| = 1 \land |S| = 1 \text{ then} \\
\quad \{t\} := T; \\
\quad \{s\} := S; \\
\quad \text{if } \bullet t = \{s\} \land t^* = \{s\} \text{ then} \\
\quad\quad \text{return True;} \\
\quad\quad \text{else} \\
\quad\quad\quad \text{return False;} \\
\quad\quad \text{end} \\
\text{else} \\
\quad \text{return False;} \\
\text{end} 
\]
Workflow Petri net check

Algorithm A.5: Initial Places

- **Input**: \( N = (S, T, F) \)
- **Output**: All places with an empty preset

\[
C := \emptyset; \\
\text{foreach } s \in S \text{ do} \\
\quad \text{if } ^*s = \emptyset \text{ then} \\
\quad \quad C := C \cup \{s\}; \\
\text{end} \\
\text{end} \\
\text{return } C;
\]

The algorithm for checking whether there is a single final place can be obtained by replacing the preset for the postset. Next the path property needs to be calculated. We use for this a standard Breadth First Search (BFS) from [8], pages 531–537. The algorithm used here for BFS returns the mapping of colors on the nodes, \((S \cup T) \rightarrow \{\text{white, gray, black}\}\). A black colored node means the node is visited. The BFS is used twice, once to visit all nodes reachable from \(i\), by \(\text{DoBFSForward}\), and once to visit all nodes that can reach \(f\), by visiting all nodes backwards from \(f\), by \(\text{DoBFSBackward}\).

Algorithm A.6: PathProperty

- **Input**: \( N = (S, T, F) \), \( i, f \)
- **Output**: True if all nodes lie on a path from \(i\) to \(f\), False otherwise

\[
\text{ColorsForward} := \text{DoBFSForward}(N, i); \\
\text{ColorsBackward} := \text{DoBFSBackward}(N, f); \\
\text{foreach } n \in (S \cup T) \text{ do} \\
\quad \text{if } \neg(\text{ColorsForward}(n) = \text{black} \land \text{ColorsBackward}(n) = \text{black}) \text{ then} \\
\quad \quad \text{return False}; \\
\quad \text{end} \\
\text{end} \\
\text{return True};
\]
Algorithm A.7: Workflow check

**Input:** $N = (S, T, F)$

**Output:** True if $N$ is a Workflow Petri net, False otherwise

1. $cIs := \text{InitialPlaces}(N)$;
2. $cFs := \text{FinalPlaces}(N)$;
3. if $|cIs| = 1$ then
   1. $\{i\} := cIs$
   else
   1. return False
   end
4. if $|cFs| = 1$ then
   1. $\{f\} := cFs$
   else
   1. return False
   end
5. return $\text{PathProperty}(N, i, f)$;

ST-net check

Algorithm A.8: CheckST

**Input:** Workflow Petri net $N = (S, T, F)$

**Output:** True if $N$ is an ST-net, False otherwise

1. $\Delta f := \text{CompDistEnd}(N)$;
2. $\Delta i := \text{CompDistInit}(N)$;
3. $X := (S \cup T) \setminus \{f\}$;
4. while $X \neq \emptyset$ do
   1. $x := \text{pick}(X)$;
   2. $M := \text{FindFactor}(N, x, \Delta f, \Delta i)$;
   3. if $(M \neq S \cup T) \land \text{CheckST}(M)$ then
      1. return $\text{CheckST}(\text{Quotient}(N, M))$;
   else
   1. $X := X \setminus \{x\}$;
   end
5. return $\text{CheckSMMG}(N)$;

In this algorithm, the function $\text{CompDistEnd}$ and $\text{CompDistInit}$ calculate the shortest distance from each node to the sink place $f$ of the Workflow Petri net $N$, and the distance to the source place $i$, respectively. Both functions return a mapping $N \rightarrow N$. The $\text{pick}$ operation picks a single element from the given set. The function $\text{FindFactor}$ returns the smallest subWorkflow Petri net in which node $x$ appears. As a single node cannot be a Workflow Petri net, the $\text{FindFactor}$ will always give a larger subnet. If it returns the complete net $N$, it terminates directly, otherwise the quotient of $N$ and $M$ is used. As $M$ is always bigger than a single place, the algorithm is recursively called on a strict smaller net.
The function *Quotient* calculates the quotient of the net $N$ with factor $M$. The function *CheckSMMG* returns whether the given Petri net is a State Machine Workflow Petri net or an acyclic Marked Graph Workflow Petri net.

<table>
<thead>
<tr>
<th>Algorithm A.9: Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Workflow Petri net $N = (S, T, F)$, a factor $M = (S_M, T_M, F_M)$, $i$ the source of $M$, and $f$ the sink of $M$</td>
</tr>
<tr>
<td><strong>Output</strong>: The quotient $L$, such that $N = L \otimes_n M$</td>
</tr>
<tr>
<td>if $\text{type}(i) = \text{place}$ then</td>
</tr>
<tr>
<td>$p = \text{new place};$</td>
</tr>
<tr>
<td>$S_L := (S \setminus S_M) \cup {p};$</td>
</tr>
<tr>
<td>$T_L := (T \setminus T_M);$</td>
</tr>
<tr>
<td>$F_L := (F \setminus F_M) \cup (\cdot_M^{-1} \times {p}) \cup ({p} \times f_M^*);$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$t = \text{new transition};$</td>
</tr>
<tr>
<td>$S_L := (S \setminus S_M);$</td>
</tr>
<tr>
<td>$T_L := (T \setminus T_M) \cup {t};$</td>
</tr>
<tr>
<td>$F_L := (F \setminus F_M) \cup (\cdot_M^{-1} \times {t}) \cup ({t} \times f_M^*);$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return $L = (S_L, T_L, F_L);$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm A.10: CheckSMMG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Petri net $PNN$, source node $i \in (S \cup T)$, target node $f \in (S \cup T)$</td>
</tr>
<tr>
<td><strong>Output</strong>: True if $N$ is a State Machine Workflow Petri net or an acyclic Marked Graph Workflow Petri net</td>
</tr>
<tr>
<td>if $\text{type}(i) = \text{type}(f)$ then</td>
</tr>
<tr>
<td>if $\text{WorkflowCheck}(N)$ then</td>
</tr>
<tr>
<td>if $\text{type}(i) = \text{place}$ then</td>
</tr>
<tr>
<td>return $\text{StateMachineCheck}(N);$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $\text{AcyclicCheck}(N) \land \text{MarkedGraphCheck}(N);$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return False;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return False;</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
The given algorithm for getting a factor is asymmetric. We transform this algorithm in a symmetric algorithm. In this algorithm, the set $X_l$ holds all nodes in the factor. Nodes in the set $X_l$ are candidate source nodes, and their preset has to be examined. Nodes in the set $X_r$ are candidate sink nodes, and their postset has to be examined. The set $A$ contains the nodes in $X_l$ with the lowest distance, $\alpha$, to the source place $i$. The set $\Omega$ contains the nodes in $X_r$ with the lowest distance, $\omega$, to the sink place $f$.

