Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Petri Net Synthesis and Modal Specifications

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Naturwissenschaften

angenommene Dissertation

von Uli Christian Schlachter

geboren am 12.12.1989 in Oldenburg

Gutachter:
Prof. Dr. Eike Best
Prof. Dr. Javier Esparza

Tag der Disputation:
12. November 2018

Zusammenfassung

Bei der Petrinetz-Synthese soll ein gegebenes endliches beschriftetes Transitionssystem (labelled transition system; LTS) durch ein injektiv beschriftetes Petrinetz gelöst werden, was bedeutet, dass der Erreichbarkeitsgraph des Petrinetzes isomorph zum gegebenen LTS ist. Petrinetz-Synthese fing mit den Arbeiten von Ehrenfeucht und Rozenberg im Jahr 1990 an und wurde seitdem in verschiedene Richtungen erweitert.

Eine solche Erweiterung stellt diese Doktorarbeit dar. Ein generischer Algorithmus wird vorgestellt, der es erlaubt Petrinetz-Synthese auf eine gegebene Kombination von Teilklassen einzuschränken, beispielsweise schlichte und schlingenfreie Petrinetze. Da für eine gegebene Eingabe Petrinetz-Synthese nicht möglich sein könnte, wird ein Algorithmus zur minimalen Überapproximation eingeführt. Dieser Algorithmus funktioniert für einige der zuvor behandelten Teilklassen, während er für andere Teilklassen eventuell nicht terminiert. Die Überapproximation basiert auf dem Synthesealgorithmus, welcher fehlschlägt falls ein so genanntes Separierungsproblem nicht lösbar ist. Die Information über unlösbare Separierungsprobleme wird verwendet, um Zustände des aktuellen LTS zusammen zu fassen, und um neue Kanten zum LTS hinzuzufügen

Außerdem wird die Synthese von modalen Spezifikationen untersucht, und zwar von modalen Transitionssystemen, dem modalem μ-Kalkül und einer Teilmenge des μ-Kalküls, die konjunktiver ν-Kalkül heißt. Der ν-Kalkül und modale Transitionssysteme sind gleich ausdrucksmächtig und durch eine Reduktion von Zwei-Zählermaschinen wird für beide Spezifikationssprachen gezeigt, dass Petrinetz-Synthese unentscheidbar ist. Als nächstes wird ein Algorithmus zur Synthese von k-beschränkten Petrinetzen aus dem kompletten μ-Kalkül eingeführt, wobei $k \in \mathbb{N}$ *a priori* gegeben ist. Dies zeigt, dass diese Einschränkung das Problem sogar für den ausdrucksmächtigeren modalen μ-Kalkül entscheidbar macht. Der Algorithmus erweitert sein aktuelles LTS durch das Verhalten, das von der gegebenen Spezifikation verlangt wird. Um Petrinetz-Lösbarkeit zu garantieren wird die minimale Überapproximation aus dem ersten Teil der Arbeit verwendet.

Alle vorgestellten Algorithmen wurden in dem Tool APT implementiert. Diese Implementierung wird für eine Fallstudie zum sogenannten Philosophenproblem verwendet und betont einige Vorteile der Petrinetz-Synthese aus modalen Spezifikationen.

Abstract

In Petri net synthesis, a given finite labelled transition system (lts) should be solved by an injectively labelled Petri net, which means that the reachability graph of the Petri net is isomorphic to the given lts. Petri net synthesis goes back to the work of Ehrenfeucht and Rozenberg in 1990, and has since then been extended in various directions.

This thesis continues this work. We present a generic algorithm that supports targeting Petri net synthesis into a given combination of subclasses, such as plain and pure Petri nets. Since Petri net synthesis may not be possible for a given input, we introduce an algorithm for minimal over-approximation. This algorithm works for some of the subclasses that were previously handled, while for others the algorithm might not terminate. The over-approximation is based on the synthesis algorithm, which fails if some so-called separation problems are unsolvable. The information about unsolvable separation problems is used to merge states and add new outgoing edges to the current lts.

Furthermore, we investigate synthesis from modal specifications, namely modal transition systems, the modal μ-calculus, and a subset of the μ-calculus, which is called the conjunctive ν-calculus. The ν-calculus and modal transition systems are equally expressive and we show via a reduction from two-counter machines for both specification languages that Petri net synthesis is undecidable. Next, we introduce an algorithm for synthesising k-bounded Petri nets from the full modal μ-calculus, where $k \in \mathbb{N}$ is given *a priori*, showing that this restriction makes the problem decidable even for the more expressive modal μ-calculus. The algorithm extends its current lts by the behaviour required by the given specification. To ensure Petri net solvability, the minimal over-approximation from the first part of this thesis is used.

All presented algorithms were implemented in the tool APT. This implementation is used for a case study with the dining philosophers problem that highlights some advantages of Petri net synthesis from modal specifications.

Acknowledgements

This thesis would not have been possible without the help and support of many different people. I am grateful to Eike Best for giving me the opportunity to work with him and for already supporting me during my undergraduate studies. He introduced me to the topics of this thesis and always provided valuable feedback on my work. I also want to thank my committee consisting of Eike Best, Javier Esparza, Ernst-Rüdiger Olderog, and Mani Swaminathan for their helpful comments and questions.

Many people kindly proofread this thesis. This eliminated a great deal of errors and all the remaining ones were likely introduced by myself in later versions of this document. I am indebted to Evgeny Erofeev, Sabrina Frohn, Manuel Gieseking, Stefan Oehmke, Christoph Peuser, Tina Schlachter, Maike Schwammberger, and Harro Wimmel for suffering through this.

Writing this thesis was made more satisfactory by the nice working environment in the theoretical computer science groups in the university of Oldenburg. A 'Thank you' goes to my current and former colleagues Erzana Abdelwahab, Jan Steffen Becker, Marion Bramkamp, Björn Engelmann, Evgeny Erofeev, Hans Fleischhack, Nils-Erik Flick, Manuel Gieseking, Andrea Göken, Martin Hilscher, Stephanie Kemper, Alexander Kluge, Sven Linker, Heinrich Ody, Okan Özkan, Christoph Peuser, Jonas Prellberg, Hendrik Radke, Christian Sandmann, Maike Schwammberger, Valentin Spreckels, Mani Swaminathan, Patrick Uven, Ira Wempe, Elke Wilkeit, Harro Wimmel, and Nick Würdemann.

The members of the parallel systems group were always available for discussions and for talking about rough ideas. Thank you for your time and thoughts: Eike Best, Evgeny Erofeev, Valentin Spreckels, and Harro Wimmel.

My work was funded by the Deutsche Forschungsgemeinschaft under project ARS and I had the honour to participate in the research training group SCARE which is coordinated by the wonderful Ira Wempe, who always supports the SCAREies with her talent in organising and planning.

I would not be were I am today without my parents who supported me during my studies and by Sabrina Frohn, who, among many other things, made me finally sit down, get my stuff together and write this thesis. Thank you Sabrina for all your help and all the chores that you took off of my shoulders. 131.

Contents

List of Figures

1. Introduction

Petri nets have applications in a variety of fields, for example, in business process modelling [Aal16], and biology [PWM03]. They were reportedly invented for modelling of chemical processes [PR08], and are a tool to model the behaviour of a system. Among their advantages is their inherent notion of concurrency. This allows them to describe distributed systems compactly [BCD02; BD11; GGS13].

There are various semantics for Petri nets, e.g. step, pomset, or interleaving semantics [GGS11]. Given a Petri net, its interleaving behaviour is described by its *reachability graph*. This graph contains all reachable states of the Petri net and the possible edges between them. It allows defining properties of the behaviour of the Petri net, for example, that every state has an outgoing edge, which is the absence of deadlocks. In *system analysis*, a Petri net is used to model a system and the desired properties of the system are then checked on this model. For example, a business process should be, among other things, deadlock-free. Checking this property can be time-consuming and requires advanced techniques to be efficient [Fah+09].

While in system analysis, a given system is analysed for its properties, *system synthesis* is the opposite operation. The properties are given and a system enjoying these properties should be produced. For Petri net synthesis, this can mean that a prototypical reachability graph—a labelled transition system—is given. The synthesis procedure should then produce a Petri net that 'has this behaviour'. This means that the reachability graph of the resulting Petri net should be isomorphic to the given original labelled transition system, but other notions of equivalence are possible as well, e.g. language equivalence.

One common requirement is that each label that exists in the given transition system becomes exactly one transition in the resulting Petri net, i.e. no two transitions generate the same label in the reachability graph. Such Petri nets are called *unlabelled*. In contrast, multiple transitions can generate the same label in *labelled* Petri nets. Every labelled transition system (lts) can be synthesised into a labelled Petri net, but we will later see examples of lts that cannot be synthesised into unlabelled Petri nets. Thus, labelled Petri nets are more expressive than unlabelled Petri nets. However, the trivial translation of an lts into a labelled Petri net provides no insights. This translation represents each state of the lts as a place of the Petri net. An edge is turned into a transition. In contrast to this, in unlabelled Petri nets all edges with the same label are represented by the same transition and the synthesis procedure has to ensure the correct interplay between the transitions. Thus, these Petri nets tend to be smaller than the original lts. They also provide information about the behaviour, for example about

independent parts of the system. In practice, it would be desirable for Petri net synthesis to produce labelled Petri nets with few transitions. This would combine the expressivity of labelled Petri nets with the succinctness of unlabelled Petri nets. The existing results in this direction are however not as advanced as the results for synthesis of unlabelled Petri nets. So far, only heuristics are known and exact algorithms are missing [CKLY98; CCK08; Car12]. In this thesis we will also focus on unlabelled Petri nets.

A labelled transition system precisely describes the interleaving behaviour of a system. In practice, it is desirable to have more flexible specifications, e.g. allowing some part of the behaviour to be left out. For example, when designing a communication protocol, the loss of messages has to be handled. However, the specification should not require the protocol to lose messages, but only specify how to deal with this, if it occurs [Bru97]. One way to achieve this is via modal specifications. Here, the specification is split into *required* and *allowed behaviour*. An implementation must present all the required behaviour and its own behaviour must be within the behaviour allowed by the specification. In this thesis, Petri net synthesis from modal specifications is investigated, which means that the reachability graph of the resulting Petri net has to satisfy a given specification. Examples for modal specifications are modal transition systems [Lar89; Kre17] and the modal μ-calculus [Koz83; AN01]. Badouel, Bernardinello, and Darondeau mention Petri net synthesis from modal transition systems as an interesting problem that is neither known to be decidable nor undecidable [BBD15]. Also, one of the problems investigated in [Dar05] is closely related to Petri net synthesis from modal transition systems. Their paper investigates supervisory control via Petri nets, i.e. a Petri net is used to forbid some behaviour in a system to ensure some property. Since it was not known whether Petri net synthesis from modal transition systems is decidable, Darondeau used a stronger constraint instead that is not related to modal transition systems.

In this thesis, the following problem is considered: Given a modal specification, is there a Petri net with a finite reachability graph satisfying this specification? We will show that the full problem is undecidable, but will identify a decidable subproblem that is still quite expressive. These investigations are also made with respect to various subclasses of Petri nets, for example, pure[1] and k-bounded[2] Petri nets. Approximate synthesis will be a helpful tool for this goal. This means that an approximate solution is provided if no exact solution is possible, instead of producing only a negative answer.

The structure of this thesis is as follows: The main content of this document is split into two parts. In the first part, Petri net synthesis from labelled transition systems up to isomorphism is investigated. In the second part, we consider Petri net synthesis from modal specifications. For this, we will formally introduce Petri nets and labelled transition systems in Chapter 2. The first part of this thesis begins in Chapter 3 with an introduction to region theory as basis for the Petri net synthesis. This is based on existing results from [BBD15; BD96]. Chapter 4 extends this for targeted Petri net synthesis, which means that various properties, plus boolean combinations of these, can

[1] There are no flows in both directions between any transition and place.
[2] No place contains more than k tokens in any reachable marking.

be required for the desired Petri net. In Chapter 5, we investigate approximative Petri net synthesis. The author implemented the algorithms that are presented in this first part of the document in the tool APT. This tool is discussed in Chapter 6.

The second part of the document starts in Chapter 7 by introducing modal transition systems, the modal μ-calculus, and the conjunctive ν-calculus. The ν-calculus is a syntactic subset of the modal μ-calculus that is expressively equivalent to modal transition systems. In Chapter 8, we show that Petri net synthesis from the conjunctive ν-calculus is undecidable. By their equivalence to the ν-calculus, synthesis from modal transition systems is also undecidable. In Chapter 9, we present an algorithm for Petri net synthesis from modal specifications by restricting the problem to k-bounded Petri nets for a fixed number k. The author also implemented this algorithm and the resulting tool is introduced in Chapter 10. Chapter 11 contains a case study, which explores some of the possibilities that arise with Petri net synthesis from modal specifications instead of the less expressive labelled transition systems. Finally, Chapter 12 summarises the results of this thesis and indicates some open problems.

2. Petri Nets and Labelled Transition Systems

Petri nets consist of places and transitions, which are connected by flows. Flows can have a number as their weight, and a weight of one is often left implicit. In figures, the places are represented by circles and transitions by rectangles. A place can hold tokens. The current state of a Petri net, called a *marking*, is the distribution of tokens on places.

Definition 2.0.1 (Petri net [Rei85])**.** *A* Petri net *is a tuple* $N = (P, T, F, M_0)$*, where* P *and* T *are finite and disjoint sets of* places *and* transitions*, respectively, and* $F\colon ((P \times T) \cup (T \times P)) \to \mathbb{N}$ *is a* flow relation *specifying* flow weights*.* M_0 *is the* initial marking*, where a* marking *is a function* $P \to \mathbb{N}$*.*

An example of a Petri net $N = (P, T, F, M_0)$ is shown on the left in Figure 2.1. It has the places $P = \{p_0, p_1, p_2, p_3\}$ and transitions $T = \{a, b, c\}$. Its initial marking is described by the mapping M_0 that satisfies $M_0(p_0) = M_0(p_2) = 1$ and $M_0(p_1) = M_0(p_3) = 0$.

Definition 2.0.2 (Pre- and postset)**.** *For a place* $p \in P$ *of a Petri net* $N = (P, T, F, M_0)$*, its* preset *is the set* ${}^\bullet p = \{t \in T \mid F(t, p) > 0\}$*, and its* postset *is* $p^\bullet = \{t \in T \mid F(p, t) > 0\}$*. Similarly, for a transition* $t \in T$ *define* ${}^\bullet t = \{p \in P \mid F(p, t) > 0\}$ *to be its* preset*, and* $t^\bullet = \{p \in P \mid F(t, p) > 0\}$ *to be its* postset*.*

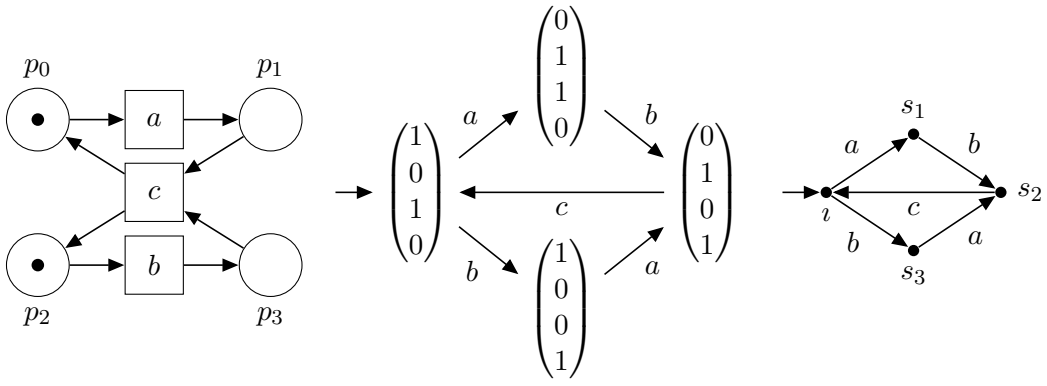


Figure 2.1.: Example of a Petri net (left), its behaviour (middle), and an lts (right).

The flows describe a relation between places and transitions. In the example there is a flow from place p_0 to transition a. We say that p_0 is in the preset of a, and that a is in the postset of p_0. Since there are no flows from p_0 to other transitions, nor from other places to a, the postset of p_0 is $p_0^\bullet = \{a\}$ and the preset of a is ${}^\bullet a = \{p_0\}$.

Petri nets are not just static constructs. They have a current state, represented by a marking, that assigns each place a number of tokens. Transitions can fire, which changes the current marking, but only if enough tokens are available. The requirements and effects of transitions are described by the flows: The incoming flows of a transition describe how many tokens it consumes while the outgoing flows describe how many tokens are produced.

Definition 2.0.3 (Enabledness and firing)**.** *A transition $t \in T$ is* enabled *in a marking M if $\forall p \in P\colon F(p,t) \leq M(p)$. This is written as $M[t\rangle$. An enabled transition can* fire*, leading to the marking M' defined by $\forall p \in P\colon M'(p) = M(p) - F(p,t) + F(t,p)$. We write $M[t\rangle M'$, and this syntax is inductively extended to label sequences $\sigma \in T^*$: $M[\varepsilon\rangle M$ is always true, and $M[wt\rangle M'$ holds if there is a marking M'' with $M[w\rangle M''[t\rangle M'$.*

For example, in the Petri net from Figure 2.1, transition a can fire, because it only needs a token from place p_0, which currently has one token. The transition consumes the token from place p_0 and produces a token on place p_1. The new marking M is $M(p_0) = 0$, $M(p_1) = 1 = M(p_2)$, and $M(p_3) = 0$.

The behaviour of a process, such as a Petri net, can be visualised by means of a *labelled transition system*. Such a system consists of states and edges, where edges connect two states and have a label, i.e. an lts is an edge-labelled directed graph. Two examples of lts are shown in the centre and on the right of Figure 2.1. A special state, the initial state, is marked by an incoming unlabelled arrow.

Definition 2.0.4 (Labelled transition system [Arn94])**.** *A* labelled transition system (lts) *is a structure $A = (S, \Sigma, \to, \imath)$, where S is a set of* states*, Σ is an alphabet containing* labels *or* events*, $\to \subseteq S \times \Sigma \times S$ is an* edge relation *and $\imath$ is its* initial state*.*

For example, the lts on the right of Figure 2.1 has $\imath$ as its initial state. Its alphabet is not explicitly given, but can be inferred to be at least $\Sigma = \{a, b, c\}$, since these are the labels that appear on the edges.

The elements of a Petri net N and an lts A will be canonically called P, T, F, M_0 and $S, \Sigma, \to, \imath$, respectively.

Definition 2.0.5 (Paths)**.** *An edge $(s, t, s') \in \to$ is written as $s \xrightarrow{t} s'$, which means that s' is* reachable *from s through the execution of t. This is extended to words $w \in \Sigma^*$ via $s \xrightarrow{\varepsilon} s$ and $s \xrightarrow{w} s' \xrightarrow{a} s'' \Rightarrow s \xrightarrow{wa} s''$. If some $s' \in S$ with $s \xrightarrow{w} s'$ exists, we write $s \xrightarrow{w}$ and call this a* path*. If no such s' exists, we write $s \xrightarrow{w}\!\!\!\!\!\!/\;$.*

Our example lts has an edge from $\imath$ to s_1 via a. This can be written as $\imath \xrightarrow{a} s_1$. There is another edge from s_1 to s_2 via b. This means we can go from $\imath$ to s_2 by first following an a-labelled edge to s_1 and then an edge with label b to s_2. To express this briefly,

we write $\imath \xrightarrow{ab} s_2$. It is also possible to go from $\imath$ to s_3 with the sequence $abcb$, hence $\imath \xrightarrow{abcb} s_3$. Since the sequence abb is not possible, we can write $\imath \xrightarrow{abb}\!\!\!\!\!\!\not\;\;$.

The behaviour of a Petri net can be represented as an lts, which is named its *reachability graph*. Markings are the states and an edge from M to M' with label t exists if transition t can fire in M and leads to M', i.e. $M[t\rangle M'$. The reachability graph of the Petri net shown on the left in Figure 2.1 is displayed in the middle of the same figure, with markings M written as vectors $M = \begin{pmatrix} M(p_0) & M(p_1) & M(p_2) & M(p_3) \end{pmatrix}^\intercal$.

Definition 2.0.6 (Reachability graph)**.** *The set of* reachable markings *of a Petri net* $N = (P, T, F, M_0)$ *is* $\mathfrak{E}(N) = \{M \mid \exists \sigma \in T^* \colon M_0[\sigma\rangle M\}$*. Its* reachability graph *is the lts* $\mathrm{RG}(N) = (\mathfrak{E}(N), T, \to, M_0)$ *with* $\to = \{(M, t, M') \in \mathfrak{E}(N) \times T \times \mathfrak{E}(N) \mid M[t\rangle M'\}$.

Reachability graphs of Petri nets satisfy some properties which not all lts enjoy. For example, reachability graphs are deterministic. This means that in a marking, a transition t cannot lead to two different successor markings. Also, reachability graphs are reachable by definition, meaning that it is possible to go to every state from the initial state. If a Petri net only has finitely many reachable markings, it is called bounded. Not every Petri net is bounded.

Definition 2.0.7 (Finite, deterministic, reachable, and bounded)**.** *An lts A is called* finite *if S and Σ (and hence also $\to$) are finite. It is* deterministic *if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies $s' = s''$ for all $a \in \Sigma$ and $s, s', s'' \in S$. It is called* reachable *if for every $s \in S$ there is a $w \in \Sigma^*$ so that $\imath \xrightarrow{w} s$. A Petri net N is* bounded *if its reachability graph* $\mathrm{RG}(N)$ *is finite.*

Lemma 2.0.8. *The reachability graph of a Petri net is deterministic and reachable.*

Proof. By the Petri net firing rule, M' is uniquely determined in $M[t\rangle M'$, so reachability graphs are deterministic. The definition of the reachability graph uses the set $\mathfrak{E}(N)$ of reachable markings, so reachability graphs are reachable by definition. □

Some other common properties of Petri nets are defined next. The running example of Figure 2.1 is plain, pure, and 1-bounded.

Definition 2.0.9 (Plain, pure, k-bounded)**.** *A Petri net N is* plain *if its flows have only weights zero or one, i.e.* $\mathrm{cod}(F) \subseteq \{0, 1\}$*. A Petri net is* pure *if there are no flows in both directions between some place and transition, i.e. $\forall p \in P \colon {}^\bullet p \cap p^\bullet = \emptyset$. A Petri net is* k-bounded *for $k \in \mathbb{N}$ if no place has more than k tokens in some reachable marking, i.e. $\forall M \in \mathfrak{E}(N) \colon \forall p \in P \colon M(p) \leq k$.*

When we are not interested in the exact order of events in a sequence, we can use the Parikh vector of the sequence. This vector describes how often each event appears in a sequence. For example, since b appears twice in $abcb$, we have $\Psi(abcb)(b) = 2$.

Definition 2.0.10 (Parikh vector)**.** *The* Parikh vector $\Psi(w) \colon \Sigma \to \mathbb{N}$ *of a word $w \in \Sigma^*$ is defined for $a \in \Sigma$ via $\Psi(\varepsilon)(a) = 0$ and $\Psi(wb)(a) = \Psi(w)(a) + \begin{cases} 0 & \text{if } a \neq b \\ 1 & \text{if } a = b \end{cases}$.*

A well-known property of Petri nets is the *marking equation*. When a marking M' can be reached from M via a sequence w, i.e. $M[w\rangle M'$, then the marking equation relates these three elements. This equation uses the incidence matrix C which describes the change of tokens when a transition t fires.

Definition 2.0.11 ($C(N)$)**.** *The* incidence matrix $C(N)$ *(or just C if N is clear) of a Petri net $N = (P, T, F, M_0)$ is the $(P \times T)$-matrix defined by $C(p,t) = F(t,p) - F(p,t)$.*

For example, the Petri net N from Figure 2.1 has the following incidence matrix:

$$C(N) = \begin{array}{c} \\ p_0 \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} -1 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \end{pmatrix} \end{array}$$

Lemma 2.0.12 (Marking equation)**.** *Given a Petri net N with markings M and M'. If $M[w\rangle M'$ for $w \in T^*$, then $M' = M + C \cdot \Psi(w)$.*

Proof. By induction: If $w = \varepsilon$, then $C \cdot \Psi(\varepsilon)$ is the null vector and so the equation is equivalent to $M = M'$, which is true by definition of the firing rule.

If $w = w't$ with $t \in T$, then there is a marking M'' so that $M[w'\rangle M''[t\rangle M'$. Consider some place $p \in P$. By the Petri net firing rule, we have $M'(p) = M''(p) - F(p,t) + F(t,p)$. Since $F(t,p) - F(p,t) = C(p,t)$, the difference between the two markings is the t-row of the matrix C, i.e. $C \cdot \Psi(t)$, and we have $M' = M'' + C \cdot \Psi(t)$. By the induction assumption we have $M'' = M + C \cdot \Psi(w')$, so $M' = M + C \cdot \Psi(w') + C \cdot \Psi(t)$. By linearity and since $\Psi(w') + \Psi(t) = \Psi(w't) = \Psi(w)$ this results in $M' = M + C \cdot \Psi(w)$, which was to be shown. □

In the initial marking M_0 of N of Figure 2.1, the sequence ab can occur, and leads to some marking M, i.e. $M_0[ab\rangle M$. The marking equation can express this relation as follows:

$$M = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = M_0 + C \cdot \Psi(ab)$$

With these definitions, we can start from a given Petri net and visualise its behaviour as an lts. We might not be interested in the exact markings that are reached in the Petri net, but only in the possible firing of transitions. Isomorphisms allow to abstract away from markings. For example, the reachability graph shown in the middle of Figure 2.1 and the lts on the right in the same figure are isomorphic.

Definition 2.0.13 (Isomorphism)**.** *Two lts A_1, A_2 with $A_i = (S_i, \Sigma, \to_i, \imath_i)$ are* isomorphic *if there exists a bijection $f\colon S_1 \to S_2$ so that $f(\imath_1) = \imath_2$ and for all $s, s' \in S_1, a \in \Sigma\colon s \xrightarrow{a}_1 s' \iff f(s) \xrightarrow{a}_2 f(s')$. In this case, f is an* isomorphism *and we abstract away from state names and write $A_1 = A_2$.*

Part I.

Petri Net Synthesis from Labelled Transition Systems

3. Introduction to Region Theory

The previous chapter introduced Petri nets and lts. The reachability graph defines an lts based on a Petri net. This lts describes the Petri net's behaviour. The opposite operation, constructing a Petri net from an lts, is Petri net synthesis, and is the topic of this chapter.

Problem 3.0.1 (Petri net synthesis from finite lts)**.** *Given a finite lts A, is there a Petri net N with* $\mathrm{RG}(N) = A$*, i.e. a Petri net N with a reachability graph isomorphic to A?*

If the reachability graph of N is isomorphic to A, we say that N *solves* A. Of course, this problem should not just be decided, but a Petri net N is sought. The concept of a *region* allows to do this, so this chapter introduces region theory. A region of an lts describes a potential place of a Petri net that solves this lts.

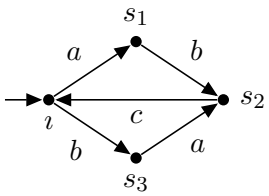
This chapter is based on [BBD15; BD96] while the definition of a region goes back to [ER90a; ER90b]. Only the basic concepts needed for the algorithm will be introduced. For a description of some of the possible optimisations, the author recommends to look into [BBD15].

3.1. Regions

A region is defined based on an lts. It defines a possible place in a Petri net that solves the lts. A region has three parts: $\mathcal{R}$ assigns a number of tokens to each state, while $\mathcal{B}$ and $\mathcal{F}$ assign weights to each event in the alphabet. The number of tokens of the initial state, $\mathcal{R}(\imath)$, corresponds to the initial marking of the place, while $\mathcal{B}(t)$ and $\mathcal{F}(t)$ describe how many tokens transition t consumes and produces, respectively.

Definition 3.1.1 (Region)**.** *A* region *of an lts $A = (S, \Sigma, \to, \imath)$ is a triple of functions $r = (\mathcal{R}, \mathcal{B}, \mathcal{F}) \in (\mathbb{N}^S) \times (\mathbb{N}^\Sigma) \times (\mathbb{N}^\Sigma)$ such that for all $(s, t, s') \in \to$, both $\mathcal{R}(s) \geq \mathcal{B}(t)$ and $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t)$ hold.*

The first requirement above describes that (the place described by) the region may not prevent an occurrence that is present in the lts. The second requirement expresses that the number of tokens $\mathcal{R}(s)$ is consistent with the token change $\mathcal{F}(t) - \mathcal{B}(t)$, e.g. if this value is three, then $\mathcal{R}(s)$ must increase by three along any edge with event t. These requirements are similar to the requirements on a place for enabling a transition ($M[t\rangle$) and for the firing of a transition ($M[t\rangle M'$).



$$\begin{array}{llll} \mathcal{R}_1(\iota)=0 & \mathcal{R}_1(s_1)=0 & \mathcal{R}_2(\iota)=1 & \mathcal{R}_2(s_1)=1 \\ \mathcal{R}_1(s_2)=1 & \mathcal{R}_1(s_3)=1 & \mathcal{R}_2(s_2)=0 & \mathcal{R}_2(s_3)=0 \\ \mathcal{B}_1(a)=0 & \mathcal{F}_1(a)=0 & \mathcal{B}_2(a)=0 & \mathcal{F}_2(a)=0 \\ \mathcal{B}_1(b)=0 & \mathcal{F}_1(b)=1 & \mathcal{B}_2(b)=1 & \mathcal{F}_2(b)=0 \\ \mathcal{B}_1(c)=1 & \mathcal{F}_1(c)=0 & \mathcal{B}_2(c)=0 & \mathcal{F}_2(c)=1 \end{array}$$

Figure 3.1.: An lts and two of its regions $r_1 = (\mathcal{R}_1, \mathcal{B}_1, \mathcal{F}_1)$ and $r_2 = (\mathcal{R}_2, \mathcal{B}_2, \mathcal{F}_2)$.

Two examples for regions are shown in Figure 3.1. The region r_1 corresponds to place p_3 of the Petri net from Figure 2.1, while region r_2 corresponds to place p_2.

Since the expression $\mathcal{F}(t) - \mathcal{B}(t)$, which describes the change in tokens for transition t, is needed often, this is called the *effect* of t and gets its own symbol $\mathcal{E}(t)$. Also, these functions are lifted to vectors to express the total outcome of each path with that Parikh vector, as follows.

Definition 3.1.2 (Effect, vector extension)**.** *Given an lts A and one of its regions $(\mathcal{R}, \mathcal{B}, \mathcal{F})$, the* effect function *is the derived function $\mathcal{E}$ defined by $\mathcal{E}(t) = \mathcal{F}(t) - \mathcal{B}(t)$.*

The function $\mathcal{B}$ ($\mathcal{F}$ and $\mathcal{E}$, resp.) is inductively extended to Σ-vectors $\pi\colon \Sigma \to \mathbb{N}$ by defining $\mathcal{B}(\pi) = \sum_{t\in\Sigma} \pi(t) \cdot \mathcal{B}(t)$ (analogously for $\mathcal{F}$ and $\mathcal{E}$).

Some fundamental properties of regions are shown next.

Lemma 3.1.3. *If $s \xrightarrow{w} s'$ in some lts, then for any region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ it holds that $\mathcal{R}(s') = \mathcal{R}(s) + \mathcal{E}(\Psi(w))$.*

Proof. By induction on the length of w. For the base case $s \xrightarrow{\varepsilon} s$ by definition $\mathcal{E}(\Psi(\varepsilon)) = 0$ holds. The induction step follows from the definition of a region and of $\mathcal{E}$ ($s \xrightarrow{t} s' \Rightarrow \mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t) = \mathcal{R}(s) + \mathcal{E}(t)$). □

A corollary is possible, because every state of a reachable lts can be reached from its initial state:

Corollary 3.1.4. *In a reachable lts, the function $\mathcal{R}$ of a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is fully determined by $\mathcal{R}(\iota)$.*

For another corollary, consider a so-called *cycle* $s \xrightarrow{w} s$, which is a sequence which leads from s back to s. The above Lemma 3.1.3 implies that w cannot change the number of tokens in this situation:

Corollary 3.1.5. *If $s \xrightarrow{w} s$, then $\mathcal{E}(\Psi(w))(t) = 0$ for all $t \in \Sigma$.*

It is well known that in a (bounded) Petri net a complement place can be constructed for a given place. This place behaves opposite to the original place, for example, gaining as many tokens as the other place loses in the firing of a transition. A consequence is that the sum of the number of tokens on a place and its complement is invariant when firing transitions. A similar construction is also possible for regions.

Lemma 3.1.6. *For a region $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ of a finite lts $A = (S, \Sigma, \to, \imath)$, there is a complementary region $\overline{r} = (\overline{\mathcal{R}}, \overline{\mathcal{B}}, \overline{\mathcal{F}})$ and a number $k \in \mathbb{N}$ with $\overline{\mathcal{B}} = \mathcal{F}$ and $\overline{\mathcal{F}} = \mathcal{B}$, such that for all $s \in S$ we have $\overline{\mathcal{R}}(s) = k - \mathcal{R}(s)$.*

Proof. Only a suitable k has to be found so that $\overline{\mathcal{R}}$ does not produce negative numbers and so that for each edge $s \xrightarrow{t} s'$ we have $\overline{\mathcal{R}}(s) \geq \overline{\mathcal{B}}(t)$. Since A is finite, there are only finitely many states and edges. This means we can e.g. choose k to be the minimum value so that $\overline{r}$ is a region. The remaining properties of a region are inherited by $\overline{r}$ from r. For example, for an edge $s \xrightarrow{t} s'$, it automatically holds that $\overline{\mathcal{R}}(s') = k - \mathcal{R}(s') = k - (\mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t)) = \overline{\mathcal{R}}(s) - \overline{\mathcal{B}}(t) + \overline{\mathcal{F}}(t)$, as required. □

For example, the two regions in Figure 3.1 are complements of each other with $k = 1$.

Intuitively, a region of an lts A is a possible place that can be added to a Petri net that should solve A. In this way we can construct a Petri net from regions. This intuition is formalised next.

Definition 3.1.7 (Corresponding Petri net)**.** *Let $A = (S, \Sigma, \to, \imath)$ be an lts and R a set of regions of A. The* corresponding Petri net *$N(R) = (R, \Sigma, F, M_0)$ has the regions R as places and the alphabet Σ of A as its set of transitions. Its flow function F is defined by $F((\mathcal{R}, \mathcal{B}, \mathcal{F}), t) = \mathcal{B}(t)$ and $F(t, (\mathcal{R}, \mathcal{B}, \mathcal{F})) = \mathcal{F}(t)$, and the initial marking M_0 is defined by $M_0((\mathcal{R}, \mathcal{B}, \mathcal{F})) = \mathcal{R}(\imath)$.*

A place of a Petri net also defines a region on the reachability graph of the Petri net. This leads to a Galois connection further investigated in [BBD15].

Lemma 3.1.8. *Let $N = (P, T, F, M_0)$ be a bounded Petri net and $p \in P$ one of its places. The* extension $[[p]]$ of p *is the region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ of* $\mathrm{RG}(N)$ *defined by $\mathcal{R}(M) = M(p)$, $\mathcal{B}(t) = F(p, t)$, and $\mathcal{F}(t) = F(t, p)$.*

Proof. We show that $[[p]]$ is indeed a region. We have to show that for all $M[t\rangle M'$ in the reachability graph of N, both $\mathcal{R}(M) \geq \mathcal{B}(t)$ and $\mathcal{R}(M') = \mathcal{R}(M) - \mathcal{B}(t) + \mathcal{F}(t)$ hold. By definition of $[[p]]$, this is equivalent to $M(p) \geq F(p, t)$ and $M'(p) = M(p) - F(p, t) + F(t, p)$, which holds by definition of the firing of transitions (Def. 2.0.3). □

This definition and this lemma allow to freely switch between a Petri net and a set of regions without loss of information. In other words, a place and a region can be considered to be equivalent.

3.2. Separation Problems

This section deals with the question which regions are actually needed so that the constructed Petri net solves A. The question we want to answer is: For which sets R of regions of an lts A does $A = \mathrm{RG}(N(R))$ hold? This will solve Problem 3.0.1.

To answer this, we define *separation problems*. There are two kinds of separation problems: A state separation problem consists of two states and is solved by a region which assigns different numbers of tokens to the two states. This region ensures that the two states correspond to different markings in a Petri net, and are not identified into the same marking. An event/state separation problem has a state and a label that is not enabled in that state. It is solved by a region which prevents the corresponding transition to fire in the state. For the corresponding Petri net, this means that the transition cannot fire in the marking that represents the state.

Definition 3.2.1 (Separation problems)**.** *Let $A = (S, \Sigma, \to, \imath)$ be an lts. A* state separation problem (SSP) *$\{s, s'\}$ is a pair of states $s, s' \in S$ with $s \neq s'$. It is* solved *by a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ when $\mathcal{R}(s) \neq \mathcal{R}(s')$. An* event/state separation problem (ESSP) *(s, t) consists of a state $s \in S$ and a label $t \in \Sigma$ so that $s \overset{t}{\not\to}$. It is solved by a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ when $\mathcal{R}(s) < \mathcal{B}(t)$.*

The sets SSP_A and ESSP_A contain all SSP instances of A, and all ESSP instances of A, respectively. The set of all separation problems of A is $\mathrm{SP}_A = \mathrm{SSP}_A \cup \mathrm{ESSP}_A$.

Each pair of states is a state separation problem, so the lts from Figure 3.1 has $\mathrm{SSP}_A = \{\{\imath, s_1\}, \{\imath, s_2\}, \{\imath, s_3\}, \{s_1, s_2\}, \{s_1, s_3\}, \{s_2, s_3\}\}$, and its event/state separation problems are $\mathrm{ESSP}_A = \{(\imath, c), (s_1, a), (s_1, c), (s_2, a), (s_2, b), (s_3, b), (s_3, c)\}$. In the example from Figure 3.1, among others, region r_1 solves the state separation problems $\{\imath, s_2\}$ and $\{s_1, s_3\}$, but not $\{\imath, s_1\}$. r_1 also solves the event/state separation problem (s_1, c), because $\mathcal{R}_1(\imath) = 0$, which is smaller than $\mathcal{B}_1(c) = 1$. The region r_2 does not solve this problem, but for example, solves (s_2, b), which is not solved by r_1.

The answer to the above question, about when a set of regions solves an lts, is that every separation problem has to be solved:

Theorem 3.2.2 ([BBD15])**.** *A set of regions R of a reachable lts A satisfies $A = \mathrm{RG}(N(R))$ if and only if each state separation problem and each event/state separation problem of A is solved by some region in R.*

Proof sketch. ($\Rightarrow$): Assuming A is solved by the Petri net $N(R)$, it can easily be seen that each separation problem is solved by a region in R: If $\neg(s[t\rangle)$, then in the marking M corresponding to s, there is a place of the Petri net $N(R)$, i.e. a region $r \in R$, that disables transition t. This means the region r solves the ESSP instance (s, t). Analogously, for two different markings $M \neq M'$ there is a place solving the corresponding SSP instance.

($\Leftarrow$): For the other direction, assume each separation problem is solved by a region in R, A is reachable, and let $A = (S, \Sigma, \to, \imath)$. Let f be a function mapping states of A to markings of $N(R)$ defined by $f(s)((\mathcal{R}, \mathcal{B}, \mathcal{F})) = \mathcal{R}(s)$, i.e. the state $s \in S$ of A is mapped to the marking of $N(R)$ in which a place $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is assigned $\mathcal{R}(s)$ tokens. It can be shown that f is indeed an isomorphism, as follows.

If $s \overset{t}{\to} s'$ in A, we have to show that $f(s) \overset{t}{\to} f(s')$ in $\mathrm{RG}(N)$. This follows from the definition of regions: No place in $f(s)$ may disable transition t and firing the transition

has the expected effect on the function $\mathcal{R}$, which was used to define f. For the other direction, if $s \overset{t}{\nrightarrow}$, then (s, t) is an ESSP instance of A. By assumption there is a region r solving this instance, so disabling transition t in the marking $f(s)$. Thus, $s \xrightarrow{t} s' \Leftrightarrow f(s) \xrightarrow{t} f(s')$.

It remains to show that f is a bijection. It is injective, because all state separation problems have a solution in R, thus $s \neq s' \Rightarrow \exists(\mathcal{R}, \mathcal{B}, \mathcal{F}) \in R\colon \mathcal{R}(s) \neq \mathcal{R}(s') \Rightarrow f(s) \neq f(s')$. f is surjective, because Petri net reachability graphs are reachable by Lemma 2.0.8. Thus, any marking M in $\mathrm{RG}(N)$ is reachable via some path $M_0 \xrightarrow{w} M$. It was already shown that edges in $\mathrm{RG}(N)$ are also present in A, so there must be some state s of A with $\imath \xrightarrow{w} s$ and $f(s) = M$. □

Note that the previous theorem does not assume A to be finite, but the condition that A is reachable cannot be left out. For example, consider the lts $A = (\{\imath, s_1\}, \emptyset, \emptyset, \imath)$, which consists of just two disconnected states. This lts has no event/state separation problems and just one state separation problem $\{\imath, s_1\}$. This state separation problem can easily be solved by some region r, but $A = \mathrm{RG}(N(\{r\}))$ will not hold.

3.3. An Algorithm for Petri Net Synthesis

While the previous section characterised when an lts can be solved by a Petri net, it did not answer the question how to find such a Petri net. An algorithm taken from [BBD15], that answers this question, is presented next. The algorithm repeatedly computes a new region of A that solves a given separation problem until solutions to all separation problems are found.

We assume that we are given a finite and reachable lts A as input. Reachability is not a restriction here, since by Lemma 2.0.8, an lts that is not reachable cannot be isomorphic to the reachability graph of any Petri net. This assumption allows us to simplify the following argument.

The first step of the algorithm is to fix an arbitrary spanning tree of the input A. This spanning tree assigns to each state s a path to reach s from the initial state.

Definition 3.3.1 (Spanning tree)**.** *A* spanning tree *of a reachable lts $(S, \Sigma, \to, \imath)$ is an lts $(S, \Sigma, \to', \imath)$ with $\to' \subseteq \to$ so that for every $s \in S$ there is a unique $w_s \in \Sigma^*$ with $\imath \xrightarrow{w_s}' s$. The* reaching Parikh vector Ψ_s *of a state s in a spanning tree is $\Psi_s = \Psi(w_s)$.*

The reaching Parikh vector is used as follows: By Lemma 3.1.3 and Definition 3.1.2 we now have $\mathcal{R}(s) = \mathcal{R}(\imath) + \mathcal{E}(\Psi_s) = \mathcal{R}(\imath) + \sum_{t \in \Sigma} \Psi_s(t) \cdot (\mathcal{F}(t) - \mathcal{B}(t))$ for any region and any state $s \in S$. Thus, a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is fully determined by $\mathcal{R}(\imath)$, $\mathcal{B}$, and $\mathcal{F}$ (Corollary 3.1.4). These are $2|\Sigma|+1$ natural numbers: $\mathcal{R}(\imath)$ is a single number and $\mathcal{B}$ and $\mathcal{F}$ are each described by one number per element in Σ. This means a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ can be represented as a vector $r \in \mathbb{N}^{2n+1}$ where $n = |\Sigma|$. For this, fix an arbitrary

enumeration $\Sigma = \{t_1, t_2, \ldots, t_n\}$ of Σ and define the vector representation via $r_0 = \mathcal{R}(\imath)$, $r_1 = \mathcal{B}(t_1)$, ..., $r_n = \mathcal{B}(t_n)$, $r_{n+1} = \mathcal{F}(t_1)$, ..., $r_{2n} = \mathcal{F}(t_n)$).