If $|A| > 1$, there are multiple candidate source places, and thus we know that the source place is at least in the preset of $A$. If it is a singleton, it is a candidate source node, and we examine the other nodes in the set $X_l$. If $X_l = A$, and $|A| = 1$, we have found the source node. As long as this is not the case, we pick a node in $X_l$, based on the length of $A$. This node cannot be a source node and is thus put in $X$. The nodes in the preset, and not yet visited are put in the set $X_l$. The candidate sink places are treated in a similar way.
Algorithm A.12: FindFactor

**Input:** Workflow Petri net $N = (S, T, F)$, $x \in N$, $\Delta_i$ distances to source $f \in S$, $\Delta_f$ distances to sink $f \in S$

**Output:** the nodes of the factor in $N$, The initial place and final place

\[X := \{t\} \cup ^*t \cup t^*;\]
\[X_l := ^*t \cup t^*;\]
\[X_r := ^*t \cup t^*;\]
\[\alpha := \{-z \in X : \Delta_i(z)\};\]
\[A := \{z \in X|\Delta_i(z) = \alpha\};\]
\[\Omega := \{z \in X|\Delta_f(z) = \omega\};\]

while $(X_l \neq A \lor |A| \neq 1) \lor (X_r \neq \Omega \lor |\Omega| \neq 1)$ do

if $X_l \neq A \lor |A| \neq 1$ then

if $|A| = 1$ then
\[y := \text{pick}(X_l \setminus A);\]
else
\[y := \text{pick}(A);\]
end
\[X_l := X_l \setminus \{y\} \cup (^*y) \setminus X;\]
\[X := X \cup ^*y;\]
\[X_r := X_r \cup (^*y) \setminus X;\]
if $y \in A$ then
\[\alpha := \alpha - 1;\]
\[A := \emptyset;\]
end
\[A := A \cup \{z \in X_l|\Delta_i(z) = \alpha\};\]
end

else if $X_r \neq \Omega \lor |\Omega| \neq 1$ then

if $|\Omega| = 1$ then
\[y := \text{pick}(X_r \setminus \Omega);\]
else
\[y := \text{pick}(\Omega);\]
end
\[X_r := X_r \setminus \{y\} \cup (y^*) \setminus X;\]
\[X := X \cup y^*;\]
\[X_l := X_l \cup (y^*) \setminus X;\]
if $y \in \Omega$ then
\[\omega := \omega - 1;\]
\[\Omega := \emptyset;\]
end
\[\Omega := \Omega \cup \{z \in X_l|\Delta_f(z) = \omega\};\]
end

return $(X, A, \Omega)$;
Batch Workflow Petri net

<table>
<thead>
<tr>
<th>Algorithm A.13: Batch Workflow Petri net check</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Workflow Petri net $N = (S, T, F)$, source place $i$, sink place $f$</td>
</tr>
<tr>
<td><strong>Output:</strong> True if $N$ is a batch Workflow Petri net, false otherwise</td>
</tr>
<tr>
<td>if $\text{MaxSiphon}(S \setminus {i}) \neq \emptyset$ then</td>
</tr>
<tr>
<td>return False;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>if $\text{MaxTrap}(S \setminus {f}) \neq \emptyset$ then</td>
</tr>
<tr>
<td>return False;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return True;</td>
</tr>
</tbody>
</table>

In [?], the authors give an algorithm to transform a Workflow Petri net into a batch Workflow Petri net.

<table>
<thead>
<tr>
<th>Algorithm A.14: Transform $N$ into a batch Workflow Petri net</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Workflow Petri net $N = (S, T, F)$</td>
</tr>
<tr>
<td><strong>Output:</strong> If $N$ is a batch Workflow Petri net, $N$, otherwise the empty set</td>
</tr>
<tr>
<td>$X := \text{MaxSiphon}(S \setminus {i})$;</td>
</tr>
<tr>
<td>$S := S \setminus X$;</td>
</tr>
<tr>
<td>$T := T \setminus X^*$;</td>
</tr>
<tr>
<td>$F := F \cap ((S \times T) \cup (T \times S))$;</td>
</tr>
<tr>
<td>if Workflow check($N$) then</td>
</tr>
<tr>
<td>if $\text{MaxTrap}(P \setminus {f}) = \emptyset$ then</td>
</tr>
<tr>
<td>return $N$;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $\emptyset$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $\emptyset$;</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Reduction algorithms