Given such a vector, the formula $\mathcal{R}(s) = \mathcal{R}(\imath) + \sum_{t\in\Sigma} \Psi_s(t) \cdot (\mathcal{F}(t) - \mathcal{B}(t))$ that was mentioned above reconstructs the number of tokens $\mathcal{R}(s)$ for a state $s \in S$. In the vectorial representation, this can be expressed as the following function:

$$\text{tokens}(r, s) := r_0 + \sum_{i=1}^{n} \Psi_s(t_i) \cdot (r_{n+i} - r_i)$$

An arbitrary vector does not necessarily represent a region, because the requirements from the definition of a region have to be fulfilled. Namely, if a state s has an outgoing edge with label t, then there must not be less tokens available in s than required by t ($\mathcal{R}(s) \geq \mathcal{B}(t)$). Also, some consistency between the token differences between states and the effect of edges is required, which means that if $s \xrightarrow{t} s'$, then $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t)$. However, $\mathcal{R}(s)$ is defined by the initial token count $\mathcal{R}(\imath)$ and the subset of edges $\rightarrow'$ that represent the spanning tree. Thus, for these edges this requirement holds automatically and just the remaining edges $\rightarrow \setminus \rightarrow'$ explicitly need this requirement. Altogether, we get the following predicate:

$$\begin{aligned}\text{isRegion}(r) := & \bigwedge_{(s,t_i,s')\in\rightarrow} (\text{tokens}(r, s) \geq r_i) \\ & \wedge \bigwedge_{(s,t_i,s')\in\rightarrow\setminus\rightarrow'} (\text{tokens}(r, s') = \text{tokens}(r, s) - r_i + r_{n+i})\end{aligned}$$

The next lemma will be used to restrict our search to vectors describing regions[1].

Lemma 3.3.2. *A vector $r \in \mathbb{N}^{2n+1}$ represents a region if and only if* isRegion(r) *holds.*

Proof. ($\Rightarrow$) Assuming that r represents the region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$, we have $\mathcal{R}(s) = \text{tokens}(r, s)$ by Lemma 3.1.3 and the definition of Ψ_s as the Parikh vector of a path from $\imath$ to s. isRegion(r) is now guaranteed to hold by the definition of a region.

($\Leftarrow$) $\mathcal{R}$ is defined by $\mathcal{R}(s) = \text{tokens}(r, s)$ and the remaining functions are directly given in the representation. We want to show that this is a region, which means that we have to show that $\mathcal{R}(s) \geq \mathcal{B}(t)$ and $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t)$ for all $(s, t_i, s') \in \rightarrow$. The first condition is directly given in our assumptions, and so is the second condition for $(s, t_i, s') \in \rightarrow \setminus \rightarrow'$. Thus, only $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t) + \mathcal{F}(t)$ for $(s, t_i, s') \in \rightarrow'$ remains to be shown.

This means that we are looking at an edge that is part of the spanning tree we are considering. Thus, Ψ_s and $\Psi_{s'}$ are almost identical, except that $\Psi_{s'}$ contains one more t_i which means that $\mathcal{E}(\Psi_{s'}) = \mathcal{E}(\Psi_s) - \mathcal{B}(t_i) + \mathcal{F}(t_i)$. We deduce that $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(t_i) + \mathcal{F}(t_i)$. □

[1]This lemma also indirectly shows that the choice of spanning tree is not important.

The predicate isRegion(r) characterises regions in a linear-algebraic fashion. The question whether a given region solves a given separation problem can also be expressed algebraically: A region $r \in \mathbb{N}^{2n+1}$ solves a given ESSP instance (s, t_i) by Definition 3.2.1 if $\text{tokens}(r, s) < r_i$, while an SSP instance $\{s, s'\}$ is solved when $\text{tokens}(r, s) \neq \text{tokens}(r, s')$ holds. Thus, it makes sense to define the inequality $\text{SP}(r, pr)$ that characterises when a region r solves a separation problem $pr \in \text{SP}_A$ as:

$$\text{SP}(r, pr) := \begin{cases} (\text{tokens}(r, s) < r_i) & \text{if } pr = (r, t_i) \in \text{ESSP}_A \\ (\text{tokens}(r, s) \neq \text{tokens}(r, s')) & \text{if } pr = \{s, s'\} \in \text{SSP}_A \end{cases}$$

This definition allows to formulate a first generic algorithm for Petri net synthesis. This algorithm considers each separation problem individually, formulates it as a linear inequality system in the way outlined above, and collects regions from the solutions of these systems. It is shown in Algorithm 1.

Algorithm 1 General algorithm for Petri net synthesis [BBD15].

```
1:  procedure SynthesiseLts(A)                    ▷ A is finite and reachable
2:      Let R = ∅
3:      for pr ∈ SP_A do                          ▷ For each separation problem
4:          Find r ∈ ℕ^{2|Σ|+1} satisfying isRegion(r) ∧ SP(r, pr)
5:          if unsolvable then return error
6:          else Add region r to R
7:          end if
8:      end for
9:      return N(R)                    ▷ Petri net corresponding to the regions R
10: end procedure                      ▷ Its reachability graph will be isomorphic to A
```

This algorithm is correct, since by Lemma 3.3.2, isRegion(r) characterises regions while the characterisation of separation problems by $\text{SP}(r, pr)$ is derived directly from the definition of separation problems. Thus, the algorithm finds a set of regions solving each separation problem when such a set exists. By Theorem 3.2.2 such a set exists exactly if the lts is Petri net solvable.

The algorithm produces linear-algebraic problems of the form $\mathbb{A} \cdot x \geq b$ that must be solved[2]. Here, $\mathbb{A}$ is a given matrix over the integers, b is a vector of integers, and x is the natural vector being sought.

Example 3.3.3. As an example, consider the lts from Figure 3.1 from page 12. This lts is repeated in Figure 3.2, which also shows a spanning tree. This spanning tree assigns $\Psi_\imath = \Psi(\varepsilon)$, $\Psi_{s_1} = \Psi(a)$, $\Psi_{s_2} = \Psi(ab)$, and $\Psi_{s_3} = \Psi(b)$ as the reaching Parikh vector for each state (see Definition 3.3.1). In the following, an alphabetical order of events is assumed, i.e. a region is represented by $r_0 = \mathcal{R}(\imath)$, $r_1 = \mathcal{B}(a)$, $r_2 = \mathcal{B}(b)$, $r_3 = \mathcal{B}(c)$, $r_4 = \mathcal{F}(a)$,

[2] An equality $y = 0$ can be expressed as two inequalities $y \geq 0 \wedge -y \geq 0$. The inequality $\text{tokens}(r, s) \neq \text{tokens}(r, s')$ in the definition of $\text{SP}(r, pr)$ can be strengthened to $>$, because either a region or its complement will satisfy this requirement (see Lemma 3.1.6).

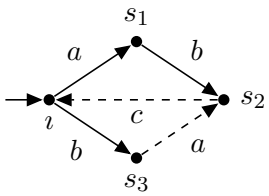


Figure 3.2.: A spanning tree (in solid) of the lts from Figure 3.1. Other edges are dashed.

$r_5 = \mathcal{F}(b)$, and $r_6 = \mathcal{F}(c)$. For example, this means that for a given vector $r \in \mathbb{N}^7$, the function tokens(r, s_1) evaluates to tokens$(r, s_1) = r_0 + 1 \cdot (r_4 - r_1) + 0 \cdot (r_5 - r_2) + 0 \cdot (r_6 - r_3)$, because w_{s_1} has one a, but no b or c.

The first part of isRegion(r) requires that $r_i \leq$ tokens(r, s) for each edge $(s, t_i, s') \in \rightarrow$, resulting in the following inequalities:

For $(\imath, a, s_1)$:	$r_1 \leq r_0$
For $(\imath, b, s_3)$:	$r_2 \leq r_0$
For (s_1, b, s_2):	$r_2 \leq r_0 - r_1 + r_4$
For (s_3, a, s_2):	$r_1 \leq r_0 - r_2 + r_5$
For $(s_2, c, \imath)$:	$r_3 \leq r_0 - r_1 + r_4 - r_2 + r_5$

The second part of isRegion(r) considers edges that are not part of the spanning tree. For example, the edge $(s_2, c, \imath)$ results in the equality tokens$(r, s_2) - r_3 + r_6 =$ tokens$(r, \imath)$. Expanding the definition of tokens, this results in $r_0 + 1 \cdot (r_4 - r_1) + 1 \cdot (r_5 - r_2) + 0 \cdot (r_6 - r_3) - r_3 + r_6 = r_0$. Since r_0 appears on both sides, the inequality can be simplified. In more abstract terms, this inequality says that the effect of *abc* should be zero, i.e. going through this cycle may not change the number of tokens.

For the edge (s_3, a, s_2), the equality is tokens$(r, s_3) - r_1 + r_4 =$ tokens(r, s_2). After expansion, the same formula appears on both sides of the equality sign. This will become more prominent in the following matrix representation, because it results in rows containing only zeros.

The following equations are constructed:

For (s_3, a, s_2):	$r_0 - r_2 + r_5 - r_1 + r_4 = r_0 - r_1 + r_4 - r_2 + r_5$
For $(s_2, c, \imath)$:	$r_0 - r_1 + r_4 - r_2 + r_5 - r_3 + r_6 = r_0$

All these inequalities and equations can be collected into a matrix $\mathbb{A}$ so that $\mathbb{A} \cdot r \geq 0$ has the same meaning as the individual inequalities. This results in the following system, where rows in the matrix are ordered as above and each equality $y = 0$ is replaced with the two inequalities $y \geq 0$ and $-y \geq 0$:

$$\begin{array}{r} (\iota,a,s_1): \\ (\iota,b,s_3): \\ (s_1,b,s_2): \\ (s_3,a,s_2): \\ (s_2,c,\iota): \\ (s_3,a,s_2): \\ -(s_3,a,s_2): \\ (s_2,c,\iota): \\ -(s_2,c,\iota): \end{array} \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 0 & 0 & 1 & 0 \\ 1 & -1 & -1 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & -1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & -1 & -1 & -1 \end{pmatrix} \cdot r \geq \mathbf{0}$$

This inequality system describes the predicate isRegion(r). A solution to a separation problem pr can now be computed by solving isRegion(r) $\wedge$ SP(r, pr). For example, the event/state separation problem (ι, c) corresponds to the inequality $r_0 \leq r_3$ and is solved by the region $r_1 = (0, 0, 0, 1, 0, 1, 0)$ from Figure 3.1 on page 12.

3.4. Bibliographical Remarks

This chapter introduced region theory and presented an algorithm for Petri net synthesis. A region of an lts describes a potential place of a Petri net that solves this lts. A condition, based on so-called separation problems, was identified that guarantees that the Petri net corresponding to a set of regions solves a given lts. This allowed to formulate an algorithm for Petri net synthesis. This algorithm computes for each separation problem a region that solves it.

The concepts introduced in this chapter are well known in the literature. This presentation is based on [BBD15]. The concept of a region originated in the work by Ehrenfeucht and Rozenberg on representation theorems for partial (set) 2-structures [ER90a; ER90b]. This led to the investigation of morphisms between transition systems and elementary[3] nets [NRT92], i.e. investigations related to category theory. The concepts were generalised to Petri nets that are not 1-bounded [Muk92], and Petri nets with different kinds of flows [BD96], so called *types of nets*. The introduction of separation problems [DR96] was an important step towards algorithms for Petri net synthesis.

The development of the first practical algorithm for elementary net synthesis [CKLY98; CKLY95] lead to the development of the tool Petrify [Cor+99]. This tool is tailored for applications in circuit design [Cor+97]. The resulting Petri net is used to represent the current state of the circuit, where a place with a token represents a logical 1 for the corresponding signal. It also allows to require properties like pure or free-choice from the resulting Petri net.

Other tools for Petri net synthesis are Synet and Genet. These both produce Petri nets instead of the elementary nets that Petrify produces. Synet [Cai99] can also produce

[3] Elementary nets are equivalent to plain, pure, and 1-bounded Petri nets.

pure Petri nets and allows to specify so-called locations that further restrict the structure of the resulting Petri net [BCD02]. Genet [CCK09] specifically targets k-bounded Petri nets and is based on a novel algorithm that computes all minimal k-bounded regions [CCK10; Car+08; CCK08]. Also, Genet supports a restriction to marked graphs and can over-approximate its input if no exact solution is possible.

The above already suggests an interest in targeting specific subclasses of Petri nets, e.g. k-bounded or pure. This was mainly done in the context of tool development, but also in the general integer-linear-programming-based approach from [CGS07] and [WDHS08], which come from a process-discovery context. Targeting specific subclasses will be the topic of the next chapter.

Best and Devillers developed dedicated algorithms for targeted synthesis of marked graphs, T-nets and choice-free Petri nets [BD14; BD15b; BD15a]. These algorithms simplify the inequality systems that have to be solved, and for connected marked graphs they even allow to eliminate these systems completely, which leads to very efficient algorithms. Similar results exist for Petri net synthesis from binary words and lts with an alphabet of size two [BESW16; EW17; Ero18].

The results mentioned so far considered finite labelled transition systems that should be solved up to isomorphism. However, there are also other results. Darondeau investigated the synthesis problem for regular and context-free languages [Dar98], where the language[4] of the synthesised net should be equal to a given input. This approach also allows to synthesise unbounded Petri nets, i.e. Petri nets with infinitely many reachable markings, thus removing the finiteness precondition [Dar03]. [LMJ07] is a survey on Petri net synthesis from languages. There are also results on synthesis for context-free graphs up to isomorphism [Dar01]. This approach was extended for path-automatic specifications [BD04] and also allows to produce unbounded Petri nets. Synthesis of languages is the topic of Section 5.5.

The state of the art was recently summarised by Badouel, Bernardinello, and Darondeau in [BBD15].

[4]The language contains all initially enabled sequences, i.e. $\{w \in \Sigma^* \mid M_0[w\rangle\}$.

4. Synthesis for Subclasses of Petri Nets

The previous chapter introduced an algorithm for Petri net synthesis. This algorithm computes a Petri net solution to a given lts, meaning that its input is an lts A and its output is a Petri net N with $A = \mathrm{RG}(N(R))$.

The present chapter will introduce a variant of this algorithm doing targeted synthesis. This means that, instead of asking for any Petri net solution, a subclass of nets is targeted. For example, plain Petri nets could be such a subclass, which means that no weights may be present and each flow may have at most weight one.

For example, the tool Petrify [Cor+99] produces plain, pure, and 1-bounded Petri nets. It has applications in circuit design, where the desired behaviour of a circuit is used as specification. The Petri net synthesized by Petrify provides a possible encoding of the internal states of the circuit, which means that a token on a place indicates that the corresponding signal is activated. Other tools can target different subclasses of Petri nets, but there is so far no completely generic approach to subclasses. What is missing is a systematic approach, starting from a synthesis algorithm that allows the entire class of P/T-nets and deriving an algorithm that can be used for a large class of targets, and which may be useful in a variety of circumstances.

To accomplish targeted synthesis, the algorithm from the last chapter is extended with another parameter, additionalProperties, which provides further inequalities that are added to each inequality system. The resulting algorithm is shown in Algorithm 2. The difference to Algorithm 1 is the addition of the predicate additionalProperties to the arguments and Line 4.

This chapter is based on the author's publications [BS15; Sch16b; SS17].

Algorithm 2 A variant of Algorithm 1 allowing targeted synthesis.

```
1: procedure SynthesiseLts(A, additionalProperties)      ▷ A is finite and reachable
2:     Let R = ∅
3:     for pr ∈ SP_A do                                  ▷ For each separation problem
4:         Find r satisfying isRegion(r) ∧ SP(r, pr) ∧ additionalProperties(r)
5:         if unsolvable then return error
6:         else Add region r to R  end if
7:     end for
8:     return N(R)                          ▷ Petri net corresponding to the regions R
9: end procedure                          ▷ Its reachability graph will be isomorphic to A
```

4.1. Targeted Subclasses

In this section, various subclasses with corresponding predicates are identified. The predicates are for individual places/regions, because the algorithm computes a Petri net one place/region at a time, These predicates restrict the inequality system so that only places/regions belonging to the subclasses are found. This also restricts the Petri net $N(R)$ that is computed, because each region corresponds to a place. However, only subclasses that can be checked on individual regions/places can be targeted this way. We will later see examples for inexpressible subclasses that are not definable on single places.

The predicates can be combined arbitrarily to produce the additionalProperties parameter of the algorithm. For example, by conjunctively combining the individual predicates, the subclass of plain, pure, and 1-bounded Petri nets that was mentioned above can be targeted.

The basic algorithm produces a linear inequality system that has to be solved in the natural numbers. Here, we generalise to a *satisfiability modulo theories* problem for the theory of integers. This means that first-order formulas can now be used, which allows, for example, disjunction and quantification in the predicates.

4.1.1. Simple Subclasses

The following subclasses can be expressed by predicates directly:

Plain A Petri net is called *plain* if its flows have only weights zero or one. Formally, this means that the flow function F of the Petri net satisfies $\mathrm{cod}(F) \subseteq \{0, 1\}$. The corresponding predicate enforces an upper limit of one for each flow weight of a region:

$$\mathrm{isPlain}(r) := \bigwedge_{i=1}^{2n} r_i \leq 1$$

Pure A Petri net is called *pure* if each of its places is pure. A place p is *pure* if no transition is in both its preset and its postset, i.e. ${}^\bullet p \cap p^\bullet = \emptyset$. Thus, at least one of the two flow weights connecting a transition with p has to be zero:

$$\mathrm{isPure}(r) := \bigwedge_{i=1}^{n} (r_i = 0 \vee r_{n+i} = 0)$$

Conflict-free A *conflict-free* [LR78] Petri net is plain and satisfies additionally $\forall p \in P\colon |p^\bullet| > 1 \Rightarrow p^\bullet \subseteq {}^\bullet p$. This means that for each place, there is either at most a single transition t_i consuming tokens from it, or all transitions t_i consuming from p must also produce a token there ($r_i \leq r_{i+n}$).

$$\text{isCF}(r) := \text{isPlain}(r) \wedge \left(\sum_{i=1}^{n} r_i \leq 1 \vee \bigwedge_{i=1}^{n} r_i \leq r_{i+n} \right)$$

Homogeneous In a *homogeneous* Petri net [JCL04; HDK15], the outgoing flows of each place p have the same weight, i.e. $\forall t_1, t_2 \in p^\bullet\colon F(p,t_1) = F(p,t_2)$. In the predicate for a region, this is expressed as pairs of weights being either zero or equal to each other:

$$\text{isHomogeneous}(r) := \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (r_i = 0 \vee r_j = 0 \vee r_i = r_j)$$

Generalised T-net In a generalised T-net each place $p \in P$ has at most one transition in its preset and at most one in its postset, i.e. $|{}^\bullet p| \leq 1 \geq |p^\bullet|$. So, the weights of all but one transition in each $\mathcal{B}$ and $\mathcal{F}$ (of the p-region) must be zero, which can be expressed as a disjunction of all-but-one flow weight sums being zero:

$$\text{isGTNet}(r) := \bigvee_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} r_j = 0 \wedge \bigvee_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} r_{j+n} = 0$$

Generalised marked graph In addition to the predicate for a generalised T-net, each place $p \in P$ must be connected to two transitions, i.e. $|{}^\bullet p| = 1 = |p^\bullet|$. In the predicate, we force the potentially non-zero weights to be truly positive:

$$\text{isGMGraph}(r) := \bigvee_{i=1}^{n} \left(r_i > 0 \wedge \sum_{\substack{j=1 \\ j\neq i}}^{n} r_j = 0 \right) \wedge \bigvee_{i=1}^{n} \left(r_{i+n} > 0 \wedge \sum_{\substack{j=1 \\ j\neq i}}^{n} r_{j+n} = 0 \right)$$

Standard T-net / marked graph combine the generalised subclass with plainness:

$$\text{isTNet}(r) := \text{isPlain}(r) \wedge \text{isGTNet}(r)$$
$$\text{isMGraph}(r) := \text{isPlain}(r) \wedge \text{isGMGraph}(r)$$

***k*-boundedness** The number of tokens on any place can never exceed $k \in \mathbb{N}$, i.e. $\forall M \in \mathfrak{E}(N)\colon \forall p \in P\colon M(p) \leq k$. In the lts, a reachable marking corresponds to a state $s \in S$, so we can simply use the existing function for counting tokens in a state:

$$\text{iskBounded}(r,k) := \bigwedge_{s\in S} \text{tokens}(r,s) \leq k$$

k-marking The initial number of tokens on any place is divisible by $k \in \mathbb{N}$:

$$\text{iskMarking}(r, k) := \exists \ell \in \mathbb{N} \colon r_0 = \ell \cdot k$$

Behavioural conflict-free (BCF) A Petri net is *behaviourally conflict-free* ([GGS11] calls this *semantically conflict-free*) if for every place p and every reachable marking activating two different transitions t_i and t_j, at most one of t_i and t_j consumes tokens from p, i.e. $\forall M \in \mathfrak{E}(N) \colon \forall t_i, t_j \in T \colon t_i \neq t_j \wedge M[t_i\rangle \wedge M[t_j\rangle \Rightarrow {}^\bullet t_i \cap {}^\bullet t_j = \emptyset$. To express this, we use the set $\text{EN} = \left\{ \{t_i, t_j\} \subseteq \Sigma \mid i \neq j \wedge \exists s \in S \colon s \xrightarrow{t_i} \wedge\, s \xrightarrow{t_j} \right\}$ of simultaneously enabled transitions and require that for each entry of EN at most one transition consumes tokens:

$$\text{isBCF}(r) := \bigwedge_{\{t_i, t_j\} \in \text{EN}} r_i = 0 \vee r_j = 0$$

Binary conflict-free (BiCF) The *binary conflict-free* [GGS11] subclass is similar to BCF. A Petri net is BiCF if, whenever t_i and t_j are both enabled, then each place in their common preset contains at least as many token as t_i and t_j consume together. We have to consider every state of the lts, because the number of tokens on each place is relevant. Define $\text{EN}' = \{(s, \{t_i, t_j\}) \in S \times 2^\Sigma \mid i \neq j \wedge s \xrightarrow{t_i} \wedge\, s \xrightarrow{t_j}\}$ to be the set of states with pairs of concurrently enabled transitions.

$$\text{isBiCF}(r) := \bigwedge_{(s, \{t_i, t_j\}) \in \text{EN}'} \text{tokens}(r, s) \geq r_i + r_j$$

Distributed according to loc A distributed Petri net can be thought of as a structural partition of the net into several locations. Locations can send messages to each other, but these might take an arbitrary time to reach their destination. Each location has a local state and cannot query the state of other locations. Based on [BD11], let $\texttt{LOCS}$ be some arbitrary, fixed, finite, and suitable large set of so-called locations. Given a mapping $\text{loc} \colon T \to \texttt{LOCS}$, a Petri net is *distributed according to* loc, if all $t_1, t_2 \in T$ with $\text{loc}(t_1) \neq \text{loc}(t_2)$ satisfy ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$ (no shared read access between locations). This means, each place/region belongs to one location. A place belonging to more than one location, or to none at all, would not allow any read access. This is also reflected in the following predicate, which requires some location $l \in \texttt{LOCS}$ to exist so that only transitions with this location consume tokens. The predicate's location-related parts can be evaluated statically and eliminated before giving the inequality system to a solver:

$$\text{isDistributed}(r, \text{loc}) := \bigvee_{l \in \texttt{LOCS}} \bigwedge_{i=1}^{n} (\text{loc}(t_i) \neq l \Rightarrow r_i = 0)$$

A similar approach is used in [BCD02], which deals specifically with the synthesis of distributable Petri nets.

Place-output-nonbranching (ON) This is defined as $|p^\bullet| \leq 1$ for all places $p \in P$, i.e. only one transition can consume tokens from each place. This can be expressed by putting each transition into its own location via the identity location mapping[1] with $\mathrm{id}(t) = t$:

$$\mathrm{isON}(r) := \mathrm{isDistributed}(r, \mathrm{id})$$

4.1.2. Non-Trivial Subclass: Equal-Conflict Petri Nets

The subclasses of Petri nets that were previously defined could all be easily expressed in a predicate. In this section, the more complicated subclass of equal-conflict Petri nets is analysed and a suitable predicate is defined.

An equal-conflict Petri net is both homogeneous and weighted free-choice: Homogeneous was already expressed as a predicate in the previous section and means that all outgoing flows of a place have the same weight. In a weighted free-choice Petri net, transitions with non-disjoint presets have the same preset.

Definition 4.1.1 (Equal-conflict)**.** *A Petri net $N = (P, T, F, M_0)$ is* weighted free-choice *if $\forall t_1, t_2 \in T\colon {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \Rightarrow {}^\bullet t_1 = {}^\bullet t_2$. N is an* equal-conflict Petri net *if it is both homogeneous and weighted free-choice.*

We cannot directly express the equal-conflict subclass in a predicate due to the weighted free-choiceness condition. It is possible to require one out of a set of chosen presets, as was e.g. done for BCF and distributed Petri nets. However, how do we know what the allowed presets for an equal-conflict Petri nets are? In the case of a distributed Petri net, the assignment of transitions to locations provided the information about allowed presets. For BCF nets the necessary information is derived from the structure of the lts. For weighted free-choice it is not immediately clear what the allowed presets are. If we simply require newly computed places to be consistent with earlier computed places, i.e. no unequal conflicts may be introduced, then it could not be guaranteed that a solution is found if one exists: A region that was found early, in a situation where many other regions would solve the given separation problem, could forbid the only possibility to solve a separation problem only considered later.

Thus, our approach has two phases, similar to the predicates for BCF and BiCF: First structural constraints on the desired Petri net are identified based on the lts. This corresponds to the calculation of the sets EN and EN′. Afterwards, this information is used to fix the set of possible postsets of a computed region in the predicate.

Figure 4.1 shows that equal-conflict Petri nets are less expressive than general Petri nets. It shows an lts A and a Petri net N so that $\mathrm{RG}(N) = A$, but no equal-conflict Petri net can generate A, as will be shown later. The Petri net N is not equal-conflict, for example, because the presets of transitions t_3 and t_2 are neither disjoint nor equal.

[1] Equivalently, one could use $\mathrm{isON}'(r) := \bigvee_{i=1}^{n} \sum_{j \in \{1,\ldots,n\} \setminus \{i\}}^{n} r_j = 0$, specifying that at most one transition consumes tokens.

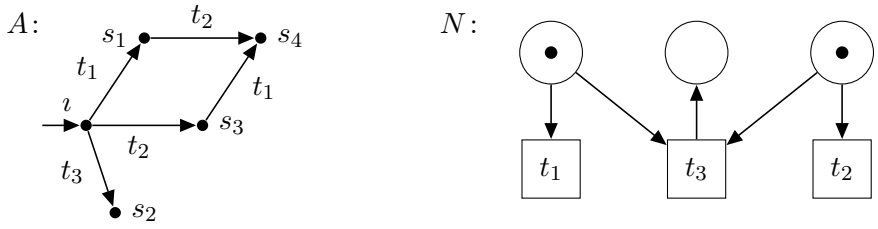


Figure 4.1.: An lts A which is not isomorphic to the reachability graph of any equal-conflict Petri net and a (non-equal-conflict) Petri net N with $\mathrm{RG}(N) = A$.

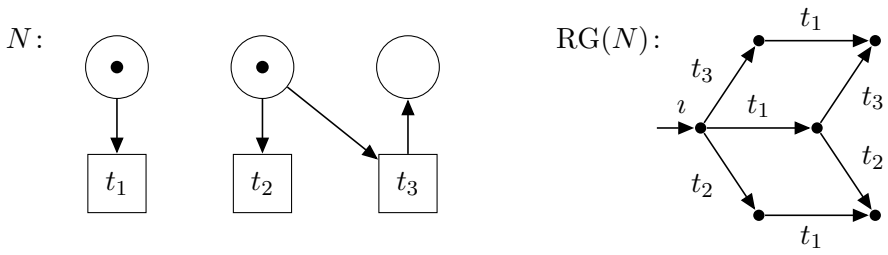


Figure 4.2.: An equal-conflict Petri net N and its reachability graph $\mathrm{RG}(N)$.

Enabling-Equivalent Transitions

This section investigates the following two properties of equal-conflict Petri nets:

Definition 4.1.2 (Preset-equal, enabling-equivalent)**.** *Two transitions $t_1, t_2 \in T$ of a Petri net N are* preset-equal *if they have the same preset, ${}^\bullet t_1 = {}^\bullet t_2$. Two labels $t_1, t_2 \in \Sigma$ of an lts A are* enabling-equivalent *if no state $s \in S$ only enables one of the two labels, i.e. $\forall s \in S\colon (s \xrightarrow{t_1} \Leftrightarrow s \xrightarrow{t_2})$.*

As an example of this definition, consider the Petri net N from Figure 4.2. The transition t_1 is not preset-equal with any other transition, but t_2 and t_3 are preset-equal. A similar relationship can be seen in $\mathrm{RG}(N)$: There are states where only t_1 is enabled, so it is not enabling-equivalent with any other label, but t_2 and t_3 are enabling-equivalent.

The following lemmas show that these two properties imply each other in equal-conflict Petri nets and their reachability graphs.

Lemma 4.1.3. *Let N be a homogeneous Petri net. Then two preset-equal transitions $t_1, t_2 \in T$ of N are enabling-equivalent in* $\mathrm{RG}(N)$.

Proof. Let $t_1, t_2 \in T$ be two preset-equal transitions. Since N is homogeneous, we get for each $p \in P$ that $F(p, t_1) = F(p, t_2)$. By the definition of enabledness, any given marking enables either both transitions or none of them. □

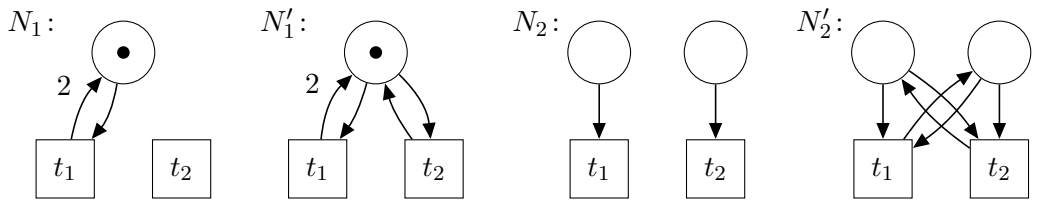


Figure 4.3.: Examples of the construction of Lemma 4.1.4.

This lemma allows us to show that the lts A from Figure 4.1 cannot be solved by an equal-conflict Petri net N. Assume for a contradiction that an equal-conflict Petri net N exists. We can see that t_1 and t_3 are not enabling-equivalent, because in s_3 only t_1 is enabled. By the previous lemma, they are not preset-equal in N. Since we are desiring an equal-conflict solution, they must thus have disjoint presets. However, in the initial state both t_1 and t_3 are enabled, but firing one of them disables the other. This is only possible if they have non-disjoint presets in N, hence we get a contradiction. This means that there is no equal-conflict Petri net that solves A.

For the opposite implication, namely deriving preset-equality from enabling-equivalence, we can only show a weaker result: There is a Petri net where enabling-equivalent transitions are preset-equal. Examples of this construction are provided in Figure 4.3. In the reachability graphs of N_1 and N_2, t_1 and t_2 are enabling-equivalent, but they are not preset-equal in the Petri nets. In N_1 both transitions are always enabled while in N_2 neither transition can fire. The Petri nets N_1' and N_2' are produced by the construction from the following Lemma.

Lemma 4.1.4. *Let N be an equal-conflict Petri net. Then there is an equal-conflict Petri net N' with $\mathrm{RG}(N) = \mathrm{RG}(N')$ so that enabling-equivalent transitions in $\mathrm{RG}(N')$ are preset-equal in N'.*

Proof. The following step can inductively be used to modify N so that any transitions, which are enabling-equivalent but not preset-equal, become preset-equal. The result of this construction will be the Petri net N'.

Let $t_1, t_2 \in T$ be enabling-equivalent, but not yet preset-equal. For any transition t_3 preset-equal with t_2 (including t_2 itself) and any place $p \in {}^\bullet t_1$, add a so-called side-condition between t_3 and p with weight $F(p, t_1)$: This means both $F(p, t_3)$ (previously zero by assumption) and $F(t_3, p)$ (possibly non-zero) are increased by $F(p, t_1)$. Also, do the analogous operation with t_1 and t_2 swapped. The resulting net will still be equal-conflict by construction, but it will also satisfy ${}^\bullet t_1 = {}^\bullet t_2$. In particular, this means that transitions that are not preset-equal have disjoint presets. Also, the behaviour of the Petri net was not modified, because the effects of transitions were not modified and no transition becomes disabled in a marking that previously enabled it, because by enabling-equivalence enough tokens for the added flows are available. □

By the previous two lemmas, we can restrict our attention to Petri nets where enabling-equivalent transitions are preset-equal, without incorrectly classifying some lts as unsolvable by an equal-conflict Petri net. This can be used to define a predicate on regions, as follows.

Let an lts A over the set of labels Σ be given that should be solved by an equal-conflict Petri net N with $A = \mathrm{RG}(N)$. First, the enabling-equivalent labels in A are computed. This produces a partitioning $E \subseteq 2^\Sigma$ of the set of labels into enabling-equivalent classes. Next, we say that a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is *compatible* with the desired equal-conflict Petri net $N = (P, \Sigma, F, M_0)$, if it is homogeneous, meaning that for all transitions $t_1, t_2 \in \Sigma$ if $\mathcal{B}(t_1) > 0 \wedge \mathcal{B}(t_2) > 0$ then $\mathcal{B}(t_1) = \mathcal{B}(t_2)$, and the postset of the corresponding place is in $E \cup \{\emptyset\}$, i.e. we additionally allow the empty postset. These two conditions (homogeneous and postset in E) are expressed by the following predicate:

$$\mathrm{isEC}(r) := \bigvee_{e \in E \cup \{\emptyset\}} \left(\left(\bigwedge_{t_i, t_j \in e} r_i = r_j \wedge r_j > 0 \right) \wedge \left(\bigwedge_{t_i \notin e} r_i = 0 \right) \right)$$

Thus, we found a predicate to express equal-conflict Petri net synthesis.

Example 4.1.5. As an example of this predicate, let us return to the lts A from Figure 4.1. We already argued that this lts cannot be solved by an equal-conflict Petri net, and we will now look more formally at this problem. Since no two labels are enabling-equivalent, we have $E = \{\{t_1\}, \{t_2\}, \{t_3\}\}$. Our predicate indicates that at most one transition can consume tokens from any single place.

We will try to find a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ that solves the event/state separation problem (s_2, t_1), i.e. that satisfies $\mathcal{R}(s_2) < \mathcal{B}(t_1)$. From this formula we immediately conclude $0 < \mathcal{B}(t_1)$, because $\mathcal{R}(s_2) \in \mathbb{N}$. By definition of a region and the edge $\imath \xrightarrow{t_3} s_2$, we have $\mathcal{R}(\imath) - \mathcal{B}(t_3) + \mathcal{F}(t_3) = \mathcal{R}(s_2) < \mathcal{B}(t_1)$. Because of the edge $\imath \xrightarrow{t_1} s_1$ we know by definition of a region that $\mathcal{B}(t_1) \leq \mathcal{R}(\imath)$. Adding the last two inequalities produces $-\mathcal{B}(t_3) + \mathcal{F}(t_3) < 0$, which implies that $0 < \mathcal{B}(t_3)$, because these values are non-negative. Thus, we have shown that $0 < \mathcal{B}(t_1)$ and $0 < \mathcal{B}(t_3)$, i.e. both t_1 and t_3 are in the preset of the sought place. This is not compatible with the predicate, because there is no $e \in E \cup \{\emptyset\}$ with $t_1, t_3 \in e$, since t_1 and t_3 are not enabling-equivalent at state s_3. This means that the event/state separation problem (s_2, t_1) is unsolvable and so A cannot be solved by an equal-conflict Petri net.

Theorem 4.1.6. *Given a finite lts A, the synthesis procedure produces an equal-conflict Petri net $N(R)$ with $\mathrm{RG}(N(R)) = A$ if and only if there is an equal-conflict Petri net N with $\mathrm{RG}(N) = A$.*

Proof. ($\Rightarrow$): If the algorithm finds a Petri net $N(R)$, then each place of $N(R)$ corresponds to a region satisfying the predicate. This means that $N(R)$ is homogeneous and the postset of each place is in $E \cup \{\emptyset\}$. Since elements of E are disjoint, $N(R)$ is an

equal-conflict Petri net. Since all separation problems were solved by the algorithm, $\mathrm{RG}(N(R)) = A$ by Theorem 3.2.2.

($\Leftarrow$): To show that a solution is always found if one exists, assume a suitable Petri net N exists. By Lemma 4.1.4 we can assume that enabling-equivalent transitions in $\mathrm{RG}(N)$ are preset-equal in N. The contraposition of Lemma 4.1.3 shows that non-enabling-equivalent transitions must not be preset-equal, which means that their presets must be disjoint by weighted free-choice. Thus, the extensions $[[p]]$ of the places $p \in P$ of N satisfy the above predicate and are a possible result of the algorithm. □

Incorporating Additional Subclasses

So far we can synthesise equal-conflict Petri nets via the new predicate. However, the algorithm allows to combine multiple predicates so that, for example, pure equal-conflict Petri nets are found. In this section, we show that these predicates can be combined with the new predicate for equal-conflict synthesis.

Lemma 4.1.4 is not applicable for synthesising pure equal-conflict Petri nets, because it only shows the existence of an equivalent non-pure Petri net, as the example N_2 and N_2' in Figure 4.3 demonstrates. Additionally, Figure 4.4 has an example N_3 and N_3' showing that plainness is not preserved, either. All subclasses from Section 4.1, except for plain and pure, are unaffected by the construction from Lemma 4.1.4, or are already more restrictive than equal-conflictness. The following Theorem 4.1.7 closes this gap. Examples of its constructions are given in Figure 4.4. The Petri nets N_3 and N_4 have enabling-equivalent transitions t_1 and t_2 that are not preset-equal. The transitions in N_3 both cannot fire, while in N_4 the transitions are enabled after t_3 fired. While N_3 is plain and pure, the construction from Lemma 4.1.4 would produce N_3', which is neither plain nor pure. Instead, Theorem 4.1.7 preserves these subclasses and produces the Petri net N_3''. Similarly, N_4 is transformed into N_4'' and both nets are plain.

Theorem 4.1.7. *Let N be a bounded equal-conflict Petri net that is pure or plain. Then there is an equal-conflict Petri net N' with $\mathrm{RG}(N) = \mathrm{RG}(N')$ so that enabling-equivalent transitions in $\mathrm{RG}(N')$ are preset-equal in N'. Additionally, N' is pure if N is pure, and plain if N is plain.*

Proof. First we modify the transitions which can never occur, so that we can assume that all remaining transitions fire in at least one reachable marking. Let $D \subseteq T$ be the set of all such dead transitions. We construct N' by removing all flows connected to a transition in D. Then, we add a new place p with $M_0(p) = 0$ and add flows so that $\forall t \in D\colon F(p,t) = 1$. This place ensures that all transitions in D cannot occur. Because dead transitions do not influence the behaviour of the Petri net, we have $\mathrm{RG}(N) = \mathrm{RG}(N')$. Also, this construction preserves plainness and pureness.

Next, let $T' \subseteq T$ be a set of pairwise enabling-equivalent transitions containing $t_1, t_2 \in T'$ that are not preset-equal. By simple liveness (each transition fires in some reachable

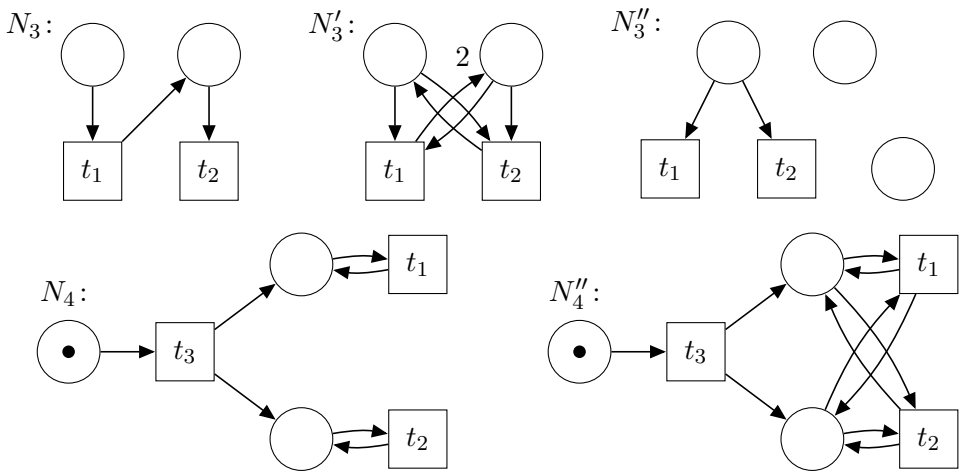


Figure 4.4.: Motivating examples for the construction of Theorem 4.1.7.

marking), there are two cases: Either there are reachable markings M, M' with $M[t_1\rangle M'$ and $\neg M'[t_1\rangle$, or t_1 can be fired infinitely often once it is enabled.

In the first case, by enabling-equivalence we get $M[t_2\rangle$ and $\neg M'[t_2\rangle$. Thus, firing t_1 disables t_2 and by the firing rule we deduce ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. Since N is weighted free-choice, this implies that ${}^\bullet t_1 = {}^\bullet t_2$. Since t_2 was arbitrary, all transitions that are enabling-equivalent with t_1 are in fact preset-equal with it, so the conclusion already holds for the given Petri net N itself.

In the latter case, by the firing rule we have[2] $\forall p \in P\colon F(t_1, p) \geq F(p, t_1)$. If N is pure (and possibly plain), it follows that $F(p, t_1) = 0$ and so t_1 has an empty preset. By enabling-equivalence we get analogously ${}^\bullet t_2 = \emptyset$ and hence ${}^\bullet t_1 = \emptyset = {}^\bullet t_2$. We can choose $N' = N$ again.

If N is plain, then by boundedness we get $\forall p \in P\colon F(t_1, p) = F(p, t_1) \wedge F(t_2, p) = F(p, t_2)$. By plainness, the only option for t_1 and t_2 not to be preset-equal is if (without loss of generality) $F(t_1, p) = 1$ and $F(t_2, p) = 0$ for some place $p \in P$. Let $T'' \subseteq T'$ contain all transitions $t \in T'$ with $F(t, p) = 0$. We can add a side-condition between each $t \in T''$ and p of weight one without modifying the behaviour of the Petri net, because by enabling-equivalence with t_1 there is always a token in p when $t \in T''$ is enabled in N. This produces the plain and equal-conflict Petri net N' with $\mathrm{RG}(N') = \mathrm{RG}(N)$. □

4.2. An Unsupported Subclass

Many structural subclasses of a Petri net, such as plainness and pureness, but also behavioural subclasses like k-boundedness, were expressed as a predicate already. However,

[2] We could even deduce equality from boundedness.

there are subclasses for which this treatment is not possible, and the synthesis procedure that was introduced cannot be targeted to them. An example of such a subclass are P-nets, which are dual to T-nets: In a generalised P-net, each transition $t \in T$ has at most one place in its preset and at most one place in its postset. This means that each transition $t \in T$ satisfies $|t^\bullet| \leq 1 \geq |{}^\bullet t|$.

The algorithm that was introduced calculates individual regions, which correspond to single places of the final Petri net solution. Thus, only subclasses that can be checked locally on individual places can be expressed as predicates for the algorithm. Since the P-net subclass cannot be checked locally on places, but instead requires the complete surroundings of transitions, no predicate for P-nets can be defined directly.

Examples of other subclasses that cannot easily be expressed are graph-properties like weak/strong connectedness. All of these subclasses cannot be decided on a single place, i.e. they are not local to places, but instead require the surrounding of transitions or even the full Petri net.

Equal-conflictness is not local to single places, either, and thus by the above argument it should not be expressible. Still, in the previous section, a way to overcome this problem was found. So, it might also be possible to incorporate other subclasses. For example, weak connectivity in a non-pure Petri net can always be ensured by adding an otherwise useless new place with a side-condition to every transition.

4.3. Conclusion

The previous chapter presented an algorithm for Petri net synthesis, which means that for a given lts, a Petri net should be computed whose reachable graph is isomorphic to the lts. This chapter extended the algorithm for *targeted synthesis*: Instead of computing any Petri net solution, the Petri net should be from a specific subclass of Petri nets. We identified predicates to express the subclasses plain, pure, conflict-free, homogeneous, generalised T-net, generalised marked graph, k-bounded, k-marking, behaviourally conflict-free, binary conflict-free, distributed, and equal-conflict. Most of these subclasses were easy to incorporate, but the case of equal-conflict Petri nets was more complicated, because it required a preprocessing step that computes a structural property of the desired Petri net based on the given lts. These subclasses can be combined, for example, allowing the synthesis of free-choice Petri nets, which are plain and equal-conflict Petri nets. Also, the subclass of P-nets was identified as being inexpressible for the algorithm. A similar observation for the same subclass of Petri nets was made in [WDHS08].

An open question for the algorithm is its complexity. While Petri net synthesis for general and pure Petri nets is possible in polynomial time [BBD95; BBD15], elementary net synthesis is NP-complete [BBD97]. Since elementary nets are equivalent to plain, pure,

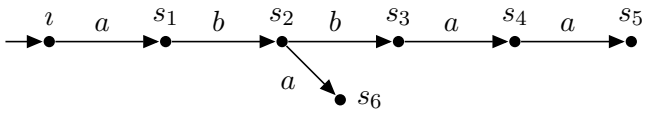


Figure 4.5.: An lts that can be solved exactly, but only if non-minimal regions are considered.

and 1-bounded Petri nets, this means that the complexity of the presented algorithm depends on the specific subclass being targeted.

A construction similar to Lemma 4.1.4 is called *equalisation* in [RTS97], but used in a different context. In [CKLY98] the synthesis of free-choice Petri nets is handled by label splitting, which means that in the resulting Petri net, different transitions might produce the same label, which our approach avoids. Another algorithm is sketched in [WDHS08]. This is a recursive approach based on integer linear programming (ILP) where the computation of a new place, which would violate the free-choice condition, is handled by discarding the already-found places. The authors point out that their approach negatively influences running times since places are computed repeatedly.

The tool Genet is based on [CCK10; Car+08; CCK08]. Genet can synthesise k-bounded Petri nets, but the algorithm does not guarantee success, which means that it can fail to find a Petri net solution even though one exists. This is because it only computes minimal regions. A region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is *minimal*, if there is no other non-trivial region $(\mathcal{R}', \mathcal{B}', \mathcal{F}')$ with either $\mathcal{R}' \leq \mathcal{R}$ (pointwise), or $\mathcal{R}' = \mathcal{R}$ and $\mathcal{B}' \geq \mathcal{B}$ and $\mathcal{F}' \geq \mathcal{F}$ (also pointwise). Here, non-trivial means that there is some $s \in S$ with $\mathcal{R}(s) \neq 0$ and intuitively a region is smaller than another region if it assigns less tokens to states, or, if the token assignment is the same, then the number of side-conditions[3] is higher. Figure 4.5 contains an lts, which cannot be solved when just using minimal regions. The following list contains all six minimal regions of this lts, where a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ is identified with the vector of vectors[4] $((\mathcal{R}(\imath), \mathcal{R}(s_1), \ldots, \mathcal{R}(s_6)), (\mathcal{B}(a), \mathcal{B}(b)), (\mathcal{F}(a), \mathcal{F}(b)))$:

$$r_1 = ((1,0,1,2,1,0,0),(1,0),(0,1)) \qquad r_2 = ((2,2,1,0,0,0,1),(0,1),(0,0))$$
$$r_3 = ((1,2,1,0,1,2,2),(0,1),(1,0)) \qquad r_4 = ((0,2,1,0,2,4,3),(0,1),(2,0))$$
$$r_5 = ((0,1,1,1,2,3,2),(0,1),(1,1)) \qquad r_6 = ((0,0,1,2,2,2,1),(0,0),(0,1))$$

It can easily be verified that for all of these regions we have $\mathcal{R}(s_6) \geq \mathcal{B}(b)$, which means that none of these regions solves the ESSP instance (s_6, b). However, the region $r_7 = ((3,2,2,2,1,0,1),(1,2),(0,2))$ does prevent b in s_6 ($\mathcal{R}(s_6) = 1 < 2 = \mathcal{B}(b)$). This region is not minimal because it is larger than r_1, but only with this region can the lts from Figure 4.5 be solved. Thus, Genet incorrectly declares the lts unsolvable.

[3] By the requirement on the token assignment, the effect $\mathcal{E} = \mathcal{F} - \mathcal{B}$ will be the same between both regions. The effect stays the same if both $\mathcal{F}$ and $\mathcal{B}$ are increased by one, which represents the addition of one side-condition.

[4] For readability, this does not use the vector representation that was introduced previously.

5. Minimal Over-Approximations

In the previous chapters, a given lts was synthesised into a Petri net up to isomorphism. When some lts was unsolvable, synthesis failed.

In this chapter, a possible way to handle failures is investigated. Instead of failing synthesis, the lts is modified and amended so that it can be solved by a Petri net. We will see that this is possible in a minimal way, where minimality is understood according to a structural preorder called lts homomorphism, which is introduced next.

This chapter is based on the author's publication [Sch18].

5.1. LTS Homomorphisms

To do minimally over-approximative synthesis into Petri nets, we first need to define the notion according to which minimality is understood. This will be lts homomorphisms (e.g. [AM98; BBD15]), which are presented in this section.

The simulation preorder (e.g. [Arn94]) between lts is well-known: A simulation is a relation between states of two lts. When the first of two related states has an outgoing edge with label a, the second state must have a suitable a-edge as well, so that the states that are reached are in the relation again. An lts homomorphism is similar to a simulation relation, but a function is used instead of a relation. This means that each state of the first lts must have a simulating state in the second lts, and that this state must be unique. As we will see later, this allows to transfer regions from one of the lts to the other.

Definition 5.1.1 (Lts morphism [SW17; AD93])**.** *An* lts homomorphism from lts A_1 to lts A_2 *with* $A_i = (S_i, \Sigma, \to_i, \imath_i)$ *is a function* $f\colon S_1 \to S_2$ *so that* $f(\imath_1) = \imath_2$ *and* $\forall (s,t,s') \in \to_1$ *also* $(f(s), t, f(s')) \in \to_2$*. If such a function* f *exists, write* $A_1 \sqsubseteq A_2$ *(via* f*).*

For example, the two lts in Figure 5.1 can simulate each other in the classic sense of simulation preorder. The initial states $\imath$ and $\imath'$ are related to each other and so are the states s_1 and s_1' reached via a. Both states in A_1 that can be reached via b (s_2 and s_3) are related to s_2' in A_2, which is also reached via b. Thus, a possible simulation relation is $\{(\imath, \imath'), (s_1, s_1'), (s_2, s_2'), (s_3, s_2')\}$. Since the edges between related states are present in both directions, this relation can be inverted by swapping the first and second element of all pairs[1].

[1] Thus, this is in fact a bisimulation [Arn94].

Figure 5.1.: Two lts that simulate each other, but an lts homomorphism only exists in one direction.

However, an lts homomorphism only exists from A_1 to A_2. The relation that was just constructed is left-total and right-unique, thus a function and an lts homomorphism. When trying to find an lts homomorphism from A_2 to A_1, the single state s'_2 reached via b in A_2 must be mapped to two different states s_2 and s_3 in A_2, which is not possible.

Similar to the simulation preorder, an lts homomorphism is also a preorder:

Lemma 5.1.2 ($\sqsubseteq$ is a preorder [SW17])**.** *The relation $\sqsubseteq$ between lts is reflexive and transitive.*

Proof. Reflexivity means that for all lts $A \sqsubseteq A$ holds. This holds with the identity homomorphism id. For transitivity we have to show that $A_1 \sqsubseteq A_2 \sqsubseteq A_3$ implies $A_1 \sqsubseteq A_3$. Let f_1 be the lts homomorphism witnessing $A_1 \sqsubseteq A_2$ and f_2 the function for $A_2 \sqsubseteq A_3$. It can easily be seen that[2] $f_1 \circ f_2$ witnesses $A_1 \sqsubseteq A_3$ since homomorphisms are closed under composition. □

Another observation about lts homomorphisms is their relation to isomorphisms:

Lemma 5.1.3 ([SW17])**.** *Two lts A_1 and A_2 are isomorphic via f if and only if f and f^{-1} are lts homomorphisms.*

Proof. By definition of isomorphisms (Def. 2.0.13) and homomorphisms (Def. 5.1.1). □

By adding more constraints on the lts than just the existence of an lts homomorphism, stronger results can be shown. The following lemma deals with a case where the lts homomorphism is unique:

Lemma 5.1.4 ([SW17])**.** *Let A_1 and A_2 be lts with $A_i = (S_i, \Sigma, \rightarrow_i, \imath_i)$ so that $A_1 \sqsubseteq A_2$ via f and $A_1 \sqsubseteq A_2$ via f'. If A_1 is reachable and A_2 is deterministic, then $f = f'$.*

Proof. We prove, by induction on the length of words $w \in \Sigma^*$, that if $\imath_1 \xrightarrow{w} s$, then $f(s) = f'(s)$. Since A_1 is reachable, we reach all states of A_1 in this way, showing that $f(s) = f'(s)$ for all $s \in S_1$. The induction basis follows from the definition of $\sqsubseteq$: $f(\imath_1) = \imath_2 = f'(\imath_1)$.

For the induction step, assume that $\imath_1 \xrightarrow{w} s$ with $f(s) = f'(s)$ and consider any edge $s \xrightarrow{a} s'$. Since $A_1 \sqsubseteq A_2$ via f and f', we have $f(s) \xrightarrow{a} f(s')$ and $f'(s) \xrightarrow{a} f'(s')$.

[2] In this this document, function composition is defined as $(f \circ g)(x) = g(f(x))$.

Because A_2 is deterministic, there cannot be two different states that are reached from $f(s) = f'(s)$ via label a. We conclude $f(s') = f'(s')$. □

Lemma 5.1.2 showed that $\sqsubseteq$ is a preorder. The next lemma strengthens this by showing that $\sqsubseteq$ is a partial order for reachable and deterministic lts. For this, antisymmetry is missing and has to be shown.

Lemma 5.1.5 ([SW17])**.** *Let A_1 and A_2 be reachable and deterministic lts so that $A_1 \sqsubseteq A_2$ via f_1 and $A_2 \sqsubseteq A_1$ via f_2. Then A_1 and A_2 are isomorphic.*

Proof. Consider $A_1 \sqsubseteq A_1$ (via id) and $A_1 \sqsubseteq A_2 \sqsubseteq A_1$ via $f_1 \circ f_2$. Since Lemma 5.1.4 is applicable, we conclude $f_1 \circ f_2 = \mathrm{id}$. Now, f_1 must be injective, since otherwise f_2 could not recover the original element. Similarly, f_2 must be surjective, since every element appears in its image. With an analogous argument for $f_2 \circ f_1 = \mathrm{id}$ we see that f_1 and f_2 are both bijective. Since their composition is the identity function, they are each other's inverses, i.e. $f_1^{-1} = f_2$. By Lemma 5.1.3, this means that A_1 and A_2 are isomorphic. □

In the beginning of this section, it was announced that lts homomorphisms allow to translate regions to smaller lts. This is formalised in the next lemma.

Lemma 5.1.6 ([SW17])**.** *Let A_1 and A_2 be lts with $A_1 \sqsubseteq A_2$ via f. If $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ is a region of A_2, then $f \circ r := (f \circ \mathcal{R}, \mathcal{B}, \mathcal{F})$ is a region of A_1.*

Proof. For an edge $s \xrightarrow{t} s'$ of A_1, $f(s) \xrightarrow{t} f(s')$ is an edge of A_2. Since r is a region of A_2, we now have $\mathcal{R}(f(s)) \geq \mathcal{B}(t)$ in A_2, which is the first condition for a region in A_1. For the second part the same argument is used: We have $\mathcal{R}(f(s')) = \mathcal{R}(f(s)) - \mathcal{B}(t) + \mathcal{F}(t)$, because r is a region of A_2 and this formula is needed to show that $f \circ r$ is a region of A_1. Thus, $(f \circ \mathcal{R}, \mathcal{B}, \mathcal{F})$ is a region of A_1. □

The last lemma of this section shows that every set of regions of an lts produces an lts larger than the original lts, according to lts homomorphisms.

Lemma 5.1.7. *Let A be a reachable lts and R a set of regions of A, then $A \sqsubseteq \mathrm{RG}(N(R))$.*

Proof. Let $A = (S, \Sigma, \to, \imath)$ and define a function $f\colon S \to \mathfrak{E}(N(R))$ via $f(s)((\mathcal{R}, \mathcal{B}, \mathcal{F})) = \mathcal{R}(s)$ for each $(\mathcal{R}, \mathcal{B}, \mathcal{F}) \in R$. This functions maps the state s to the marking of $N(R)$ where each place/region $(\mathcal{R}, \mathcal{B}, \mathcal{F}) \in R$ is assigned $\mathcal{R}(s)$ tokens. We show that f is an lts homomorphism, i.e. satisfies $f(\imath) = M_0$ and $s \xrightarrow{a} s'$ implies $f(s)[a\rangle f(s')$.

By definition of the initial marking M_0 of $N(R)$, $\mathcal{R}(\imath) = M_0((\mathcal{R}, \mathcal{B}, \mathcal{F}))$ for all $(\mathcal{R}, \mathcal{B}, \mathcal{F}) \in R$, which implies that $f(\imath) = M_0$. If $s \xrightarrow{a} s'$ in A, then for any region $(\mathcal{R}, \mathcal{B}, \mathcal{F}) \in R$, by definition of a region $\mathcal{R}(s) \geq \mathcal{B}(a)$, i.e. transition a is enabled in the marking $f(s)$, and $\mathcal{R}(s') = \mathcal{R}(s) - \mathcal{B}(a) + \mathcal{F}(a)$, i.e. $f(s)[a\rangle f(s')$. Thus, f is an lts homomorphism. □

5.2. Existence of Minimal Petri Net Solvable Over-Approximations

Now that lts homomorphisms were introduced, the goal of this chapter can be stated more formally: In Theorem 5.2.2, it will be shown that, given a finite lts A, there is a minimal over-approximation by a finite and Petri net solvable lts $\mathrm{Approx}(A)$, meaning that there is a unique (up to isomorphism) lts larger than or equal to A that can be solved by a Petri net and is minimal according to $\sqsubseteq$ with this property.

Lemma 5.2.1. *For a finite lts A there is a set R of regions of A so that $N(R)$ is bounded and for any set $\widehat{R}$ of regions of A also $\mathrm{RG}(N(R)) \sqsubseteq \mathrm{RG}(N(\widehat{R}))$ holds.*

Proof. Let R' be the set of all regions of A. This might be an infinite set, so $N(R')$ might not be a (finite) Petri net, but for the moment we lift the restriction that a Petri net has only finitely many places. The firing rule and the construction of the reachability graph are not affected by this. $\mathrm{RG}(N(R'))$ is finite, because by Lemma 3.1.6, for each region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ of A, there is a complement region $(\overline{\mathcal{R}}, \overline{\mathcal{B}}, \overline{\mathcal{F}})$ so that $\mathcal{R}(s) + \overline{\mathcal{R}}(s)$ is constant. The same also holds for the corresponding places: The sum of their numbers of tokens stays constant when firing transitions. Thus, for each place of $N(R')$, there are only finitely many reachable numbers of tokens, which means that $N(R')$ is bounded and $\mathrm{RG}(N(R'))$ is finite.

$\mathrm{RG}(N(R'))$ is solvable by a finite Petri net: Because the lts is finite, it has only finitely many separation problems. Each separation problem is solvable by construction: If two markings are different, then some region/place differs in its number of tokens and so the corresponding SSP instance is solvable. If a transition is disabled in a given marking, there is some region/place that prevents the transition from firing, and thus solves the corresponding ESSP instance. Since the number of separation problems of a finite lts is finite, there is a finite set of regions R so that $N(R)$ solves $\mathrm{RG}(N(R'))$ by Theorem 3.2.2, which states that if all separation problems are solved by regions in R, then $\mathrm{RG}(N(R)) = \mathrm{RG}(N(R'))$.

Finally, for any set of regions $\widehat{R}$ of A also $\mathrm{RG}(N(R)) \sqsubseteq \mathrm{RG}(N(\widehat{R}))$ holds: Since R' contains all regions, we have $\widehat{R} \subseteq R'$. Define f to be the function that restricts markings of $N(R')$ to those regions/places present in $N(\widehat{R})$. This function is an lts homomorphism by construction, showing $\mathrm{RG}(N(R')) \sqsubseteq \mathrm{RG}(N(\widehat{R}))$. Since $\mathrm{RG}(N(R'))$ and $\mathrm{RG}(N(R))$ are isomorphic, it follows that $\mathrm{RG}(N(R)) \sqsubseteq \mathrm{RG}(N(\widehat{R}))$. □

The lts $\mathrm{RG}(N(R))$ from this lemma is the minimal Petri net solvable over-approximation $\mathrm{Approx}(A)$:

Theorem 5.2.2. *Given a finite and reachable lts A, there is a minimal Petri net solvable over-approximation $\mathrm{Approx}(A)$, that is finite, unique up to isomorphism, and satisfies $A \sqsubseteq \mathrm{Approx}(A)$. Minimality means that for any finite lts B with $A \sqsubseteq B$, which can be solved by a Petri net, also $\mathrm{Approx}(A) \sqsubseteq B$ holds.*

Proof. Let $\text{Approx}(A)$ be the finite lts $\text{RG}(N(R))$ as defined in the previous lemma. By Lemma 5.1.7, $A \sqsubseteq \text{Approx}(A)$ holds. Now assume some lts B with $A \sqsubseteq B$ so that B is solved by some Petri net N. Let R' be the set of regions containing for each place p of N its extension $[[p]]$ (see Lemma 3.1.8). By definition of the corresponding Petri net (Definition 3.1.7), we have $N(R') = N$ and by assumption $A \sqsubseteq B$, hence also $A \sqsubseteq \text{RG}(N(R'))$ since $\text{RG}(N(R')) = B$. Now Lemma 5.1.6 is applicable, which says that each region of $\text{RG}(N(R'))$ can be transferred to A, so the set R' of regions of B is transferred into a set R'' of regions of A. This step did not change the corresponding Petri net, i.e. $N(R') = N(R'')$, because the definition of the corresponding Petri net only considers the number of tokens in the initial state and the weights of events, which stayed the same. Now, the previous lemma is applicable, which says that $\text{Approx}(A) = \text{RG}(N(R)) \sqsubseteq \text{RG}(N(R'')) = B$, which was to be shown.

For uniqueness, assume that there are two sets of regions R and $\tilde{R}$ according to the previous lemma with $\text{Approx}(A) = \text{RG}(N(R))$ and $\text{Approx}(A)' = \text{RG}(N(\tilde{R}))$. Since they are over-approximations, we have $\text{Approx}(A) \sqsubseteq \text{Approx}(A)'$ and $\text{Approx}(A)' \sqsubseteq \text{Approx}(A)$. By Lemma 2.0.8 both lts are deterministic and reachable since they are reachability graphs of Petri nets. Thus, Lemma 5.1.5 is applicable and we conclude that $\text{Approx}(A)$ and $\text{Approx}(A)'$ are isomorphic. □

5.3. Computing Minimal Over-Approximations

The previous section showed the existence of a minimal over-approximation by a Petri net solvable lts via a brute-force construction: Use all regions to construct a Petri net. This set is not necessarily finite, so this does not lend itself to actually computing an over-approximation yet. This section will present an algorithm to compute this lts.

The computation will be done iteratively: We begin with the original lts and try to solve it with a Petri net. If the lts is already solvable by a Petri net, then it is obviously its own minimal over-approximation, since nothing was over-approximated yet. If it is instead not solvable, then some separation problems must be unsolvable. This information will be used to modify the lts.

In particular, if a state separation problem $\{s, s'\}$ is unsolvable, then we know that the states s and s' cannot be differentiated by a Petri net place. In other words, these two states must be the same in any Petri net solution, so we identify these two states.

If an event/state separation problem (s, t) is unsolvable, then by definition there is no Petri net place that can disable transition t in the marking corresponding to state s while allowing all desired behaviour. If the edge cannot be prevented, then it must be allowed, so an outgoing edge with label t is added to state s. As the target of this edge, a new state is added to the lts that has no other connections.

This process produces a modified lts. Petri net synthesis is then attempted with this modified lts. This can again fail, in which case the whole process is repeated.

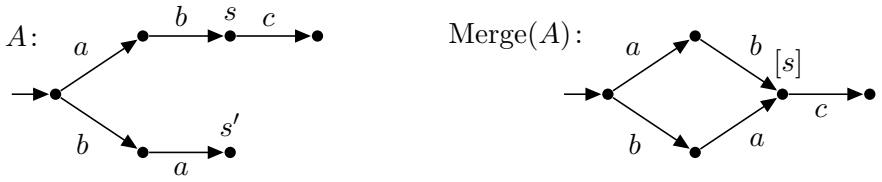


Figure 5.2.: An lts A with unsolvable separation problem $\{s, s'\}$ and the lts Merge(A).

5.3.1. The LTS Expansion Operator

To formalise this idea, we need symbols for the sets of unsolvable separation problems:

Definition 5.3.1 (Sets of unsolvable separation problems)**.** *For a reachable lts* $A = (S, \Sigma, \to, \imath)$*, define* $\mathrm{SSP}_{unsolv}(A)$ *to be its set of* unsolvable state separation problems *and* $\mathrm{ESSP}_{unsolv}(A)$ *to contain all of its unsolvable* event/state separation problems*, where a separation problem is* unsolvable *if no region solving it exists.*

A further complication is that individual state separation problems do not provide enough information: What if a state s should be merged with both state s' and s''? It turns out that in this case s' and s'' cannot be separated either, because state separation actually produces equivalence classes of states:

Definition 5.3.2 ($\equiv_A$)**.** *For a reachable lts* $A = (S, \Sigma, \to, \imath)$*, the relation* $\equiv_A \subseteq S \times S$ *is the following set:* $\equiv_A = \{(s, s) \mid s \in S\} \cup \{(s, s') \mid \{s, s'\} \in \mathrm{SSP}_{unsolv}(A)\}$.

Lemma 5.3.3. *For a reachable lts* A*,* $\equiv_A$ *is an equivalence relation.*

Proof. The relation is reflexive and symmetric by definition: $s \equiv_A s$ always holds and if $s \equiv_A s'$, then also $s' \equiv_A s$, since $\{s, s'\} = \{s', s\}$.

For transitivity, assume three states s, s', s'' so that $s \equiv_A s'$ and $s' \equiv_A s''$. If two of these states are the same, then the conclusion holds automatically. Thus, assume that all three states are different. This means that $\{s, s'\} \in \mathrm{SSP}_{unsolv}(A)$ and $\{s', s''\} \in \mathrm{SSP}_{unsolv}(A)$, which in turns implies that $\mathcal{R}(s) = \mathcal{R}(s') = \mathcal{R}(s'')$ holds for every region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$. Thus, no region can separate s and s'', so $\{s, s''\} \in \mathrm{SSP}_{unsolv}(A)$ and $s \equiv_A s''$. □

The equivalence relation is used to define the state-merged lts Merge(A): Each state s is replaced with its equivalence class $[s]$.

Definition 5.3.4. *For a reachable lts* $A = (S, \Sigma, \to, \imath)$*, define the* state-merged lts *to be* $\mathrm{Merge}(A) = (S/\equiv_A, \Sigma, \to/\equiv_A, [\imath])$*, where* $\to/\equiv_A = \{([s], t, [s']) \mid (s, t, s') \in \to\}$.

An example of this construction is shown in Figure 5.2. The lts A has $\mathrm{SSP}_{unsolv}(A) = \{\{s, s'\}\}$, so the construction of the state-merged lts identifies these two states to produce the state $[s] = \{s, s'\}$.

The lts Merge(A) handles unsolvable state separation problems, so event/state separation problems remain to be handled. As outlined above, for each such problem, a new

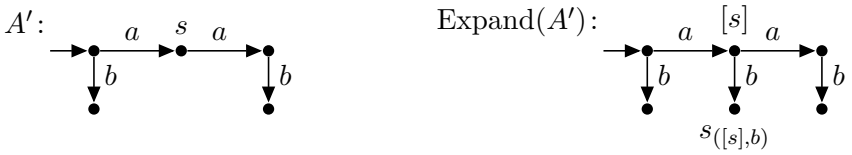


Figure 5.3.: An lts A' with an unsolvable event/state separation problem (s, b) and the lts Expand(A').

state and edge is added to produce the expansion Expand(A). However, the merging of states could already have added an outgoing edge with a suitable label, so this has to be checked to avoid introducing non-determinism. Otherwise, the unsolvable event/state separation problem (s', c) from the lts A in Figure 5.2 could lead to the addition of another outgoing c-edge to the state $[s]$ of Merge(A).

Definition 5.3.5 (Expansion Expand(A))**.** *For a reachable lts* $A = (S, \Sigma, \to, \imath)$*, let* Merge$(A) = (S', \Sigma, \to', [\imath])$ *and define the expansion* Expand$(A) = (S' \cup S'', \Sigma, \to' \cup \to'', [\imath])$*, where* $S'' = \{s_{([s],t)} \mid ([s], t) \in \Lambda\}$ *is the set of added states,* $\to'' = \{([s], t, s_{([s],t)}) \mid ([s], t) \in \Lambda\}$ *are added edges, and* $\Lambda = \{([s], t) \in (S/\equiv_A) \times \Sigma \mid (s, t) \in \mathrm{ESSP}_{unsolv}(A) \wedge \forall s' \in [s] \colon s' \overset{t}{\not\to}\}$ *is the set of remaining ESSP instances, where, without loss of generality*[3]*,* $S' \cap S'' = \emptyset$ *is assumed.*

This construction is illustrated in Figure 5.3. The lts A' has no unsolvable state separation problems, so no states are merged in Merge(A'). The ESSP instance (s, b) is unsolvable, because b is enabled in the initial state, then has to be disabled by transition a when going to state s, and then again enabled by the next a, which is not possible. This unsolvable ESSP instance is handled by adding a new state and edge in Expand(A'). In detail, $\Lambda = \{([s], b)\}$ since the separation problem was not handled by merging states. Thus, a state $s_{([s],b)}$ is added that is reachable from $[s]$ via b.

The algorithm for computing the minimal Petri net solvable over-approximation is to recursively apply the expansion operation until a fixed point is hopefully reached.

Definition 5.3.6 (Fixed point over-approximation)**.** *Given a reachable lts* A*, its* fixed point over-approximation *is the lts* A^**. This lts* A^* *is the fixed point of the chain defined by* $A_0 = A$ *and* $A_{i+1} = \mathrm{Expand}(A_i)$*.*

The next section will show that A^* is isomorphic to Approx(A) from Theorem 5.2.2. This means that the iterative definition of A^* actually terminates and provides an algorithm for computing Approx(A).

An example of this algorithm is shown in Figure 5.4. The lts A_0 is to be minimally over-approximated. Petri net synthesis of this lts fails and produces the following sets of unsolvable separation problems: Because $\imath \xrightarrow{ba} \imath$ forms a cycle, ba cannot appear on the non-cycle $s_2 \xrightarrow{ba} s_4$ and we have $\mathrm{SSP}_{unsolv}(A_0) = \{\{s_2, s_4\}\}$. Event a is enabled in

[3]States can be suitably renamed if needed.

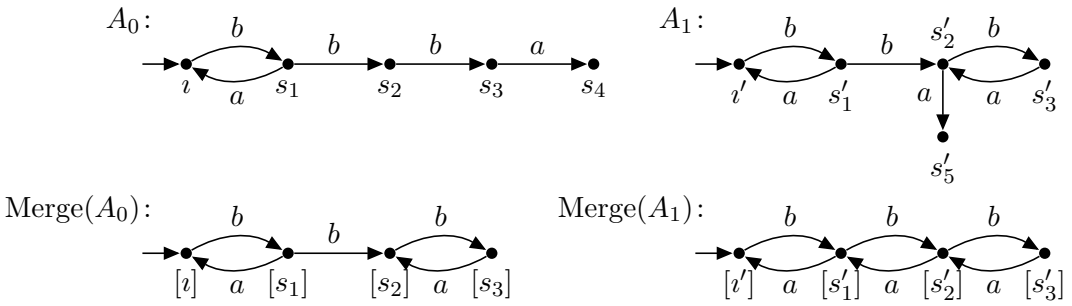


Figure 5.4.: An example of the fixed point algorithm.

s_1 and s_3 and $s_1 \xrightarrow{b} s_2 \xrightarrow{b} s_3$, so b has to first disable a and then enable it again, which is not possible, i.e. (s_2, a) is unsolvable. By the previous SSP instance, s_2 and s_4 have the same number of tokens in any region. Since s_2 enables b, s_4 must, too. By the first unsolvable ESSP instance, additionally (s_4, a) is unsolvable, so $\mathrm{ESSP}_{unsolv}(A_0) = \{(s_2, a), (s_4, a), (s_4, b)\}$. Since the lts is not solvable, the expansion operator has to be applied. For this, first the lts $\mathrm{Merge}(A_0)$ is constructed. This identifies the states s_2 and s_4, resulting in the lts $\mathrm{Merge}(A_0)$.

Next, the unsolvable event/state separation problems have to be handled. Thanks to the identification of states, the unsolvable event/state separation problem (s_4, b) was already handled, because s_4 was identified with state s_2, which had an outgoing b-edge. Thus, $\Lambda = \{([s_2], a), ([s_4], a)\}$ remains. However, since $[s_2] = [s_4] = \{s_2, s_4\}$ this really is $\Lambda = \{([s_2], a)\}$ and only a single edge and state are added by the expansion operator. The resulting lts $\mathrm{Expand}(A_0) = A_1$ is shown in Figure 5.4 as well. To simplify the following steps, its states were renamed.

Next, Petri net synthesis of A_1 is attempted. This fails again due to unsolvable separation problems, namely $\mathrm{SSP}_{unsolv}(A_1) = \{\{s'_1, s'_5\}\}$ and $\mathrm{ESSP}_{unsolv}(A_1) = \{(s'_5, a), (s'_5, b)\}$. Handling the state separation problems produces the lts $\mathrm{Merge}(A_1)$ displayed in Figure 5.4. This step already handles the unsolvable event/state separation problems: The events a and b cannot be prevented in s'_5. However, this state is identified with s'_1, which has outgoing edges with these events. Thus, $\Lambda = \emptyset$ and $\mathrm{Expand}(A_1) = \mathrm{Merge}(A_1)$. Petri net synthesis for this lts succeeds, which means that $\mathrm{SSP}_{unsolv}(\mathrm{Expand}(A_1)) = \emptyset = \mathrm{ESSP}_{unsolv}(\mathrm{Expand}(A_1))$ and $\mathrm{Expand}(\mathrm{Expand}(A_1)) = \mathrm{Expand}(A_1)$, i.e. a fixed point is reached. This lts can be solved by a Petri net, for example, the net N from Figure 5.5.

5.3.2. Fixed point Over-Approximation is Minimal Petri Net Solvable Over-Approximation

We begin with some basic lemmas about $\mathrm{Expand}(A)$. These lemmas will then be used to show that the fixed point over-approximation A^* really is the minimal over-

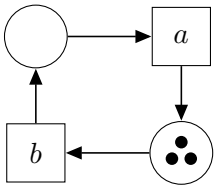


Figure 5.5.: A Petri net N with $\mathrm{RG}(N) = \mathrm{Merge}(A_1)$ with the lts from Figure 5.4.

approximation $\mathrm{Approx}(A)$ by a Petri net solvable lts according to Theorem 5.2.2.

Lemma 5.3.7. *Let $A = (S, \Sigma, \to, \imath)$ be a reachable lts, then $A \sqsubseteq \mathrm{Expand}(A)$.*

Proof. Let $\mathrm{Expand}(A) = (S', \Sigma, \to', [\imath])$. The canonical homomorphism $f\colon S \to S'$ defined via $f(s) = [s]$ is an lts homomorphism, because the added edges in $\to'$ are not relevant: By definition we have $f(\imath) = [\imath]$ and for each edge $(s, t, s') \in \to$, we have $([s], t, [s']) \in \to'$ by definition of $\to'$. □

Lemma 5.3.8. *For a reachable lts $A = (S, \Sigma, \to, \imath)$, the expansion $\mathrm{Expand}(A)$ is deterministic and reachable.*

Proof. First, we show that $\mathrm{Merge}(A)$ is deterministic and reachable. Reachability is easily inherited from A: Any path $\imath \xrightarrow{w} s$ in A can inductively be translated into $[\imath] \xrightarrow{w} [s]$ in $\mathrm{Merge}(A)$, and all its states can be reached in this way.

For determinism, assume that $[s] \xrightarrow{a} [s']$ and $[s] \xrightarrow{a} [s'']$ in $\mathrm{Merge}(A)$. We want to show that $[s'] = [s'']$. Since we have the edges $[s] \xrightarrow{a} [s']$ and $[s] \xrightarrow{a} [s'']$ in $\mathrm{Merge}(A)$, there must be states $\tilde{s}$, $\tilde{s}'$, $\tilde{s}''$, and $\hat{s}$ in A with edges $\tilde{s} \xrightarrow{a} \tilde{s}'$ and $\hat{s} \xrightarrow{a} \tilde{s}''$ and $\tilde{s} \equiv_A s \equiv_A \hat{s}$. Let $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ be an arbitrary region of A. By transitivity, $\tilde{s} \equiv_A \hat{s}$ and so we have $\mathcal{R}(\tilde{s}) = \mathcal{R}(\hat{s})$. Thus, by the definition of a region it follows that both target states have the same token counts, $\mathcal{R}(\tilde{s}') = \mathcal{R}(\tilde{s}) - \mathcal{B}(a) + \mathcal{F}(a) = \mathcal{R}(\hat{s}) - \mathcal{B}(a) + \mathcal{F}(a) = \mathcal{R}(\tilde{s}'')$. Since the region r was arbitrary, this means that these two states cannot be separated and we have $\tilde{s}' \equiv_A \tilde{s}''$. Since these states were chosen from $[s']$ and $[s'']$, respectively, $[s'] = [s'']$ follows, showing that $\mathrm{Merge}(A)$ is deterministic.

For $\mathrm{Expand}(A)$, reachability is obviously inherited from $\mathrm{Merge}(A)$ since every new state is reachable from an already reachable state of $\mathrm{Merge}(A)$. Also, the edges that are added to $\mathrm{Merge}(A)$ to construct $\mathrm{Expand}(A)$ are constructed such that no non-determinism is introduced. This is guaranteed through the set Λ in the definition (Definition 5.3.5). □

Lemma 5.3.9. *Let $A = (S, \Sigma, \to, \imath)$ be a reachable lts. There is a bijection between regions of A and regions of $\mathrm{Expand}(A)$ that preserves the value $\mathcal{R}(\imath)$ of the initial state and the functions $\mathcal{B}$ and $\mathcal{F}$ of a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$.*

Proof. By Lemma 5.3.7, $A \sqsubseteq \mathrm{Expand}(A)$ via some function f holds. Thus, we can invoke Lemma 5.1.6 on a region $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ of $\mathrm{Expand}(A)$ to get a region $f \circ r = (f \circ \mathcal{R}, \mathcal{B}, \mathcal{F})$ of A. We want to show that this mapping is a bijection.

For injectivity, assume two regions $(\mathcal{R}, \mathcal{B}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{B}', \mathcal{F}')$ of Expand(A) with $(f \circ \mathcal{R}, \mathcal{B}, \mathcal{F}) = (f \circ \mathcal{R}', \mathcal{B}', \mathcal{F}')$. This directly provides $\mathcal{B} = \mathcal{B}'$ and $\mathcal{F} = \mathcal{F}'$. It remains to derive $\mathcal{R} = \mathcal{R}'$ from $f \circ \mathcal{R} = f \circ \mathcal{R}'$. In particular, this last condition means that $\mathcal{R}$ and $\mathcal{R}'$ assign the same value to the initial state $f(\imath)$. We can use Corollary[4] 3.1.4, which states that the function $\mathcal{R}$ of a region is fully determined by $\mathcal{R}(\imath)$ and arrive at $\mathcal{R} = \mathcal{R}'$.

For surjectivity, we need to construct a region r' of Expand(A) from a region $r = (\mathcal{R}, \mathcal{F}, \mathcal{B})$ of A so that $r = f \circ r'$. By definition of $f \circ r'$, this means that $r' = (\mathcal{R}', \mathcal{F}, \mathcal{B})$, i.e. only $\mathcal{R}'$ is still missing. Invoking Corollary 3.1.4 again, $\mathcal{R}'$ is fully determined by $\mathcal{R}'(\imath) = \mathcal{R}(f(\imath))$ and this will already imply $\mathcal{R} = f \circ \mathcal{R}'$. Now that r' is defined, we need to show that r' is indeed a region of Expand(A). This means we need to show that $\mathcal{R}'$ does not produce negative values and that for all edges $s \xrightarrow{a} s'$ of Expand(A), both $\mathcal{R}'(s) \geq \mathcal{B}(t)$ and $\mathcal{R}'(s') = \mathcal{R}'(s) - \mathcal{B}(t) + \mathcal{F}(t)$ hold. The initial state is not assigned a negative value by definition and all other states have at least one incoming edge by reachability, which we use below.

Consider some edge $s \xrightarrow{a} s'$ of Expand(A). By construction of Expand(A), there must be a state $\hat{s}$ of A with $f(\hat{s}) = s$. Now, there are two possibilities. Either there is a state $\hat{s}'$ of A so that $f(\hat{s}') = s'$, and $\hat{s} \xrightarrow{a} \hat{s}'$, or s' was added in Expand(A) because $(\hat{s}, a)$ is an unsolvable ESSP instance in A. In the first case, r' inherits the region properties from r directly by $\mathcal{R}'(s) = \mathcal{R}(f(\hat{s}))$ and $\mathcal{R}'(s') = \mathcal{R}(f(\hat{s}'))$. In the second case, because $(\hat{s}, a)$ is an unsolvable ESSP instance, there is no region of A with $\mathcal{R}''(\hat{s}) \leq \mathcal{B}''(a)$, so in our case $\mathcal{R}'(s) = \mathcal{R}(\hat{s}) > \mathcal{B}(a)$ must hold. Also, $\mathcal{R}'(s') = \mathcal{R}'(s) - \mathcal{B}(a) + \mathcal{F}(a)$ holds since $s \xrightarrow{a} s'$ is the only incoming edge to s' and thus $\mathcal{R}(s')$ is defined by this equation. This value is non-negative by $\mathcal{R}'(s) > \mathcal{B}(a)$. □

Lemma 5.3.10. *Let $A = (S, \Sigma, \rightarrow, \imath)$ be a reachable lts and $N = (P, T, F, M_0)$ a Petri net. If $A \sqsubseteq \mathrm{RG}(N)$, then also* Expand$(A) \sqsubseteq \mathrm{RG}(N)$.

Proof. Let g be the homomorphism witnessing $A \sqsubseteq \mathrm{RG}(N)$. Each place p of N corresponds to a region of $\mathrm{RG}(N)$ via its extension $[[p]]$. By Lemma 5.1.6, $g \circ [[p]]$ is a region of A. By Lemma 5.3.9, there is an equivalent region of Expand(A). To summarise, each place of N corresponds to a region of Expand(A). Let R be the set of all regions of Expand(A) generated from N in this way. By Lemma 5.1.7, we now have Expand$(A) \sqsubseteq \mathrm{RG}(N(R))$. The construction preserves the initial marking and weights of events, so $N = N(R)$ holds and the lemma follows. □

These lemmas will now be used to show that the fixed point over-approximation A^* is the minimal Petri net solvable over-approximation Approx(A), and also that A^* exists, which means that the fixed point, as which it was defined, actually exists, i.e. the chain $(A_i)_{i \in \mathbb{N}}$ becomes stationary after finitely many steps.

Theorem 5.3.11. *Given a finite and reachable lts A, the chain defined by $A_0 = A$ and $A_{i+1} =$ Expand(A_i) reaches a fixed point A^* (up to isomorphism), there is a Petri net N*

[4] Expand(A) is reachable by Lemma 5.3.8, which is needed for this corollary.

solving A^*, and $\mathrm{RG}(N)$ *is isomorphic to the least Petri net solvable over-approximation* $\mathrm{Approx}(A)$ *of A, i.e. for all Petri nets N' with $A \sqsubseteq \mathrm{RG}(N')$, also $A^* \sqsubseteq \mathrm{RG}(N')$.*

Proof. First we show that the fixed point always exists and then that it is the minimal Petri net solvable over-approximation.

To show that the fixed point exists, consider an arbitrary lts A_i in the chain $(A_i)_{i\in\mathbb{N}}$. We want to show that there are only finitely many possibilities for A_i and so the ascending chain must eventually reach a fixed point. We begin by showing an upper bound on the number of states of A_i. By Lemma 5.2.1, there is a minimal Petri net over-approximation $N(R')$ of A, where $N(R')$ is bounded. Let $m \in \mathbb{N}$ be the number of reachable markings in $N(R')$. In Lemma 5.2.1, $N(R')$ was constructed from all regions of A, so for each region of A there is an equivalent region of $\mathrm{RG}(N(R'))$, meaning that $\mathcal{R}(\imath)$, $\mathcal{B}$, and $\mathcal{F}$ are the same.

We have $A \sqsubseteq A_i \sqsubseteq \mathrm{RG}(N(R'))$ by iterative application of Lemma 5.3.7, which says that $A \sqsubseteq \mathrm{Expand}(A)$, and Lemma 5.3.10, $A \sqsubseteq \mathrm{RG}(N(R')) \Rightarrow \mathrm{Expand}(A) \sqsubseteq \mathrm{RG}(N(R'))$, respectively. Let $n_i \in \mathbb{N}$ be the number of states of $\mathrm{Merge}(A_i)$. We have $n_i \leq m$, i.e. $\mathrm{Merge}(A_i)$ cannot have more states than $\mathrm{RG}(N(R'))$, as follows: Pick a word w_s for each state s of $\mathrm{Merge}(A_i)$ so that $\imath \xrightarrow{w_s} s$. Each state separation problem in $\mathrm{Merge}(A_i)$ is solvable by definition, so select a set of regions that solve all state separation problems of $\mathrm{Merge}(A_i)$. Next, these regions are transferred to $\mathrm{RG}(N(R'))$ as outlined above. By $\mathrm{Merge}(A_i) \sqsubseteq \mathrm{RG}(N(R'))$, every word w_s is also enabled in the initial marking of $N(R')$. The transferred regions ensure that none of these words reach the same marking in $N(R')$. Thus, $\mathrm{RG}(N(R'))$ has at least as many states as $\mathrm{Merge}(A_i)$. This also provides an upper bound on the size of $\mathrm{Expand}(A_i)$: It has at most $m \cdot (1 + |T|)$ states, since at most one state is added per state and label.

Since there are only finitely many different[5] lts with an upper bound on the number of states and a fixed alphabet, at least one lts A' must appear infinitely often in the chain $(A_i)_{i\in\mathbb{N}}$. By Lemma 5.3.8, each result of the expand function is deterministic and reachable, which applies to each A_i with $i > 0$. Thus, by Lemma 5.1.5 the preorder $\sqsubseteq$ is in fact a partial order in this setting. If some element appears twice in a partially ordered sequence, it must also appear twice consecutively[6] and is a fixed point of the underlying function. Thus $A' = A^*$ is a fixed point of the Expand-function.

Next we want to show that A^* can be solved by a Petri net and that it is the least over-approximation $\mathrm{Approx}(A)$ of A. Since $\mathrm{Expand}(A^*) = A^*$ holds, we have $\mathrm{SSP}_{unsolv}(A^*) = \emptyset = \mathrm{ESSP}_{unsolv}(A^*)$ by definition of the Expand-function. By Theorem 3.2.2, an lts without unsolvable separation problems is Petri net solvable, so A^* can be solved by a Petri net $N(R)$ for a suitable set R of regions that solve all separation problems. By iterated application of Lemma 5.3.10, we have that for all Petri nets N' with $A \sqsubseteq \mathrm{RG}(N')$ also $A^* \sqsubseteq \mathrm{RG}(N')$. □

[5] Up to isomorphism.

[6] $A \sqsubseteq A_j \sqsubseteq A \Rightarrow A = A_j$ by Lemma 5.1.5. So if $A_i = A_k$ for $i \leq j \leq k$, then also $A_i = A_j = A_k$.

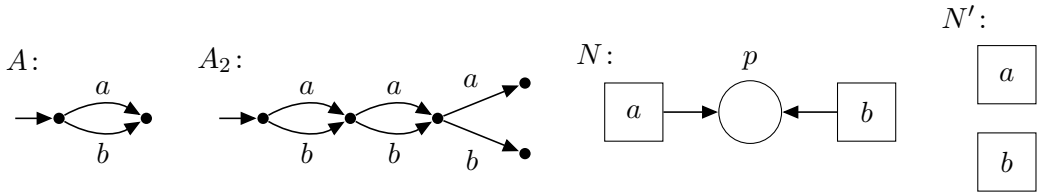


Figure 5.6.: An lts A, an over-approximation A_2 and two Petri nets N and N' further over-approximating A.

5.4. Subclasses of Petri Nets

The algorithm for computing the minimal Petri net solvable over-approximation can easily be defined for subclasses of nets: The construction of Expand(A) is based on the sets of unsolvable separation problems. These sets can also be defined relatively to a subclass. The basic arguments for the correctness of this construction stay the same, but it is not guaranteed to terminate.