Abstraction rule

<table>
<thead>
<tr>
<th>Algorithm A.15: Abstraction Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $N = (S, T, F)$</td>
</tr>
<tr>
<td><strong>Output:</strong> Reduced net $N$</td>
</tr>
<tr>
<td><strong>foreach</strong> $t \in T$ do</td>
</tr>
<tr>
<td>$s := \text{giveSinglePrePlace}(N,t)$;</td>
</tr>
<tr>
<td>if $s$ is defined then</td>
</tr>
<tr>
<td>$F := (F \setminus ({s} \times {t}) \cup ({t} \times {s}) \cup {s \times t})$;</td>
</tr>
<tr>
<td>$S := S \setminus {s}$;</td>
</tr>
<tr>
<td>$T := T \setminus {t}$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return $N$;</td>
</tr>
</tbody>
</table>

In this algorithm, we check for each transition $t \in T$ if it has a singleton preset, such that the conditions of the abstraction rule hold on the place $s \in S$ in the preset and the transition. This is calculated by the function $\text{giveSinglePrePlace}$. Assuming this function is correct, the algorithm above removes all arcs between the preset of $s$ and all arcs between $t$ and the postset of $t$, and the arc $(s, t)$. In this way, both $s$ and $t$ are not connected anymore, and can be safely removed. We thus have all arcs of the net $N$, without any arc connected to $s$ or $t$, in other words, $F := F \cap ((\hat{S} \times \hat{T}) \cup (\hat{T} \times \hat{S}))$.

The abstraction rule connects all transitions of the preset of $s$ with all places in the postset of transition $t$. Now $s$ and $t$ can be removed.

<table>
<thead>
<tr>
<th>Algorithm A.16: giveSinglePrePlace</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $N = (S, T, F), t \in T$</td>
</tr>
<tr>
<td><strong>Output:</strong> If $t$ satisfies the conditions of definition, the input place $s$</td>
</tr>
<tr>
<td>if $</td>
</tr>
<tr>
<td>${s} := t$;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return $\emptyset$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>if $</td>
</tr>
<tr>
<td>return $\emptyset$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td><strong>foreach</strong> $n \in \cdot s$ do</td>
</tr>
<tr>
<td>if $n^* \cap t^* \neq 0$ then</td>
</tr>
<tr>
<td>return $\emptyset$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return $s$;</td>
</tr>
</tbody>
</table>
This algorithm first checks if the preset of the given transition $t$ is a singleton, and if the postset is not empty. If the check fails, the algorithm stops and returns the empty set. This check is exactly condition 2. The next step is to check if the postset of this place $s$ is a singleton, and if its preset is not empty. If and only if this check fails, condition 1 does not hold. The last check performed is condition 3, which check fails as soon as there is a transition in the preset of $s$ that is connected to an element in the postset of $t$. If this last condition also holds, we know all three conditions hold, and thus that the abstraction rule is applicable.

Linear dependent places

**Algorithm A.17: Linear Dependent Places Rule**

<table>
<thead>
<tr>
<th>Input: $N = (S, T, F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Reduced net $N$, incidence matrix $N$</td>
</tr>
<tr>
<td>if $</td>
</tr>
<tr>
<td>$p := \text{Pick(giveDependentPlaces}(N))$;</td>
</tr>
<tr>
<td>if $p$ is defined then</td>
</tr>
<tr>
<td>$F := F \setminus (\ast p \times {p}) \cup ({p} \times p^*)$;</td>
</tr>
<tr>
<td>$S := S \setminus {p}$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return $N$;</td>
</tr>
</tbody>
</table>

As the above algorithm shows, the incidence matrix is calculated, and then from the set of linear dependent nodes, a node is picked and removed from the Petri net. This removal is done in two steps: first all arcs connected to $p$ in any direction are removed, and secondly the place $p$ itself is removed, as is defined by $N \setminus \{p\}$. 

82
Algorithm A.18: giveDependentPlaces

**Input:** incidence Matrix N

**Output:** The set of dependent places

\[ m := 0; \]
\[ R := \emptyset; \]
\[ C := \emptyset; \]
\[ (N, O) := \text{EchelonForm}(N); \]

\[ \text{while } m < \text{rows}(N) \text{ do} \]
\[ \text{\quad } j := 0; \]
\[ \text{\quad } n := \text{cols}(N); \]
\[ \text{\quad } \text{isNull} := \text{True}; \]
\[ \text{\quad } \text{while } j < n \land \text{isNull} \text{ do} \]
\[ \text{\quad \quad } \text{if } N(m, j) \neq 0 \text{ then} \]
\[ \text{\quad \quad \quad } \text{isNull} := \text{False}; \]
\[ \text{\quad \quad } j := j + 1; \]
\[ \text{\quad end} \]
\[ \text{\quad if } \text{isNull} \text{ then} \]
\[ \text{\quad \quad } C := C \cup \{p_m\} \]
\[ \text{\quad end} \]
\[ \text{\quad } m := m + 1; \]
\[ \text{end} \]

\[ \text{foreach } s \in C \text{ do} \]
\[ \text{\quad if } \text{IsPositiveLinearCombination}(s, O) \land (s \cup -s) \neq \emptyset \text{ then} \]
\[ \text{\quad \quad } R := R \cup \{s\}; \]
\[ \text{end} \]
\[ \text{end} \]

The set \( C \) denotes the candidates for being an implicit place. If the place is an implicit place, it is added to the set \( R \). In the algorithm, the incidence matrix is brought to echelon form. Then, each place \( p_m \) in the incidence matrix is visited, and checked if it is a candidate incidence place. A candidate place is a place which has only zeros in its row. Secondly, all candidates are checked if it is a positive linear combination, and if is not an isolated place, then it is an implicit place.

To calculate whether a place is a positive linear combination, we administrate the actions performed to calculate the echelon form of the incidence matrix in a second matrix \( O \). This second matrix is a square matrix of size \((|S| \times |S|)\). A row \( i \) is positive linear dependent if \( O(i, i) \) is positive, and the other elements of \( O(i, \cdot) \) are negative. If \( O(i, i) \) is negative, the row is multiplied with \(-1\). In the algorithm we check the formula:

\[ O(i, i) > 0 \land (\forall j \in (0..|S|) : O(i, j) < 0) \]
Algorithm A.19: IsPositiveLinearCombination

Input: Place $s$, Operations matrix $O$

allPositive := True;
nrPos := 0;

$j := 0$;
i := RowOf($s$);

if $O(i, i) < 0$ then
    multiplyRow($O$, $i$,
end

while $j < Cols(O) \land$ allPositive do
    allPositive := $O(i, j) \leq 0 \land$ allPositive;
    if $O(i, j) < 0$ then
        nrPos := nrPos + 1;
    end
    $j := j + 1$;
end

return allPositive \land O(i, i) > 0

In this algorithm, the function multiplyRow($O$, $i$, $-1$) multiplies row $i$ of the matrix $O$ with $-1$.

Fusion of series places

Algorithm A.20: Fusion Of Series Places

Input: $N = (S, T, F)$
Output: Reduced net $N$

foreach $t \in T$ do
  if $\bullet t = 1 \land |t^*| = 1$ then
    $\{s\} := ^* t$;
    if $|s^*| = 1 \land s \neq \emptyset \land t \not\in s$ then
      $S := S \setminus \{s\}$;
      $T := T \setminus \{t\}$;
      $F := (F \setminus ((s^* \times \{s\}) \cup \{(s, t)\} \cup \{(t^* \times s^*)\}) \cup (s^* \times t^*)$;
  end
end

return $N$;
Fusion of series transitions

**Algorithm A.21: Fusion Of Series Transitions**

**Input:** \( N = (S, T, F) \)

**Output:** Reduced net \( N \)

foreach \( s \in S \) do
  if \( |s| = 1 \land |s^*| = 1 \) then
    \( \{t\} := s^*; \)
    if \( |t^*| = 1 \land t^* \neq \emptyset \land s \not\in t^* \) then
      \( S := S \setminus \{s\}; \)
      \( T := T \setminus \{t\}; \)
      \( F := (F \setminus (s^* \times \{s\}) \cup \{(s, t)\}) \cup (\{t^*\} \times t^*) \cup (\{s^*\} \times t^*); \)
    end
  end
end
return \( N \);

Fusion of parallel places

**Algorithm A.22: Fusion Of Parallel Places**

**Input:** \( N = (S, T, F) \)

**Output:** Reduced net \( N \)

foreach \( t \in T \) do
  \( C := \emptyset; \)
  if \( |t^*| > 0 \) then
    foreach \( s \in t^* \) do
      if \( |s| = 1 \land |s^*| = 1 \) then
        \( C := C \cup \{s\}; \)
      end
    end
    if \( |C| > 1 \) then
      foreach \( s \in C \) do
        \( \{u\} := s^*; \)
        foreach \( r \in C \) do
          if \( \{u\} = r^* \) then
            \( S := S \setminus \{s\}; \)
            \( F := F \setminus (\{s^*\} \cup (\{s\} \times s^*)); \)
            \( C := C \setminus \{s\}; \)
          end
        end
      end
    end
  end
end
return \( N \);
Fusion of parallel transitions

<table>
<thead>
<tr>
<th>Algorithm A.23: FusionOfParallelTransitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ( N = (S, T, F) )</td>
</tr>
<tr>
<td><strong>Output:</strong> Reduced net ( N )</td>
</tr>
<tr>
<td>foreach ( s \in S ) do</td>
</tr>
<tr>
<td>( C := \emptyset );</td>
</tr>
<tr>
<td>if (</td>
</tr>
<tr>
<td>foreach ( t \in s^* ) do</td>
</tr>
<tr>
<td>if (</td>
</tr>
<tr>
<td>( C := C \cup { t }; )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>if (</td>
</tr>
<tr>
<td>foreach ( t \in C ) do</td>
</tr>
<tr>
<td>{ r } := t^*;</td>
</tr>
<tr>
<td>foreach ( u \in C ) do</td>
</tr>
<tr>
<td>if ( { r } = u^* ) then</td>
</tr>
<tr>
<td>( T := S \setminus { t }; )</td>
</tr>
<tr>
<td>( F := F \setminus ((t \times { t }) \cup ({ t } \times t^*)); )</td>
</tr>
<tr>
<td>( C := C \setminus { t }; )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return ( N );</td>
</tr>
</tbody>
</table>

Note that this algorithm is a generalization of this rule: if there are \( n \) parallel places, \( n - 1 \) places are removed.
Elimination of self-loop places

**Algorithm A.24: Elimination Self-loop Places**

SetLine Input: \( N = (S, T, F) \), marking \( m \) of \( N \)
Output: Reduced net \( N \)

foreach \( s \in S \) do
  if \( m(s) > 0 \land |\cdot s| = 1 \land \cdot s = s^* \) then
    \( \{t\} := \cdot s; \)
    \( S := S \setminus \{s\}; \)
    \( F := F \setminus \{(s, t), (t, s)\}; \)
  end
end
return \( N; \)

Elimination of self-loop transitions

**Algorithm A.25: Elimination Self-loop Transitions**

SetLine Input: \( N = (S, T, F) \)
Output: Reduced net \( N \)

foreach \( t \in T \) do
  if \( |\cdot t| = 1 \land \cdot t = t^* \) then
    \( \{s\} := \cdot t; \)
    \( T := T \setminus \{t\}; \)
    \( F := F \setminus \{(s, t), (t, s)\}; \)
  end
end
return \( N; \)

Identity transition rule

**Algorithm A.26: Identity transition rule**

Input: Petri net \( N = (S, T, F) \)
Output: reduced net \( N \)

foreach \( t \in T \) do
  if \( \cdot t = t^* \) then
    \( T := T \setminus \{t\}; \)
    \( F := F \setminus ((\cdot t \times \{t\}) \cup (\{t\} \times t^*)); \)
  end
end
Identical transition rule