For example, consider place-output-nonbranching Petri nets, which means that for each place, at most one transition consumes tokens from it. Repeatedly applying the expansion operation with respect to place-output-nonbranching synthesis to the lts A in Figure 5.6 does not reach a fixed point[7]. This means that Merge(A) and Expand(A) are defined as before, but the sets of unsolvable separation problems now contain all separation problems which cannot be solved by any region satisfying the place-output-nonbranching property. Because of the two parallel edges with labels a and b, these two labels must have the same effect in any region. Place-output-nonbranching means that each place has at most one transition in its postset, so it is not possible for a and b to both consume tokens, which means these transitions must have empty presets. This, in turn, means that they are always enabled. Thus, each expansion step appends two new states to this lts that are reached via a and b, respectively. The next step will then merge these two states since they correspond to an unsolvable state separation problem and append two new states. An intermediate result $A_2 = \text{Expand}(\text{Expand}(A))$ is shown in Figure 5.6. If this operation were continued infinitely often, the result would be the reachability graph of the Petri net N from the same figure. This Petri net is unbounded, meaning that it has infinitely many reachable markings.

Thus, this example shows that Theorem 5.2.2, which states that the minimal over-approximation is finite, i.e. by a bounded Petri net, does not hold for place-output-nonbranching Petri net over-approximation. If the requirement for boundedness is added to the definition of the minimal over-approximation, i.e. we ask for the minimal Petri net solvable over-approximation by a *bounded* Petri net, the result is the Petri net N' from Figure 5.6. This Petri net has two transitions that are always enabled and do not modify

[7] Without the restriction to place-output-nonbranching Petri nets, A is Petri net solvable, i.e. $\text{Expand}(A) = A$.

the marking. However, this Petri net cannot be computed by the presented algorithm. This is because boundedness is shown based on Lemma 3.1.6, which introduced complement regions, i.e. regions where the forward and backward weight functions $\mathcal{F}$ and $\mathcal{B}$ are swapped. This Lemma does not hold for arbitrary subclasses of nets. For example, the place p of the Petri net N in Figure 5.6 is no longer place-output-nonbranching when complemented, because the complement has two transitions in its postset. Complementary regions were used to show that only finitely many markings are reachable. Without complementary regions, it is possible that the minimal over-approximation has infinitely many reachable markings, as this example shows. Since the algorithm constructs the minimal over-approximation iteratively, it cannot produce an infinite lts in a finite number of steps.

In the remainder of the section we will show that the algorithm works for some other subclasses. From the above considerations we can see that a subclass should allow the complementation of regions to be compatible with the algorithm. Looking at the subclasses introduced in Section 4.1, only plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded, plus combinations of these subclasses, allow complementation, because the other subclasses make different requirements for forward and backward flows. Since all of these subclasses make the same requirements on the forward and backward flows of a place, Lemma 3.1.6 holds for them:

Lemma 5.4.1. *Let r be a region of a finite lts A and $\overline{r}$ its complement according to Lemma 3.1.6. Then $\overline{r}$ belongs to a combination of the net subclasses plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded, if and only if r does.*

Proof. The construction of Lemma 3.1.6 transforms a region $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ into the complement $\overline{r} = (k-\mathcal{R}, \mathcal{F}, \mathcal{B})$. This construction preserves the considered subclasses. □

Lemma 5.1.6 also holds with respect to these subclasses:

Lemma 5.4.2. *Let A_1 and A_2 be finite lts with $A_1 \sqsubseteq A_2$ via f, and let $r = (\mathcal{R}, \mathcal{B}, \mathcal{F})$ be a region of A_2. Then the region $f \circ r := (f \circ \mathcal{R}, \mathcal{B}, \mathcal{F})$ of A_1 belongs to a combination of the net subclasses plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded, if and only if r does.*

Proof. Lemma 5.1.6 showed that $f \circ r$ is a region of A_1. The functions $\mathcal{B}$ and $\mathcal{F}$ are the same in r and $f \circ r$. Since the subclasses plain, pure, T-net and marked graph restrict only the functions $\mathcal{B}$ and $\mathcal{F}$, these subclasses are preserved and only k-boundedness still has to be considered. For k-boundedness we have to show that $f \circ \mathcal{R}$ does not produce values above k. However, the image of this function is a subset of the image of $\mathcal{R}$, so k-boundedness is preserved, too. □

The above lemma is needed to show that Lemmas 5.3.9 and 5.3.10 also hold for our subclasses:

Lemma 5.4.3. *Let $A = (S, \Sigma, \to, \imath)$ be a reachable lts. There is a bijection between regions of A and regions of* Expand(A) *that preserves the value $\mathcal{R}(\imath)$ of the initial state and the functions $\mathcal{B}$ and $\mathcal{F}$ of a region $(\mathcal{R}, \mathcal{B}, \mathcal{F})$, where all regions belong to a combination of the subclasses plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded.*

Lemma 5.4.4. *Let $A = (S, \Sigma, \to, \imath)$ be a reachable lts and $N = (P, T, F, M_0)$ a Petri net. If $A \sqsubseteq \mathrm{RG}(N)$, then also* Expand$(A) \sqsubseteq \mathrm{RG}(N)$*, where the expansion operator is understood relative to a combination of the subclasses plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded and N is also a member of the same subclass.*

Proof for Lemma 5.4.3 and 5.4.4. The details of both proofs are unchanged, so they will not be repeated. The only new insight is for k-boundedness: If an unsolvable event/state separation problem (s, t) exists, the expansion operator adds a new state $s_{([s],t)}$ that is reachable from s via t. This state could be assigned more than k tokens, but then the region's complement would assign a negative number of tokens. Thus, the complement region would in fact solve the event/state separation problem (s, t), which was assumed to be unsolvable, and we arrive at a contradiction. □

We can now prove the correctness of the algorithm with respect to subclasses:

Theorem 5.4.5. *Given a finite and reachable lts A, the chain defined by $A_0 = A$ and $A_{i+1} =$* Expand(A_i) *with respect to a combination of the net subclasses plain, pure, (generalised) T-net, (generalised) marked graph, and k-bounded, reaches a fixed point A^* (up to isomorphism), there is a Petri net N from the same subclass solving A^*, and* RG(N) *is the least Petri net solvable over-approximation* Approx(A) *of A, i.e. for all Petri nets N' with $A \sqsubseteq \mathrm{RG}(N')$, also $A^* \sqsubseteq \mathrm{RG}(N')$.*

Proof. The arguments from the proof of Theorem 5.3.11 can mostly be reused: By the argument above, a minimal over-approximation exists, and since Lemma 5.4.1 allows complementation of places, the minimal over-approximation must be finite (see proof of Lemma 5.2.1).

We have $A \sqsubseteq A_i \sqsubseteq \mathrm{RG}(N(R'))$ by iterative application of Lemma 5.3.7 (which says that $A \sqsubseteq$ Expand(A) and is not influenced by subclasses), and Lemma 5.4.4 ($A \sqsubseteq \mathrm{RG}(N(R')) \Rightarrow$ Expand$(A) \sqsubseteq \mathrm{RG}(N(R'))$), respectively. There is an upper bound on the number of states of each lts A_i. Since there are only finitely many different lts with an upper bound on the number of states and a fixed alphabet, at least some lts A' must appear infinitely often and this lts is then a fixed point A^* of the expansion operator by monotonicity. See the proof of Theorem 5.3.11 for more details.

Because A^* is a fixed point of the expansion operator, it has no unsolvable separation problems, which means by Theorem 3.2.2 that it can be solved by a Petri net. By iterative application of Lemma 5.4.4, for all Petri nets N' with $A \sqsubseteq \mathrm{RG}(N')$ and belonging to our subclass, also $A^* \sqsubseteq \mathrm{RG}(N')$ holds, so that A^* is the minimal over-approximation. □

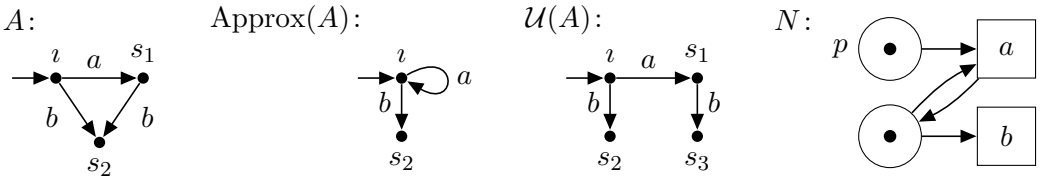


Figure 5.7.: An lts A, its minimal over-approximation Approx(A), its limited unfolding $\mathcal{U}(A)$, and a Petri net N with $\mathrm{RG}(N) = \mathcal{U}(A)$.

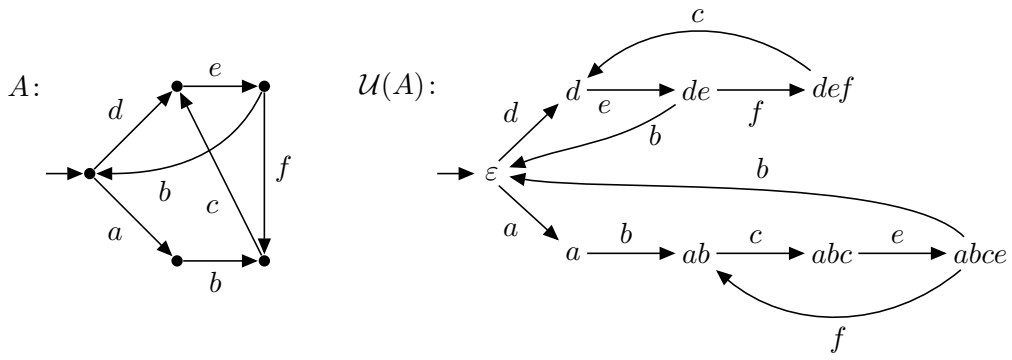


Figure 5.8.: An lts A and its limited unfolding $\mathcal{U}(A)$ [BBD15, Figure 2.6].

5.5. Over-Approximation of Regular Languages

The language of an lts is the set of sequences that are possible in its initial state, i.e. $L(A) = \{w \in \Sigma^* \mid \iota \xrightarrow{w}\}$. This definition can be extended to Petri nets through their reachability graphs, i.e. $L(N) = L(\mathrm{RG}(N))$. Language inclusion $L(A) \subseteq L(B)$ is a preorder between lts and can be used to define a minimal over-approximation, too. By the definition above, the languages that are considered here are prefix closed, i.e. if $ww' \in L$, then also $w \in L$. Regular languages [HU79] are a well-known class of languages. A regular language can be represented by an lts if and only if it is prefix closed.

The minimal over-approximation according to $\sqsubseteq$ that was so far studied in this chapter does not guarantee minimality according to language-inclusion. For example, on the left of Figure 5.7, there is an lts A and its minimal over-approximation Approx(A). These two lts do not have the same language, since $L(A) = \{\varepsilon, a, b, ab\}$ and $L(\mathrm{Approx}(A)) = \{a\}^* \cup \{wb \mid w \in \{a\}^*\}$. For example, aa is only possible in Approx(A). However, the Petri net N has a reachability graph isomorphic to $\mathcal{U}(A)$, both shown in the same figure, and has the same language as A. Thus, A can be solved exactly up to language-equivalence, but Approx(A), its minimal over-approximation according to $\sqsubseteq$, is not a Petri net with the same language. This means that minimality according to $\sqsubseteq$ does not guarantee minimality according to language-inclusion.

Minimal over-approximation with respect to language inclusion was already studied in the literature. In [BBD15], a limited unfolding $\mathcal{U}(A)$ of an lts A is defined. It changes the lts so that paths that reach the same state, e.g. $\imath \xrightarrow{b} s_2$ and $\imath \xrightarrow{ab} s_2$ in A from Figure 5.7, now reach different states in $\mathcal{U}(A)$. Examples of limited unfoldings are shown in Figures 5.7 and 5.8. In detail, in Definition 5.5.1 the states of the given lts are replaced with words that reach this state. Each state has the same outgoing edges as in the original lts. However, when some state is visited twice by a given word, instead of continuing the unfolding, a loop is introduced to the prefix that already reached the current state. For example, in the lts A of Figure 5.8, both *ab* and *abcef* reach the same state. Thus, there is no state *abcef* in $\mathcal{U}(A)$, but instead the outgoing edge with label f from state *abce* goes to *ab*.

Definition 5.5.1 (Limited unfolding [BBD15])**.** *Let $A = (S, \Sigma, \to, \imath)$ be a deterministic lts. Its* limited unfolding $\mathcal{U}(A)$ *is the lts $\mathcal{U}(A) = (S', \Sigma, \to', \varepsilon)$ with $S' = \{w \in L(A) \mid \forall w_1, w_2, w_3 \in \Sigma^*, s, s' \in S\colon (w = w_1 w_2 w_3 \wedge w_2 \neq \varepsilon \wedge \imath \xrightarrow{w_1} s \xrightarrow{w_2} s') \Rightarrow s \neq s'\}$ being the set of words from $L(A)$ that do not visit any state twice and $\to' = \{(w, t, w') \in S' \times \Sigma \times S' \mid w' = wt \in S' \vee \exists s \in S, w'' \in \Sigma^*\colon w'w'' = w \wedge \imath \xrightarrow{wt} s \wedge \imath \xrightarrow{w'} s\}$, which either appends a label to the current word if the corresponding state was not visited before, or else goes back to the (unique) prefix that corresponds to the new state.*

The interest in the limited unfolding stems from the following theorem, which relates the unfolding to lts homomorphisms:

Theorem 5.5.2 ([BBD15, Proposition 7.10])**.** *Let A be a finite, deterministic, and reachable lts. For any bounded Petri net N, $L(A) \subseteq L(N)$ if and only if $\mathcal{U}(A) \sqsubseteq \mathrm{RG}(N)$.*

Proof. ($\Leftarrow$): Assuming $\mathcal{U}(A) \sqsubseteq \mathrm{RG}(N)$ via f, we have $L(\mathcal{U}(A)) \subseteq L(N)$: For any $\imath \xrightarrow{w}$ in $\mathcal{U}(A)$, also $f(\imath) \xrightarrow{w}$ in $\mathrm{RG}(N)$ by definition of an lts homomorphism. Since $L(A) = L(\mathcal{U}(A))$ by construction, the conclusion follows.

($\Rightarrow$): Assuming $L(A) \subseteq L(N)$, we define a function f from states of $\mathcal{U}(A)$ to markings of N. The states of $\mathcal{U}(A)$ are words $w \in L(A) \subseteq L(N)$, so w can be mapped to the marking M of N that is reached via w, i.e. $M_0[w\rangle f(w)$. Such a marking exists by assumption and is unique since $\mathrm{RG}(N)$ is deterministic by Lemma 2.0.8 and $\mathcal{U}(A)$ is deterministic by construction. We want to show that f is an lts homomorphism, in which case it witnesses $\mathcal{U}(A) \sqsubseteq \mathrm{RG}(N)$. Obviously $f(\varepsilon) = M_0$, so it only remains to show that if $w \xrightarrow{t} w'$ in $\mathcal{U}(A)$, then also $f(w)[t\rangle f(w')$ in $\mathrm{RG}(N)$. If $w' = wt$, then $f(w')$ is determined via $f(w)[t\rangle f(w')$ and nothing remains to show.

Thus, assume that w can be decomposed into $w = w'w''$ so that there is a state s of A with $\imath \xrightarrow{wt} s$ and $\imath \xrightarrow{w'} s$. In this case there is a cycle $s \xrightarrow{w''t} s$ in $\mathcal{U}(A)$. Since s is reached via w', for all $k \in \mathbb{N}$ we have $w'(w''t)^k \in L(\mathcal{U}(A)) = L(A) \subseteq L(N)$. Now, $w''t$ may not change the marking of the Petri net by the marking equation from Lemma 2.0.12: The change in number of tokens when firing $w''t$ is $C \cdot \Psi(w''t)$ by this lemma. Since every $w'(w''t)^k$ can be fired from the initial marking of N, we get infinitely many different

reachable markings if $w''t$ changes the marking of the Petri net. This contradicts the assumption that N is a bounded Petri net. Thus, $C \cdot \Psi(w''t)$ is the null vector, which means that $wt = w'w''t$ reaches the same marking in N as w', i.e. $f(w)[t\rangle f(w')$. □

Combining the above theorem with Theorem 5.2.2, which showed that $\text{Approx}(A)$ is the minimal Petri net solvable over-approximation according to $\sqsubseteq$, allows to do minimal over-approximation up to language-inclusion by calculating $\text{Approx}(\mathcal{U}(A))$:

Corollary 5.5.3. *Given a finite and deterministic lts A, for any bounded Petri net N with $L(A) \subseteq L(N)$, also $L(\text{Approx}(\mathcal{U}(A))) \subseteq L(N)$.*

Furthermore, this can be used for synthesis up to language equivalence, because if any Petri net is language-equivalent to A, then by the previous corollary, also $L(A) = L(\text{Approx}(\mathcal{U}(A)))$:

Corollary 5.5.4. *Given a finite and deterministic lts A, there is a bounded Petri net N with $L(A) = L(N)$ if and only if $L(A) = L(\text{Approx}(\mathcal{U}(A)))$.*

Since the arguments in this section are based on languages and not specific subclasses of Petri nets, these two corollaries also hold with respect to the subclasses of nets that were considered in Section 5.4.

Another attempt at minimal over-approximation with respect to language inclusion was made in [CCK10; Car+08], but their result is incorrect. The algorithm from these papers generate the lts $\text{Approx}(A)$ as the minimal language-based over-approximation of the lts A in Figure 5.7. However, its limited unfolding $\mathcal{U}(A)$, and thus its language $L(A)$, can be solved exactly as shown by the Petri net N from the same figure. The error is in a statement similar to Lemma 5.1.6, but for language inclusion: If $L(A) \subseteq L(B)$ for two deterministic lts, then every region of B can be transferred into a region of A[8]. However, the extension $[[p]]$ of the place p of N in Figure 5.7 is a region of $\mathcal{U}(A)$ that cannot be transferred into a region of A, since it would have to assign two different token counts to the state s_2 of A: By $\imath \xrightarrow{b} s_2$, s_2 gets one token, but by $\imath \xrightarrow{ab} s_2$ it has no tokens.

Thus, this approach for over-approximating regular languages with bounded Petri nets does not work.

5.6. Conclusion

In this chapter, an algorithm for minimal Petri net over-approximation was introduced. The algorithm modifies a given lts so that it becomes solvable by a Petri net. Minimality is here understood with respect to a structural preorder, which we call *lts homomorphism*. This procedure is also possible for some, but not all, of the subclasses of nets that

[8] The papers actually use the classic simulation preorder. Lemma 3 and Lemma 11.3 show that $L(A) \subseteq L(B)$ implies that B can simulate A. Lemma 2 and Lemma 11.2 then claim that simulation implies that regions can be transferred.

were introduced in the previous chapter. This construction was lifted to minimal over-approximation up to language-inclusion via a construction from [BBD15].

In future work, more subclasses of Petri nets for which this algorithm works should be identified. For example, even though we have seen that targeting place-output-nonbranching Petri nets introduces problems, it might very well be possible that k-bounded place-output-nonbranching Petri nets can be produced. The intuition here is that no arbitrarily long paths can be generated since at some point the k-boundedness condition causes state separation problems to become unsolvable. This would then result in a shorter path and thus enforce an upper bound on the length of paths, which would ensure termination of the algorithm.

In the literature, mainly approximation according to language inclusion was investigated [BBD95; Dar98; Dar03; LMJ07], which is different to our preorder that also considers the structure of the lts. An exception is [BBD15], which calls an lts homomorphism a *simulation* and introduces an algorithm for minimal over-approximation. For this, *extremal regions* are introduced and it is shown that taking all extremal regions produces a minimal over-approximation. Their algorithm works for bounded Petri nets, which is the setting that was examined here, as well as for unbounded Petri nets, which our algorithm cannot produce. It can also be amended for pure, distributed and place-output-nonbranching Petri nets [BBD15; Dar00], but most[9] of the subclasses considered in Section 5.4 are not expressible. For language-based over-approximation based on extremal regions, they introduced the limited unfolding that was also presented here.

There is also the field of *process discovery*, where observations are used to generate a model of a business process [Aal16]. Such a model can be a Petri net that only approximates the observations. However, the focus in process discovery is to generate simple Petri nets with e.g. few places, instead of having a minimal approximation.

An open question about the algorithm that was introduced in this chapter is its complexity. As indicated in Section 3.4, this will depend on the specific subclass of Petri nets that is used. For simplicity, we now consider only general Petri nets without further restrictions. The approach based on extremal regions has at least exponential complexity since there can be exponentially many extremal regions for an lts [BBD15]. Our approach needs the sets $\mathrm{SSP}_{unsolv}(A)$ and $\mathrm{ESSP}_{unsolv}(A)$, which can be computed in polynomial time [BBD95; BD96], so a single application of the expansion operator can be performed in polynomial time. However, the number of iterations that the algorithm needs is unknown and therefore needs to be approximated. Also, it might be possible to directly approximate the size of $\mathrm{Approx}(A)$ based on the size of A.

One promising approach for this are geometrical characterisations of separation problems [EW17; BDS17; SW18]. These characterisations use the reaching Parikh vectors Ψ_s for a state s according to an arbitrary spanning tree that were introduced in Definition 3.3.1. In an lts with only trivial cycles[10] this characterisation shows that all state separation

[9] Binary conflict-free Petri nets might be possible, but this was not examined in the literature.
[10] The lts must be a directed acyclic graph and no two different states s and s' may have $\Psi_s = \Psi_{s'}$.

problems are solvable and that an event/state separation problem (s, t) is unsolvable if Ψ_s is in the convex hull of all $\Psi_{s'}$ for states s' with $s' \xrightarrow{t}$. The size of the convex hull provides an upper bound for the number of states that have to enable t, thus the size of the union of all the convex hulls provide an upper bound on the size of the minimal over-approximation. The size of a convex hull is polynomial. To summarise, the conjecture is that Approx(A) has polynomial size in the size of the input and can be computed in polynomial time when not targeting a subclass.

Another interesting challenge is maximal under-approximations of lts, i.e. instead of adding behaviour to the input, some behaviour is removed. However, it is easy to find examples that have no unique under-approximation. For example, consider an lts where ab reaches another state than ba. Since these two paths have the same Parikh vector, they must reach the same marking in a Petri net by Lemma 2.0.12 (marking equation), and this behaviour cannot be reproduced by a Petri net. To under-approximate this, the last event from either of the paths can be removed. Thus, there is no unique maximal under-approximation.

6. Implementation

In 2012 and 2013, twelve master students at the University of Oldenburg, among them the author of this document, developed the tool APT [Bor+13; BS15]. This group was founded, instructed and supervised by Prof. Dr. Eike Best with help from PD. Dr. Elke Wilkeit, Dr. Hans Fleischhack, and Dipl.-Inform. Thomas Strathmann. APT stands for **A**nalysis of **P**etri nets and **T**ransition systems and provides various algorithms dealing with Petri nets and lts, for example, computing the reachability graph of a Petri net or checking an lts for determinism. Besides providing algorithms, of course it also provides the necessary data structures to represent Petri nets and lts and supports various file formats. It is available online at `https://github.com/CvO-Theory/apt` as a command line application, plus a graphical user interface is available at `https://github.com/CvO-Theory/apt-gui`.

Beginning in 2014, the author of this thesis extended APT with an implementation of Petri net synthesis. This led to synthesis targeting subclasses as presented in Chapter 4 and the algorithm for minimal Petri net solvable over-approximation from Chapter 5. In this chapter, we will see how this implementation can be used and the design is briefly presented.

6.1. The User Interface

APT is organised into *modules*. The synthesis algorithm from Chapter 4 is available in the `synthesize`-module. Since describing the exact file format is beyond the scope of this document, the following examples will use the `regular_language_to_lts`-module to generate an lts from the unique minimal automaton[1] of a regular language. The resulting lts is given via a pipe to another invocation of APT, which then reads it from its standard input, which is requested via the parameter `-`.

For example, the extended[2] regular expression $((ab|ba)c)\{2\}$ produces the lts shown in Figure 6.1. This lts can be synthesised into a Petri net via:[3]

```
./apt.sh reg '((ab|ba)c){2}' | ./apt.sh synthesize pure -
```

[1] Since finite automata have final states while lts do not, this module actually generates an lts whose language is the prefix closure of the given regular language.

[2] For a regular expression `r`, `r{n}` is equivalent to n repetitions of r.

[3] This assumes a Unix system. Under Windows a temporary file is needed since pipes are not supported. Also, APT allows to use a prefix of the name of a module as long as this is unique.

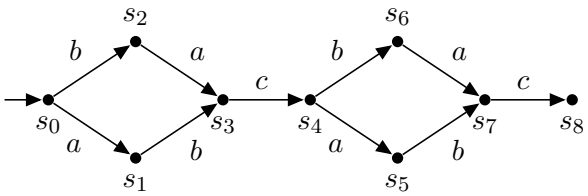


Figure 6.1.: Lts generated from the extended regular expression $((ab|ba)c)\{2\}$.

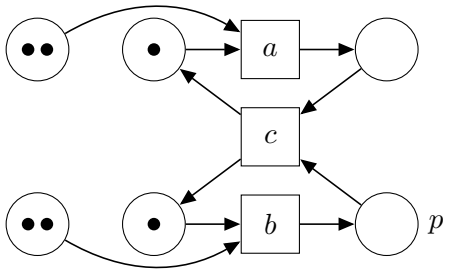


Figure 6.2.: Pure Petri net generated for the lts in Figure 6.1.

This command first uses APT to generate an lts from the given regular expression and then passes that on to the `synthesize`-module. The next argument, `pure`, requests that a pure solution to this lts is produced. This call produces the Petri net shown in Figure 6.2.

If we want more information about this Petri net, we can request the pseudo-property `verbose`, which means that APT lists the event/state separation problems that are solved by each region:

```
$ ./apt.sh reg '((ab|ba)c){2}' | ./apt.sh synthesize pure,verbose -
success: Yes
solvedEventStateSeparationProblems:
Region { init=0, 0:a:0, 0:b:1, 1:c:0 }:
    separates event c at states [s0, s1, s4, s5, s8]
[...]
```

The region shown in this example disables transition c in, for example, state s_0. It has no token initially (`init=0`), but transition b produces one (`0:b:1`) while transition c consumes one (`1:c:0`). It corresponds to the place p in Figure 6.2.

If, instead of asking for a pure solution, we want a 1-bounded solution, then synthesis fails, because some separation problems are unsolvable.

```
$ ./apt.sh reg '((ab|ba)c){2}' | ./apt.sh synthesize 1-bounded -
success: No
```

```
failedStateSeparationProblems:
    [[s1, s5], [s2, s6], [s0, s4, s8], [s3, s7]]
failedEventStateSeparationProblems: {a=[s8], b=[s8]}
```

Here we can see that a variety of separation problems were not solvable. The module provides the equivalence classes for unsolvable state separation problems and it provides the list of unsolvable event/state separation problems.

Chapter 5 introduced an algorithm for dealing with unsolvable lts by over-approximating the input. This algorithm is implemented in the `overapproximate_synthesize`-module and can be used via `./apt.sh reg '((ab|ba)c){2}' | ./apt.sh over 1-bounded -`. With this invocation, APT produces the Petri net which was already shown in Figure 2.1 on page 5 and that is equivalent to the Petri net in Figure 6.2, except for the two places that limit a and b to fire at most twice.

To see a full list of supported properties, the modules can be called without providing an input, e.g. `./apt.sh synth`. For example, this shows that the `synthesize`-module supports the option `upto-language-equivalence`, which internally computes the limited unfolding mentioned in Section 5.5. This operation is also available directly as the `limited_unfolding`-module. Thus, even though minimal over-approximation up to language equivalence is not available directly, it can be achieved by over-approximating the limited unfolding of an lts (see Corollary 5.5.3):[4]

```
./apt.sh reg 'b|ab' | ./apt.sh limited - - | ./apt.sh over none -
```

This command first calls the `regular_language_to_lts`-module to generate an lts. This lts is unfolded by the next invocation of APT. Finally, the minimal over-approximation according to Petri nets without any restrictions—subclass `none`—is computed.

6.2. Implemented Synthesis Algorithms

The synthesis approach that was developed in Chapters 3 and 4 supports a large variety of subclasses. It characterises regions via a satisfiability modulo theories (SMT) problem for the theory of integers, i.e. linear inequalities together with boolean combinations that have to be solved in the integers. Specifically, the implementation uses the `QF_LIA` logic defined in the SMT-LIB standard [BST10] and uses the SMTInterpol library [CHN13] to solve the constructed systems.

While this approach is very generic and supports many subclasses, it is also not very efficient. Thus, a variety of algorithms from the literature were implemented additionally. Based on the input lts and the selected subclass, the best applicable algorithm

[4] The input here is the lts A from Figure 5.7 on page 47.

is selected. The implementation of the following classes can be found in the subfolder `src/module/uniol/apt/analysis/synthesize/separation/` of the APT source code.

- For pure and distributed Petri nets, `BasicPureSeparation` implements a dedicated algorithm from [BBD15] that could have polynomial performance[5].
- `BasicImpureSeparation` extends `BasicPureSeparation` to also provide solutions that are not pure [BBD15].
- `PlainPureSeparation` can produce pure and plain Petri nets. This algorithm is a trivial extension of `BasicPureSeparation` that is not found in the literature.
- A dedicated algorithm for 1-bounded synthesis based on [CKLY98; BBD15] is implemented in `ElementarySeparation`. This algorithm also supports the subclasses plain, pure, and distributed.
- The algorithm for 1-bounded synthesis was extended by its authors to k-bounded synthesis [Car+08; CCK10]. This is implemented in `KBoundedSeparation`. The minimisation step of this algorithm is not implemented for reasons outlined in Section 4.3.
- The state spaces of connected marked graph Petri nets were characterised graph theoretically in [BD14] in such a way that a solution can be directly constructed based on the distance between special states that are only reached by, or are only left by a single event. This is implemented in `MarkedGraphSeparation` and is the only implementation which does not reduce the existence of regions to an inequality system or a combinatorial problem, which means that it can be a lot faster than the other implementations. Thus, this implementation is used when the input lts satisfies the needed structural conditions even when no marked graph solution is explicitly requested.
- Synthesis of place-output-nonbranching nets is examined in [BDS18]. This algorithm is implemented in `OutputNonbranchingSeparation` and consists of structural preconditions as well as smaller inequality systems that need to be solved.
- The general algorithm used when no specialised implementation exists can be found in `InequalitySystemSeparation`.

In addition to these implementations of algorithms that directly produce regions, there is also the `FactorisationSynthesizer`. This class is based on the concepts of products of lts and sums of Petri nets, which will be formally introduced in Section 8.3.4. Given two Petri nets N_1 and N_2, their disjoint[6] sum $N_1 \oplus N_2$ is produced by placing the two

[5] The algorithm produces homogeneous inequality systems which can be solved in polynomial time. However, the actually used solver does not have this time guarantee.

[6] Disjointness refers to their alphabets, meaning that e.g. only one of the Petri nets has a transition for label a.

Petri nets next to each other in a single Petri net. The equivalent operation on lts is the disjoint[7] product $A_1 \otimes A_2$.

In [Dev18], conditions for factorising an lts A into factors A_1 and A_2 so that $A = A_1 \otimes A_2$ were developed. Checking these conditions efficiently is an open problem and instead in [DS18] an efficient algorithm for the context of Petri net synthesis was developed. This algorithm is implemented in APT. It allows to decompose a complicated input into two smaller lts that can be synthesised separately and faster. If a factorisation is not possible, this is detected quickly, so that there is only a low overhead due to the attempted factorisation.

Even though the characterisation used for `MarkedGraphSeparation` only works for connected marked graphs, the factorisation allows APT to also synthesise unconnected marked graphs efficiently.

6.3. Optimisation Strategies

The general algorithm outlined in Chapters 3 and 4 is neither very efficient nor does it produce simple Petri nets. For example, the Petri nets can have redundant places whose removal does not modify the shape of the reachability graph. As a first optimisation, before calculating a new region to solve a given separation problem, the list of already found regions is checked for a region that already solves this separation problem. This is a lot faster than computing a new region and leads to smaller Petri nets since each region corresponds to a place.

Next, for an lts $A = (S, \Sigma, \rightarrow, \imath)$ there are at most $|S| \times |\Sigma|$ event/state separation problems while there are $\frac{1}{2}|S| \times (|S| + 1)$ state separation problems. This means that, in practice, many more state separation problems exist, so the implementation first solves all event/state separation problems and then computes the equivalence classes of states that are not yet separated. Most of the time no unsolvable state separation problems remain.

The two heuristics above improve the performance of APT. The next heuristic specifically reduces the number of places in a Petri net solution. The intuition is that regions computed later might also solve separation problems that were already considered earlier. Thus, as a post-processing step, for each separation problem the list of solving regions is computed. From each of these sets one region is picked so that the number of regions is heuristically minimised.

More details on these optimisation strategies can be found in [Sch16b].

[7] Their alphabets are disjoint.

6.4. Minimising the Number of Places

The algorithm that was explained in Chapters 3 and 4 produces a system $\text{isRegion}(r) \wedge \text{SP}(r, pr)$ solvable by some vector r, if r is a region and solves the separation problem pr. The following variant of this algorithm calculates the minimal possible number of places for a given input. It is implemented in the `synthesize`-module via the `minimize` option.

To find the minimal number of places, first an upper bound ℓ on the number of places is needed. This bound can be found by doing normal synthesis as previously explained. This produces a Petri net with some number ℓ of places. Now, a system is constructed to find a solution with $\ell - 1$ places: This means that r^1 to $r^{\ell-1}$ are regions and for each separation problem $pr \in \text{SP}_A$, one of the regions r^1 to $r^{\ell-1}$ solves it. This system is:

$$\bigwedge_{1 \leq i \leq \ell-1} \text{isRegion}(r^i) \wedge \bigwedge_{pr \in \text{SP}_A} \bigvee_{1 \leq i \leq \ell-1} \text{SP}(r^i, pr)$$

If this system is unsolvable, then there is no Petri net with $\ell - 1$ places that solves the given lts. Otherwise, such a solution is found and the procedure is retried with $\ell - 2$ places. This continues until the minimal number of necessary places is found.

If the minimal number of places according to some subclass from Chapter 4 is sought, we can simply replace $\text{isRegion}(r^i)$ with $\text{isRegion}(r^i) \wedge \text{additionalProperties}(r^i)$.

One may be tempted to use a binary search for the minimal number of places instead of the linear search outlined above. However, experimental results suggest that this leads to a longer running time, because the solver—here that is SMTInterpol—needs a lot longer to conclude unsolvability of an inequality system than to find a solution, if one exists. Thus, the number of unsolvable inputs to the solver should be kept small.

6.5. Conclusion and Performance of the Implementation

APT is a tool for working with Petri nets and lts. Originally, APT could only call other tools for Petri net synthesis. The author extended APT with implementations of Petri net synthesis and over-approximation. These implementations were presented in this chapter.

The performance of APT was compared with other implementations and some proposed algorithms in some publications. In [BS15], the basic algorithm implemented in `InequalitySystemSeparation` was compared with Synet [Cai99; BCD02], Petrify [Cor+99; CKLY98; CKLY95], and Genet [CCK09; CCK10; Car+08; CCK08]. Here, APT showed comparable performance and was for some of the considered examples faster than some of its competitors. An example for this is shown in Table 6.1, which shows measurements of the time to synthesise a bit net Petri net. A bit is a Petri net with two places and two transitions that swap a single token between the places. A bit

n	APT	APT-pure	Synet	Petrify	Genet
8	0.60	0.86	138.49	0.13	0.05
10	1.56	2.32	—	1.25	0.31
12	5.71	6.31	—	17.73	2.28
14	24.69	30.48	—	403.67	16.10
16	183.76	212.23	crash	—	132.13
18	—	—	crash	OOM	—

Table 6.1.: Time in seconds for synthesising a bit net Petri net of size n. Dashes indicate that the 10 minutes time limit was exceeded. OOM means out of memory.

net of size n has n disconnected bits. APT produces unrestricted Petri nets while APT-pure targeted pure Petri nets. For large inputs, Synet crashed with a stack overflow and Petrify exited with a memory allocation error.

In [BDS18], an implementation of a proposed algorithm was compared with other algorithms. The implementation based on the characterisation of marked graphs from [BD14] won almost all cases. However, it was not applicable in all the considered situations. The heuristics from Section 6.3 and the minimisation from Section 6.4 were evaluated in [Sch16b]. Here, it was shown that the optimisation strategy improves performance immensely, while minimisation is expensive.

A new synthesis algorithm was proposed in [Wol18]. A prototype implementation of this algorithm was compared only with APT, because, after some experimentation, it was concluded that APT has the strongest performance among the considered tools.

Part II.

Petri Net Synthesis from Modal Specifications

7. Introduction to Modal Specifications

In the first part of this thesis, a given labelled transition system (lts) was assumed. The task at hand was to find a Petri net with a reachability graph isomorphic to the lts—or over-approximating the lts if no exact solution exists. This is called Petri net synthesis. However, lts are quite limited as a specification language used to describe a system that should be produced. The input already describes the exact behaviour of the system and no possible choices remain. To allow for more flexibility in the synthesis procedure, the second part of the document investigates modal specifications as the starting point for synthesis. So far, there are few[1] approaches for this, even though Petri net synthesis from modal specifications was already desired in the literature, both for an actual application [Dar05] and out of theoretical interest [BBD15].

In an lts, there are two possibilities for a label in some state: Either an edge with the label is present, and then the solution must also have it, or there is no edge with the label and so the label is also not allowed in a solution. Modal specifications add a third possibility: Some behaviour can be allowed without requiring it to be present.

For example, a simple model of a vending machine is that, after a coin was inserted, a product, for example, a cup of coffee, is made available and the machine returns to its initial state. The behaviour of this vending machine could be modelled via an lts. Such an lts is shown on the left of Figure 7.1. However, when we want to generalise this model, we run into problems. For example, we cannot allow the machine to also offer tea without requiring this. Also, we cannot model the possibility for a machine to clean itself when it is currently not serving a customer. We need a new kind of edge that *allows* some label without requiring it.

The needed new kind of edges are the *must and may modalities* in modal specifications. On the right of Figure 7.1, a modal transition system is depicted. Its dashed arrows

[1] The author only knows about [BD04].

Figure 7.1.: Models of a vending machine [Kre17].

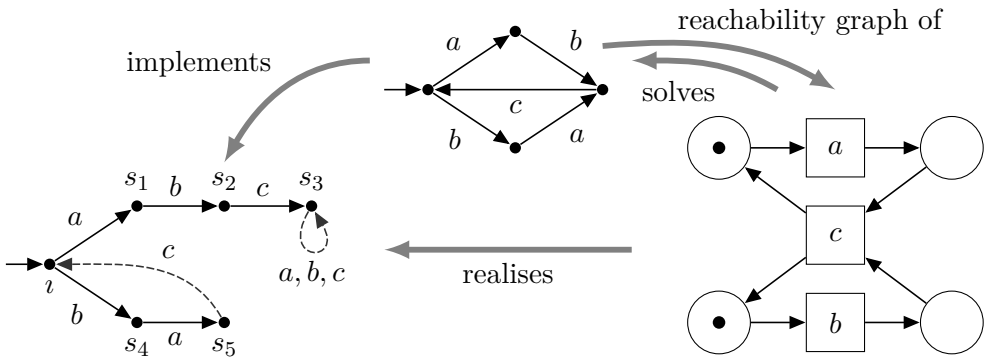


Figure 7.2.: The relation between implementation, realisation, and solution.

represent may edges. The meaning of these edges still requires the extended model of a vending machine to offer coffee, but it could additionally also offer tea. Also, the machine may clean itself when it is in its initial state, but this is just a possibility and not a requirement. Also, cleaning and serving tea are independent of each other and the relation between these is not specified.

While previously lts were solved up to isomorphism, the interpretation of modal transition systems is closer to bisimulation. This can be seen in the left half of Figure 7.2: The lts in the middle is an *implementation* of the modal transition system on the left, but they are clearly not isomorphic to each other.

The concept of modal specifications is not new and goes back to Hennessy-Milner-Logic [HM80; HM85], but in the literature these concepts are used to specify families of lts: A modal specification language comes with a notion expressing that a given lts implements the modal specification. Thus, there is a notion of *implementation* that relates modal specifications and lts, and there is a notion of *solution* as a Petri net that relates lts and Petri nets. We combine these notions to define *Petri net realisations* of modal specifications: A Petri net realises a modal specification if its reachability graph implements the specification. This relation is visualised in Figure 7.2 and is a natural combination of modal specifications and Petri nets.

Modal specifications are already well established in the literature. In this thesis, two flavours of modal specifications will be used: The modal transition systems that were already outlined above, and the modal μ-calculus, which consists of formulas.

There are reasons to use both of these specification languages. The modal μ-calculus is quite expressive and, for example, more powerful than the well-known logics LTL, CTL, and CTL* [CGR11], as well as modal transition systems. So, by providing algorithms for this really expressive language, a variety of other specification languages can be handled as well.

A downside of the modal μ-calculus is that it gives specifications as formulas. It is well

known that graphical specifications are easier to understand and to reason about than formulas (see e.g. [Sat+15]). For this reason, *modal transition systems (mts)* are also used.

The μ-calculus is more expressive than mts. For example, every mts can be implemented, i.e. `false` is not expressible, and a modal transition system cannot express disjunctive properties, such as a vending machine that should sell at least either tea or coffee, without requiring both options. There are extensions of mts that allow to express this [Kre17], for example, disjunctive modal transition systems [LX90], but they will not be used here. Instead, the limitations of mts will be used to show more general results. We show that Petri net synthesis from mts is undecidable. This result automatically transfers to the modal μ-calculus and other specification languages that are more expressive than mts. This proof will actually be done on a subset of the modal μ-calculus that is just as expressive as mts, so that conjunction can be used for better readability.

In the following Section 7.1, modal transition systems are formally introduced. Section 7.2 presents the modal μ-calculus, and finally Section 7.3 restricts the modal μ-calculus syntactically, so that the result is equivalent to mts.

7.1. Modal Transition Systems

Modal transition systems [Lar89] are an extension of lts. A single mts specifies a family of lts, which are called its implementations. Similar to an lts, an mts has states and labelled edges between these states. Mts generalise this by having two kinds of edges, *must edges* and *may edges*. Roughly speaking, a must edge has to be present in any implementation while a may edge is optional and can be left out. These edges are similar to edges in lts in that they have a source state, a label, and a target state.

Definition 7.1.1 (Modal transition system)**.** *A* modal transition system (mts) M *is a tuple* $M = (S, \Sigma, \rightarrow, \dashrightarrow, \imath)$*, where* S *is a finite set of* states*,* Σ *is an alphabet,* $\imath \in S$ *is the* initial state*,* $\dashrightarrow \subseteq S \times \Sigma \times S$ *is the set of* may edges *and* $\rightarrow \subseteq S \times \Sigma \times S$ *is the set of* must edges *satisfying* $\rightarrow \subseteq \dashrightarrow$*. Both* $s \xrightarrow{w} s'$ *and* $s \overset{w}{\dashrightarrow} s'$ *are defined analogously to* $s \xrightarrow{w} s'$ *in lts for words* $w \in \Sigma^*$*.*

An example of an mts is given on the left of Figure 7.2. Its initial state $\imath$ is indicated with an incoming arrow. The other states are s_1 to s_5. May edges are drawn in grey and are dashed. An example of a may edge is the edge going from s_5 to $\imath$ with label c. Must edges are drawn as solid lines and implicitly also represent the underlying may edge. Formally, this mts is $(S, \Sigma, \rightarrow, \dashrightarrow, \imath)$ with states $S = \{\imath, s_1, s_2, s_3, s_4, s_5\}$, alphabet $\Sigma = \{a, b, c\}$, must edges $\rightarrow = \{(\imath, a, s_1), (s_1, b, s_2), (s_2, c, s_3), (\imath, b, s_4), (s_4, a, s_5)\}$, and may edges $\dashrightarrow = \rightarrow \cup \{(s_3, a, s_3), (s_3, b, s_3), (s_3, c, s_3), (s_5, c, \imath)\}$.

Some possible implementations of this mts are shown in Figure 7.3. The first lts has the minimal required behaviour from the specification. The implementation relation for mts only considers the presence of allowed (may edges) and required (must edges)

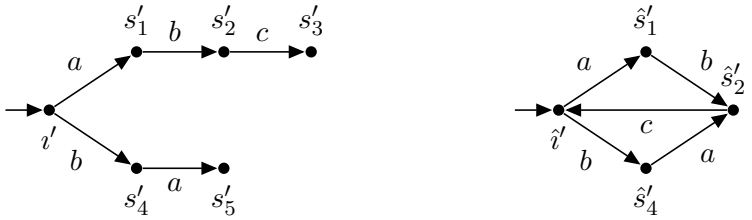


Figure 7.3.: Example implementations for the mts on the left of Figure 7.2.

behaviour, but, e.g., does not require that paths that lead to the same state in the specification also do so in the implementation. Put differently, the implementation relation is similar to bisimulation and not to isomorphism. The second example highlights this. Each state implements the corresponding state of the specification, e.g. $\hat{\imath}'$ implements $\imath$. Additionally, state $\hat{s}'_2$ also implements state s_5. Plus, the path $\imath \xrightarrow{abc} s_3$ corresponds to a loop $\hat{\imath}' \xrightarrow{abc} \hat{\imath}'$ in the implementation. Thus, $\hat{\imath}'$ also implements s_3. Since every label has a may edge around s_3, all states that are reachable from $\hat{\imath}'$ in the implementation, i.e. all states, also implement s_3.

The next definition formally introduces when an lts implements an mts. It is based on the definition of *refinement* of [Lar89], but is specialised to the case that the more concrete specification is an lts. This definition uses a relation between states of the mts and the lts. For two related states, if the state of the mts has a must edge, then the state of the lts must have an edge with the same label, and these two edges lead to states that are again in relation to each other. Similarly, each edge of the implementation must be allowed by a may edge in the specification.

Definition 7.1.2 (Implementation)**.** *An lts $A = (S_A, \Sigma, \rightarrow, \imath_A)$ is an* implementation via $R \subseteq S_A \times S_M$ *of an mts $M = (S_M, \Sigma, \rightarrow, \dashrightarrow, \imath_M)$ if $(\imath_A, \imath_M) \in R$ and for all $(q, s) \in R$ and all $a \in \Sigma$:*

- $\forall q' \in S_A \colon q \xrightarrow{a} q' \Rightarrow \exists s' \in S_M \colon s \dashrightarrow s' \wedge (q', s') \in R$,
- $\forall s' \in S_M \colon s \xrightarrow{a} s' \Rightarrow \exists q' \in S_A \colon q \xrightarrow{a} q' \wedge (q', s') \in R$.

If $(q, s) \in R$, we say that q implements *s* (via R)*. If A is an implementation of M via some relation R, we call R an* implementation relation *and write $A \models M$ (via R).*

For example, the second lts in Figure 7.3 implements the mts from Figure 7.2 via the relation $R = \{(\hat{\imath}', \imath), (\hat{s_4}', s_4), (\hat{s_2}', s_5), (\hat{s_1}', s_1), (\hat{s_2}', s_2), (\hat{\imath}', s_3), (\hat{s_1}', s_3), (\hat{s_2}', s_3), (\hat{s_4}', s_3)\}$.

It was already announced above that every mts can be implemented. This is formalised in the following lemma.

Lemma 7.1.3. *Given an mts $M = (S, \Sigma, \rightarrow, \dashrightarrow, \imath)$, both $(S, \Sigma, \rightarrow, \imath)$ and $(S, \Sigma, \dashrightarrow, \imath)$ implement M.*

Proof. The implementation is with the identity relation $\mathrm{id} = \{(s, s) \mid s \in S\}$. This satisfies $(\imath, \imath) \in \mathrm{id}$, as required, and each edge $s \xrightarrow{a} s'$ in the lts is also a may edge in the mts, since by definition of mts $\rightarrow \subseteq \dashrightarrow$ holds. For the same reason, every must edge of the mts is implemented correctly. □

Similar to lts, an mts is called *deterministic* if no state has two different outgoing edges with the same label.

Definition 7.1.4 (Determinism)**.** *An mts $M = (S, \Sigma, \rightarrow, \dashrightarrow, \imath)$ is* deterministic *if the lts $(S, \Sigma, \dashrightarrow, \imath)$ is deterministic.*

7.2. Modal μ-Calculus

The modal μ-calculus [Koz83; AN01] is an extension of the Hennessy-Milner-Logic [HM80; HM85] with fixed points. Just like modal transition systems, the μ-calculus can be used to describe classes of lts. While modal transition systems are an automaton-based specification language, meaning that they have states and edges, the μ-calculus is a logical specification language based on formulas.

An example of a formula of the modal μ-calculus is `false`, the inconsistent specification that is never satisfied. This already highlights a difference to modal transition systems, because every modal transition system is implementable (see Lemma 7.1.3). We will later see that the μ-calculus is indeed more expressive than mts, meaning that every mts can be translated into a formula of the μ-calculus, but not vice versa.

The modalities that are available in the modal μ-calculus are the box modality $[a]$, which is the universal modality, and the diamond modality $\langle a \rangle$, which is the existential modality, where $a \in \Sigma$ is some label. These modalities specify what should happen after a label a occurred. In terms of an lts, this means that the state, in which we currently evaluate the formula, is left via an a-edge leading to another state. For example, $[a]$`false` expresses that after label a the inconsistent specification must hold, which, of course, is not possible. Thus, this formula asserts that no a-edge is present.

The difference between the two modalities is similar to the difference between an existential and an universal quantifier: $[a]$`false` means that after all possible a-edges `false` holds while $\langle a \rangle$`false` represents that there is an a-edge that leads to a state satisfying `false`. For deterministic lts, which will be our focus, this corresponds[2] to the may and must edges of mts: An existential modality and a must edge both require some edge to be present in the lts, while a universal modality and a may edge do not require it.

The logic outlined so far, together with the usual logical connectives of disjunction, conjunction, and negation, is the Hennessy-Milner-Logic [HM80]. Since all formulas must be finite and a formula can only have finitely many modalities, this logic only

[2] This is only a rough correspondence, because the μ-calculus implicitly allows everything that is not forbidden, while mts forbid everything not explicitly allowed (cf. Figure 7.5 on page 76).

allows to specify finite behaviour. For example, $\langle b\rangle\texttt{true} \vee \langle a\rangle\langle b\rangle\texttt{true} \vee \langle a\rangle\langle a\rangle\langle b\rangle\texttt{true}$ expresses that after at most two a-edges, b is possible, but the logic cannot express that after a finite number of a-edges, b becomes possible. The addition to the calculus that makes this possible are the fixed point operators μ and ν, together with variables such as X. They allow to write recursive formulas such as $\mu X.(\langle a\rangle X \vee \langle b\rangle\texttt{true})$. This formula holds in a state that enables label b, i.e. which satisfies $\langle b\rangle\texttt{true}$, but this formula also holds in a state that allows a sequence of a's and then enables b.

Definition 7.2.1 (Syntax of μ-calculus)**.** *Given a set of variables* Var *and an alphabet* Σ*, the set of all formulas of the* modal μ-calculus *is defined recursively:*

$$\beta_1, \beta_2 ::= \texttt{true} \mid \texttt{false} \mid X \mid \beta_1 \wedge \beta_2 \mid \beta_1 \vee \beta_2 \mid \langle a\rangle\beta_1 \mid [a]\beta_1 \mid \nu X.\beta_1 \mid \mu X.\beta_1$$

where $X \in \mathsf{Var}$ *is a variable and* $a \in \Sigma$ *is an event. A variable* $X \in \mathsf{Var}$ *is* free in β*, if it is not under the scope of any fixed point operator* μX *or* νX*. A formula without free variables is* closed*.*

This definition does not include negation. Negation is later introduced as an abbreviation in Definition 7.2.5.

Definition 7.2.2 (Precedence rules)**.** *To avoid ambiguity, we assume that the unary operators negation (defined later) and the modalities have highest precedence, followed by binary operators (conjunction and disjunction). The fixed point operators have lowest precedence. Where necessary, parentheses can be used.*

For example, $\nu X.\neg[a]\neg X \vee \langle b\rangle\texttt{true}$ is interpreted as $\nu X.((\neg([a](\neg X))) \vee (\langle b\rangle\texttt{true}))$.

The recursive interpretation of the formula $\mu X.(\langle a\rangle X \vee \langle b\rangle\texttt{true})$ is as follows: The inner formula of the fixed point is $\langle a\rangle X \vee \langle b\rangle\texttt{true}$ and the fixed point binds the variable X. Substituting $\texttt{false}$ for X results in the formula $\langle a\rangle\texttt{false} \vee \langle b\rangle\texttt{true}$, which requires a b-edge to be present, because the first part of the disjunction, $\langle a\rangle\texttt{false}$, is unsatisfiable, since it requires an a-edge to a state satisfying $\texttt{false}$, which is not possible. Thus, this formula is equivalent to its second part, which is $\langle b\rangle\texttt{true}$. A state satisfies $\langle a\rangle\langle b\rangle\texttt{true} \vee \langle b\rangle\texttt{true}$ if it allows either of the sequences ab or b. This second formula was generated from the first by a substitution from our specification: In $\langle a\rangle X \vee \langle b\rangle\texttt{true}$, we substitute $\langle b\rangle\texttt{true}$ for the variable X. If we continue like this, in the next iteration we get the formula $\langle a\rangle(\langle a\rangle\langle b\rangle\texttt{true} \vee \langle b\rangle\texttt{true}) \vee \langle b\rangle\texttt{true}$. This formula now requires either of the sequences aab, ab, or b to be present. If this substitution were continued infinitely often, the resulting (infinite) formula would express that after a finite sequence of label a, label b becomes enabled. This is the meaning of the fixed point formula $\mu X.(\langle a\rangle X \vee \langle b\rangle\texttt{true})$.

The difference between the fixed points ν and μ is intuitively in whether they allow infinite sequences. For example, $\mu X.(\langle a\rangle X \vee \langle b\rangle\texttt{true})$ only holds if after a finite sequence consisting only of event a, event b becomes possible. However, $\nu X.(\langle a\rangle X \vee \langle b\rangle\texttt{true})$ is also satisfied by a system in which an infinite a-sequence is possible, which means that the fixed point is recursed infinitely often.

The interpretation of a formula of the μ-calculus is defined as a set of states of an lts. This set will contain all states satisfying the formula. Since the syntax allows free occurrence of variables, a valuation val will be needed that assigns states to free variables.

Definition 7.2.3 (Semantics of μ-calculus)**.** *The* interpretation $[\![\beta]\!]_A^{\mathsf{val}}$ *of a formula β of the modal μ-calculus with respect to an lts $A = (S, \Sigma, \to, \imath)$ and a valuation $\mathsf{val} \colon \mathsf{Var} \to 2^S$ is defined inductively:*

$$
\begin{aligned}
[\![\mathtt{true}]\!]_A^{\mathsf{val}} &= S \\
[\![\mathtt{false}]\!]_A^{\mathsf{val}} &= \emptyset \\
[\![X]\!]_A^{\mathsf{val}} &= \mathsf{val}(X) \\
[\![\langle a \rangle \beta_1]\!]_A^{\mathsf{val}} &= \{s \in S \mid \exists s' \in S \colon s \xrightarrow{a} s' \wedge s' \in [\![\beta_1]\!]_A^{\mathsf{val}}\} \\
[\![[a] \beta_1]\!]_A^{\mathsf{val}} &= \{s \in S \mid \forall s' \in S \colon s \xrightarrow{a} s' \Rightarrow s' \in [\![\beta_1]\!]_A^{\mathsf{val}}\} \\
[\![\beta_1 \wedge \beta_2]\!]_A^{\mathsf{val}} &= [\![\beta_1]\!]_A^{\mathsf{val}} \cap [\![\beta_2]\!]_A^{\mathsf{val}} \\
[\![\beta_1 \vee \beta_2]\!]_A^{\mathsf{val}} &= [\![\beta_1]\!]_A^{\mathsf{val}} \cup [\![\beta_2]\!]_A^{\mathsf{val}} \\
[\![\nu X.\beta_1]\!]_A^{\mathsf{val}} &= \bigcup \{V \subseteq S \mid [\![\beta_1]\!]_A^{\mathsf{val}[V \leftarrow X]} \supseteq V\} \\
[\![\mu X.\beta_1]\!]_A^{\mathsf{val}} &= \bigcap \{V \subseteq S \mid [\![\beta_1]\!]_A^{\mathsf{val}[V \leftarrow X]} \subseteq V\}
\end{aligned}
$$

where $\mathsf{val}[V \leftarrow X]$ for $V \subseteq S$ and $X \in \mathsf{Var}$ is equivalent to the valuation val, except for $\mathsf{val}[V \leftarrow X](X) = V$. If β has no free variables, then the valuation can be left out and we write $[\![\beta]\!]_A = [\![\beta]\!]_A^{\mathsf{val}}$.

A state s and a valuation val together implement *a formula β, written $s, \mathsf{val} \models \beta$, if and only if $s \in [\![\beta]\!]_A^{\mathsf{val}}$. An lts $A = (S, \Sigma, \to, \imath)$ and a valuation val* implement *a formula β, written $A, \mathsf{val} \models \beta$, if and only if $\imath, \mathsf{val} \models \beta$. If β has no free variables, the valuation does not influence the result and we write $s \models \beta$ for a state s, and $A \models \beta$ for an lts A. For a Petri net N, define* realisation *via its reachability graph: $N, \mathsf{val} \models \beta$ if and only if $\mathrm{RG}(N), \mathsf{val} \models \beta$, and $N \models \beta$ if and only if $\mathrm{RG}(N) \models \beta$. Then, N* realises *β.*

As an example, consider the mts from Figure 7.4. It has two paths. In the upper path, the word *abc* is required to be present and this reaches a state where everything is allowed, which corresponds to `true`. So, this path roughly corresponds to $\langle a \rangle \langle b \rangle \langle c \rangle \mathtt{true}$. In the lower path, the word *ba* is required and afterwards, if a *c*-edge is present, we reach the initial state $\imath$ again. This can be expressed as $\nu X.\langle b \rangle \langle a \rangle [c] X$, which uses the greatest fixed point operator since this loop can be taken infinitely often. Combining these two parts, we arrive at the formula $\nu X.\langle a \rangle \langle b \rangle \langle c \rangle \mathtt{true} \wedge \langle b \rangle \langle a \rangle [c] X$.

Note however that this formula does not have the same meaning as the mts from Figure 7.4. This is, for example, because the mts forbids *c* in the initial state, because in an mts a may edge has to be present for some behaviour to be allowed, while in the μ-calculus, everything not explicitly forbidden is allowed. Thus, the subformula $[c]\mathtt{false}$ needs to be added to the above formula. Doing this in all required places results in a

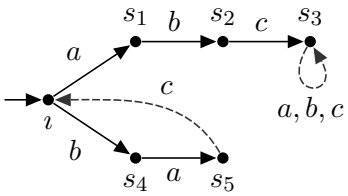


Figure 7.4.: An example of an mts that was already shown in Figure 7.2.

quite large and unwieldy formula, so this will only be provided here for the upper path containing states $\imath$, s_1, s_2, and s_3:

$$\langle a\rangle([a]\texttt{false} \wedge [c]\texttt{false} \wedge \langle b\rangle([a]\texttt{false} \wedge [b]\texttt{false} \wedge \langle c\rangle\texttt{true})) \wedge [c]\texttt{false} \wedge \langle b\rangle(\dots)$$

The relation between mts and the μ-calculus will be further examined in Section 7.3.

The recursive meaning of fixed points that was informally introduced above, namely that we can unroll a fixed point like $\mu X.\beta(X)$ into $\beta(\texttt{false})$, $\beta(\beta(\texttt{false}))$, etc., has a formal basis shown in the next Theorem:

Theorem 7.2.4 (Theorem 1.2.11 of [AN01])**.** *Let $A = (S, \Sigma, \to, \imath)$ be a finite lts, $X \in \mathsf{Var}$ a variable, $\mathsf{val}\colon \mathsf{Var} \to 2^S$ a valuation, and β a formula of the modal μ-calculus. Define the sequences $(x_i)_{i\in\mathbb{N}}$ and $(y_i)_{i\in\mathbb{N}}$ via $x_0 = [\![\texttt{false}]\!]_A^{\mathsf{val}}$, $y_0 = [\![\texttt{true}]\!]_A^{\mathsf{val}}$, $x_{i+1} = [\![\beta]\!]_A^{\mathsf{val}[x_i \leftarrow X]}$, and $y_{i+1} = [\![\beta]\!]_A^{\mathsf{val}[y_i \leftarrow X]}$. Then $[\![\mu X.\beta]\!]_A^{\mathsf{val}} \supseteq x_{i+1} \supseteq x_i$ and $[\![\nu X.\beta]\!]_A^{\mathsf{val}} \subseteq y_{i+1} \subseteq y_i$ for all $i \in \mathbb{N}$. Furthermore, there is an $i \in \mathbb{N}$ so that $[\![\mu X.\beta]\!]_A^{\mathsf{val}} = x_i$ and $[\![\nu X.\beta]\!]_A^{\mathsf{val}} = y_i$.*

The definition of the μ-calculus does not include negation, because negation is defined as an abbreviation (e.g. [MSS99; BW15]).

Definition 7.2.5 (Negation)**.** *Negation in the μ-calculus is defined by the following rules, where $\beta[V \leftarrow X]$ is the formula β with all free occurrences of X replaced[3] by V:*

$$\begin{array}{lll}
\neg\texttt{true} \equiv \texttt{false} & \neg\texttt{false} \equiv \texttt{true} & \neg(\beta_1 \wedge \beta_2) \equiv \neg\beta_1 \vee \neg\beta_2 \\
\neg\langle a\rangle\beta \equiv [a]\neg\beta & \neg[a]\beta \equiv \langle a\rangle\neg\beta & \neg(\beta_1 \vee \beta_2) \equiv \neg\beta_1 \wedge \neg\beta_2 \\
\neg\nu X.\beta \equiv \mu X.\neg\beta[\neg X \leftarrow X] & \neg\mu X.\beta \equiv \nu X.\neg\beta[\neg X \leftarrow X] & \neg\neg\beta \equiv \beta
\end{array}$$

It is possible to define negation directly and leave out some other operators, but that requires introducing some well-nestedness condition: Any occurrence of a variable X bound by a fixed point νX or μX must have an even number of negations, e.g. $\nu X.\langle a\rangle\neg X$ is not allowed. This condition is imposed for the following technical reason: The semantics of a formula violating the condition is not well-defined, because negation is not monotonous and thus fixed points involving negation might not exist.

As an example for negation, consider the formula $\neg[a]\texttt{false}$. Intuitively $[a]\texttt{false}$ means that a is not possible, or more precisely that after any a-edge we reach an inconsistent

[3] Previously, this syntax was used for semantic substitution while here it means syntactic substitution.

state. Its intuitive negation is that an edge with label a is required, and indeed the above rules produce $\neg[a]\texttt{false} \equiv \langle a\rangle\texttt{true}$, which expresses that an a-edge is present.

The formula $\nu X.\langle a\rangle X$ expresses that an infinite sequence of a-edges exists. Its negation evaluates to $\neg\nu X.\langle a\rangle X \equiv \mu X.\neg\langle a\rangle\neg X \equiv \mu X.[a]\neg\neg X \equiv \mu X.[a]X$. This formula follows all edges with label a, due to the box modality. Since the least fixed point intuitively only allows finite recursion, this formula does not hold if an infinite sequence of a-edges exists, since then the fixed point is iterated infinitely often. Thus, this formula states that all paths labelled exclusively with a are finite.

7.2.1. Vectorial Fixed Points and Systems of Equations

In this section, vector equality systems with formulas of the modal μ-calculus are solved. This means that a vectorial system is turned into several independent and closed formulas, that each describe one component of the solution of the system.

Namely, we are dealing with the semantics of terms such as

$$\begin{pmatrix}\Psi_1\\ \vdots\\ \Psi_n\end{pmatrix} = \nu \begin{pmatrix}X_1\\ \vdots\\ X_n\end{pmatrix} . \begin{pmatrix}\Phi_1(X_1,\ldots,X_n)\\ \vdots\\ \Phi_n(X_1,\ldots,X_n)\end{pmatrix}$$

where Φ_1 to Φ_n are formulas of the modal μ-calculus that can use the variables X_1 to X_n, which in turn are bound by the fixed point operator, in this case ν. We are looking for formulas Ψ_1 to Ψ_n, which describe a closed solution to this fixed point, meaning that they describe a common greatest fixed point of the whole equation system.

Our running example is the following system:

$$\begin{pmatrix}\Psi_1\\ \Psi_2\\ \Psi_3\end{pmatrix} = \nu \begin{pmatrix}X_1\\ X_2\\ X_3\end{pmatrix} . \begin{pmatrix}[a]X_1 \wedge [b]X_2\\ \langle c\rangle(X_2 \wedge X_3)\\ [b]X_1\end{pmatrix}$$

In a solution to this system, Ψ_1 expresses that all a-successors of a state satisfying Ψ_1 also satisfy Ψ_1, while all its b-successors also satisfy Ψ_2, due to the requirements in the first row of the system. For a state to satisfy Ψ_2, it has to have a c-edge leading to a state which satisfies Ψ_2 and Ψ_3. Finally, all b-successors of states satisfying Ψ_3 must satisfy Ψ_1.

Similar to Gaussian elimination in linear algebra, it is possible to eliminate variables in a vectorial fixed point to find closed formulas for the Ψ_i.

Theorem 7.2.6 (Gaussian elimination principle [AN01])**.** *Let a vectorial fixed point with $\theta \in \{\mu, \nu\}$ and $n > 1$ be given, which we write as:*

$$\begin{pmatrix}\Psi_1\\ \vdots\\ \Psi_n\end{pmatrix} = \theta \begin{pmatrix}X_1\\ \vdots\\ X_n\end{pmatrix} . \begin{pmatrix}\Phi_1\\ \vdots\\ \Phi_n\end{pmatrix}$$

Then with $\Phi'_i = \theta X_i.\Phi_i$ *for* $1 \leq i \leq n$ *the system is equivalent to:*

$$\begin{pmatrix} \Psi_1 \\ \vdots \\ \Psi_n \end{pmatrix} = \theta \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} . \begin{pmatrix} \Phi_1[\Phi'_i \leftarrow X_i] \\ \vdots \\ \Phi_{i-1}[\Phi'_i \leftarrow X_i] \\ \Phi'_i \\ \Phi_{i+1}[\Phi'_i \leftarrow X_i] \\ \vdots \\ \Phi_n[\Phi'_i \leftarrow X_i] \end{pmatrix}$$

The above theorem provides a way to eliminate a variable X_i from the system by substituting X_i with $\theta X_i.\Phi_i$ and replacing the i-th row of the system with this expression.

As an example of this theorem, we calculate closed formula solutions to our example. We begin with $i = 1$ and apply the theorem to eliminate variable X_1. This means we substitute X_1 with $\Phi'_1 = \nu X_1.[a]X_1 \wedge [b]X_2$ and replace the formula in the first row with this formula (the highlighted block is where X_1 was replaced with Φ'_1):

$$\begin{pmatrix} \Psi_1 \\ \Psi_2 \\ \Psi_3 \end{pmatrix} = \nu \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} . \begin{pmatrix} \nu X_1.[a]X_1 \wedge [b]X_2 \\ \langle c \rangle (X_2 \wedge X_3) \\ [b] \; (\nu X_1.[a]X_1 \wedge [b]X_2) \end{pmatrix}$$

Next, we pick[4] $i = 3$ and substitute $\Phi'_3 = \nu X_3.[b](\nu X_1.[a]X_1 \wedge [b]X_2)$ for X_3. Since X_3 does not actually appear inside the fixed point, we can instead use the equivalent formula without the fixed point, $\Phi'_3 \equiv [b](\nu X_1.[a]X_1 \wedge [b]X_2)$.

$$\begin{pmatrix} \Psi_1 \\ \Psi_2 \\ \Psi_3 \end{pmatrix} = \nu \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} . \begin{pmatrix} \nu X_1.[a]X_1 \wedge [b]X_2 \\ \langle c \rangle (X_2 \wedge \; [b](\nu X_1.[a]X_1 \wedge [b]X_2) \;) \\ [b](\nu X_1.[a]X_1 \wedge [b]X_2) \end{pmatrix}$$

Finally we handle $i = 2$ to get a solution without free variables in the right part:

$$\begin{pmatrix} \Psi_1 \\ \Psi_2 \\ \Psi_3 \end{pmatrix} = \nu \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} . \begin{pmatrix} \nu X_1.[a]X_1 \wedge [b] \; (\nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2))) \\ \nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2)) \\ [b](\nu X_1.[a]X_1 \wedge [b] \; (\nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2))) \;) \end{pmatrix}$$

Since the vectorial fixed point no longer binds any variables, we arrived at a closed solution to the vectorial fixed point:

$$\begin{aligned} \Psi_1 &= \nu X_1.[a]X_1 \wedge [b](\nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2))) \\ \Psi_2 &= \nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2)) \\ \Psi_3 &= [b](\nu X_1.[a]X_1 \wedge [b](\nu X_2.\langle c \rangle (X_2 \wedge [b](\nu X_1.[a]X_1 \wedge [b]X_2)))) \end{aligned}$$

This example shows that the resulting formulas can be quite large, compared to the input. In fact, an exponential blow-up is possible.

[4] This highlights that we do not have to process the variables in order. Plus, for this example, this results in a smaller solution.

7.3. Equivalence of Deterministic Modal Transition Systems and the Conjunctive ν-Calculus

Both mts and the modal μ-calculus are modal specifications. In this section we relate them to each other. Namely, a syntactic subset of the modal μ-calculus, which is called the *conjunctive ν-calculus*, is shown to be similarly expressive as deterministic modal transition systems.

Definition 7.3.1 (Conjunctive ν-calculus [Feu05b; FP07; Feu05a])**.** *A formula of the* conjunctive ν-calculus *is defined recursively as follows, where $X \in \mathsf{Var}$ and $a \in \Sigma$:*

$$\beta_1, \beta_2 ::= \texttt{true} \mid X \mid \rightarrow^a \mid \not\rightarrow^a \mid \beta_1 \wedge \beta_2 \mid \langle a \rangle \beta_1 \mid [a]\beta_1 \mid \nu X.\beta_1$$

A formula of the ν-calculus can be translated into the modal μ-calculus via $\rightarrow^a \equiv \langle a \rangle \texttt{true}$ and $\not\rightarrow^a \equiv [a]\texttt{false}$. The semantics of a formula of the ν-calculus is the semantics of its translation into the μ-calculus.

The conjunctive ν-calculus does not allow negation, disjunctions, or least fixed points. Some new atomic operations are introduced for expressions that are no longer possible, but still needed. Their semantics is defined via syntactic substitution into the modal μ-calculus. For example, one of these operators is $\not\rightarrow^a$, which is replaced with $[a]\texttt{false}$ and expresses that no a-edge is present. Since `false` is not expressible by mts, it cannot be allowed in the ν-calculus and thus $\not\rightarrow^a$ has to be introduced to express this specific situation, i.e. $[a]\texttt{false}$, which is expressible in mts.

Compared to the full modal μ-calculus, some things are missing[5]. There is no least fixed point, disjunction, and no negation, so the ν-calculus is not a complete predicate logic. There are two weak exceptions to this: $\not\rightarrow^a$ contains some flavour of negation, in that it forbids an edge with label a, and $[a]\beta_1$ expresses that either a is not enabled, or β_1 holds afterwards, which is a kind of disjunction.

7.3.1. Translating Deterministic Modal Transition Systems into the Conjunctive ν-Calculus

There are various translations for similar settings in the literature, e.g. [Lar89; Ben+13; FLT14]. The basic idea of the translation is to associate to each state s of the mts a variable X_s. This variable will be part of a vectorial fixed point system used to connect the formulas for the individual states to each other. The solution for the variable $X_\imath$, which corresponds to the initial state of the mts, will be the equivalent formula of the conjunctive ν-calculus.

To formalise this, let a deterministic mts $M = (S, \Sigma, \rightarrow, \dashrightarrow, \imath)$ be given. For a state $s \in S$ and an event $a \in \Sigma$, there are three possibilities: Either there is a must edge with

[5] We will see in Section 7.3.2 that `false` is expressible in the conjunctive ν-calculus.

label a emanating from s, or a may edge but no must edge is present, or neither a must edge nor a may edge exists. Due to determinism, there cannot be multiple such edges. If a may edge $s \overset{a}{\dashrightarrow} s'$ is present, then the formula for X_s must contain $[a]X_{s'}$, which expresses that with label a we may go to the formula corresponding to state s'. If this is additionally a must edge $s \xrightarrow{a} s'$, then the formula will be $\rightarrow^a \wedge [a]X_{s'}$. The additional part $\rightarrow^a$ expresses the requirement that an a-edge must be present. If no may edge is present, then a can be forbidden via $\not\dashrightarrow^a$.

Definition 7.3.2 (From mts to ν-calculus)**.** *The formula for event a and state s is:*

$$edge_a(s) = \begin{cases} [a]X_{s'} & \text{if } \exists s' \in S\colon (s,a,s') \in \dashrightarrow \setminus \rightarrow \\ [a]X_{s'} \wedge \rightarrow^a & \text{if } \exists s' \in S\colon (s,a,s') \in \rightarrow \\ \not\dashrightarrow^a & \text{if } \forall s' \in S\colon (s,a,s') \notin \dashrightarrow \end{cases}$$

The formula $state_s = \bigwedge_{a\in\Sigma} edge_a(s)$ for state s is the conjunction of the individual formulas for the edges. This leads to the vectorial fixed point system in the variables X_s with $s \in S$, where the definition of the variable X_s for state s is $state_s$:

$$\begin{pmatrix} \Psi_\imath \\ \vdots \\ \Psi_{s_n} \end{pmatrix} = \nu \begin{pmatrix} X_\imath \\ \vdots \\ X_{s_n} \end{pmatrix} . \begin{pmatrix} state_\imath \\ \vdots \\ state_{s_n} \end{pmatrix}$$

A solution to this vectorial fixed point system produces a closed formula $\Phi_\imath$ that describes the behaviour of the initial state. For a deterministic modal transition system M, the corresponding formula Φ_M *is the solution $\Phi_\imath$ for its initial state.*

Theorem 7.3.3. *Given a deterministic mts $M = (S_M, \Sigma, \rightarrow, \dashrightarrow, \imath_M)$ and a deterministic lts $A = (S_A, \Sigma, \rightarrow, \imath_A)$, then $A \models M$ if and only if $A \models \Phi_M$.*

Proof. ($\Rightarrow$) Assume that $A \models M$ via R. By the definition above, Φ_M is a fixed point with variables X_s for each state $s \in S$ of the mts. We define a valuation val via $\mathsf{val}(X_s) = \{q \in S_A \mid (q,s) \in R\}$ that assigns to the variable X_s the set of all states of the lts that are in relation to state s. It can now be verified that with this valuation, $(q,s) \in R$ implies that state q of the lts satisfies the formula $state_s$, i.e. $q \in [\![state_s]\!]_A^{\mathsf{val}}$: Since $state_s$ is just the conjunction of the formulas $edge_a(s)$ of the individual edges, it is enough to verify that each of these formulas are satisfied.

If the state s in the mts has no outgoing a-edge, then by $A \models M$ via R, the state q of the lts cannot have an outgoing a-edge either. Thus, it satisfies $edge_a(s) = \not\dashrightarrow^a$.

If the state s in the mts has an outgoing must edge for a ($(s,a,s') \in \rightarrow$ for some state s'), then by $A \models M$ via R, there must be an edge $q \xrightarrow{a} q'$ in the lts for some state q', so that $(q',s') \in R$. Thus, $q' \in \mathsf{val}(X_{s'})$ and s satisfies $edge_a(s) = [a]X_{s'} \wedge \rightarrow^a$.

If the state s in the mts has an outgoing may edge for a, but no must edge, ($(s,a,s') \in \dashrightarrow \setminus \rightarrow$ for some state s'), then there are two possibilities. If q has no outgoing a-edge, it satisfies $edge_a(s) = [a]X_{s'}$ and nothing remains to be shown. If there is a state q' so

that $q \xrightarrow{a} q'$, then by $A \models M$ via R the element (q', s') must be in R. Thus, as in the previous case, $q' \in \mathsf{val}(X_{s'})$ and s satisfies $edge_a(s)$.

($\Leftarrow$) Assume that $A \models \Phi_M$. Since Φ_M was constructed as a vectorial fixed point system, A must also satisfy the fixed point system itself. This means that there is a valuation val so that $\imath_A \in [\![state_{\imath_M}]\!]_A^{\mathsf{val}}$. We define a relation R via $R = \{(q, s) \in S_A \times S_M \mid q \in \mathsf{val}(X_s)\}$ and will show that $A \models M$ via R. By $\imath_A \in [\![state_{\imath_M}]\!]_A^{\mathsf{val}} = \mathsf{val}(X_{\imath_M})$, we now have $(\imath_A, \imath_M) \in R$, as required for $A \models M$.

Let $(q, s) \in R$ be an arbitrary element. We have to show that $\forall q \xrightarrow{a} q' \colon \exists s \overset{a}{\dashrightarrow} s' \colon (q', s') \in R$ and $\forall s \xrightarrow{a} s' \colon \exists q \xrightarrow{a} q' \colon (q', s') \in R$. We begin with the second condition.

Let $(s, a, s') \in \rightarrow$ be a must edge of M. By definition we have $edge_a(s) = [a]X_{s'} \wedge {\rightarrow^a}$. Since this formula is part of $state_s$, which is satisfied by q, q must have an outgoing a-edge (${\rightarrow^a}$). By the first part of the formula, the state q', that is reached via a, must satisfy $X_{s'}$. Thus, $q' \in \mathsf{val}(X_{s'})$ and $(q', s') \in R$, as required.

For the first condition, let $q \xrightarrow{a} q'$ be an edge of A. By assumption $q \in [\![edge_a(s)]\!]_A^{\mathsf{val}}$. If $edge_a(s) = {\not\rightarrow^a}$, then q could not have an outgoing a-edge, so this is not the case. This means by definition of $edge_a(s)$ that s must have an outgoing may edge with label a, i.e. there is a state s' of M with $(s, a, s') \in \dashrightarrow$. Thus, $[a]X_{s'}$ necessarily appears in $edge_a(s)$. Since q satisfies this formula and $q \xrightarrow{a} q'$ is an edge in A, we must have $q' \in \mathsf{val}(X_{s'})$ and so $(q', s') \in R$. □

7.3.2. Translating Closed Formulas of the Conjunctive ν-Calculus into Deterministic Modal Transition Systems

Translating a free variable X into an mts makes no sense, because the meaning of X is only given by the valuation when the formula is evaluated. Thus, only closed formulas, i.e. formulas without free variables, can be translated into mts.

The construction for translating a closed formula into an mts is inspired by [FP07; Feu05b; Feu05a], where formulas are translated into a language-based model called *modal specifications*. The construction is a bit involved, because subformulas with free variables have to be handled appropriately. The translation will only be sketched here.

One complication is that the conjunctive ν-calculus is more expressive than mts: The formula ${\rightarrow^a} \wedge {\not\rightarrow^a}$ requires the event a to be enabled, but at the same time forbids this. This formula is unsatisfiable and thus expresses `false`, which cannot be expressed by an mts (see Lemma 7.1.3). This is handled by only providing a translation for formulas which are not equivalent to `false`. Thus, the translation either produces a deterministic mts, or concludes that the given formula is unsatisfiable.

Translating most formulas is relatively straightforward: The atomic propositions specify paths that must be present and paths after which some event may not be possible. For example, `true` corresponds to a single-state mts in which each event has a may edge

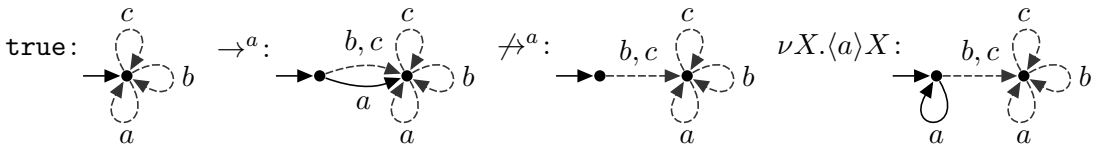


Figure 7.5.: Examples of the translations into mts assuming an alphabet of $\Sigma = \{a, b, c\}$.

that goes back to the initial state. An example of this construction that assumes that the alphabet is $\Sigma = \{a, b, c\}$ is shown on the left of Figure 7.5.

While the mts so far have only a single state, the mts for $\rightarrow^a$ requires two states and is interpreted as $\langle a \rangle \texttt{true}$: The initial state has a must edge for a going to a state that represents `true`. All other labels go via may edges to a state for `true`, from both the initial state as well as the state for `true` itself. An example is shown in the middle of Figure 7.5. The modalities $\langle a \rangle \beta$, $[a]\beta$, and $\not\rightarrow^a$ are handled like this as well: Either a must and a may edge, just a may edge, or no may edge, respectively, with label a goes to the state for formula β, depending on the kind of modality that is present. All other events have may edges going to the state for `true`. An example for $\not\rightarrow^a$ is shown in Figure 7.5.

The only remaining operators are the conjunction $\beta_1 \wedge \beta_2$ and the fixed point $\nu X.\beta$. These are more complicated to handle. For a fixed point $\nu X.\beta$, the formula β is translated and the free variable X corresponds to a single state. Afterwards, the state for X is identified with the initial state of the resulting mts. This captures the intuition that when reaching the formula X, the whole fixed point formula may be traversed again. Figure 7.5 has an example for the formula $\nu X.\langle a \rangle X$.

A conjunction $\beta_1 \wedge \beta_2$ can be handled by constructing the mts for the individual formulas. Then, a synchronised product of the two mts is constructed, meaning that each state of the mts for $\beta_1 \wedge \beta_2$ is a pair (s_1, s_2) of states of β_1 and β_2. The edges between these states are constructed as dictated by the individual mts, meaning that both s_1 and s_2 must participate in an a-edge for (s_1, s_2) and the edge is not possible if one of the underlying states does not allow it. An edge is a must edge if at least one of the underlying edges is a must edge. This is very similar to the standard construction for the union and intersection of languages of deterministic finite automata [HU79].

As outlined above, this construction fails when a must edge is present on one state, but the other state of the pair does not have a corresponding may edge. In this case, the resulting state is said to be *inconsistent* and is equivalent to `false`. Any state with a must edge to an inconsistent state is also inconsistent. All inconsistent states and their connected edges are removed in the final mts, i.e. any edge which would lead to `false` is handled by forbidding the edge. If the initial state is inconsistent, the full formula is unsatisfiable and equivalent to `false`. For example, when constructing the mts for $\rightarrow^a \wedge \not\rightarrow^a$, we can use the two mts for $\rightarrow^a$ and $\not\rightarrow^a$ from Figure 7.5. We see that one of them has a must edge with label a emanating from the initial state, while the other

one does not have a may edge with that label. Thus, the initial state of the product automaton is inconsistent and the formula $\rightarrow^a \wedge \not\rightarrow^a$ is equivalent to `false`.

The direct construction that was sketched so far is a simplification of the actual construction from [FP07; Feu05b; Feu05a], which translates between the conjunctive ν-calculus and so-called modal specifications. The presented construction sketch could inductively be proven correct, but this will not be done here.

7.4. Bibliographical Remarks

This chapter introduced modal transition systems and the modal μ-calculus. A syntactic fragment of the modal μ-calculus, which is called the conjunctive ν-calculus, was presented and its equivalence with mts was shown via constructions inspired from [Lar89; Ben+13; FLT14] and [Feu05b; FP07; Feu05a].

Modal transition systems were originally introduced by Larsen and Thomsen [LT88; Lar89] and have been extended since then in various directions [Ant+08; Kre17], for example, to express disjunction [LX90], and have been applied to specify e.g. network protocols [Bru97].

A translation between mts and Hennessy-Milner-Logic without negation was already introduced in [BL92] based on a normal form for formulas. A translation between mts with disjunctive edges and the disjunctive ν-calculus was shown in [Ben+13] and [FLT14]. These translations are similar to the approach presented here, where states of an mts are represented by variables. The converse translation is based on a conjunctive normal form of formulas. Restricted to deterministic mts, this last construction produces formulas of the conjunctive ν-calculus.

Another specification that can be used as input for Petri net synthesis are *path-automatic specifications*. While the implementation relation for mts and the interpretation of the μ-calculus is related to bisimulation, path-automatic specifications have an interpretation closer to isomorphism. Petri net synthesis for path-automatic specifications was shown to be decidable by Badouel and Darondeau [BD04]. Since the next chapter shows that this problem is undecidable for mts, path-automatic specifications are, in a sense, a weaker specification language.

Petri nets were directly extended with modalities in [EHH12; Bri16], producing *modal Petri nets* and turning Petri nets into a modal specification language. The present document does not investigate such extensions that increase the expressivity of Petri nets and even limits itself to injectively labelled Petri nets. The reachability graph of such a net is deterministic. In deterministic systems, several problems on modal transition systems become easier [BKLS09].

8. Undecidability of Bounded Modal Realisation

In this chapter, Petri net synthesis from modal specifications is examined, meaning that for a given modal specification, a Petri net realisation should be found. This problem can be stated for mts, for the modal μ-calculus, and for the conjunctive ν-calculus. Since satisfiable formulas of the conjunctive ν-calculus and mts are equivalent, as shown in Section 7.3, and since the ν-calculus is less expressive than the full modal μ-calculus, it is enough to derive undecidability for the conjunctive ν-calculus, because undecidability results for other kinds of modal specifications follow.

To be precise, the following two problems are shown to be undecidable:

Problem 8.0.1 (Bounded realisation problem)**.** *Given a formula β of the conjunctive ν-calculus, is there a bounded Petri net N so that $N \models M$?*

Problem 8.0.2 (Bounded and pure realisation problem)**.** *Given a formula β of the conjunctive ν-calculus, is there a pure and bounded Petri net N so that $N \models M$?*

In this chapter, we present a reduction from the bounded execution problem of two-counter machines, which is undecidable and introduced in Section 8.1. This problem asks if there are bounds (b_0, b_1) that are not exceeded in the execution of the two-counter machine.

For the reduction to ν-calculus, a family of Petri nets $N_{\text{sim}}(b_0, b_1)$ is introduced in Section 8.2. Petri nets from this family can simulate two counters as long as the first counter does not exceed the value b_0 and the second counter does not exceed b_1. The program of a two-counter machine $\mathcal{C}$ can be encoded into a formula $\Phi_{\mathcal{C}}$ so that $N_{\text{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$ if and only if the unique execution of the machine does not exceed the bounds. Thus, the existence of values (b_0, b_1) so that $N_{\text{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$ encodes the bounded execution problem and is undecidable. Next, a formula $\Phi_{N_{\text{sim}}}$ is introduced in Section 8.3 that characterises the Petri nets $N_{\text{sim}}(b_0, b_1)$ independently of the concrete bounds of the counter. This characterisation also works if only pure Petri nets are considered.

In the end, a bounded Petri net N satisfying $N \models \Phi_{N_{\text{sim}}} \wedge \Phi_{\mathcal{C}}$ exists if and only if the execution of the two-counter machine $\mathcal{C}$ is bounded. Since the latter problem is undecidable, the existence of a suitable Petri net N is undecidable, too.

This chapter is based on the author's publication [Sch16a], but we extend the approach from bounded Petri nets to pure and bounded Petri nets.

8.1. Two-Counter Machines

Two-counter machines, also known as Minsky machines, are a simple yet Turing-complete model of computation. They are specified by a program, which is a list of instructions. When the machine is started, its two counters are initialised to zero. Its execution consists of a series of configurations, where each configuration describes the current value of the two counters and a pointer to the next instruction to be executed. In our setting there are three types of instructions: Increment instructions, test-and-decrement instructions, and a halt instruction. An example of a program is given in Algorithm 3.

Algorithm 3 Example of the program of a two-counter machine.

```
1: INC_0 2
2: INC_1 3
3: DEC_0 2 ELSE 4
4: DEC_1 3 ELSE 5
5: HALT
```

When an increment instruction is executed, the counter that the instruction refers to is incremented by one and execution continues at a specified instruction. We represent such an instruction as $\texttt{INC}_i\ k$, which means that counter $i \in \{0, 1\}$ is incremented and the next instruction to execute has index k. For example, the machine in Algorithm 3 begins its execution with the first instruction. The execution of this instruction increments the counter zero and continues execution with the second instruction.

A test-and-decrement instruction is represented as $\texttt{DEC}_i\ k\ \texttt{ELSE}\ k'$. If the value of counter i is positive, it is decremented by one and execution continues with the k-th instruction. Alternatively, if the value of counter i is zero, execution continues with instruction k'[1]. For example, the third instruction in Algorithm 3 decrements counter zero. If the counter was non-zero, execution continues with instruction two. Otherwise, the values of the counters are not modified and execution continues with instruction four.

Finally, the halt instruction HALT terminates the execution of the machine.