**Algorithm A.27: identical transition rule**

Input: Petri net \( N = (S, T, F) \)
Output: reduced net \( N \)

\[ \text{visitedNodes} := \emptyset ; \]

\[
\text{foreach } t \in T \text{ do}
\]

\[
\text{foreach } s \in (\text{t} \cup \text{t}^*) \land \neg \text{visitedNodes}(s) \text{ do}
\]

\[
\text{foreach } u \in s^* \land u \neq t \text{ do}
\]

\[
\text{if } \text{u} = \text{t}^* \land \text{u}^* = \text{t}^* \text{ then}
\]

\[
T := T \setminus \{t\};
\]

\[
F := F \setminus ((\text{t} \times \{t\}) \cup (\{t\} \times t^*));
\]

\[
\text{end}
\]

\[
\text{visitedNodes} := \text{visitedNodes} \cup \{s\};
\]

\[
\text{end}
\]

In this algorithm, the set of visited nodes is kept, to ensure that nodes are not visited more than once.
Appendix B

Yasper: a tool for workflow modeling and analysis

Kees van Hee, Olivia Oanea, Reinier Post, Lou Somers, Jan Martijn van der Werf
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee, o.i.oanea, r.d.j.post, l.j.a.m.somers}@tue.nl,
j.m.e.m.v.d.werf@student.tue.nl

Abstract. This paper presents Yasper, a tool for modeling, analyzing and simulating workflow systems, based on Petri nets. Yasper puts Petri net modeling in the hands of business analysts and software architecture designers. They can specify systems in familiar terms (XOR choice, workflow, cases, roles, processing time and cost), and can directly run manual and automatic simulations on the resulting models to analyze correctness and performance. Yasper was designed to cooperate with other tools, such as Petri net analyzers, and off-the-shelf software for data (color) handling and forms handling.

B.1 Introduction

In dealing with processes performed or controlled by software, there is a great need for good modeling techniques to support process design and analysis at an abstract level. Many techniques have been developed for this purpose; industry predominantly uses diagram-based techniques. These are used in commercial tools for software design, business process design, workflow process management, and coordination software (middleware). Some techniques are covered by industry standards; for example, UML activity diagrams, BPMN, BPEL, and to some extent Event-driven Process Chains (EPCs).

Petri nets are formal and well-researched; at the same time, they are diagram-based and close to techniques popular in industry such as UML activ-

1The content of this appendix is identical to [16].
ity diagrams and EPCs. However, they are not optimally easy to use. For instance, when a step is a choice, having multiple possible outcomes, it must be modeled with multiple transitions; this is often counter-intuitive. Yasper offers fork and join constructs.

It is good practice in process modeling to first specify the process flow, including dependency on resources. This allows basic analysis of correctness and performance. Particularly useful are animation and simulation, since they quickly expose most correctness and performance issues, and are generally applicable, unlike formal analysis techniques. Yasper supports this first stage of modeling. It tries to make modeling and simulation as easy as possible.

For more detailed modeling, such as the use of data (token color), high-level Petri net tools can be used, such as CPN Tools [29], ExSpecT [1], or ReNeW [23].

Yasper’s simulation is specifically designed for workflow nets: Petri nets that take individual cases from a given initial point to a given final point. Business processes and workflows typically have this structure. The simulation report indicates completion of cases and offers standard performance indicators.

B.2 Features

In Yasper it is possible to model and animate arbitrary place-transition nets, and, on top of this, it has the following features:

**Hierarchy** allows nets to be distributed across subnets. The interface of a subnet is defined as in ExSpecT [1], by pins, which are references to places in the surrounding net.

**Choice (XOR)** Like many techniques and tools, such as UML activity diagrams, Yasper has a diamond symbol to express alternative execution paths.

**Roles** are users or resources that can be assigned to transitions. During animation, roles are displayed with the transitions they can execute, as is done in workflow systems. Simulation measures the utilization of roles.

**Inhibitor and reset arcs** They are useful to test for empty places or to empty places.

**Data stores** are places having always a single token. They are connected to transitions through biflow arcs and are used to represent databases.

**Time, cost, and probabilities** are present to support stochastic simulation. Transitions can either be timed or untimed. Timed transitions can be assigned a mean time and standard deviation, in which case processing time is determined randomly with a gamma distribution. Furthermore, a transition can have an execution cost, consisting of a fixed (setup) cost and a variable cost per execution time unit. Weights can be assigned to the output arcs of a choice to determine their chance of being chosen during simulation.

**Case identity, emitors and collectors** are used for manual and automatic simulation of individual cases. A case can be represented by more than one (case) tokens. Cases are handled independently, except that they may compete for the same resource.

Workflow cases start in emitors, are passed by transitions through
case sensitive places, and end in collectors.

Transition execution is case sensitive: the tokens consumed from case sensitive input places must have the same case identity, which is passed to tokens produced to case sensitive places.

A basic correctness criterion for workflow is proper termination (also called soundness [2]) — in Yasper terms, a workflow net is sound if the collector consumes the last token of a case.

**Automatic simulation** records proper termination of individual cases and gives statistics on the workflow execution, e.g. waiting time, cycle time, work time and the total processing cost.

The results are displayed aggregated (averaged) over the cases, and optionally per case. Further, the utilization (％ of time occupied) is shown for each role.

Figure B.1 is a screenshot of Yasper in editing mode that shows most of the above mentioned features. More details on the semantics can be found in [7, 36].

Figure B.2 shows the results of an automatic simulation run performed on the same net.

---

**B.3 Combining Yasper with other tools**

There are two ways in which Yasper can be used in combination with other tools.

First, Yasper uses the standard PNML format [6] which facilitates exchange with other Petri net tool formats. At present, we use filter programs to Petri net analyzers like LoLa [33], INA [32] and Woflan [39].

Secondly, most of Yasper’s libraries are reused in other tools (e.g. the business process modeling tool of Deloitte).
B.4 Conclusions and future work

In this paper we have presented Yasper, a tool designed to support easy modeling and simulation of workflow Petri nets.

For the future, we plan to extend Yasper with a workflow engine to make it support data stored in relational databases, and interface with other document management systems.