Definition 8.1.1 (Two-counter machines [Min67])**.** *A* two-counter machine *is a tuple* $\mathcal{C} = (\ell, \gamma)$*, where* $\ell \in \mathbb{N}$ *is the number of states and* $\gamma\colon \{1, \dots, \ell\} \to \Gamma_\ell$ *maps states to instructions, where* $\Gamma_\ell = \{\texttt{INC}_i\ k \mid i \in \{0, 1\}, k \in \{1, \dots, \ell\}\} \cup \{\texttt{DEC}_i\ k\ \texttt{ELSE}\ k' \mid i \in \{0, 1\}, k, k' \in \{1, \dots, \ell\}\} \cup \{\texttt{HALT}\}$ *is the set of possible instructions.*

A configuration *of* (ℓ, γ) *is a tuple* $c = (k, (j_0, j_1))$*, where* $k \in \{1, \dots, \ell\}$ *is the current state and* $j_0, j_1 \in \mathbb{N}$ *are the values of the counters. The* execution *of* (ℓ, γ) *is the*

[1] This was implicit in Minsky's original definition. We give the next instruction explicitly in both cases.

maximal sequence of configurations, which begins with $(1, (0, 0))$*, and a configuration* $c = (k, (j_0, j_1))$ *is followed by* $c' = (k', (j_0', j_1'))$ *if* $\gamma(k) \neq \texttt{HALT}$ *and*

- *if* $\gamma(k) = \texttt{INC}_i\ k''$*, then* $k' = k''$*,* $j_i' = j_i + 1$*, and* $j_{1-i}' = j_{1-i}$*,*
- *if* $\gamma(k) = \texttt{DEC}_i\ k''\ \texttt{ELSE}\ k'''$*, then*
  - *either* $j_i = 0$*,* $k' = k'''$*,* $j_0' = j_0$*, and* $j_1' = j_1$*,*
  - *or* $j_i \neq 0$*,* $k' = k''$*,* $j_i' = j_i - 1$*, and* $j_{1-i}' = j_{1-i}$*.*

Since the next configuration of a two-counter machine is uniquely determined, two-counter machines are deterministic, i.e. there is a unique execution.

The execution of the two-counter machine from Algorithm 3 is the following sequence:
$(1, (0, 0))$, $(2, (1, 0))$, $(3, (1, 1))$, $(2, (0, 1))$, $(3, (0, 2))$, $(4, (0, 2))$,
$(3, (0, 1))$, $(4, (0, 1))$, $(3, (0, 0))$, $(4, (0, 0))$, $(5, (0, 0))$.
This machine begins by incrementing its first counter once (instruction 1). In a loop, it then increments the second counter and decrements the first (instructions 2–3). When the first counter reaches zero, another loop is done decrementing the second counter (instructions 3–4). Instruction 5 terminates the execution.

Furthermore, the following notions will be important:

Definition 8.1.2 (Halt, bounded execution)**.** *A two-counter machine* $\mathcal{C}$ halts *if its execution is finite, i.e. a configuration with instruction* `HALT` *is reached. The execution is called* bounded by (b_0, b_1) *if all of its configurations* $(k, (j_0, j_1))$ *satisfy* $j_0 \leq b_0$ *and* $j_1 \leq b_1$*. The execution is called* bounded *if such* (b_0, b_1) *exist.*

The machine from Algorithm 3 halts, because we just saw that its execution reached an instruction `HALT`. We can also see that its execution is bounded, for example, by $(42, 42)$. The tightest possible bound is $(1, 2)$, meaning that both of these numbers are actually reached. However, no configuration reaches both bounds at the same time.

While two-counter machines are quite a simple model of computation, they are still Turing-complete. In particular, their halting problem is undecidable:

Theorem 8.1.3 ([Min67])**.** *Given a two-counter machine* $\mathcal{C}$*, it is undecidable if* $\mathcal{C}$ *halts.*

For technical reasons, we are interested in a slightly different problem that is still undecidable: The bounded execution problem. This problem is to determine whether the execution of a given two-counter machine is bounded.

While the halting problem cannot be encoded directly into the conjunctive ν-calculus, because this would require a least fixed point operator, we will later see that the bounded execution problem can be encoded into a formula.

Theorem 8.1.4. *Given a machine $\mathcal{C}$, it is undecidable if its execution is bounded.*

Proof. Assume for contradiction that we could decide whether a given two-counter machine has a bounded execution. This would allow to decide the halting problem as follows. Since this problem is undecidable by Theorem 8.1.3, the bounded execution problem must be undecidable as well.

Given a two-counter machine $\mathcal{C}$, decide whether it has a bounded execution. If it does not, then by definition this means that for any number $n \in \mathbb{N}$ there is a configuration in its execution where some counter has a value $\geq n$. Since there are infinitely many natural numbers and two-counter machines can only increment their counters one step at a time, the execution of $\mathcal{C}$ must also have infinitely many configurations. Thus, the machine does not halt.

If the execution is bounded, we can decide the halting problem by brute force: All three values k, j_0, and j_1 in an arbitrary configuration $(k, (j_0, j_1))$ in the execution of $\mathcal{C}$ are bounded, hence there are only finitely many different configurations. We simulate the execution of $\mathcal{C}$ and check if any configuration appears twice. If the simulation halts, then $\mathcal{C}$ obvious halts. Otherwise, eventually some configuration is seen for a second time and since two-counter machines are deterministic, we can be sure that $\mathcal{C}$ is in a loop and does not halt. □

8.2. Simulating Two-Counter Machines with Petri Nets and the Conjunctive ν-Calculus

This section introduces a family of pure and bounded Petri nets $N_{\text{sim}}(b_0, b_1)$ with transitions $\Sigma = \{\oplus, \ominus, ⓣ, ⓪, \boxplus, \boxminus, \boxed{\text{T}}, \boxed{0}\}$ and parameters $b_0, b_1 \in \mathbb{N}$. They are defined such that a two-counter machine $\mathcal{C}$ can be simulated on the reachability graph of such a net via a formula $\Phi_{\mathcal{C}}$ of the conjunctive ν-calculus if and only if its execution is bounded by (b_0, b_1).

8.2.1. The Family of Petri Nets

A prototypical member of the family $N_{\text{sim}}(b_0, b_1)$ is depicted in Figure 8.1. The Petri nets from this family can simulate two bounded counters and have two disconnected parts, each simulating one of the counters. The values of the counters are the number of tokens on p_0 and p_1, respectively. Each counter has an initial value of zero, a capacity of b_0, and b_1, respectively (this is because the complement places $\overline{p_i}$ have b_i tokens initially), and can be incremented, decremented and tested for zero via transitions $\oplus$, $\ominus$ and ⓣ ($\boxplus$, $\boxminus$ and $\boxed{\text{T}}$, respectively). After a zero test, transition ⓪ (or $\boxed{0}$) has to fire before the simulation can continue. Every reachable marking M satisfies, by the structure of the net, either $M(p_0) + M(\overline{p_0}) = b_0$ or $M(e_0) = 1$, and similarly either $M(p_1) + M(\overline{p_1}) = b_1$

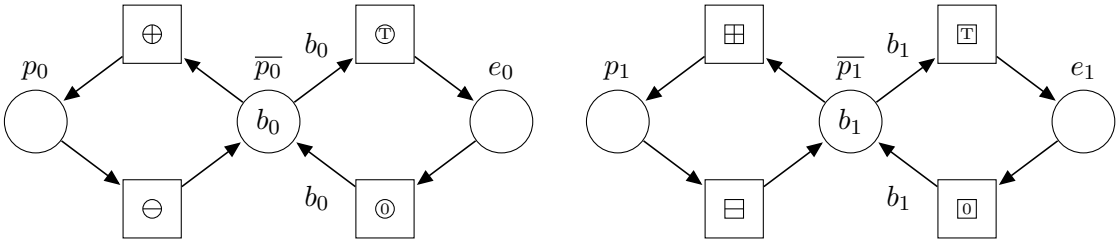


Figure 8.1.: The family of Petri nets $N_{\text{sim}}(b_0, b_1)$ with $b_0, b_1 \in \mathbb{N}$.

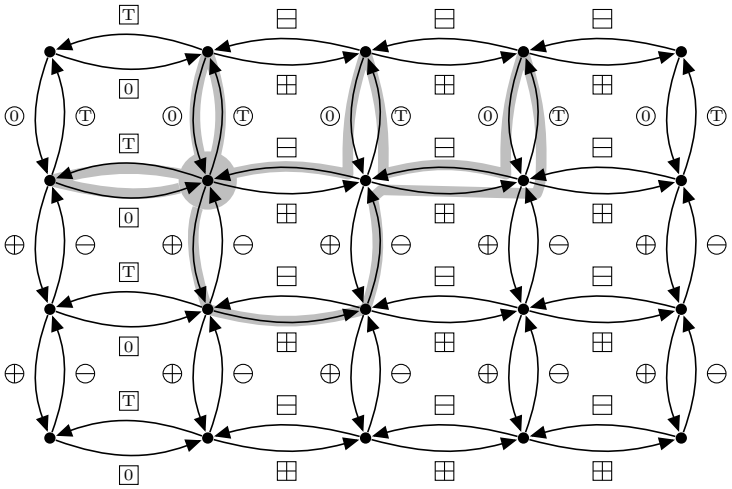


Figure 8.2.: The reachability graph of $N_{\text{sim}}(2, 3)$. For space reasons, the initial state is highlighted in grey instead of being marked with an arrow. Additionally, the firing sequence ⊕⊞⊖⊞Ⓣ⓪⊟Ⓣ⓪⊟Ⓣ⓪T0 is highlighted.

or $M(e_1) = 1$. As an example of the behaviour of these Petri nets, the reachability graph $\text{RG}(N_{\text{sim}}(2, 3))$ is shown in Figure 8.2.

Lemma 8.2.1. *For $w \in \{\oplus, \ominus, \text{Ⓣ⓪}, \boxplus, \boxminus, \text{T0}\}$ and $M \in \mathfrak{E}(N_{sim}(b_0, b_1))$: $M[w\rangle M'$ if and only if:*

- *For $w = \oplus$ we have $M'(p_0) = M(p_0) + 1 \leq b_0$ and $M'(p_1) = M(p_1)$.*
- *For $w = \ominus$ we have $M'(p_0) = M(p_0) - 1 \geq 0$ and $M'(p_1) = M(p_1)$.*
- *For $w = \text{Ⓣ⓪}$ we have $M' = M$ and $M(p_0) = 0$.*
- *Analogously for $w \in \{\boxplus, \boxminus, \text{T0}\}$ (the second counter).*

Proof. This lemma follows from the structure and behaviour of each net $N_{\text{sim}}(b_0, b_1)$. □

8.2.2. Formula for a Two-Counter Machine

We define a formula $\Phi_{\mathcal{C}}$ describing a two-counter machine $\mathcal{C}$. By exploiting the structure of $N_{\text{sim}}(b_0, b_1)$, such a formula is satisfied on $N_{\text{sim}}(b_0, b_1)$ if and only if the execution of $\mathcal{C}$ is bounded by (b_0, b_1).

Definition 8.2.2 (Simulating formula)**.** *Given a two-counter machine $\mathcal{C} = (\ell, \gamma)$, the formula Φ_k for a state $k \in \{1, \ldots, \ell\}$ in the variables X_1 to X_ℓ is defined by:*

$$\Phi_k = \begin{cases} \langle \oplus \rangle X_{k'} & \text{if } \gamma(k) = \texttt{INC}_0\ k' \\ \langle \boxplus \rangle X_{k'} & \text{if } \gamma(k) = \texttt{INC}_1\ k' \\ [\ominus] X_{k'} \wedge [\textcircled{T}] \langle \textcircled{0} \rangle X_{k''} & \text{if } \gamma(k) = \texttt{DEC}_0\ k'\ \texttt{ELSE}\ k'' \\ [\boxminus] X_{k'} \wedge [\boxed{\text{T}}] \langle \boxed{0} \rangle X_{k''} & \text{if } \gamma(k) = \texttt{DEC}_1\ k'\ \texttt{ELSE}\ k'' \\ \texttt{true} & \text{if } \gamma(k) = \texttt{HALT} \end{cases}$$

This is used to define the following vectorial equation. The formula $\Psi_{\mathcal{C}}$ of the machine $\mathcal{C}$ is a closed solution for Ψ_1, the formula for the initial state (see Theorem 7.2.6).

$$\begin{pmatrix} \Psi_1 \\ \vdots \\ \Psi_\ell \end{pmatrix} = \nu \begin{pmatrix} X_1 \\ \vdots \\ X_\ell \end{pmatrix} \cdot \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_\ell \end{pmatrix}$$

Intuitively we can understand that Φ_k is fulfilled if the behaviour of state k can be simulated. The free variables are used to connect the individual formulas for the states with each other. For the increment operation, the corresponding event has to be possible and afterwards the following state should be simulated. The decrement operation is more complicated, because there are two possibilities for the following state. To implement this, the structure of $N_{\text{sim}}(b_0, b_1)$ is exploited. In every relevant reachable marking, exactly one of the transitions $\ominus$ and $\textcircled{T}$ ($\boxminus$ and $\boxed{\text{T}}$, resp.) is enabled. Thus, the $[a]$-operator can be used to express this choice, even though the conjunctive ν-calculus does not have a disjunction operator. The increment operation uses the $\langle a \rangle$-operator instead, which will make the simulation fail if a counter needs to be incremented beyond the bound b_0 or b_1 of $N_{\text{sim}}(b_0, b_1)$, because this transition is disabled.

The following examples show the construction:

Example 8.2.3. We will construct $\Phi_{\mathcal{C}}$ for the two-counter machine from Algorithm 3 on page 80. We repeat its execution here, but this time we use the transition names that are used in the family $N_{\text{sim}}(b_0, b_1)$ to label the edge from one configuration to another:

$$(1,(0,0)) \xrightarrow{\oplus} (2,(1,0)) \xrightarrow{\boxplus} (3,(1,1)) \xrightarrow{\ominus} (2,(0,1)) \xrightarrow{\boxplus} (3,(0,2)) \xrightarrow{\textcircled{T}\textcircled{0}} (4,(0,2)) \xrightarrow{\boxminus}$$
$$(3,(0,1)) \xrightarrow{\textcircled{T}\textcircled{0}} (4,(0,1)) \xrightarrow{\boxminus} (3,(0,0)) \xrightarrow{\textcircled{T}\textcircled{0}} (4,(0,0)) \xrightarrow{\boxed{\text{T}}\boxed{0}} (5,(0,0))$$

The word $\oplus\boxplus\ominus\boxplus\textcircled{T}\textcircled{0}\boxminus\textcircled{T}\textcircled{0}\boxminus\textcircled{T}\textcircled{0}\boxed{\text{T}}\boxed{0}$ is constructed from the individual edges above and represents the execution of the machine. This path is highlighted in Figure 8.2.

The individual formulas for each state are shown in the following system, where a closed formula as a solution to Ψ_1 is needed for $\Phi_{\mathcal{C}}$:

$$\begin{pmatrix}\Psi_1\\ \Psi_2\\ \Psi_3\\ \Psi_4\\ \Psi_5\end{pmatrix} = \nu \begin{pmatrix}X_1\\ X_2\\ X_3\\ X_4\\ X_5\end{pmatrix} . \begin{pmatrix}\langle\oplus\rangle X_2\\ \langle\boxplus\rangle X_3\\ [\ominus]X_2 \wedge [\textcircled{T}]\langle\textcircled{0}\rangle X_4\\ [\boxminus]X_3 \wedge [\boxed{\mathrm{T}}]\langle\boxed{0}\rangle X_5\\ \mathtt{true}\end{pmatrix}$$

The Gaussian elimination principle (see Theorem 7.2.6) now eliminates variables. This results in a formula that can be simplified[2] into:

$$\Phi_{\mathcal{C}} = \langle\oplus\rangle(\nu X_2.\langle\boxplus\rangle(\nu X_3.[\ominus]X_2 \wedge [\textcircled{T}]\langle\textcircled{0}\rangle([\boxminus]X_3 \wedge [\boxed{\mathrm{T}}]\langle\boxed{0}\rangle\mathtt{true})))$$

As we saw above, the execution of $\mathcal{C}$ is bounded by $(2,3)$. We can verify that $N_{\mathrm{sim}}(2,3) \models \Phi_{\mathcal{C}}$: This means that the initial state of the reachability graph from Figure 8.2 satisfies the formula, i.e. we begin with the formula $\langle\oplus\rangle X_2$ for X_1. The event $\oplus$ leads to a state where $\langle\boxplus\rangle X_3$ has to hold, the formula for X_2. After $\boxplus$, the formula $[\boxminus]X_2 \wedge [\boxed{\mathrm{T}}]\langle\boxed{0}\rangle X_4$ is considered. Because $\boxed{\mathrm{T}}$ is not enabled in the current state, we continue with the state reached by $\boxminus$ and the formula for X_2. In this way we follow $\oplus\boxplus\ominus\boxplus\textcircled{T}\textcircled{0}\boxminus\textcircled{T}\textcircled{0}\boxminus\textcircled{T}\textcircled{0}\boxed{\mathrm{T}}\boxed{0}$ through the reachability graph of $N_{\mathrm{sim}}(2,3)$ (compare Figure 8.2), which is the word representing the execution of $\mathcal{C}$.

Example 8.2.4. An example of a machine with a bounded execution, but which does not halt, is the machine $\mathcal{C}'$ whose only instruction is $\mathtt{DEC}_0$ 1 $\mathtt{ELSE}$ 1. This machine loops in its initial configuration by repeatedly testing its counter zero for value zero. Its formula is $\Phi_{\mathcal{C}'} = \nu X_1.[\ominus]X_1 \wedge [\textcircled{T}]\langle\textcircled{0}\rangle X_1$. Here we have $N_{\mathrm{sim}}(0,0) \models \Phi_{\mathcal{C}'}$ and the 'infinite word' representing a correct simulation is $(\textcircled{T}\textcircled{0})^\omega$. This word is allowed by all simulating Petri nets $N_{\mathrm{sim}}(b_0, b_1)$.

If instead we take the machine $\mathcal{C}''$ with the instruction $\mathtt{INC}_0$ 1, which does infinitely many increments, we get a machine with an unbounded execution. This execution is represented by $\oplus^\omega$. No instance of $N_{\mathrm{sim}}(b_0, b_1)$ allows an infinite sequence of increments. The corresponding formula is $\Phi_{\mathcal{C}''} = \nu X_1.\langle\oplus\rangle X_1$.

The next two lemmas show that the formula $\Phi_{\mathcal{C}}$ simulates the machine $\mathcal{C}$ on Petri nets $N_{\mathrm{sim}}(b_0, b_1)$. Together they show that $\Phi_{\mathcal{C}}$ holds on $N_{\mathrm{sim}}(b_0, b_1)$ exactly if $\mathcal{C}$ is bounded.

Lemma 8.2.5 ([Feu05b])**.** *If the execution of a two-counter machine $\mathcal{C}$ is bounded by $(b_0, b_1) \in \mathbb{N} \times \mathbb{N}$, then $N_{sim}(b_0, b_1) \models \Phi_{\mathcal{C}}$.*

Proof. Given a two-counter machine $\mathcal{C} = (\ell, \gamma)$, inductively define a sequence of words w_i that describe its execution, as follows below. These words will all be in the language of $N_{\mathrm{sim}}(b_0, b_1)$ and will witness that $\Phi_{\mathcal{C}}$ is fulfilled.

[2] The formulas $\nu X.\Phi$ and Φ are equivalent if X does not appear as a free variable in Φ. This is used to remove variables X_1, X_4, and X_5.

The initial configuration is associated with the word $w_0 = \varepsilon$. If the action done in configuration c_i is an increment of the first counter, then $w_{i+1} = w_i\oplus$. Similarly, if the counter is decremented, then $w_{i+1} = w_i\ominus$. If instead a zero test was successful, then $w_{i+1} = w_i\textcircled{T}\textcircled{0}$. The corresponding operations on the second counter are handled analogously.

By Lemma 8.2.1, all these words are in the language of the Petri net since, by assumption, no bound of a counter is exceeded. Thus, we can define a marking M_i of $N_{\mathrm{sim}}(b_0, b_1)$ for each i, namely the marking reached by w_i. Note that the empty word $w_0 = \varepsilon$ reaches the initial marking M_0 of the Petri net. This gives us a relation between the i-th configuration c_i, the i-th reaching word w_i, and the i-th marking M_i.

Next, the markings M_i are grouped into sets $V_k = \{M_i \mid c_i = (k, (j_0, j_1))\}$, where $k \in \{1, \ldots, \ell\}$ is a state of $\mathcal{C}$. The set V_k contains all markings M_i where the corresponding configuration c_i is in state k. Via this, define a valuation val by $\mathsf{val}(X_k) = V_k$. This valuations assigns a variable X_k representing state k to the set V_k.

With this valuation we want to show that $[\![\Phi_k]\!]_A^{\mathsf{val}} \supseteq V_k$ for all k, where Φ_k is the formula corresponding to state k (Definition 8.2.2) and $A = \mathrm{RG}(N_{\mathrm{sim}}(b_0, b_1))$ is the lts under consideration. When this holds, then $\mathsf{val}(X_k)$ is a so called pre-fixed point[3], which is contained in the greatest fixed point, because the greatest fixed point is the union of all pre-fixed points (cf. Definition 7.2.3). Because $\Phi_{\mathcal{C}}$ is defined to be the greatest fixed point, this means that $\mathsf{val}(X_1) \subseteq [\![\Phi_{\mathcal{C}}]\!]_A$. By construction we now have $M_0 \in V_1 = \mathsf{val}(X_1) \subseteq [\![\Phi_{\mathcal{C}}]\!]_A$, which by definition means that $A \models \Phi_{\mathcal{C}}$ and was to be shown.

Thus, it remains to show that for $M_i \in V_k$ also $M_i \in [\![\Phi_k]\!]_A^{\mathsf{val}}$. This is done by case analysis on $\gamma(k)$. If $\gamma(k) = \texttt{HALT}$, then $\Phi_k = \texttt{true}$, so $[\![\Phi_k]\!]_A^{\mathsf{val}}$ contains all states of A, including M_i.

If $\gamma(k) = \texttt{INC}_0\ k'$, then $\Phi_k = \langle\oplus\rangle X_{k'}$. By construction, $M_{i+1} \in V_{k'}$ holds, because the following configuration c_{i+1} exists and is in state k'. Also, $M_i[\oplus\rangle M_{i+1}$ by Lemma 8.2.1. Since $\mathsf{val}(X_{k'}) = V_{k'}$ we thus have $M_i \in [\![\Phi_k]\!]_A^{\mathsf{val}}$ by definition of the semantics.

If $\gamma(k) = \texttt{DEC}_0\ k'\ \texttt{ELSE}\ k''$, then $\Phi_k = [\ominus]X_{k'} \wedge [\textcircled{T}]\langle\textcircled{0}\rangle X_{k''}$ and a case analysis is needed. If the first counter has a non-zero value in configuration c_i, then transition $\textcircled{T}$ is disabled in M_i (Lemma 8.2.1), so the second part of the conjunction is satisfied. For the first part, the transition $\ominus$ is enabled and reaches a marking $M_{i+1} \in V_{k'}$ where the first counter's value was decremented by one (Lemma 8.2.1). Since $\mathsf{val}(X_{k'}) = V_{k'}$, we have $M_i \in [\![\Phi_k]\!]_A^{\mathsf{val}}$. If the first counter has a value of zero in configuration c_i, then transition $\ominus$ is disabled in M_i, so the first part of the conjunction is satisfied. The word $\textcircled{T}\textcircled{0}$ is enabled and reaches the marking $M_{i+1} = M_i$ again (Lemma 8.2.1). Since $\mathsf{val}(X_{k''}) = V_{k''}$ we have $M_i \in [\![\Phi_k]\!]_A^{\mathsf{val}}$.

The remaining part for the second counter can be shown analogously. □

[3] A pre-fixed point under X of β, val, and A is a set V so that $[\![\beta]\!]_A^{\mathsf{val}[V \leftarrow X]} \supseteq V$.

Lemma 8.2.6 ([Feu05b]). *Given a two-counter machine $\mathcal{C}$ and a pair of numbers $(b_0, b_1) \in \mathbb{N} \times \mathbb{N}$ so that $N_{sim}(b_0, b_1) \models \Phi_{\mathcal{C}}$, the execution of $\mathcal{C}$ is bounded by (b_0, b_1).*

Proof. Let $A = \mathrm{RG}(N_{\mathrm{sim}}(b_0, b_1))$. By assumption we have $M_0 \in [\![\Phi_{\mathcal{C}}]\!]_A$. Since $\Phi_{\mathcal{C}}$ is defined to be the solution of a vectorial greatest fixed point, there is a valuation val so that $\mathsf{val}(X_k) = [\![\Phi_k]\!]^{\mathsf{val}}_A$ for $k \in \{1, \ldots, \ell\}$ with $M_0 \in \mathsf{val}(X_1)$.

We want to show that $c_i = (k_i, (v_{0,i}, v_{1,i}))$, the i-th configuration in the execution of $\mathcal{C}$, is bounded by (b_0, b_1). We do this by inductively showing that c_i corresponds to a marking $M_i \in \mathsf{val}(X_{k_i})$ where the marking M_i represents the counter values $(v_{0,i}, v_{1,i})$. Since the counters modelled by $N_{\mathrm{sim}}(b_0, b_1)$ are bounded by (b_0, b_1), this means that all counter values are bounded, too.

For the induction basis, the configuration $c_0 = (1, (0, 0))$ is related to the initial marking M_0, which does indicate counter values $(0, 0)$ and is by assumption in $\mathsf{val}(X_1)$.

Next, assume that $c_i = (k_i, (v_{0,1}, v_{1,i}))$ corresponds suitably to a marking $M_i \in \mathsf{val}(X_{k_i})$ and show the same for $i + 1$. By assumption we have $M_i \in \mathsf{val}(X_k) = [\![\Phi_k]\!]^{\mathsf{val}}_A$, where Φ_k is defined based on the instruction $\gamma(k)$. If $\gamma(k) = \texttt{HALT}$, then there is no following configuration c_{i+1} and nothing remains to be shown. If $\gamma(k) = \texttt{INC}_0\ k'$, then $\Phi_k = \langle\oplus\rangle X_{k'}$, so there is a marking $M_{i+1} \in \mathsf{val}(X_{k'})$ with $M_i[\oplus\rangle M_{i+1}$. By the structure of $N_{\mathrm{sim}}(b_0, b_1)$ (see Lemma 8.2.1), the value of the first counter was incremented in M_{i+1} compared to M_i, i.e. M_{i+1} corresponds to the counter values in configuration c_{i+1}. To summarise, $M_{i+1} \in \mathsf{val}(X_{k'})$ with M_{i+1} modelling the counter values for c_{i+1} and k' being the state in c_{i+1}, which was to be shown.

If $\gamma(k) = \texttt{DEC}_0\ k'\ \texttt{ELSE}\ k''$, then $\Phi_k = [\ominus]X_{k'} \wedge [\textcircled{T}]\langle\textcircled{0}\rangle X_{k''}$. If the value of the first counter in M_i is non-zero, then $\ominus$ is enabled while $\textcircled{T}$ is disabled (see Lemma 8.2.1), so there is a marking $M_{i+1} \in \mathsf{val}(X_{k'})$ with $M_i[\ominus\rangle M_{i+1}$. Similarly, if the first counter is zero in M_0, then $\textcircled{T}$ is enabled while $\ominus$ is disabled, so there is a marking $M_{i+1} \in \mathsf{val}(X_{k''})$ with $M_i[\textcircled{T}\textcircled{0}\rangle M_{i+1}$. The rest of both cases is similar to the case for the increment instruction. Also, the same result for the second counter can be shown analogously. □

The last two lemmas showed that $\mathcal{C}$ is bounded by (b_0, b_1) if and only if $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$. Because the bounded execution problem is undecidable (Theorem 8.1.4), a corollary follows:

Corollary 8.2.7. *Given $\mathcal{C}$, it is undecidable if $\exists b_0, b_1 \in \mathbb{N}$ such that $N_{sim}(b_0, b_1) \models \Phi_{\mathcal{C}}$.*

8.3. Encoding the Family of Petri Nets $N_{\mathsf{sim}}(b_0, b_1)$

In this section we will characterise the reachability graphs of Petri nets $N_{\mathrm{sim}}(b_0, b_1)$ with a formula $\Phi_{2\mathrm{ctr}}$ of the conjunctive ν-calculus. To be precise, our goal is to construct a formula $\Phi_{2\mathrm{ctr}}$ so that any bounded Petri net N with $N \models \Phi_{2\mathrm{ctr}}$ has isomorphic behaviour

to some Petri net from the family $N_{\text{sim}}(b_0, b_1)$, i.e. when $N \models \Phi_{2\text{ctr}}$, then there should be $b_0, b_1 \in \mathbb{N}$ so that $\text{RG}(N) = \text{RG}(N_{\text{sim}}(b_0, b_1))$.

This will allow to show the bounded realisation problem of the conjunctive ν-calculus to be undecidable: By Corollary 8.2.7, the existence of numbers (b_0, b_1) satisfying $N_{\text{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$ is undecidable. Since the formula $\Phi_{2\text{ctr}}$ encodes $N_{\text{sim}}(b_0, b_1)$ without mentioning concrete bounds, but only their existence, it will be undecidable if a bounded, or a bounded and pure, Petri net satisfying $\Phi_{2\text{ctr}} \wedge \Phi_{\mathcal{C}}$ exists.

8.3.1. Auxiliary Formulas

We begin by introducing some auxiliary formulas that will be useful in later sections for encoding counters. The first formula signifies that some word $a_1 a_2 \dots a_n \in \Sigma^*$ can be fired infinitely often:

$$\text{NoEffect}(a_1 a_2 \dots a_n) = \nu X_1.(\langle a_1 \rangle \langle a_2 \rangle \dots \langle a_n \rangle X_1)$$

Lemma 8.3.1. *$N \models \text{NoEffect}(w)$ for a bounded Petri net N if and only if both $M_0[w\rangle$ and $C(N) \cdot \Psi(w) = 0$ hold.*

Proof. Let $\text{RG}(N) = (\mathfrak{E}(N), T, \to, M_0)$ be the reachability graph of N. By definition $[\![\text{NoEffect}(w)]\!]_{\text{RG}(N)} = \bigcup\{V \subseteq \mathfrak{E}(N) \mid \{M \in \mathfrak{E}(N) \mid \exists M' \in V, M[w\rangle M'\} \supseteq V\}$. This means it is the largest subset W of $\mathfrak{E}(N)$ that satisfies $M \in W \Rightarrow \exists M' \in W : M[w\rangle M'$, i.e. if M is in W, then a marking M' is reachable from M via w that is also in W.

Thus, the premise $M_0 \in [\![\text{NoEffect}(w)]\!]_{\text{RG}(N)}$ means that from M_0 a marking M_1 is reachable via w, from which in turn a marking M_2 is reachable via w, etc. Since N is a bounded Petri net, there are only finitely many reachable markings in N. Thus, infinitely many of the M_i must be equal. However, if some of them are equal, then by the marking equation (Lemma 2.0.12), if $M + C \cdot \Psi(w^i) = M$, then $0 = C \cdot \Psi(w^i) = C \cdot i \cdot \Psi(w) = C \cdot \Psi(w)$, meaning that firing w does not change the current marking, and all M_i are equal.

For the opposite direction, if $M_0[w\rangle$ and $C \cdot \Psi(w) = 0$, then $M_0[w\rangle M_0$ (Lemma 2.0.12) and all words w^i with $i \in \mathbb{N}$ are enabled in N. By the reasoning above this means that $M_0 \in [\![\text{NoEffect}(w)]\!]_{\text{RG}(N)}$, i.e. $N \models \text{NoEffect}(w)$. □

Note that the μ-calculus can only differentiate lts up to bisimulation [Pop94; Arn94], while the above actually expresses a stronger property. The key insight is that a sequence w^ω is only possible in a bounded Petri net if the sequence w returns to the initial marking. This lemma is a crucial ingredient for the characterisation of $N_{\text{sim}}(b_0, b_1)$ later.

Next, we define a formula $\text{Global}(\beta)$ that requires a formula β to hold in all reachable states. Here, X_1 is a fresh variable which does not appear in β.

$$\text{Global}(\beta) = \nu X_1(\beta \wedge \bigwedge_{a \in \Sigma} [a] X_1)$$

Lemma 8.3.2. *For a reachable lts $A = (S, \Sigma, \to, \imath)$, we have $\imath \in [\![\mathrm{Global}(\beta)]\!]^{\mathsf{val}}_A$ if and only if $S = [\![\beta]\!]^{\mathsf{val}}_A$.*

Proof. By the semantics $[\![\beta \wedge \bigwedge_{a\in\Sigma}[a]X_1]\!]^{\mathsf{val}}_A = \{s \in [\![\beta]\!]^{\mathsf{val}}_A \mid \forall a \in \Sigma, s' \in S\colon s \xrightarrow{a} s' \Rightarrow s' \in \mathsf{val}(X_1)\}$, meaning that this formula holds in a state s if β holds and each directly reachable state satisfies X_1. The fixed point produces $[\![\mathrm{Global}(\beta)]\!]^{\mathsf{val}}_A = \bigcup\{V \subseteq S \mid \{s \in [\![\beta]\!]^{\mathsf{val}}_A \mid \forall a \in \Sigma, s' \in S\colon s \xrightarrow{a} s' \Rightarrow s' \in V\} \supseteq V\}$. This set contains a state s if s satisfies β and each recursive successor of s does, too. This is equivalent to $[\![\mathrm{Global}(\beta)]\!]^{\mathsf{val}}_A = \{s \in [\![\beta]\!]^{\mathsf{val}}_A \mid \forall w \in \Sigma^*, s' \in S\colon s \xrightarrow{w} s' \Rightarrow s' \in [\![\beta]\!]^{\mathsf{val}}_A\}$: The latter set is a superset of the earlier set by induction on w, while the subset relation can be shown directly. So for a state s to be in this set, s and all states reachable from it have to satisfy β.

Thus, if $\imath \in [\![\mathrm{Global}(\beta)]\!]^{\mathsf{val}}_A$, then all states of A satisfy β, because A is reachable by assumption. When all states satisfy β, then they also satisfy $\mathrm{Global}(\beta)$. To summarise, $\imath \in [\![\mathrm{Global}(\beta)]\!]^{\mathsf{val}}_A$ if and only if $S = [\![\beta]\!]^{\mathsf{val}}_A$. □

The last auxiliary formula that we define expresses that some events are independent. Namely, given two disjoint sets $A, B \subseteq \Sigma$, we express that along any path in the lts, the order of events from these two sets can be swapped. To achieve this, the formula needs for each event $a \in A \cup B$ an inverse event $\delta(a)$ that, in a sense, undoes the event a. This means that in a Petri net, transitions a and $\delta(a)$ must have opposite effects, i.e. $C \cdot \Psi(a) = -C \cdot \Psi(\delta(a))$, so that the sequence $a\delta(a)$ does not change the current marking.

$$\mathrm{Indep}_h(A, B, \delta) = \mathrm{Global}(\bigwedge_{a\in A}\bigwedge_{b\in B}[a][b]\langle\delta(b)\rangle\langle\delta(a)\rangle\langle b\rangle\langle a\rangle\mathtt{true})$$
$$\mathrm{Indep}(A, B, \delta) = \mathrm{Indep}_h(A, B, \delta) \wedge \mathrm{Indep}_h(B, A, \delta)$$

What the formula $\mathrm{Indep}_h(A, B, \delta)$ actually expresses is that in a state, where the word ab is enabled ($[a][b]$), also $ab\delta(b)\delta(a)ba$ has to be possible. By the assumption about effects, $\delta(b)$ undoes the previous b and $\delta(a)$ undoes a, so that in the state where ab is possible, also ba has to be enabled. $\mathrm{Indep}(A, B, \delta)$ then uses this formula to express that swapping in both directions is possible

Lemma 8.3.3. *Let $N = (P, T, F, M_0)$ be a Petri net and A and B be two disjoint subsets of T. Then $N \models \mathrm{Indep}_h(A, B, \delta)$ if and only if for all reachable markings $M \in \mathfrak{E}(N)$ and all $a \in A$ and $b \in B$ it holds that $M[ab\rangle$ implies $M[ab\delta(b)\delta(a)ba\rangle$.*

Proof. By the semantics of the μ-calculus and Lemma 8.3.2. □

Lemma 8.3.4. *Let $N = (P, T, F, M_0)$ be a Petri net, A and B be two disjoint subsets of T, and $\delta\colon A \cup B \to T$ be a function so that the effect of a is the opposite of $\delta(a)$: for all $a \in A \cup B\colon C \cdot \Psi(a) = -C \cdot \Psi(\delta(a))$. Then $N \models \mathrm{Indep}(A, B, \delta)$ implies that for all reachable markings $M \in \mathfrak{E}(N)$ and all $a \in A$ and $b \in B$ it holds that $M[ab\rangle$ if and only if $M[ba\rangle$.*

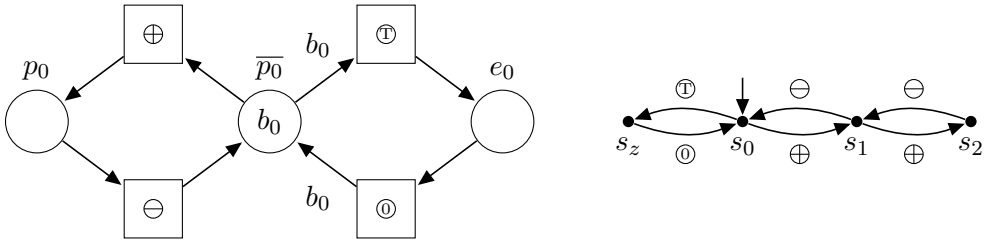


Figure 8.3.: A single counter $N^0_{\text{sim}}(b_0)$ from the Petri net $N_{\text{sim}}(b_0, b_1)$ from Figure 8.1 and the reachability graph of the counter with bound $b_0 = 2$.

Proof. Since b and $\delta(b)$ are assumed to have opposite effects, for all markings M, if $M[b\delta(b)\rangle$, then $M[b\delta(b)\rangle M$. By the same argument $M[ab\delta(b)\delta(a)\rangle M$, i.e. the effect $C \cdot \Psi(ab\delta(b)\delta(a))$ of this sequence is zero. Thus, the previous lemma can be restated as: For all reachable markings M, if $M[ab\rangle$, then $M[ba\rangle$. Since $\text{Indep}(A, B, \delta)$ is defined to make this requirement also with the meaning of A and B swapped, both directions are shown. □

8.3.2. Encoding a Single Counter

To begin with a simpler problem than two bounded counters, this section will introduce a formula that characterises a single bounded counter $N^0_{\text{sim}}(b_0)$. Such a counter and a possible reachability graph is shown in Figure 8.3. Our characterising formula Φ_{1ctr} will be independent of the bound of the counter and will consist of three parts: $\Phi_{\text{1ctr}} = \Phi_{\text{init}} \wedge \Phi_{\text{positive}} \wedge \Phi_{\text{zero}}$.

The formula Φ_{init} characterises the initial state s_0 of the counter:

$$\Phi_{\text{init}} = \text{NoEffect}(\oplus\ominus) \wedge \text{NoEffect}(ⓉⓄ) \wedge \not\to^{\ominus} \wedge \not\to^{Ⓞ}$$

By Lemma 8.3.1, the first two subformulas express that $M_0[\oplus\ominus\rangle M_0$ and $M_0[ⓉⓄ\rangle M_0$, i.e. initially the counter can both be incremented and tested for zero, and the two transitions $\oplus$ and $\ominus$ (Ⓣ and Ⓞ, resp.) have opposite effects. Also, the events $\ominus$ and Ⓞ are disabled in the initial state.

The next formula, Φ_{positive}, characterises the state of the counter when its value is positive:

$$\Phi_{\text{positive}} = \nu X.[\oplus](X \wedge \to^{\ominus} \wedge \not\to^{Ⓣ} \wedge \not\to^{Ⓞ})$$

A state with a non-zero counter is reached after transition $\oplus$ fired arbitrary often. Any such state must enable the decrement transition $\ominus$ and may not allow either of the zero test transitions. The state of $\oplus$ is unknown: It could be enabled or not. This is what makes the formula independent of the actual bound of the counter.

The last formula is Φ_{zero}, which characterises the state s_z reached by Ⓣ:

$$\Phi_{\text{zero}} = [Ⓣ](\to^{Ⓞ} \wedge \not\to^{Ⓣ} \wedge \not\to^{\oplus} \wedge \not\to^{\ominus})$$

In this state, ⓪ must be enabled[4] and all other transitions of this counter must be disabled.

To show that the formula $\Phi_{1\text{ctr}}$ characterises the reachability graph of $N^0_{\text{sim}}(b_0)$, we define the bound of the counter for an arbitrary Petri net N:

Definition 8.3.5 (Counter bound)**.** *Given any Petri net N, define*[5] $b_0(N) = \sup\{n \in \mathbb{N} \mid M_0[\oplus^n\rangle\}$ $(\subseteq \mathbb{N} \cup \{\infty\})$.

Theorem 8.3.6 ($\Phi_{1\text{ctr}}$ characterises $N^0_{\text{sim}}(b_0)$)**.** *For any bounded Petri net N with transitions $T = \{\oplus, \ominus, \text{Ⓣ}, \text{⓪}\}$, if $N \models \Phi_{1ctr}$ then $b_0(N) \neq \infty$ and $\text{RG}(N) = \text{RG}(N^0_{sim}(b_0(N)))$.*

Proof. Let M be an arbitrary reachable marking in N via $w \in \Sigma^*$, i.e. $M_0[w\rangle M$. By $N \models \Phi_{\text{init}}$ and Lemma 8.3.1, we have $C \cdot \Psi(\oplus) = -C \cdot \Psi(\ominus)$ and $C \cdot \Psi(\text{Ⓣ}) = -C \cdot \Psi(\text{⓪})$. Thus firing these transitions cancels any change to the current marking and any subwords of the form $\oplus\ominus$, $\ominus\oplus$, ⓉⓄ, and ⓄⓉ can be removed iteratively from w without affecting the reached marking. By Φ_{init}, w can only begin with $\oplus$ or Ⓣ. If w begins with Ⓣ, then by Φ_{zero}, the only possibility for the next event is ⓪. However, subwords of the form ⓉⓄ were removed, so in this case $w = \text{Ⓣ}$. If w begins with $\oplus$, then by Φ_{positive}, only $\oplus$ or $\ominus$ are possible next events. However, since subwords of the form $\oplus\ominus$ were removed, in this case only $w = \oplus^i$ for a suitable $i \in \mathbb{N}$ is possible.

To summarise, all reachable markings in N are reached by words from $\{\text{Ⓣ}\} \cup \{\oplus^i \mid i \in \mathbb{N}\}$.

Next, observe that $M_0[\text{Ⓣ}\rangle$, but not $M_0[\oplus\text{Ⓣ}\rangle$. Thus, $\oplus$ must consume a token that is needed to activate Ⓣ. Since $\oplus$ has a negative effect on some place, the number of tokens on this place provides an upper bound on how often $\oplus$ can fire consecutively. We conclude $b_0(N) \neq \infty$.

We can now compare $\text{RG}(N)$ and $\text{RG}(N^0_{\text{sim}}(b_0(N)))$. By the reasoning above, the set of reachable states of these Petri nets can be identified. To conclude that the reachability graphs are isomorphic, it only remains to be shown that this mapping between states is an isomorphism, i.e. preserves edges.

For the initial state, the same set of transitions is enabled and they lead to the expected state by definition of the mapping. For the state reached by Ⓣ, the only enabled event is ⓪ and this leads back to the initial state by the structure of $N^0_{\text{sim}}(b_0(N))$, since NoEffect(ⓉⓄ) requires ⓉⓄ not to change the current marking (Lemma 8.3.1). For any state reached by $\oplus$, only transitions $\oplus$ and $\ominus$ could be enabled. If $\oplus$ is enabled, it reaches the expected state by the definition of the mapping; if $\ominus$ is enabled, it reaches the expected state by the structure of $N^0_{\text{sim}}(b_0(N))$, or by Φ_{positive} and NoEffect($\oplus\ominus$), respectively. □

We will later need that $N_{\text{sim}}(b_0, b_1)$ implements $\Phi_{1\text{ctr}}$:

[4] This part of the formula could be dropped since the same requirement is already expressed by NoEffect(ⓉⓄ) above.

[5] If $\oplus$ is not a transition of N, then $b_0(N) = 0$.