Another line of development is to provide graphical feedback from analysis tools, and incorporate analysis techniques based on structural transformations. Further, we plan to support local transformations in the editor and correctness by construction principles, where the editing operations guarantee that Petri nets always satisfy well-formedness criteria, such as being sound workflow nets.
Appendix C

Petriweb: A Repository for Petri Nets

R. Goud, K.M. van Hee, R.D.J. Post, J.M.E.M. van der Werf

This paper describes Petriweb, a web application for managing collections of Petri nets. When a collection of nets is large or has multiple users, it becomes difficult for users to survey the collection and to find specific nets. Petriweb addresses this issue by supporting arbitrary content-based filtering. Nets can be assigned properties with values of arbitrary types. Properties can be used in searching and are displayed in search results. Their values can be manually assigned by users or derived automatically by applying a tool. This allows server-side integration of Petri net analysis tools. Properties can also define translations to output formats, on which the user can invoke client-side viewers and analyzers. Petriweb supports communities: members submit nets and property definitions, community moderators approve them. The paper discusses Petriweb’s features and architecture, and how it relies on the proper application of a common document format for Petri nets, the Petri Net Markup Language (PNML).

C.1 Needs for Model Repositories

Many tools exist to support the modeling process. One tool may be better suited for designing models, another for analysis. Therefore, in a typical development or research environment, multiple tools are used in combination. This creates the need to work on the same models with different tools. This can be addressed by defining a standard file format that all tools can use.

Models need not only be shared by tools, but also by different users. For instance, a model may need to be reviewed by a colleague of the designer. We can send the

\footnote{The content of this appendix is identical to [15].}
model, and the colleague can open and use the model, but as soon as new versions appear, it becomes hard to make sure the right version is always used. Here, a shared location for the models is needed.

A shared collection of models also encourages users to reuse existing models or parts of them. This can be useful for different kinds of users. Designers, who employ modeling to describe and design systems, can use this to streamline the modeling process. Researchers and educators can build up collections of models used as illustrations, e.g. examples or counterexamples in proofs.

Most collections of models will be assembled in the context of a specific project, with a small group of participants. But collections can also be turned into company-wide or world-wide resources. In such cases, users will rarely be familiar with all the models in the collection, and collections can grow quite large.

Various solutions can be considered to support these needs. The simplest approach is to use a shared (web) location for models, but this does not offer any help in organizing them. An improvement is to use standard version control software, such as CVS or Subversion. However, these systems still require the users to be familiar with the organization of the material in terms of file names and directory structure. For larger collections, or an open-ended user base, this does not suffice. Additional facilities for searching and browsing will be required that employ knowledge of the model contents.

Specialized repository software is needed that combines general file management with domain specific knowledge.

A specific domain to support at our university is process modeling with Petri nets. Many examples in course material and exercises are reused over the years; they are often recreated from memory or from paper. It is attractive to make such examples available in a shared repository, accessible by both students and teachers. Here, the need for both browsing and searching facilities is evident. Since users recognize Petri nets by their graphical representation, browsing the collection can only be supported with a graphical browser. In larger collections, users also need to filter the collection based on properties of the content. For instance, users may want to find examples of Petri nets that are bounded and contain a deadlock.

Petriweb is a web application for maintaining repositories of Petri nets. A Petriweb installation can host many different collections, each with their own administrators and users. The software is portable; different Petriweb installations can be easily installed.
C.2 Petriweb

C.2.1 Model retrieval

Knowledge about the contents

The main focus of Petriweb is how to retrieve a specific Petri net in a collection. To be able to search for Petri nets by giving criteria, Petriweb needs to have knowledge about the Petri net.

An example query Petriweb needs to support: “Find the smallest net present that is live, unbounded and free-choice”. Another example, from a developer’s perspective: “Find a component with this interface and performing task I”. Queries must also include metadata, e.g. the author or original publication of the net. We see that Petriweb must support different kinds of properties: structural properties, e.g. the number of places; behavioral properties, such as boundedness or liveness; and metadata.

Properties

Giving search criteria in terms of properties is a powerful manner of retrieving the correct Petri net. Petriweb supports different kinds of properties on Petri nets. These are:

- Metadata
- Application characteristics
- Structural and behavioral properties
- Transformations

By allowing metadata as properties in Petriweb, the user uploading the net can specify related information about it, such as when it was created, by whom, etc. This kind of property can narrow search results.

Application characteristics describe what the model is about: in what domain, for what purposes, etc.

Simple structural properties can be determined by simple programs, while more complex structural and behavioral properties such as free choice, liveness, boundedness, can be determined with existing Petri net analysis tools. Many such tools can be called as filters that take a net as input and produce results as output. Petriweb can incorporate this through automated properties. These are not specified by the user while uploading the net, but instead, computed automatically by invoking an XSLT stylesheet or an external command. This allows non-trivial structural
and behavioral properties to be determined automatically; they can even be used in search criteria.

This mechanism can also be used to automate conversions from PNML to other file formats, e.g. the TPN format of Woflan [?]. Calling an analysis tool is often preceded by a transformation, but the results can also be presented to the user, e.g., as the input for a client-side tool. In supporting transformations, Petriweb becomes more than a repository: it functions as a mediator between different tools and formats.

**View Petri Nets**

Not only searching on properties is important, the graphical representation of a Petri net is also a great help in finding a specific Petri net. It is hard to describe a Petri net in words, such that others can find it; the graphical display of a net is much
easier to recognize. Therefore, search results do not only list the Petri nets’ names and properties, but also display them as diagrams. This allows a combination of searching and browsing.

C.2.2 Model management

Addition

Petriweb has been designed for public and private use. Petri nets can be shared within a community. Anyone can register at Petriweb and upload their own Petri nets. To allow this, a standard file format is used for uploading Petri nets: the Petri Net Markup Language (PNML). Before adding the Petri net, Petriweb first checks its syntax against a PNML syntax definition [40]. The net is then parsed and stored.