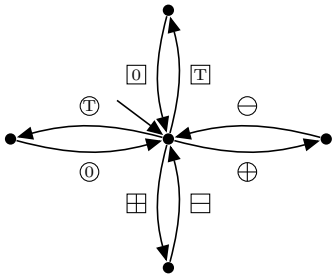


Figure 8.4.: An lts that does not model two independent counters.

Lemma 8.3.7. *For $b_0, b_1 \in \mathbb{N}_+$, $N_{sim}(b_0, b_1) \models \Phi_{1ctr}$.*

Proof. This follows as a corollary of the argumentation in the above proof: Only the transitions allowed by $\Phi_{1\mathrm{ctr}}$ are enabled in the Petri net and they reach the expected states again. This needs $b_0 > 0$, because otherwise $\oplus$ is not initially enabled and NoEffect($\oplus\ominus$) is not fulfilled. □

8.3.3. Encoding Two Counters: The Naïve Approach

The previous section introduced a formula $\Phi_{1\mathrm{ctr}}$ that encodes a single counter $N^0_{\mathrm{sim}}(b_0)$. An obvious way to encode two counters would be to create a copy $\Phi'_{1\mathrm{ctr}}$ of $\Phi_{1\mathrm{ctr}}$, where all symbols for the first counter are replaced with symbols for the second counter. Now the expectation might be that $\Phi = \Phi_{1\mathrm{ctr}} \wedge \Phi'_{1\mathrm{ctr}}$ encodes the two-counters Petri net $N_{\mathrm{sim}}(b_0, b_1)$, but this is not correct.

For example, the lts from Figure 8.4 satisfies this formula Φ, but does not model two independent counters, since when one counter is incremented, the other can no longer be incremented. Similarly, the zero tests disable each other. This is not a desired behaviour.

Instead, we want the two counters to be independent.

8.3.4. Products of Transition Systems

The two counters that are modelled should be independent in the following way:

Given two lts A_1 and A_2 with disjoint alphabets, we can define a product lts $A_1 \otimes A_2$. Each state of the product consists of a pair of states of A_1 and A_2. A label of A_1 is possible if the underlying state of A_1 allows it and the new state is the target state in A_1 while the component for lts A_2 is not modified. Labels of A_2 can occur in a similar way.

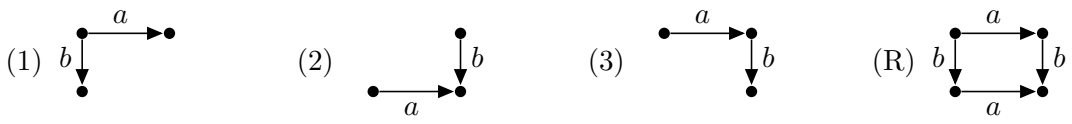


Figure 8.5.: A visualisation of the general diamond property. Each of the situations (1), (2), and (3) must be completable to (R).

Definition 8.3.8 (Disjoint product of lts [Dev18])**.** *Given two lts* $A_1 = (S_1, \Sigma_1, \to_1, \imath_1)$ *and* $A_2 = (S_2, \Sigma_2, \to_2, \imath_2)$ *with* $\Sigma_1 \cap \Sigma_2 = \emptyset$*, their* product $A_1 \otimes A_2$ *is the lts* $A_1 \otimes A_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \to, (\imath_1, \imath_2))$ *with* $\to = \{((s_1, s_2), t_1, (s_1', s_2)) \mid (s_1, t_1, s_1') \in \to_1, s_2 \in S_2\} \cup \{((s_1, s_2), t_2, (s_1, s_2')) \mid (s_2, t_2, s_2') \in \to_2, s_1 \in S_1\}$.

The corresponding operation for Petri nets is called the *disjoint sum* of Petri nets. Informally speaking, this operation puts the two Petri nets next to each other to produce their Petri net sum. The reachability graph of this sum is the disjoint product of the reachability graphs of the individual Petri nets.

We do not need products, but the opposite operation, called *factorisation*: Given an lts A and a subset $\Sigma_1 \subseteq \Sigma$ of its alphabet, under what condition is A the product of two lts with alphabets Σ_1 and $\Sigma \setminus \Sigma_1$? More specifically, we have seen that our formula $\Phi_{1\text{ctr}}$ for the characterisation of a single counter cannot easily be extended to two counters. However, if a condition for factorisability could be expressed as a formula, this would allow us to express that an lts is the product of two independent counters.

Since our interest is not just in lts, but in lts that are the reachability graphs of Petri nets, we can use a result from [Dev18; DS18]. This result is based on the *general diamond property*, which is illustrated in Figure 8.5. If two events from different factors are enabled in a state (situation (1)), then they must not interfere with each other in the sense that after one of them is executed, the other one is still possible. Also, executing both of them in sequence must reach the same state, independent of the order in which they occurred (situation (R)).

The same requirement is also made when events reach the same state (situation (2)): If both a and b, which are from different factors, can be used to reach a state s, then the state from which a originates also has an incoming edge with label b, and vice versa. Furthermore, following both events backwards reaches the same state independent from the order.

Finally, this requirement is also made when one event is followed in forward direction and the other in backward direction (situation (3)).

Definition 8.3.9 (General diamond property)**.** *Given an lts* $A = (S, \Sigma, \to, \imath)$ *and two distinct labels* $a, b \in \Sigma$*, A has the* general diamond property for a and b*, if for all states* $s, s_1, s_2 \in S$*,*

- *if* $s \xrightarrow{a} s_1$ *and* $s \xrightarrow{b} s_2$*, then there is a state* $s' \in S$ *so that* $s_1 \xrightarrow{b} s'$ *and* $s_2 \xrightarrow{a} s'$*,*

- *if* $s \xrightarrow{a} s_1$ *and* $s_2 \xrightarrow{b} s$*, then there is a state* $s' \in S$ *so that* $s' \xrightarrow{b} s_1$ *and* $s_2 \xrightarrow{a} s'$*,*
- *if* $s_1 \xrightarrow{a} s$ *and* $s \xrightarrow{b} s_2$*, then there is a state* $s' \in S$ *so that* $s_1 \xrightarrow{b} s'$ *and* $s' \xrightarrow{a} s_2$*,*
- *if* $s_1 \xrightarrow{a} s$ *and* $s_2 \xrightarrow{b} s$*, then there is a state* $s' \in S$ *so that* $s' \xrightarrow{b} s_1$ *and* $s' \xrightarrow{a} s_2$*.*

Given a subset $\Sigma_1 \subseteq \Sigma$ *of the alphabet, A is called* Σ_1-gdiam *if it presents the general diamond property for each pair* $a \in \Sigma_1$ *and* $b \in \Sigma \setminus \Sigma_1$*.*

This definition can now be used to characterise when factorisation is possible:

Theorem 8.3.10. *Let* $N = (P, T, F, M_0)$ *be a Petri net and* $T_1 \subseteq T$ *be a subset of the transitions so that* $\mathrm{RG}(N)$ *is* T_1*-gdiam. Then there are Petri nets* $N_1 = (P_1, T_1, F_1, M_{0,1})$ *and* $N_2 = (P_2, T \setminus T_1, F_2, M_{0,2})$ *so that* $\mathrm{RG}(N) = \mathrm{RG}(N_1) \otimes \mathrm{RG}(N_2)$*.*

Proof sketch. This follows from the main result of [DS18][6]: By Lemma 2.0.8, $\mathrm{RG}(N)$ is deterministic and reachable. With the same proof it can be shown that $\mathrm{RG}(N)$ is also *backwards deterministic*, which means that if $M'[t\rangle M$ and $M''[t\rangle M$, then $M' = M''$. We now have the preconditions for Theorem 2 of [DS18], which states that $\mathrm{RG}(N)$ is the product of the parts of it reachable via T_1 and the parts reachable via T_2, and that each of these parts can be solved by a Petri net, i.e. $\mathrm{RG}(N) = \mathrm{RG}(N_1) \otimes \mathrm{RG}(N_2)$. □

8.3.5. Encoding the General Diamond Property

Our first attempt for encoding two counters by just duplicating the formula for a single counter did not work, because there was a possibility of unwanted dependencies between the counters. These dependencies will be forbidden via a formula, which ensures that the resulting Petri net is $\{\oplus, \ominus, ⓣ, ⓪\}$-gdiam. By the results above, this ensures that any bounded Petri net satisfying this formula is the disjoint product of two counters.

In Section 8.3.1 an auxiliary formula was already introduced that will be used now: The formula $\mathrm{Indep}(\Sigma_0, \Sigma_1, \delta)$ expresses that events from Σ_0 and Σ_1 are independent in the sense that they can be swapped with each other on paths. However, in addition to this formula, we will need something more to express the general diamond property.

Define $\Sigma_0 = \{\oplus, \ominus, ⓣ, ⓪\}$ and $\Sigma_1 = \{⊞, ⊟, \boxed{\mathrm{T}}, \boxed{0}\}$ to be a partition of our alphabet into the events of the individual counters. Also, define $\delta \colon \Sigma \to \Sigma$ as the function that maps each event to its inverse, i.e. $\delta(\oplus) = \ominus$, $\delta(\ominus) = \oplus$, $\delta(ⓣ) = ⓪$, $\delta(⓪) = ⓣ$, $\delta(⊞) = ⊟$, $\delta(⊟) = ⊞$, $\delta(\boxed{\mathrm{T}}) = \boxed{0}$, and $\delta(\boxed{0}) = \boxed{\mathrm{T}}$. The formula for characterising the general diamond property is:

$$\Phi_{\mathrm{gdiam}} = \mathrm{Indep}(\Sigma_0, \Sigma_1, \delta) \wedge \mathrm{Global}(\bigwedge_{a \in \Sigma} [a] \to^{\delta(a)})$$

In addition to the already mentioned independence requirement, this formula also requires that in a state reached by some event a, the inverse event $\delta(a)$ is enabled.

[6] Providing the full proof would require several new notions. Thus, only the following theorem is cited.

Theorem 8.3.11. *Let N be a bounded Petri net with $N \models \Phi_{gdiam} \wedge \Phi$ where*[7] $\Phi = \text{NoEffect}(\oplus\ominus) \wedge \text{NoEffect}(\textcircled{T}\textcircled{0}) \wedge \text{NoEffect}(\boxplus\boxminus) \wedge \text{NoEffect}(\boxed{T}\boxed{0})$. *Then* $\text{RG}(N)$ *is* Σ_1*-gdiam.*

Proof. By Lemma 8.3.1, $\text{NoEffect}(w)$ implies that firing the word w does not change the current marking of the Petri net. Thus, Φ expresses that various transitions have opposite effects, namely $C \cdot \Psi(\oplus) = -C \cdot \Psi(\ominus)$, $C \cdot \Psi(\textcircled{T}) = -C \cdot \Psi(\textcircled{0})$, $C \cdot \Psi(\boxplus) = -C \cdot \Psi(\boxminus)$, and $C \cdot \Psi(\boxed{T}) = -C \cdot \Psi(\boxed{0})$. This means that the preconditions for Lemma 8.3.4 are satisfied and this lemma states that in all reachable markings M, $M[ab\rangle$ if and only if $M[ba\rangle$, for all $a \in \Sigma_0$ and $b \in \Sigma_1$.

By the Petri net marking equation from Lemma 2.0.12, this already guarantees the second and third condition in the definition of the general diamond property, because $M[a\rangle M'[b\rangle$ is given and we conclude that $M[ba\rangle$.

Next, let M be a marking with $M[a\rangle M'$ and $M[b\rangle M''$. We want to show the first condition in the definition of the general diamond property holds, which is that $M'[b\rangle\overline{M}$ and $M''[a\rangle\overline{M}$ for a suitable marking $\overline{M}$. By $\text{Global}(\bigwedge_{a\in\Sigma}[a]\to^{\delta(a)})$, Lemma 8.3.2 and the semantics of the μ-calculus, from $M[a\rangle M'$, we conclude $M'[\delta(a)\rangle M$. Since for all $a \in \Sigma_1$, also $\delta(a) \in \Sigma_1$, we are now in a similar position than before: Marking M' enables the word $\delta(a)b$, which can be swapped into $M'[b\delta(a)\rangle$ via Lemma 8.3.4. Let $\overline{M}$ be the marking reached by b: $M'[b\rangle\overline{M}$, which was to be shown. With the same steps on $M[b\rangle M''$ we also get that $M''[a\rangle\overline{M}'$, and $\overline{M} = \overline{M}'$ because $\Psi(ab) = \Psi(ba)$ and the Petri net marking equation (Lemma 2.0.12).

For the last condition in the definition of the general diamond property, we assume markings so that $M'[a\rangle M$, $M''[b\rangle M$, and have to show that there is a marking $\overline{M}$ with $\overline{M}[b\rangle M'$ and $\overline{M}[a\rangle M''$. This works similarly as before: We get $M'[a\delta(b)\rangle M''$ from $\text{Global}(\bigwedge_{a\in\Sigma}[a]\to^{\delta(a)})$. Lemma 8.3.4 allows us to produce $M'[\delta(b)\rangle\overline{M}[a\rangle M''$, and we can reverse the $\delta(b)$ again to arrive at $\overline{M}[b\rangle M'$ and $\overline{M}[a\rangle M''$. □

Let $\Phi_{N_{\text{sim}}} = \Phi_{1\text{ctr}} \wedge \Phi'_{1\text{ctr}} \wedge \Phi_{\text{gdiam}}$, where $\Phi'_{1\text{ctr}}$ is equivalent to $\Phi_{1\text{ctr}}$, but with the events of the second counter instead of the first, and $b_1(N)$ is defined analogously to $b_0(N)$, but for the second counter. We can now show that this formula characterises N_{sim}:

Theorem 8.3.12. *For any bounded Petri net N, if $N \models \Phi_{N_{sim}}$, then $b_0(N) \neq \infty \neq b_1(N)$ and* $\text{RG}(N) = \text{RG}(N_{sim}(b_0(N), b_1(N)))$.

Proof. For a bounded Petri net N with $N \models \Phi_{N_{\text{sim}}}$, by the previous theorem we know that N is Σ_1-gdiam. By Theorem 8.3.10, there are two Petri nets N_0 and N_1 so that $\text{RG}(N) = \text{RG}(N_0) \otimes \text{RG}(N_1)$ and the alphabet of N_0 is Σ_0, while the alphabet of N_1 is Σ_1. Since the combined lts satisfies the formula $\Phi_{1\text{ctr}} \wedge \Phi'_{1\text{ctr}}$, N_0 must satisfy $\Phi_{1\text{ctr}}$ and N_1 must satisfy $\Phi'_{1\text{ctr}}$, because each of those formulas only contains events that

[7] This formula is part of Φ_{init}, which in turn is part of $\Phi_{1\text{ctr}}$.

appear in the respective Petri net. By Theorem 8.3.6, we conclude that $\mathrm{RG}(N_0) = \mathrm{RG}(N^0_{\mathrm{sim}}(b_0(N)))$ and $\mathrm{RG}(N_1) = \mathrm{RG}(N^1_{\mathrm{sim}}(b_1(N)))$, where $b_0(N) \neq \infty \neq b_1(N)$, thus $\mathrm{RG}(N) = \mathrm{RG}(N^0_{\mathrm{sim}}(b_0(N))) \otimes \mathrm{RG}(N^1_{\mathrm{sim}}(b_1(N)))$. The disjoint product of two one-counter Petri nets is the two-counter Petri net $N_{\mathrm{sim}}(b_0(N), b_1(N))$. □

Using Corollary 8.2.7, which states that, given a two-counter machine $\mathcal{C}$, it is undecidable if $b_0, b_1 \in \mathbb{N}$ exist so that $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$, we arrive at the following result:

Theorem 8.3.13. *Given a two-counter machine $\mathcal{C}$, it is undecidable if a bounded Petri net N exists such that $N \models \Phi_{\mathcal{C}} \wedge \Phi_{N_{sim}}$.*

Proof. By Corollary 8.2.7 it is undecidable if $b_0, b_1 \in \mathbb{N}$ exist so that $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$. However, this is the case if and only if a bounded Petri net N satisfying $N \models \Phi_{\mathcal{C}} \wedge \Phi_{N_{\mathrm{sim}}}$ exists, as follows.

Assume $b_0, b_1 \in \mathbb{N}$ exists so that $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$. To arrive at $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}} \wedge \Phi_{N_{\mathrm{sim}}}$, we only have to show that $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{N_{\mathrm{sim}}}$ We have $\Phi_{N_{\mathrm{sim}}} = \Phi_{1\mathrm{ctr}} \wedge \Phi'_{1\mathrm{ctr}} \wedge \Phi_{\mathrm{gdiam}}$. By Lemma 8.3.7 the first two subformulas hold[8] and by Lemma 8.3.3 and the structure of $N_{\mathrm{sim}}(b_0, b_1)$ also Φ_{gdiam} holds.

Conversely, assume that a bounded Petri net N satisfying $N \models \Phi_{\mathcal{C}} \wedge \Phi_{N_{\mathrm{sim}}}$ exists. By Theorem 8.3.12, there are numbers $b_0 = b_0(N)$ and $b_1 = b_1(N)$ so that $\mathrm{RG}(N) = \mathrm{RG}(N_{\mathrm{sim}}(b_0, b_1))$, so $N_{\mathrm{sim}}(b_0, b_1) \models \Phi_{\mathcal{C}}$ follows, which was to be shown. □

This also shows that bounded Petri net synthesis from the conjunctive ν-calculus is undecidable, since we have found a family of formulas for which this problem is undecidable:

Corollary 8.3.14. *The bounded realisation problem (Problem 8.0.1) is undecidable.*

Since the nets $N_{\mathrm{sim}}(b_0, b_1)$ are all pure and the characterisation in Theorem 8.3.12 is based on the shape of the reachability graph, and not on the precise shape of the Petri net, we have indeed proven that the problem stays undecidable even when asking for pure Petri nets.

Corollary 8.3.15. *The bounded and pure realisation problem (Problem 8.0.2) is undecidable.*

In fact, the undecidability result applies for any subclass of Petri nets that contains $N_{\mathrm{sim}}(b_0, b_1)$. For example, the nets N_{sim} are distributed, and so the synthesis problem for this subclass is undecidable as well. However, we cannot infer anything about the decidability for e.g. plain, place-output-nonbranching, or k-bounded Petri nets.

[8] If one of the bounds is zero, use a higher bound instead for Lemma 8.3.7.

8.4. Conclusion

This chapter showed that, given a formula of the conjunctive ν-calculus, it is undecidable if a bounded Petri net realisation exists, where a realisation is a Petri net with a reachability graph satisfying the formula. This result easily extends to bounded and pure Petri nets.

The approach was inspired by [Feu05b]. The behaviour of a two-counter machine $\mathcal{C}$ [Min67] is encoded into a formula $\Phi_{\mathcal{C}}$. A special family of Petri nets simulating bounded counters was presented, and it was shown that satisfaction of $\Phi_{\mathcal{C}}$ by a net of this family is linked to an undecidable problem of two-counter machines. Next, the family of Petri nets was encoded into another formula $\Phi_{N_{\text{sim}}}$, so that a Petri net satisfying $\Phi_{N_{\text{sim}}}$ behaves identically to a Petri net of our family of nets. Thus, satisfaction of $\Phi_{N_{\text{sim}}} \wedge \Phi_{\mathcal{C}}$ by a bounded Petri net encodes an undecidable problem.

In contrast to other undecidability results for Petri nets based on two-counter machines, e.g. [Jan01], instead of simulating the complete two-counter machine with a Petri net, the presented approach only simulates the two counters with a Petri net, while the program of the machine is encoded into a formula.

Compared to the approach of [Feu05b], which shows the same problem[9] undecidable for pure and possibly unbounded Petri nets, the approach to encode the family of nets is completely new. While [Feu05b] encodes the individual places of a Petri net in a formula, our approach encodes the behaviour of the net, i.e. the shape of its reachability graph. The main insight for this new encoding was a formula for encoding that some sequence forms a loop in the lts. This is a property normally not expressible in the μ-calculus since bisimilar lts satisfy the same formulas [Pop94; Arn94]. However, due to the interplay with the semantics of Petri nets, this allowed us to construct a formula that encodes a family of reachability graphs up to isomorphism.

Compared to [Sch16a], which is the basis for this chapter and which shows the undecidability for bounded, but possibly impure, Petri nets, the use of factorisation is new. In [Sch16a], only independence of counter bounds was needed to show the equivalent of factorisability. This means that when both counters can be incremented in the current marking of the Petri net, they can also be incremented sequentially, i.e. when the values $(3, 4)$ and $(4, 3)$ are reachable, then $(4, 4)$ must also be reachable. Since we also show undecidability for pure Petri nets, the zero test was split into two transitions. In this setting, not only increments have to be independent, but the interplay between increments and the transitions for the zero test also has to be modelled. The characterisation of factorisability in a formula simplified this a lot, and is more general.

In Section 7.3.2, it was argued that the conjunctive ν-calculus and deterministic mts are equally expressive. Thus, the presented results apply to deterministic mts, too. It would have been possible to do the constructions directly as deterministic mts instead

[9] Note that neither result implies the other.

of formulas. However, this would have been more cumbersome and less intuitive. For example, logical conjunction allowed to combine partial specifications into larger ones. Conjunction can be defined on mts, but the ν-calculus provides this operation directly. Another reason is that in the formulas, any behaviour not explicitly forbidden is allowed. This means that for a partial specification, e.g. the formula that encoded a single counter, the rest of the system does not have to be taken into account. In mts we have to allow everything explicitly instead. As a small example, the formula $\not\rightarrow^a$ simply requires that the event a is not enabled initially. As an mts, this would require two states and $2|\Sigma| - 1$ may edges (cf. Figure 7.5 on page 76). The formula $\not\rightarrow^a$ is less cluttered and therefore easier to understand than the equivalent mts.

The state-based semantics of the μ-calculus that were used here are equivalent to a similar, language-based semantics [Feu05b]. For this semantics, [Esp94; Esp97] showed that model checking for the linear time μ-calculus is decidable even for unbounded Petri nets, while the branching time μ-calculus is more powerful and has an undecidable model checking problem.

9. Decidability of k-bounded Modal Realisation

In the previous chapter it was shown that finding bounded Petri net realisations from the conjunctive ν-calculus and mts is undecidable. Since Petri net synthesis from lts is possible and mts are an extension of lts, this raises the question where exactly the border to decidability lies.

In this chapter, we restrict the problem to k-bounded Petri nets, where the number k is another input to the problem. With this restriction, the problem becomes decidable not only for the conjunctive ν-calculus and mts, but also for the full modal μ-calculus, which is more expressive. Additionally, k-boundedness can be combined with arbitrary other subclasses, e.g. k-bounded and pure Petri nets. The reason for this is that model checking for finite lts is decidable. With a fixed bound k, there are only finitely many Petri nets over the given alphabet Σ. Each of these Petri nets can be checked against the specification by checking if it is indeed k-bounded, computing its reachability graph, and model-checking against the specification. Thus, a brute force algorithm is possible.

Theorem 9.0.1. *Given a fixed alphabet Σ and a number $k \in \mathbb{N}$, there are at most $2^{(k+1)^{1+2|\Sigma|}}$ structurally different reachability graphs of k-bounded Petri nets.*

Proof. First we consider the number of possible places. Each place has an initial marking in the range $\{0, \ldots, k\}$ and for each transition $t \in \Sigma$, there are two flow weights in the same range, since otherwise the bound would be violated when the transition fires[1]. Thus, there are $(k+1)^{1+2|\Sigma|}$ possible places.

Each k-bounded Petri net has a subset of these places, where exact duplicates do not affect the structure of the reachability graph. Thus, there are $2^{(k+1)^{1+2|\Sigma|}}$ Petri nets to consider. Some of them are not k-bounded, or have reachability graphs that are isomorphic to each other, but every possible reachability graph is covered. □

Corollary 9.0.2. *The k-bounded realisation problem for the μ-calculus is decidable.*

This decidability result is not really satisfactory as an algorithm, because it is based on a brute-force approach. In this chapter, we introduce a more efficient goal-oriented algorithm that actually uses information from the specification to incrementally construct a realisation.

[1] If necessary, transitions can be prevented from firing at all with flow weights $\leq k$.

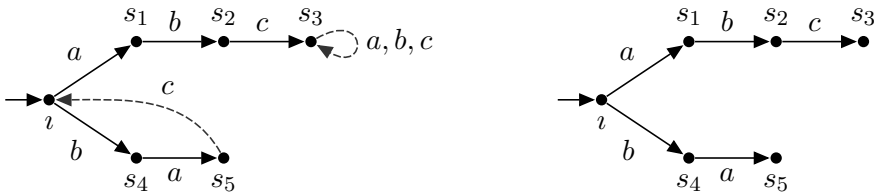


Figure 9.1.: On the left is a deterministic mts M used as an example, and on the right is a first implementation of M.

This chapter is loosely based on the author's publication [SW17], but we will develop an algorithm for formulas of the modal μ-calculus instead of the more limited disjunctive modal transition systems used in [SW17].

9.1. Introduction

To introduce the basic idea of the algorithm, this section presents a simplified example. The task will be to produce a 1-bounded realisation of the mts M depicted on the left of Figure 9.1. For this motivating example, a modal transition system, a state-based model, is used instead of a formula of the modal μ-calculus.

The current state of the algorithm is an lts, which is enlarged into an implementation of the input mts. This lts begins with just an initial state and no edges. It can easily be checked that this lts is not an implementation of M since the sequences abc and ba are required, but missing from the lts. Thus, new states and edges are added to this lts for the missing behaviour. The result is the lts shown on the right side of Figure 9.1.

However, this lts is not Petri net solvable, because the two sequences ab and ba are supposed to reach different states from the initial state. By the marking equation from Lemma 2.0.12, they must reach the same marking in a Petri net. Since our implementation needs to be the reachability graph of some Petri net, this lts has to be modified. We do that with the minimal over-approximation that was introduced earlier in Chapter 5. This will be the only place where the bound $k = 1$ is used in the algorithm. The 1-bounded minimal over-approximation of the current lts is shown on the left side of Figure 9.2.

Next, this lts is compared with the specification M. It is not an implementation of M, since again some required behaviour is missing: After the sequence bac the initial state of the specification is reached, which requires the sequences abc and ba to be present. Thus, the lts is modified again to add the missing behaviour, resulting in the lts on the right side of Figure 9.2.

Again, this lts is not the reachability graph of any Petri net, so another minimal over-approximation is generated. This time, due to the bound $k = 1$, the shape of the lts

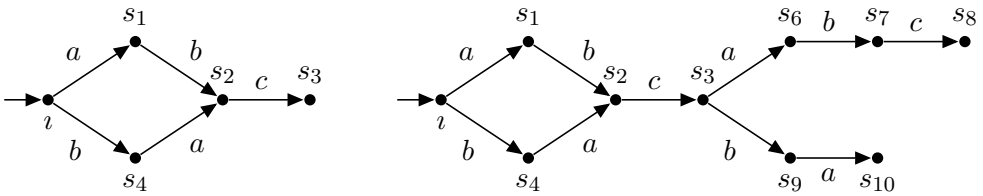


Figure 9.2.: The minimal over-approximation of the lts from the right of Figure 9.1 and the result of adding required behaviour to this lts.

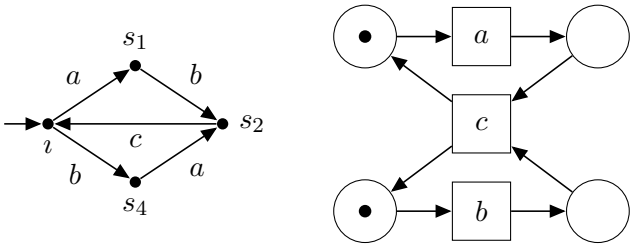


Figure 9.3.: A realisation of the mts M. On the left as an implementing lts and on the right a realising Petri net generating this lts.

changes more drastically: The sequence *abcabc* is enabled in this lts' initial state. If the subsequence *abc* would generate or consume tokens, then the whole sequence would only be possible if more than $k = 1$ token existed in some marking. Thus, the sequence *abc* cannot change the current marking and must instead form a circle in the reachability graph. This leads to the minimal over-approximation shown in Figure 9.3.

This lts is now an implementation of the specification M and is generated by a Petri net, so a realisation of M was found.

Next, we consider an example where the algorithm fails. The mts from Figure 9.4 is identical to the mts from Figure 9.1, except that the may edge $s_5 \overset{c}{\dashrightarrow} \imath$ was removed. The first iteration of the algorithm would proceed as before, constructing the lts from the left side of Figure 9.2. Next, this lts is compared with the new, restricted specification and the algorithm fails: In the lts, $\imath \overset{ba}{\longrightarrow} s_2$ reaches a state with an outgoing edge for label

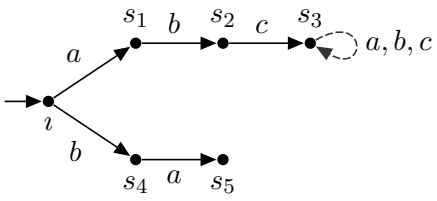


Figure 9.4.: A variant of the mts from Figure 9.1 where a may edge emanating from state s_5 was removed.

c. However, in the mts, $\imath \xrightarrow{ba} s_5$ reaches a state with no outgoing may edge for the label c. Since no allowing may edge exists, this lts cannot implement the mts. Adding more behaviour to it would not change this observation. Thus, the lts cannot be extended to construct an implementation of the specification and we conclude that the modified mts cannot be realised by a Petri net.

Since the example from Figure 9.1 will be reused as an example later, it will now be translated into the modal μ-calculus. We use $\not\dashrightarrow^{a,b}$ to express $[a]\texttt{false} \wedge [b]\texttt{false}$ succinctly:

$$\nu X. \not\dashrightarrow^{c} \wedge \langle a \rangle (\not\dashrightarrow^{a,c} \wedge \langle b \rangle (\not\dashrightarrow^{a,b} \wedge \langle c \rangle \texttt{true})) \wedge \langle b \rangle (\not\dashrightarrow^{b,c} \wedge \langle a \rangle (\not\dashrightarrow^{a,b} \wedge [c] X))$$

The inner part $\langle a \rangle (\not\dashrightarrow^{a,c} \wedge \langle b \rangle (\not\dashrightarrow^{a,b} \wedge \langle c \rangle \texttt{true}))$ of this formula expresses that after a, only b is possible, then only c is possible and then anything is allowed. This corresponds to the upper path in the mts. Similarly, the second half of the formula requires the existence of a path labelled with ba that reaches a state where c, if it is enabled, recurses back to the whole formula. At the outer level, those two formulas are combined with $\not\dashrightarrow^{c}$ and a fixed point operator.

However, to keep the example manageable, we allow an inexact translation: We leave out subformulas of the kind $\not\dashrightarrow^{a}$ and arrive at the following formula:

$$\nu X. \langle a \rangle \langle b \rangle \langle c \rangle \texttt{true} \wedge \langle b \rangle \langle a \rangle [c] X$$

9.2. Local Model Checking

Before the idea of the previous section can be turned into an algorithm, some details have to be clarified. Given a modal specification and an lts A, the idea is to add new edges to A for behaviour that is required by the specification, but that is not yet present in A. However, how is missing behaviour identified? For this, we need to relate states of A to the modal specification in some way.

Since the goal for the algorithm is to find realisations not only for modal transition systems, but also for formulas of the more expressive modal μ-calculus, this section will deal with the μ-calculus. We investigate *local model checking* for the μ-calculus, which is the following problem:

Problem 9.2.1. *Given a formula β, a finite lts A, and a state s of A, does $s \models \beta$?*

A brute-force approach to this problem could compute all states in A that satisfy β via the semantics $[\![\beta]\!]_A$ of the μ-calculus and check if s is in the set, but this algorithm would be inefficient, which is why local model checking was investigated in the literature. We will later see that one possibility for a negative result is that some required behaviour, $\langle a \rangle \beta$, is not implemented in A. This is exactly the information that the algorithm that was outlined in the previous section needs to extend its current lts.

Stirling and Walker [SW91; SW89] have introduced a tableau method for local model checking in the modal μ-calculus. Its rules are inverse natural deduction type rules, meaning that, for example, there is a rule that decomposes "Does s satisfy $\beta_1 \wedge \beta_2$?" into the two subproblems "Does s satisfy β_1?" and "Does s satisfy β_2?". Each node in the resulting tree is labelled by a *sequent*, which is of the form $s, \Delta \vdash \beta$, where s is a state of the lts, β a formula, and Δ a definition list, which will be explained later.

The rules for the tableau method are shown in Figure 9.5. They naturally capture the meaning of formulas, as was just outlined for conjunction. A complication in this scheme are fixed points, for which a decomposition is harder. The intuitive idea to overcome this is to use fixed point induction, which can also be considered an unrolling of fixed points: If we can derive from the assumption $s \in [\![\nu X.\langle a\rangle X]\!]_A^{\mathsf{val}}$ that $s \in [\![\langle a\rangle \nu X.\langle a\rangle X]\!]_A^{\mathsf{val}}$, then $s \in [\![\nu X.\langle a\rangle X]\!]_A^{\mathsf{val}}$ is indeed true. Here, $\langle a\rangle \nu X.\langle a\rangle X$ is generated by taking the inner formula of the fixed point and substituting the full formula for X. Similarly, for a least fixed point, $s \in [\![\langle a\rangle \mu X.\langle a\rangle X]\!]_A^{\mathsf{val}}$ has to be derived from $s \notin [\![\mu X.\langle a\rangle X]\!]_A^{\mathsf{val}}$. The soundness of this argument is based on the fixed point induction introduced in Theorem 7.2.4, which begins with `true` for greatest fixed points, but `false` for least fixed points.

To track this unrolling of fixed points, definition lists Δ and constant symbols U_i are introduced. A constant U_i is similar in meaning to a variable, but is used specially in the evaluation of fixed points. A constant U_i is associated with a (fixed point) formula Ψ_i and a set of states J_i. The set of states tracks in which states this constant was already expanded, ensuring that it is not expanded twice in some state.

Definition 9.2.2 (Definition list)**.** *A* definition list Δ *is a list of constant definitions* $(U_i = \Psi_i, J_i)$ *of the form* $\Delta = \langle (U_1 = \Psi_1, J_1), \ldots, (U_n = \Psi_n, J_n)\rangle$*, where each* Ψ_i *is a formula that can contain constants* U_1 *to* U_{i-1}*, and* $J_i \subseteq S$ *tracks the set of states where* U_i *was already expanded.*

There are two operations that will be needed on definition lists: Adding a new constant for a formula β is written as $\Delta \cdot (U = \beta)$, where U represents a new constant which was not yet expanded anywhere. Adding a state s to J_i is done by Δ_i^s, which updates the definition list to indicate that the constant U_i was expanded in state s.

Definition 9.2.3 (Operations $\Delta \cdot (U = \beta)$ and Δ_i^s)**.** *Given a definition list* $\Delta = \langle (U_1 = \Psi_1, J_1), \ldots, (U_n = \Psi_n, J_n)\rangle$*, the operation* $\Delta \cdot (U = \beta) = \langle (U_1 = \Psi_1, J_1), \ldots, (U_n = \Psi_n, J_n), (U = \beta, \emptyset)\rangle$ *appends the element* $(U = \beta, \emptyset)$ *to* Δ*, and* Δ_i^s *replaces the element* $(U_i = \Psi_i, J_i)$ *in* Δ *with* $(U_i = \Psi_i, J_i \cup \{s\})$*.*

The interpretation β_Δ of a formula β relative to a definition list Δ treats the constants as variables whose definition is provided by the definition list. In detail, this means that if a definition list Δ contains all constants occurring in a formula β, then define $[\![\beta_\Delta]\!]_A^{\mathsf{val}} = [\![\beta]\!]_A^{\mathsf{val}_n}$, where $\mathsf{val}_0 = \mathsf{val}$, $\mathsf{val}_{i+1} = \mathsf{val}_i[[\![\Psi_{i+1}]\!]_A^{\mathsf{val}_i}/U_i]$ up to the length n of the definition list. The next lemma shows that with this definition, β_Δ has the same interpretation as the formula β with each constant symbol U_i substituted with the associated fixed point formula Φ_i.

$$\frac{s, \Delta \vdash \langle a \rangle \beta}{s', \Delta \vdash \beta} \text{ where } s \xrightarrow{a} s' \qquad \frac{s, \Delta \vdash [a]\beta}{s_1, \Delta \vdash \beta \quad \cdots \quad s_n, \Delta \vdash \beta} \text{ where } \{s_1, \ldots, s_n\} = \{s' \mid s \xrightarrow{a} s'\}$$

$$\frac{s, \Delta \vdash \theta X.\beta}{s, \Delta \cdot (U = \theta X.\beta) \vdash U} \qquad \frac{s, \Delta \vdash U_i}{s, \Delta_i^s \vdash \beta[U_i \leftarrow X]} \text{ where } (U_i = \theta X.\beta, J_i) \in \Delta \text{ and } s \notin J_i$$

$$\frac{s, \Delta \vdash \beta_1 \vee \beta_2}{s, \Delta \vdash \beta_1} \qquad \frac{s, \Delta \vdash \beta_1 \vee \beta_2}{s, \Delta \vdash \beta_2} \qquad \frac{s, \Delta \vdash \beta_1 \wedge \beta_2}{s, \Delta \vdash \beta_1 \quad s, \Delta \vdash \beta_2}$$

Figure 9.5.: The rules for the tableau system. $\theta \in \{\nu, \mu\}$ is an arbitrary fixed point.

Lemma 9.2.4 ([SW91])**.** $[\![\beta_{\Delta \cdot (U=\Phi)}]\!]_A^{\mathsf{val}} = [\![(\beta[\Phi \leftarrow U])_\Delta]\!]_A^{\mathsf{val}}$.

Proof. By structural induction. The only non-trivial case is if $\beta = U$, because otherwise the induction hypothesis directly shows the conclusion. For $\beta = U$, the definition above evaluates the left hand side to $[\![U_{\Delta \cdot (U=\Phi)}]\!]_A^{\mathsf{val}} = [\![U]\!]_A^{\mathsf{val}_{n+1}} = [\![\Phi]\!]_A^{\mathsf{val}_n}$, where n is the length of the definition list. We can replace Φ with $U[\Phi \leftarrow U]$ and then use the above definition to arrive at the right hand side $[\![(U[\Phi \leftarrow U])_\Delta]\!]_A^{\mathsf{val}}$. □

With these notations, we can now give the rules of the tableau system. The rules are shown in Figure 9.5. Each rule is applicable if the current sequent matches the pattern shown above the line, in which case the list of new sequents below the line is produced. This can be subject to some side condition. When no more rules are applicable, the resulting leaves[2] can be used to decide if the original state satisfies the original formula. For example, this is the case when all leaves have the formula `true`, but a single leaf with formula `false` indicates a failure. However, since e.g. disjunctions are handled by picking one disjunct, this failure does not necessarily mean that the original formula does not hold, because picking the other disjunct of a disjunction could succeed.

The rule for the existential modality $\langle a \rangle \beta$ in state s (top left) requires to pick one edge $s \xrightarrow{a} s'$ and produces a sequent for state s' with formula β. The rule for the universal modality $[a]\beta$ (top right) proceeds similarly, but produces one sequent for each state reachable via a. The last row of the figure shows the rules for disjunction and conjunction: For disjunction, one of the disjuncts is picked, while conjunction produces two sequents that both must hold.

The remaining two rules in the middle row are for fixed points and constant symbols. A fixed point $\nu X.\langle a \rangle X$ is handled by the rule on the left by introducing a new constant symbol U for it, adding it to the definition list, and also using U as the new formula to check. A constant symbol on the other hand is handled by looking up the definition of the constant symbol in the definition list. The inner formula of the original fixed point is used as the new formula, but the bound variable X is substituted with the constant symbol. In the example, this would result in the formula $\langle a \rangle U$.

[2] A leaf of a tableau is a sequent where no rules can be applied.

The rule for constant symbols can only be applied if the symbol was not yet expanded in the current state. When a constant symbol appears a second time in a state s, the rule cannot be applied again. Thus, the constant symbol is a leaf. If the constant symbol belongs to a least fixed point, then the algorithm fails, while for a greatest fixed point such a leaf indicates success. This is the implementation of the fixed point unrolling that was mentioned above.

There is no rule for negation in Figure 9.5. Negation in the modal μ-calculus is not a fundamental operation, but was defined in Definition 7.2.5 as an abbreviation. It cannot be incorporated into this tableau method directly, since that would require tableaux to be able to show that a formula does not hold.

Applying these rules produces a proof tree, as follows.

Definition 9.2.5 (Proof tree, tableau)**.** *A* proof tree for $s \vdash \beta$ *is a rooted tree* $T = (V, E, r, l)$ *with vertices* V*, edges* E*, root* $r \in V$ *and a labelling function* l *that assigns to each vertex* $v \in V$ *a sequent, so that the sequent* $s, \langle\rangle \vdash \beta$ *is the label* $l(r)$ *of the root. An edge* $(v_1, v_2) \in E$ *may only exist if it corresponds to an application of a rule from Figure 9.5. This means that the children of a vertex are labelled by the result of applying one of the rules from Figure 9.5. Also, all children are generated by the same rule application. A* tableau *is a maximal proof tree, which means that none of the rules can be applied to its leaves, where a* leaf *is a is sequent without rule applications.*

Applying the rules is not necessarily deterministic: There are two rules for disjunction allowing to pick either disjunct. The existential modality $\langle a \rangle$ allows to pick one out of possibly multiple successors. This means there can be multiple tableaux for the same input.

An example from [SW91] for a tableau is shown in Figure 9.6. This figure also shows the lts that was used, which has two states that can be reached from each other via a, only one of which has label b enabled. The formula that is checked is $\nu X.\mu Y.[a]((X \wedge \langle b \rangle \texttt{true}) \vee Y)$, which expresses that label b is enabled infinitely often along all a-paths, i.e. the loop through Y, which is bound by a least fixed point, may only be taken finitely often until the case of the disjunction without Y holds, which requires label b to be enabled.

The tableau in Figure 9.6 has its root at the top. It contains a state $\imath$, the empty definition list and the full formula to check. Since the outermost syntactic element of the formula is a fixed point, only the rule that introduces constant symbols can be used. This rule introduces a new constant symbol U_1 and records its definition as the full formula in the definition list. For space reasons, the full formula was abbreviated as Φ_1. Next, this constant symbol is expanded. Writing the original formula as $\nu X.\beta$, this produces $\beta[U_1 \leftarrow X]$, i.e. the inner formula of the fixed point with all free occurrences of X substituted with the constant symbol U_1. This procedure is repeated for the inner formula, because it is another fixed point formula.

The fifth row in the tableau is now a universal modality $[a]\beta$ for some formula β. This sequent still contains the initial state $\imath$, so its outgoing a-edges are considered. This is

$$\imath, \langle\rangle \vdash \nu X.\mu Y.[a]((X \wedge \langle b\rangle \texttt{true}) \vee Y)$$

$$\imath, \langle(U_1 = \Phi_1, \emptyset)\rangle \vdash U_1$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath\})\rangle \vdash \mu Y.[a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee Y)$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \emptyset)\rangle \vdash U_2$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash [a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee U_2)$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash (U_1 \wedge \langle b\rangle \texttt{true}) \vee U_2$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash U_1 \wedge \langle b\rangle \texttt{true}$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash U_1 \qquad s_1, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash \langle b\rangle \texttt{true}$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash \mu Y.[a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee Y) \qquad s_2, \langle(U_1 = \Phi_1, \{\imath\}), (U_2 = \Phi_2, \{\imath\})\rangle \vdash \texttt{true}$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \emptyset)\rangle \vdash U_3$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{s_1\})\rangle \vdash [a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee U_3)$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{s_1\})\rangle \vdash (U_1 \wedge \langle b\rangle \texttt{true}) \vee U_3$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{s_1\})\rangle \vdash U_3$$

$$\imath, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash [a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee U_3)$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash (U_1 \wedge \langle b\rangle \texttt{true}) \vee U_3$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash U_1 \wedge \langle b\rangle \texttt{true}$$

$$s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash U_1 \qquad s_1, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash \langle b\rangle \texttt{true}$$

$$s_2, \langle(U_1 = \Phi_1, \{\imath, s_1\}), (U_2 = \Phi_2, \{\imath\}), (U_3 = \Phi_2, \{\imath, s_1\})\rangle \vdash \texttt{true}$$

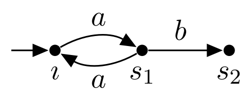


Figure 9.6.: An example tableau for $\Phi_1 = \nu X.\mu Y.[a]((X \wedge \langle b\rangle \texttt{true}) \vee Y)$. The lts is shown on the right and the tableau witnessing $\imath \models \Phi_1$ on the left. As an abbreviation, $\Phi_2 = \mu Y.[a]((U_1 \wedge \langle b\rangle \texttt{true}) \vee Y)$ was used.

only $\imath \xrightarrow{a} s_1$, so the only possible child is labelled with a sequent for s_1 and formula β. The resulting child is a disjunction. This allows to pick one disjunct non-deterministically. Here, the left disjunct is taken. It would also be possible to take the right disjunct, but this would lead to an unsuccessful tableau, because the constant symbol U_2 would again be considered in state $\imath$. Since U_2 belongs to a least fixed point, this would be an unsuccessful leaf.