Approval

While anyone can be registered at Petriweb and upload Petri nets to it, nets go through a built-in approval mechanism. A net can be in three possible states: uploaded, approved, or deleted. After a user uploads a Petri net for a community, the community moderator receives a notification and can decide whether the Petri net is approved or denied. In this way, collections remain manageable.

Retrieval

Sharing the Petri nets also means that it is possible to download and use the Petri nets from Petriweb. Petriweb supports this in two ways, by allowing downloading the original uploaded file, or a file generated from the parsed information. In this way it is possible to serve as many tools as possible.

C.2.3 Users and communities

Petriweb is primarily intended to serve the public community by sharing Petri nets. Users may not want to share their Petri nets with the total public community, but only with a small group of users. Therefore, Petriweb features a built-in mechanism to support communities. Each registered user can request a community. After the Petriweb administrator has approved the request, the user becomes moderator of the private community and can invite users to join. In this way restricted project areas can be defined and used, with the full search and properties functionality of Petriweb.
C.2.4 Architecture

Storing knowledge

Petriweb is based on the data schema given in Fig. C.3. It consists of three parts:

- the structure and marking of the Petri net,
- the graphical information to display the Petri net, and
- the properties associated with the Petri net.

Petriweb supports component based models: definitions define the behavior and structure, and can be instantiated in other definitions. Each Petri net definition is either a transition definition or a (sub)net definition. For every transition and subnet, its definition and its instance are stored. Every instance is a part of a (sub)net. The relation between ProcessInstance and Subnet depicts the hierarchy of the Petri net. Places are also part of a (sub)net. A marking is stored in the MarkingInformation table. A Petri net can have a trace of markings (a firing sequence); this information is stored in the StateHistory table. Multiple markings and traces per net are supported. Arcs are not considered as separate objects in the Petri net, but instead, both transitions and subnets have connectable pins. These pins connect instances with places or other pins. Due to the separation of definition and
instance, a distinction is made between formal and actual pins. A formal pin is part of the definition, an actual pin is an instance of the formal pin. Graphical information is part of the data schema, although not all relations are drawn in the figure.

Properties

Properties in Petriweb are stored in the table **Properties**. Each property belongs to a specific category. The table **PropCategory** contains information about the different categories. The table **NetProperty** contains the values of properties of (sub)nets in the repository. The table contains some standard properties that should always be available for every Petri net, such as its name, the location of its file and whether the (sub)net is approved. For each property defined in Petriweb (thus an entry in the table **Properties**) a column is present in this table. Automated properties can be derived either by applying a stylesheet (XSLT) or by calling a tool on the command line. The output is parsed into the database.

Communities

To support communities, it is of importance to separate the different communities from each other, in such a way that there is no connection between them. An advantage of this approach is that only properties needed in the community are stored and shown. To support this, each community receives its own database to fill. In order to administer the different communities, the data schema of Fig. C.4 is used.

Implementation

The design of Petriweb is component based (Fig. C.5). It is divided into three main components: repository, viewer and checker. The repository uses the viewer and
checker. All components are designed and implemented as stand-alone applications. It was a goal to provide easily installation on different machines.\(^2\). The implementation in PHP with MySQL meets this goal.

### C.3 PNML: the Petri Net Markup Language

Petriweb was designed to be used with as many different tools as possible. Hence it relies on a standard file format, and Petriweb is based on the Petri Net Markup Language, PNML\(^3\)[11][40], a draft ISO standard interchange format for Petri nets. Many tools already support PNML, and it is well-described, which makes it easy to write translators for tools that do not support it. Petriweb employs such translations automatically by means of automated properties.

PNML defines a basic syntax for flat colorless Petri nets, and an extension mechanism, PNTDs\(^4\)[41], to define syntax for additional features. Some extensions, e.g. structured PNML to denote hierarchy, are part of the standard. The PNML supported by Petriweb is defined by a specific PNTD we call EPNML 1.1. It encompasses basic Petri nets, as defined in [41], and most structural nets.\(^1\) Further, it defines special arc and transition types.\(^4\) These are employed by some of our tools, such as the Yasper editor [13][17].

#### C.3.1 PNML Viewer

PNML offers a way to store graphical information in the net. Therefore, Petriweb contains a basic Petri net viewer, that displays the Petri net as a Portable Network Graphics (PNG) image. The viewer supports hierarchical Petri nets; the user can browse through the different levels of a Petri net by clicking in the images. The viewer is implemented as a stand-alone web application. It can also animate firing sequences. Traces have to be uploaded in a special-purpose file format.

#### C.3.2 PNML Checker

Petriweb ensures that only correct PNML files are uploaded, by running them through a checker. This checker validates the file against the PNTD of EPNML 1.1. A PNTD is a RELAX NG\(^7\) specification that extends the standard PNML definitions. The checker can also be used as a stand-alone application.

---

\(^1\)Petriweb has been tested on two platforms: Linux and Microsoft Windows.

\(^2\)Some extra restrictions on reference nodes apply.

\(^4\)The fully documented specification can be found in [42].
C.3.3 Issues with PNML

Despite the power and clarity of the PNML standard, interoperability problems remain. Generally they are due to the following issues:

**PNML is a moving target:** The PNML standard still changes in details. At present there are three versions to cope with: the current ISO draft, the currently published RELAX NG syntax definitions, and the original PNML as used by the Petri Net Kernel [22].

The differences between the first two are minor; for instance, in structured PNML, the ISO draft requires that all elements are included in a `page` element, which the syntax definitions do not.

The last two have major incompatibilities, e.g. in the handling of hierarchy. We have found that even new tools sometimes produce PNK-compatible PNML instead of standard-conformant PNML.

**Tools implement PNML incorrectly:** Sometimes, nets being uploaded fail to validate for petty technical reasons, e.g. a failure to use XML namespaces properly.

**PNML is open to interpretation:** Some details left open in the PNML specification may lead to practical interoperability issues. For instance, the unit of measurement of graphical coordinates is not specified.

**PNTDs are too liberal:** In order to maximize interoperability between tools, PNML extensions (PNTDs) should be very conservative:

- all PNML documents that conform to the base definition should also conform to the extended definition, whenever this makes sense
- all such documents should continue to have the same interpretation

The first rule is broken by the present ISO draft, which, for structured PNML, requires every element to be in a `page`. With this requirement, basic (flat) PNML and structured PNML become disjoint sets of documents. In order to support compatibility with tools for flat Petri nets, designers of hierarchical Petri net tools will now need to support both basic and structured PNML, and take care that a net that happens to not use hierarchy is always written as basic PNML.

The second rule was broken by our own extension, EPNML. It adds a type label to arcs that determined their directions. The choice of source and target used to be defined as arbitrary. This breaks compatibility with tools that do not recognize our arc types. In EPNML 1.1, the arc type and the choice of source and target must agree.
These examples demonstrate how easy it is to accidentally break interoperability when designing Petri net extensions in PNML. If the utmost care is not taken, PNML will not achieve its intended purpose; it will just be a convenient mechanism for defining toolspecific file formats for Petri net tools.

**No support for constraints:** Precise syntax checking is crucial to a tool such as Petriweb. However, both standard PNML and our own extensions impose constraints on the use of PNML that cannot be expressed in RELAX NG (or, for that matter, in XML Schema). More accurate syntax checking can be attained by adding a mechanism to express constraints, e.g., Schematron [19].

### C.4 Future Work

Petriweb is installed at [http://www.petriweb.org/](http://www.petriweb.org/); we are populating it with collections of Petri nets. Specifically, we collect a large set of ‘interesting’ nets for researchers, as well as well-known standard Petri nets, e.g. from [9], [27], and [31].

Automated properties are a useful feature. They provide for the invocation of file format conversions and analysis tools in a batch run. However, the communication between Petriweb and tools leaves room for improvement.

A facility for batch uploading and downloading would be very convenient.

Support for communities could be further improved. At present, there is only one community with public access; it would be useful if there could be more. Further, it would be very useful if nets and property definitions could be shared among communities.

Adding support for version control will greatly aid the gradual development of models.

Another possible improvement is the unification of Petriweb properties and PNML labels. When retrieving Petri nets from Petriweb, the properties defined in Petriweb would be included into in the PNML. Conversely, when a net is uploaded, any of its attributes already present in PNML would show up as Petriweb properties.

This can be taken one step further, by automatically generating PNTDs corresponding to properties defined in Petriweb. Thus, Petriweb can be turned into a tool for creating and managing PNTDs.

At the moment, only static Petri nets are stored with a single initial marking. It is not yet possible to add occurrence sequences to a Petri net. The datamodel supports this extension, but it is not yet integrated in Petriweb.
C.5 Conclusions

The use of repositories to handle large collections of models helps the researcher and designer in his or her work. Petriweb is a repository to handle such collections. Multiple collections are supported by a single Petriweb installation.

A stable version of the Petriweb software is installed at www.petriweb.org. It is a fully functional prototype, and demonstrates the benefits of such repositories. The manual and automated property definition mechanisms allow for powerful search facilities. Automated properties also provide an elegant mechanism for batch processing on nets, used for structural and behavioral Petri net analysis and for file format conversions. A graphical Petri net viewer is integrated to support browsing.

Petriweb would be useless without a good standard file format for Petri nets. The qualities of PNML make it a good choice, but it can be improved in various respects.
List of Algorithms

3.1 Find Subworkflow Petri net ........................................ 30
A.1 State Machine check ................................................... 73
A.2 Marked Graph check .................................................... 73
A.3 Free Choice check ...................................................... 74
A.4 Atomic net check ....................................................... 74
A.5 Initial Places ........................................................... 75
A.6 PathProperty ............................................................ 75
A.7 Workflow check ........................................................ 76
A.8 CheckST ............................................................... 76
A.9 Quotient ................................................................. 77
A.10 CheckSMMG ............................................................ 77
A.11 FindFactor .............................................................. 78
A.12 FindFactor .............................................................. 79
A.13 Batch Workflow Petri net check .................................... 80
A.14 Transform \( N \) into a batch Workflow Petri net ..................... 80
A.15 Abstraction Rule ....................................................... 81
A.16 giveSinglePrePlace .................................................... 81
A.17 Linear Dependent Places Rule ....................................... 82
A.18 giveDependentPlaces ................................................. 83
List of Figures

1.1 Analysis of a Petri net, returning the results of the reduced net . . . 1
2.1 A Petri net model modelling two traffic lights on a crossway . . . 5
3.1 Smallest well-formed Free Choice Petri net . . . . . . . . . 22
3.2 The abstraction rule applied on transition $t$ . . . . . . . . . 23
3.3 Positive linear dependent place $b$ is removed . . . . . . . . . 24
3.4 Positive linear dependent transition $b$ is removed . . . . . . . 25
3.5 The six reduction rules, defined by Murata . . . . . . . . . 31
3.6 Place substitution. Place $s$ is reduced . . . . . . . . . . . . . 35
3.7 Identity transition rule, transition $t$ is an identity transition . . . 37
3.8 Identical transition rule, transitions $t1$ and $t2$ are identical . . 38
3.9 Classification of the reduction rules based on generalization . . . 40
4.1 Procedure for analyzing Petri nets, highest level . . . . . . . . 41
4.2 Sub procedure for analyzing Petri nets on soundness . . . . . . 43
4.3 Sub procedure for analyzing Petri nets on well-formedness . . . 44
5.1 Petri nets, selections and elements in a UML class diagram . . . 46
5.2 Structure of the Extended Composite Pattern . . . . . . . . . 47
5.3 UML class diagram of properties . . . . . . . . . . . . . . . 48
Bibliography


[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