Next, a conjunction appears, so now two children are produced. The right sequent has the formula $\langle b \rangle \texttt{true}$ and belongs to state s_1. This produces one sequent for `true` in s_2. The left sequent has the constant symbol U_1. This constant symbol was already expanded previously, but in state $\imath$. Since it was not yet expanded in s_1, the corresponding rule is applicable. This produces a sequent similar to the one in the third row of the proof tree, but this time for state s_1 instead of $\imath$. This results in a part of the tree to be repeated. Note that this repetition introduces a new constant symbol U_3. Also, this time the other disjunct in the disjunction is taken. It would also be possible to take the same disjunct as previously, but this would again lead to an unsuccessful proof tree. The remaining rule applications are straightforward.

The tableau has three leaves, two of which are labelled with the formula `true` and one with the constant symbol U_1. The rule for constant symbols cannot be applied here, because the constant symbol was already expanded before in the current state, as tracked in the definition list.

Coming back to the tableau system, all the rules are backwards sound, i.e. when the children of a node are true, then so is the node itself. To see this, interpret a sequent $s, \Delta \vdash \beta$ as $s \in [\![\beta_\Delta]\!]_A$. For all rules except for fixed points, backward soundness is then easy to see. Fixed points are unrolled, meaning that e.g. $\nu X.\beta$ is replaced with $\beta[\nu X.\beta \leftarrow X]$, i.e. the formula β where all free occurrences of X are replaced with the fixed point itself. This unrolling only happens implicitly in the handling of constant symbols, which are introduced when a fixed point occurs first. The unrolling is sound by Lemma 9.2.4.

The side condition for fixed point unrolling guarantees that all proof trees are finite. All other rules decrease the length of the formula under consideration, so they cannot lead to non-termination, while fixed points in general could. However, since constant symbols are only expanded once per state (and contain only finitely many inner fixed points), they again cannot produce infinite paths. Thus:

Lemma 9.2.6 ([SW91])**.** *Every tableau is finite.*

So far we can construct tableaux. The next definition uses tableaux for model-checking. For this, the leaves of a tableau are examined: By the argument above, if they hold, then the root of the tableau is true, i.e. the state at the root is a model of the formula.

Definition 9.2.7 (Successful leaf, witnessing tableau)**.** *A tableau for $s \vdash \beta$* witnesses *that $s \models \beta$ if all its leaves are successful. A leaf is* successful *if it is labelled with* `true`*, a universal modality $[a]\beta$ (which is only possible as a leaf if no outgoing edge with the*

label a exists), or a constant U_i for a greatest fixed point formula $\nu X.\beta$ (which means the constant was already expanded before in the current state).

The following possibilities for unsuccessful leaves exist: They can be labelled with `false`, a free variable X, an existential modality $\langle a \rangle \beta$ (in which case no outgoing edge with the label a exists), or a constant symbol for a least fixed point (in which case fixed point induction failed and the constant would need expansion infinitely often). The other possibilities (conjunction, disjunction, or a fixed point) cannot appear as leaves in a tableau since their rules are always applicable and have no side condition.

Tableaux can be used for model checking in the following way:

Theorem 9.2.8 ([SW91]). *$s \models \beta$ if and only if there is a successful tableau whose root is labelled with $s, \langle\rangle \vdash \beta$.*

Note that this theorem only requires the existence of a successful tableau. Thus, in practice, it will be necessary to construct all tableaux to get a definitive negative answer.

9.3. Computing Petri Net Realisations of the μ-Calculus

The tableaux introduced in the previous section allow to identify missing edges: If a leaf of the tableau has an existential formula $\langle a \rangle \beta$, then the corresponding state needs an outgoing edge with label a. With this insight, we can now formulate the algorithm that was sketched in Section 9.1.

In the previous section it was mentioned that there can be multiple tableaux for a given input. This is one source for non-determinism for the algorithm. For example, in a formula like $\langle a \rangle \texttt{true} \vee \langle b \rangle \texttt{true}$, we have a choice between adding an a-edge or a b-edge to the lts. However, some of these choices might not lead to realisation, for example, because the required behaviour cannot be reproduced by a Petri net. Thus, the algorithm for synthesis will have to follow all possibilities.

This means that the current state of the algorithm will not be just a single lts, as was indicated in the introduction, but instead a set of pairs of lts and tableaux. The tableaux track the decision between the possible branches of a disjunction[3].

So far, the algorithm starts with the lts that consists of just an initial state. All possible tableaux for this lts and the given formula are calculated. Then, the leaves with existential modalities are used to extend the lts with new states and edges. The resulting lts are minimally over-approximated with a Petri net. The already existing tableaux are then transferred to this new lts. This produces non-maximal proof trees, so the resulting proof trees are extended again into tableaux. Extending a proof tree can again produce

[3]The only other non-determinism in the construction of tableaux is the rule for existential modalities. However, in deterministic lts, such as reachability graphs, this rule is deterministic.

multiple tableaux. These steps are repeated until either a successful tableau is found, or there is no way to continue.

Before this sketch can be formalised into an algorithm, we have to clarify the abort condition of the algorithm, i.e. when can a given lts not be completed into an implementation, and how are tableaux transferred to other lts.

As an abort condition, leaves which are labelled with `false`, a free variable, or a constant symbol for a least fixed point are used:

Definition 9.3.1 (Surely false)**.** *A leaf is* surely false *if it is unsuccessful and not labelled with an existential modality. A tableau is* surely false *if one of its leaves is.*

Thus, if the algorithm produces a surely false tableau, this tableau and the corresponding lts can be discarded. If all pairs of lts and tableaux are either discarded or completed into implementations, the algorithm found all minimal realisations and terminates.

Another open problem is how to transfer a tableau to a different lts. For this, the concept of lts homomorphisms from Section 5.1 is used. In fact, this concept is the partial order for the minimality in the minimal over-approximation. The transformation will replace each state s appearing in the tableau by the state $f(s)$ to which the lts homomorphism f maps s.

Definition 9.3.2 (Transferred proof tree)**.** *Given a proof tree $T = (V, E, r, l)$ for the lts A_1 and an lts A_2 with $A_1 \sqsubseteq A_2$ via f, the* transferred proof tree $f(T)$ *is $f(T) = (V, E, r, l')$ where l' is defined by $l'(v) = [f(s), f(\Delta) \vdash \beta]$ if $l(v) = [s, \Delta \vdash \beta]$. Here, $f(\Delta) = \langle (U_1 = \Psi_1, f(J_1)), \ldots, (U_n = \Psi_n, f(J_n)) \rangle$ when $\Delta = \langle (U_1 = \Psi_1, J_1), \ldots, (U_n = \Psi_n, J_n) \rangle$.*

A problem with this definition is that it may produce proof trees which cannot be constructed ordinarily, or even invalid proof trees. This is because lts homomorphisms can identify two states and states can have new outgoing edges. For example, a constant could have been first expanded in state s and later in a different state s', but these two states are identified by the lts homomorphism. However, the condition that constants can only be expanded once in a state is only needed so that tableaux are finite and not for soundness. Thus, this deviation from the proof rules is not actually a problem and could e.g. be fixed by removing the part of the tree between the two expansions, even though this is not really needed.

The remaining problem concerns the modalities: Some state s of A_1 could have less outgoing edges with label a than the state $f(s)$ of A_2. In this case, the rule for the existential and universal modalities can produce different sequents. However, we sidestep this problem by restricting our attention to deterministic lts, since Petri net reachability graphs are deterministic by Lemma 2.0.8: If s already has an outgoing a-edge, then $f(s)$ cannot have additional outgoing a edges by determinism and there is no problem. If s has no outgoing a-edges, then the corresponding sequent must have been a leaf, so further rules can be applied to enlarge the mapped proof tree $f(T)$ into a tableau.

The above considerations amount to a proof of:

Lemma 9.3.3. *Given deterministic lts A_1 and A_2 with $A_1 \sqsubseteq A_2$ via f and a tableau T for some state s of A_1, there is a proof tree T' of A_2 for state $f(s)$, which is equivalent[4] to the transferred proof tree $f(T)$.*

The algorithm that was so far only sketched is now formalised as Algorithm 4. Its execution begins in the procedure RealiseFormula. The arguments to this procedure are the bound k for k-bounded over-approximation, and the formula β that should be solved. In Line 2 the minimal lts according to $\sqsubseteq$ is constructed, which is the lts with just an initial state and no edges. Then, in Line 3, all tableaux for this lts and the formula β are constructed. For each of them the procedure Recurse is called in Line 4 to produce all minimal realisations. The algorithm actually produces Petri net solvable implementing lts instead of realisations. We will not differentiate between these.

Algorithm 4 Algorithm for finding Petri net realisations for a formula.

```
1: procedure RealiseFormula(k, β)                    ▷ k ∈ ℕ+ and β of the μ-calculus
2:     Let A = ({ι}, Σ, ∅, ι) be the lts consisting of just an initial state ι
3:     Let 𝒯 be the set of all tableaux for ι ⊢ β
4:     return ⋃_{T∈𝒯} Recurse(k, A, T)
5: end procedure
6: procedure Recurse(k, A, T)
7:     if T is surely false then return ∅ end if
8:     for T has a leaf labelled s, Δ ⊢ ⟨a⟩β for some Δ and some β do
9:         Add a new state s_new and an additional edge s -a-> s_new to A
10:    end for
11:    if no edges were added then return {(A, T)} end if
12:    return ⋃_{(A',T')∈PNApprox(k,A,T)} Recurse(k, A', T')
13: end procedure
14: procedure PNApprox(k, A, T)
15:    A' = Approx_k(A)                              ▷ Minimal over-approximation
16:    f = (unique) homomorphism witnessing A ⊑ A'
17:    T' = f(T)                                     ▷ Transfer tableau from A to A'
18:    Calculate a set 𝒯 of new tableaux by applying proof rules to T' as long as possible
19:    return {(A', T'') | T'' ∈ 𝒯}
20: end procedure
```

The procedure Recurse checks if the given tableau is surely false (Line 7), which indicates that no implementations can be found based on it. Then, for each leaf that indicates that some state s needs an outgoing edge with label a, a new state and a corresponding edge is added. If no such leaf exists, then the tableau must be successful, because all possibilities for unsuccessful leaves were excluded. Thus, Line 11 returns the lts and its tableau as one possible Petri net solvable implementation. The original call to RealiseFormula will return a set containing all these implementations. Otherwise, Line 12

[4] When ignoring duplicate constant unrolling, as explained above.

$$\dfrac{\dfrac{\dfrac{}{\imath, \langle\rangle \vdash \nu X.(\langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X)}}{\imath, \langle(U_1 = \Phi_1, \emptyset)\rangle \vdash U_1}}{\dfrac{\imath, \langle(U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]U_1}{\imath, \langle(U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \qquad \imath, \langle(U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle b\rangle\langle a\rangle[c]U_1}}$$

Figure 9.7.: First tableau for the example.

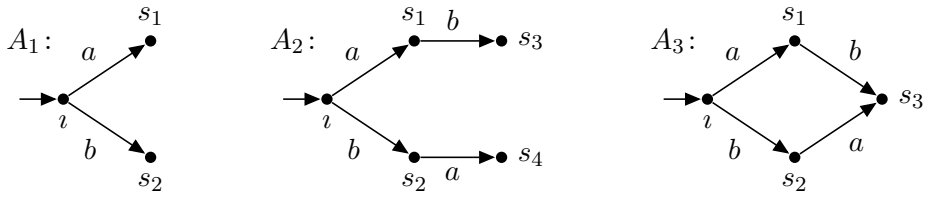


Figure 9.8.: The lts generated in the first few iterations of the algorithm.

will call PNApprox to compute a minimal over-approximation of the extended lts and to compute new tableaux. For each of these results, Recurse will call itself again.

The procedure PNApprox first computes the minimal over-approximation $\mathrm{Approx}_k(A)$ of the given lts A with respect to k-bounded Petri nets. By Theorem 5.2.2 and Theorem 5.4.5, which showed the existence of minimal Petri net solvable over-approximations, there is an lts homomorphism between an lts and its minimal over-approximation, and by Lemma 5.1.4, there is at most one such homomorphism for reachable and deterministic lts, which our lts are. In line 16, this unique lts homomorphism f is computed and then used to produce a new proof tree $f(T)$. Finally, all extensions of this proof tree into tableaux are computed and returned to Recurse, together with the over-approximating lts.

9.4. Example

Section 9.1 motivated the algorithm with an example. The example was an mts that was translated into the modal μ-calculus at the end of that section, producing the formula $\Phi_1 = \nu X.\langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X$. This formula, together with $k = 1$ as the bound for Petri nets, will now be used for an example of the algorithm.

The function call RealiseFormula(k, β) begins by constructing an lts with just an initial state $\imath$ together with all possible corresponding tableaux. For the given formula, there is only a single tableau. It is shown in Figure 9.7. In the tableau, a constant symbol is introduced and then the inner conjunction is decomposed. This tableau is given to Recurse. No leaf of the tableau is surely false, so the following loop is reached. This adds outgoing edges for labels a and b to the initial state in Line 9, because there are leaves for the formulas $\langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true}$, and $\langle b\rangle\langle a\rangle[c]U_1$, respectively. The resulting lts A_1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\imath, \langle\rangle \vdash \nu X.(\langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X)}}{\imath, \langle (U_1 = \Phi_1, \emptyset)\rangle \vdash U_1}}{\imath, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X}}{\imath, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true}} \quad \dfrac{}{\imath, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle b\rangle\langle a\rangle[c]U_1}}{s_1, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle b\rangle\langle c\rangle \mathtt{true}} \quad \dfrac{}{s_2, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle a\rangle[c]U_1}}{s_3, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash \langle c\rangle \mathtt{true}} \quad \dfrac{}{s_3, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash [c]U_1}$$

Figure 9.9.: Second tableau for the example. The corresponding lts is A_3 in Figure 9.8.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{s_3, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash [c]U_1}}{s_4, \langle (U_1 = \Phi_1, \{\imath\})\rangle \vdash U_1}}{s_4, \langle (U_1 = \Phi_1, \{\imath, s_4\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X}}{s_4, \langle (U_1 = \Phi_1, \{\imath, s_4\})\rangle \vdash \langle a\rangle\langle b\rangle\langle c\rangle \mathtt{true}} \quad \dfrac{}{s_4, \langle (U_1 = \Phi_1, \{\imath, s_4\})\rangle \vdash \langle b\rangle\langle a\rangle[c]X}$$

Figure 9.10.: A subtree added to the right leaf of the tableau in Figure 9.9 in later iterations of the algorithm.

is shown on the left of Figure 9.8. The following over-approximation in PNApprox does not modify the lts, but the tableau can be continued by adding sequents for state s_1 with the formula $\langle b\rangle\langle c\rangle\mathtt{true}$, and state s_2 and the formula $\langle a\rangle[c]U_1$, respectively, to the leaves of the tableau from Figure 9.7. The result is given to a recursive call of Recurse.

This call to Recurse proceeds similarly to the previous one. The new leaves cause new additions, and the lts A_2 shown in the middle of Figure 9.8 is produced. This time, the minimal over-approximation in PNApprox changes the lts, producing A_3 from the same figure. Transferring the tableau to this new lts leads to the tableau from Figure 9.9. This only entails replacing the states, i.e. no new nodes are generated in Line 18 of the algorithm.

The lts A_3 and the tableau from Figure 9.9 are again given to Recurse. The leaf with the formula $\langle c\rangle\mathtt{true}$ causes a c-edge to be added to s_3. The following over-approximation does not modify this lts further, but now the tableau can be continued in the other leaf: There is now a tableau rule that can be applied to $[c]U_3$ in state s_3. Figure 9.10 only shows the subtree that begins in the formula $[c]U_1$ and assumes that $s_3 \xrightarrow{c} s_4$ was added to A_3, i.e. the new state is s_4.

The following development closely mirrors what happened initially: The paths ab and ba have to be added due to the existential modalities. The over-approximation makes these two paths reach the same state and next an outgoing c-edge is added. This results in an lts similar to what was already generated in the introductory example. It is shown in Figure 9.11. Next, over-approximation generates the same lts as in the introductory example, which is repeated Figure 9.12.

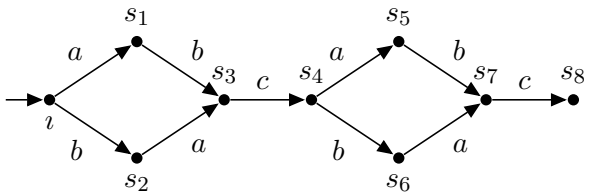


Figure 9.11.: An intermediate lts generated by the algorithm.

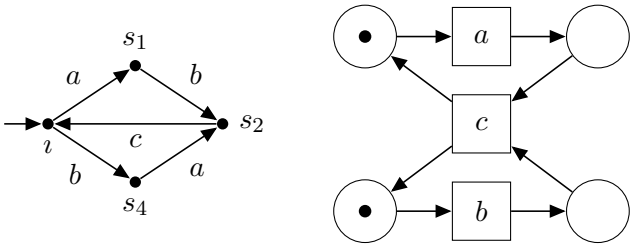


Figure 9.12.: The implementation and realisation found by the algorithm. This repeats Figure 9.3.

The tableau that corresponds to this lts is shown completely in Figure 9.13. Since all of its leaves are successful, the following call to RECURSE will not modify the lts and instead returns it as an implementation of the input formula. Thus, the algorithm terminates with an lts and a tableau. The tableau witnesses that the lts really implements the formula that was given to the algorithm.

Note that the final tableau from Figure 9.13 cannot be generated by applying the tableau rules, because the constant symbol U_1 is expanded twice in state $\imath$, which is not allowed by the tableau rules. A proper tableau could be generated by removing the part of the tableau between the two expansions of U_1. However, this side condition on the rule only exists to ensure that all tableaux are finite, and the correctness proof will argue that it can be removed as long as no infinitely large tableaux are produced.

The algorithm needed six iterations[5] to calculate an implementation for the given example, i.e. until the lts of Figure 9.11 was generated. Compared to the brute-force approach based on Theorem 9.0.1, which would generate and examine up to $2^{(k+1)^{1+2|\Sigma|}} = 2^{2^7} = 2^{128}$ Petri nets, this is clearly faster.

9.5. Correctness of the Algorithm

In this section, three results will be shown: The algorithm terminates; the algorithm is correct in the sense that it produces implementations of the given formula; and the

[5] A longest path in the lts from Figure 9.11 is labelled with *abcabc*, which has six labels. The algorithm generates one label of this path per iteration, so six iterations are needed to generate it.

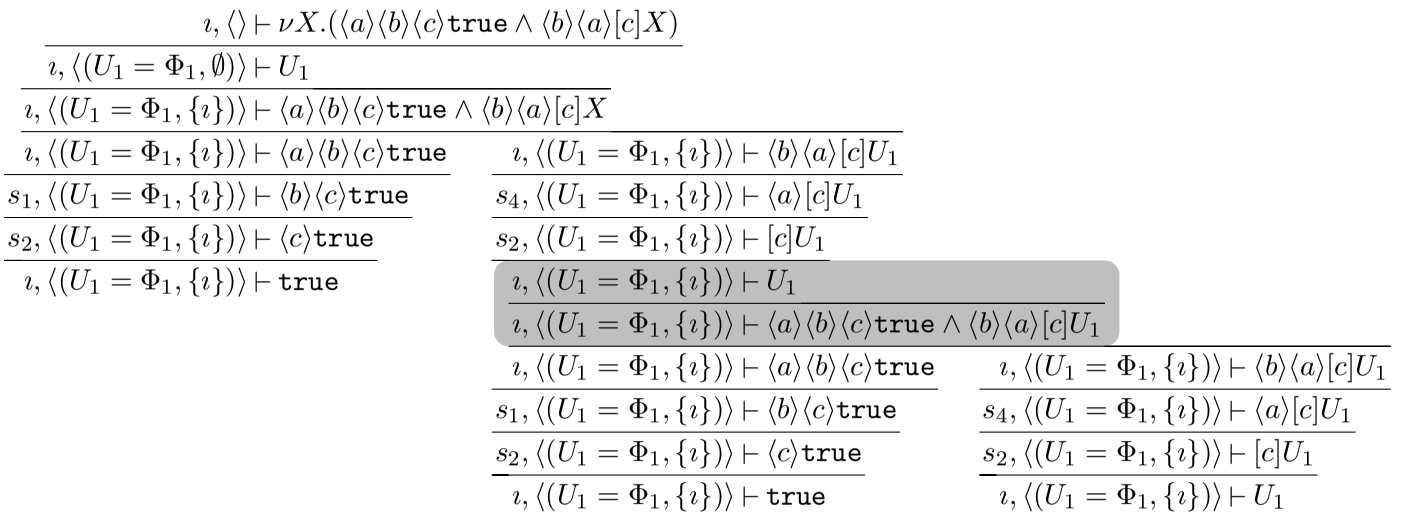


Figure 9.13.: The final tableau showing that the lts from Figure 9.12 implements the formula $\Phi_1 = \nu X.(\langle a\rangle\langle b\rangle\langle c\rangle\mathtt{true} \wedge \langle b\rangle\langle a\rangle[c]X)$. The highlighted rule application is not allowed by the tableau rules, because U_1 was already expanded in $\imath$ before, but is generated by the algorithm due to an lts homomorphism identifying two originally separate states. By the tableau rules, the sequent with formula U_1 would be a leaf.

algorithm is complete in the sense that it finds a Petri net solvable implementation if one exists.

Lemma 9.5.1. *For a formula β and $k \in \mathbb{N}_+$,* RealiseFormula(k, β) *terminates.*

Proof. If the algorithm does not terminate, this must be due to Recurse, because all loops are finite and no other procedures recursively call themselves. Since no infinitely large objects can be constructed by the algorithm, there must be an infinite recursion of Recurse with arguments $(A_i)_{i\in\mathbb{N}}$ and $(T_i)_{i\in\mathbb{N}}$ (k is always the same value). All arguments to Recurse can be solved by Petri nets by construction. By Theorem 9.0.1, there are only finitely many k-bounded Petri net reachability graphs over a fixed alphabet. Thus, there are also only finitely many lts that can be solved by k-bounded Petri nets. This means that the A_i are all elements of a finite set, i.e. some lts A (up to isomorphism) will occur infinitely often. Also $A_i \sqsubseteq A_{i+1}$ for $i \in \mathbb{N}$, because in Recurse, only new states and edges are added, which preserves $\sqsubseteq$, and the minimal Petri net solvable over-approximation $\text{Approx}_k(A)$, calculated in PNApprox, satisfies $A \sqsubseteq \text{Approx}_k(A)$ by Theorem 5.2.2.

Since Petri net reachability graphs are reachable and deterministic by Lemma 2.0.8, Lemmas 5.1.2 and 5.1.5 are applicable, which together state that $\sqsubseteq$ is a partial order. In a partially ordered sequence, if some element A occurs infinitely often, then the sequence must become stationary: If $A \sqsubseteq B \sqsubseteq A$, then $A = B$. This means that there is some $j \in \mathbb{N}$ so that for all $i \geq j$, $A_i = A$ holds.

Consider now some $i \geq j$. Let $\widetilde{A_i}$ be the lts constructed in Recurse from A_i. We have $A_i \sqsubseteq \widetilde{A_i}$ via the identity homomorphism id, because only new states and edges are added in the construction. Now, by assumption PNApprox$(k, \widetilde{A_i}, T_i)$ recreates the original lts, i.e. $\widetilde{A_i} \sqsubseteq A_{i+1}$ via some homomorphism f. Overall, $A = A_i \sqsubseteq \widetilde{A_i} \sqsubseteq A_{i+1} = A$ via $\mathrm{id} \circ f$. In this context, Lemma 5.1.4 about the uniqueness of lts homomorphisms states that we can conclude $\mathrm{id} = \mathrm{id} \circ f$ from $A \sqsubseteq A$ via id and $A \sqsubseteq A$ via $\mathrm{id} \circ f$. Thus, f must already be the identity mapping on all states except for the added states s_{new}.

If Recurse would add an edge $s \xrightarrow{a} s_{\text{new}}$ to A_i to construct $\widetilde{A_i}$, then $f(s) \xrightarrow{a} f(s_{\text{new}})$ would also be an edge in A_{i+1}. Since we already know that f is the identity mapping on states from A_i, this new edge actually is $s \xrightarrow{a} f(s_{\text{new}})$ in A_{i+1}. Since $A_i = A_{i+1}$, s must already have an outgoing edge with label a in A_i. However, then there cannot be a leaf with the sequent $s, \Delta \vdash \langle a \rangle \beta$ in T_i, because this can only be a leaf in a tableau if s has no outgoing edge with label a, because otherwise a proof rule would be applicable. Thus, Recurse does not modify the lts, which means that it is returned as an implementation. □

The next lemma shows that RealiseFormula only produces Petri net solvable implementations of its input:

Lemma 9.5.2. *Let a formula β and a number $k \in \mathbb{N}_+$ be given. Then, for each tuple $(A, T) \in$* RealiseFormula(k, β)*, the tableau T witnesses $A \models \beta$ and A can be solved by a k-bounded Petri net.*

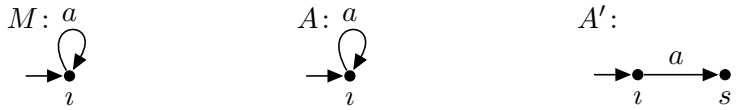


Figure 9.14.: An mts M, an implementation A, and an intermediate lts A' constructed while attempting to solve M.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\iota, \langle\rangle \vdash \nu X.\langle a\rangle X}}{\iota, \langle (U_1 = \nu X.\langle a\rangle X, \emptyset)\rangle \vdash U_1}}{\iota, \langle (U_1 = \nu X.\langle a\rangle X, \{\iota\})\rangle \vdash \langle a\rangle U_1}}{\iota, \langle (U_1 = \nu X.\langle a\rangle X, \{\iota\})\rangle \vdash U_1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\iota, \langle\rangle \vdash \nu X.\langle a\rangle X}}{\iota, \langle (U_1 = \nu X.\langle a\rangle X, \emptyset)\rangle \vdash U_1}}{\iota, \langle (U_1 = \nu X.\langle a\rangle X, \{\iota\})\rangle \vdash \langle a\rangle U_1}}{s, \langle (U_1 = \nu X.\langle a\rangle X, \{\iota\})\rangle \vdash U_1}}{s, \langle (U_1 = \nu X.\langle a\rangle X, \{\iota, s\})\rangle \vdash \langle a\rangle U_1}$$

Figure 9.15.: Tableaux for the lts A (left) and A' (right) from Figure 9.14.

Proof. For (A, T) to be returned by RealiseFormula, it must be returned by Line 11 of the algorithm in a call to Recurse. This means that it has no leaves with existential modalities and no surely false leaves (Line 7). By definition, this means that only successful leaves remain. By Theorem 9.2.8 the tableau[6] T witnesses $A \models \beta$.

It remains to show that A can be solved by a k-bounded Petri net. This is the case because every lts A given to Recurse is either the trivial lts having no edges or was generated by PNApprox as a minimal Petri net solvable over-approximation. □

We showed termination and correctness, but completeness is still missing. This means that RealiseFormula$(k, \beta) = \emptyset$ can only occur if there are no realisations of β. The idea for this proof is to assume that a realisation exists and to show that the current state of the algorithm always contains an lts and a proof tree which are 'smaller' than the realisation. For lts, the relation $\sqsubseteq$ already exists to formalise 'smaller'. For proof trees, a prefix will be defined, the idea of which is that a proof tree can be constructed from another one by repeatedly removing leaves.

However, there is one complication. Figure 9.14 shows an mts M that requires a to be possible infinitely often. This mts corresponds to the formula $\nu X.\langle a\rangle X$. A possible implementation is the lts A. The algorithm will initially conclude that the initial state needs an outgoing edge with label a and construct the lts A' from the same figure. We now want that A' is smaller than the implementation A, i.e. the algorithm can still construct A from A'. This holds, because of $A' \sqsubseteq A$.

However, the algorithm also tracks tableaux together with the lts that it constructs. In this case the tableaux are unique and shown in Figure 9.15. We can see that the intuitive

[6] See also Lemma 9.3.3 for why the constructed tableau is indeed a valid tableau even though fixed points are possibly expanded multiple times in the same state.

definition of a prefix of a tableau, the removal of leaves, is not possible here, because the tableau for A' has more nodes than the tableau for A. Making the tableau of A even shorter will not result in two equivalent tableaux.

The problem is that in the tableau for A, the edge with label a forms a loop. Thus, the constant U_1 can only be unrolled once. For A' however, a does not form a loop and so the constant can be expanded once more. To overcome this problem, tableaux themselves will be unrolled: In the tableau on the left of Figure 9.15, there is a leaf with formula U_1. This constant symbol cannot be expanded, because it was already expanded previously in the current state. The *expanded tableau* will be constructed by expanding U_1 again and continuing with the same subtree as before to construct an infinitely large tree.

Definition 9.5.3 (Expanded proof tree)**.** *The* expanded proof tree *of a tableau T is constructed from T by replacing each leaf for a constant symbol by the subtree where this constant symbol was previously expanded. This process is repeated recursively, i.e. the result is infinitely large.*

We can now define the prefix relation for tableaux and proof trees as an embedding into the expanded proof tree:

Definition 9.5.4 (Prefix)**.** *Given a proof tree $T_1 = (V_1, E_1, r_1, l_1)$ and a tableau T_2 whose expanded proof tree is $T_2' = (V_2, E_2, r_2, l_2)$, T_1* is a prefix of *T_2, if there is a function $f\colon V_1 \to V_2$ so that $f(r_1) = r_2$, $(v, v') \in E_1 \Rightarrow (f(v), f(v')) \in E_2$, and for all $v \in V_1\colon l_1(v) = l_2(v)$.*

We can now show the announced result that the algorithm will produce a realisation whenever one exists.

Lemma 9.5.5. *Let β be a formula, $k \in \mathbb{N}_+$ a number, N a Petri net, and T a tableau witnessing that $\mathrm{RG}(N) \models \beta$. Then there is $(A, T') \in \textsc{RealiseFormula}(k, \beta)$ so that $A \sqsubseteq \mathrm{RG}(N)$ and T' is a prefix of T.*

Proof. We show the following invariant of the algorithm: There is always an lts A with an associated tableau T' so that $A \sqsubseteq \mathrm{RG}(N)$ via f and $f(T')$ is a prefix of T.

Initially, the algorithm begins with the lts A that consists of just an initial state. Obviously $A \sqsubseteq \mathrm{RG}(N)$. A proof tree T' for A can be constructed by cutting T at every point where a rule for a modality is applied. Since A does not have any edges, none of these rules are applicable for A. Thus, the resulting proof tree T' is indeed a tableau for A and the invariant holds initially, because all tableaux are constructed.

Given a pair (A, T') fulfilling the invariant, Recurse either returns this pair, or constructs a new pair fulfilling the invariant. In the first case, nothing remains to be shown, so assume that Recurse modifies A in Line 9. By assumption $A \sqsubseteq \mathrm{RG}(N)$ via f and

$f(T')$ is a prefix of T, so the leaf of T' that causes a new edge to be added is also present in T. Thus, $\mathrm{RG}(N)$ must have an outgoing edge with label a in the corresponding marking. This shows that after modifying A, we still have $A \sqsubseteq \mathrm{RG}(N)$.

Next, PNApprox is called. This computes A', a minimal Petri net solvable over-approximation of A. Since $\mathrm{RG}(N)$ is the reachability graph of a Petri net, $A' \sqsubseteq \mathrm{RG}(N)$ (via some function g) still holds (see Theorem 5.3.11 on minimality of the over-approximation). Transferring the tableau to A' only renames the states[7], so $g(T')$ will still be a prefix of T. Finally, this function creates all possible tableaux starting from T'. One of these tableaux will be a prefix of T, so the invariant still holds. □

The following corollary is derived via contraposition:

Corollary 9.5.6. *For a formula β and a number $k \in \mathbb{N}_+$, if* RealiseFormula$(k, \beta) = \emptyset$*, then there is no k-bounded Petri net N with* $\mathrm{RG}(N) \models \beta$.

9.6. Conclusion

In Chapter 8, it was shown that finding pure and bounded Petri net realisations for the conjunctive ν-calculus is undecidable. The current chapter added another parameter $k \in \mathbb{N}$ to the problem and introduced an algorithm for finding k-bounded Petri net realisations. This algorithm not only works for the ν-calculus, but also for the full modal μ-calculus. It proceeds by iteratively adding the behaviour required by the given formula to an lts. The lts is minimally over-approximated via the algorithm from Chapter 5 at each step to guarantee Petri net solvability.

Because the only step specific to Petri nets is the minimal over-approximation, the presented algorithm is compatible with all subclasses from Section 5.4 for which over-approximation is possible. For example, this means that it is possible to ask for k-bounded and pure marked graph implementations of a formula of the modal μ-calculus. As mentioned in Section 5.6, it might even be possible to ask for k-bounded and place-output-nonbranching Petri nets even though the over-approximation itself might not terminate when targeting just place-output-nonbranching Petri nets.

An open question about the algorithm is its complexity, since the complexity of the underlying minimal over-approximation is not known. Because the μ-calculus allows disjunctions, the algorithm has to explore each possible branch of a disjunction separately. This allows to construct formulas like $\mu X.\langle a\rangle X \vee \langle b\rangle X \vee ([a]\mathtt{false} \wedge [b]\mathtt{false})$ that have exponentially many realisations[8]. However, for a formula without disjunctions the

[7] Up to multiple unrolling of fixed points.

[8] The formula is satisfied by any lts where no infinite sequences of a and b are possible. For example, just considering strings, i.e. lts with at most one outgoing edge in a state, there are already $2^{k+1} - 1$ possible strings over $\{a, b\}$ with length $\leq k$, but not all of them can be solved by a Petri net [Ero18; BESW16]. However, there are more realisations that do not correspond to a string.

algorithm can produce at most one realisation, i.e. does not branch. Still, even with this limitation it is not clear how many iterations are needed for termination.

There is also still potential to improve the algorithm. The presented version computes an over-approximation every time the lts was modified. Since computing the over-approximation will very likely be more complex than extending the lts, it makes sense to do multiple steps of appending new behaviour before computing a new over-approximation. This would not interfere with the presented correctness proofs. For example, it might make sense to append new behaviour to the lts until a constant symbol needs to be expanded for the second time. With this modification, the example from Section 9.4 could be solved in just two iterations. Further optimisations of the algorithm will be proposed in the case study in Chapter 11.

10. Implementation

In Chapter 6, an implementation of the algorithms from the first part of this document was presented. This implementation was done in the tool APT. The author of this thesis also implemented the algorithms introduced in the second part of this document. However, since this implementation deals with modal specifications and since the focus of APT is on Petri nets and lts, this implementation was not added to APT. Instead it is available at `https://github.com/CvO-Theory/apt-modal-mu-synthesis` in a tool that uses APT as a library.

This tool provides data structures and a parser for formulas of the modal μ-calculus, as well as some related utility functions. In addition to the modules that are already part of APT, the author implemented the modules `mts_to_formula`, `model_check`, `realise_pn`, `deadlock_free_realise_pn`, and `call_expansion`. They will be explained in this chapter.

10.1. Syntax for Formulas of the Modal μ-Calculus

Formulas need to be specified in plain text. Basic operations like `true` and `false`, parentheses, as well as the modalities `<a>` and `[a]`, and negation `!`, have obvious representations. The remaining operations cannot be easily entered directly. Conjunction $\wedge$ has to be entered as `&&` and a disjunction $\vee$ as `||`. Variables are finite strings of letters and numbers, e.g. `X` is a possible variable, but also `Step5`. While the fixed point operators μ and ν are supported directly, it is easier to type these as `mu` and `nu`. For example, the formula $\nu X.\langle a\rangle\langle b\rangle\langle c\rangle$`true`$\wedge\langle b\rangle\langle a\rangle[c]X$ is represented by the string `nu X.<a><b><c>true && <b><a>[c]X`.

Additionally, the parser supports `/* C-style comments */`, let-expressions and uninterpreted function calls. Comments allow to annotate parts of a formula with an explanation, which can make formulas easier to understand. Uninterpreted function calls are interpreted by the `call_expansion` module. This is further detailed in the next chapter.

An example of a let-expression is `let X = true||false in X&&!X`. The interpretation of such a formula is syntactic substitution, i.e. the free variable X in $X\wedge\neg X$ is substituted with `true`$\vee$`false` to produce the formula (`true`$\vee$`false`)$\wedge\neg$(`true`$\vee$`false`). The original formula with a let-expression above is interpreted via this expansion. This addition does not change the expressive power of the modal μ-calculus, but allows to represent

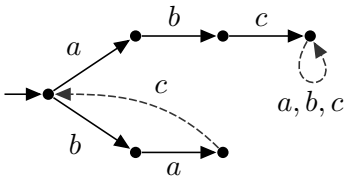


Figure 10.1.: Our running example for an mts.

the solution of vectorial fixed points more succinctly, which means that the formulas produced by the translation from mts into the ν-calculus (see Section 7.3.1) can be represented more efficiently.

10.2. Translating Modal Transition Systems into the μ-Calculus

The `mts_to_formula`-module that was implemented is based on the translation introduced in Section 7.3.1, but extended to also support non-deterministic mts. Due to this extension, the resulting formula may not be expressible in the conjunctive ν-calculus, so the μ-calculus is used.

The APT file format is only specified for lts. However, it allows to specify arbitrary *extensions* on edges of an lts and this is used to mark may edges. Specifically, an edge from `s1` to `s2` via `a` in an lts is specified as `s1 a s2`. As-is, the edge would be interpreted as a must edge together with an underlying may edge, as required by the definition of mts. To turn this into a may edge, the string `[may]` is appended.

As an example, consider the mts from Figure 10.1. It is a deterministic mts and it was translated at the end of Section 9.1 into the formula[1] $\nu X.([c]\texttt{false} \wedge \langle a\rangle([a]\texttt{false} \wedge [c]\texttt{false} \wedge \langle b\rangle([a]\texttt{false} \wedge [b]\texttt{false} \wedge \langle c\rangle\texttt{true})) \wedge \langle b\rangle([b]\texttt{false} \wedge [c]\texttt{false} \wedge \langle a\rangle([a]\texttt{false} \wedge [b]\texttt{false} \wedge [c]X)))$ of the modal μ-calculus. Doing the same translation with the module `mts_to_formula` results in the following formula[2]:

```
(let cse0 = [c]false in ((let cse1 = [b]false in (let cse2 = [a]false in
(let cse3 = (cse2&&cse1) in (let cse4 = <a>((cse2&&<b>(cse3&&<c>(
nu s5.(([a]s5&&[b]s5)&&[c]s5))))&&cse0) in (cse4&&<b>(nu s2.((<a>(cse3&&
[c]((cse4&&<b>s2)&&cse0))&&cse1)&&cse0)))))))&&cse0))
```

It can be seen that the formula produced by the module is a lot larger than the formula from Section 9.1. This is because the equation system that was constructed internally

[1] The actual translation in Section 9.1 used $\not\rightarrow^{a,b}$ as an abbreviation.

[2] `cse` stands for common subexpression elimination, which is used to introduce let-expressions.

was solved in a non-optimal order. Another reason is that the implementation cannot derive that the subformula[3] `nu s5.(([a]s5&&[b]s5)&&[c]s5)` is equivalent to `true`.

The translation from Section 7.3.1 only holds if the lts under consideration is deterministic. For non-deterministic lts, the original mts and its translation into a formula are not necessarily equivalent.

This module can also produce generic translations, i.e. translations that are also equivalent for non-deterministic lts. For the present example, this results in an even larger formula, because every must edge $s \xrightarrow{a} s'$ has to be translated into $[a]X_{s'} \wedge \langle a \rangle X_{s'}$ while in the deterministic case $\langle a \rangle X_{s'}$ is enough since only one outgoing edge with label a can exist. This equivalent formula is:

```
(let cse0 = [c]false in ((let cse1 = [b]false in (let cse2 = [a]false in
(let cse3 = (cse2&&cse1) in (let cse4 = (let cse7 = ((cse2&&(let cse8 =
(cse3&&(let cse9 = (nu s5.(([a]s5&&[b]s5)&&[c]s5)) in (<c>cse9&&[c]cse9)
)) in (<b>cse8&&[b]cse8)))&&cse0) in (<a>cse7&&[a]cse7)) in (cse4&&(let
cse5 = (nu s2.(((let cse6 = (cse3&&[c]((cse4&&(<b>s2&&[b]s2))&&cse0)) in
(<a>cse6&&[a]cse6))&&cse1)&&cse0)) in (<b>cse5&&[b]cse5)))))))&&cse0))
```

Here, a subformula equivalent to `true` is generated again. This time it is bound to the variable `cse9`, which only appears in the expression `<c>cse9&&[c]cse9`. This expression must be the translation of the c-labelled must edge reaching state `s5`.

It can be seen that the resulting formulas are very complicated, but also that let-expressions avoid the repetition of some large parts of the translation.

10.3. Model Checking

The local model checking that was presented in Section 9.2 was implemented in the `model_check` module. This module receives an lts and a formula as input and decides if the initial state of the lts satisfies the formula. To do so it generates all possible tableaux that relate the two, and checks if any of them are successful. These tableaux can also be outputted into a file in the DOT file format [GN00]. Tools for working with this file format and e.g. to visualise a DOT file in an image are available at `http://www.graphviz.org/`.

For efficiency reasons, it is possible to generate only successful tableaux instead of all tableaux. This allows the implementation to discard proof trees with unsuccessful leaves early before the full tableau is computed.

[3] Note that the variable name `s5` refers to the state s_5 of the input mts.

10.4. Finding Petri Net Realisations

The module `realise_pn` realises a formula of the modal μ-calculus with a Petri net via the algorithm that was presented in Chapter 9. This module gets two arguments, the subclass of Petri nets that should be targeted and a formula that should be realised. Its output is a list of realising Petri nets represented by their reachability graphs. Internally this uses the implementation of the minimal over-approximation that was presented in Chapter 6, as well as the local model checking that is also used for the `model_check` module.

So far the possible optimisations that were mentioned in Section 9.6 are not implemented. This means that the algorithm spends a lot of time in computing minimal over-approximations and its performance could likely be improved, for example, by over-approximating less often, or by trying to re-use regions that were previously computed for the next over-approximation. To optimise memory usage, only leaves of proof trees are saved and internal nodes are discarded.

The module `deadlock_free_realise_pn` also realises a formula with a Petri net. However, another requirement is that in each and every reachable marking, some transition is enabled, i.e. there are no deadlocks. Deadlocks can also be forbidden with a formula of the modal μ-calculus, but this module uses a more efficient approach. This module is necessary for the case study that is presented in the next chapter and is explained in Sections 11.3.4 and 11.3.5.

11. Case Study: Dining Philosophers

The previous chapters presented an algorithm for realising formulas via Petri nets and an implementation of this algorithm. Realising the modal μ-calculus with Petri nets was not analysed before. This chapter will explore a possible application of this algorithm.

Dijkstra's dining philosophers [Dij71] are a well-known distributed synchronisation problem with the possibility of deadlock. In this problem, there are a number of philosophers sitting at a round table. In front of each philosopher, there is a plate with an infinite supply of spaghetti. To eat spaghetti, a philosopher needs two forks. However, there is just one fork between every two neighbouring philosophers. This means that any single philosopher has to share a fork with his left neighbour and another fork with his right neighbour. Each philosopher is thinking for some individual amount of time until he becomes hungry. Then he grabs each of the two forks to eat spaghetti. When he is no longer hungry, he puts the forks back on the table and starts thinking again. The situation is illustrated in Figure 11.1.

This model can produce a deadlock. For example, if each philosopher grabs his left fork at the same time, then no progress is possible. Each philosopher needs his right fork to continue, but no philosopher has a right fork available. Another problem that can occur is starvation, which means that some philosopher would like to eat, but does not get a chance to do so. One possibility for this is that this philosopher is too slow, which means that his neighbours always grab the forks first whenever they become available.

In this chapter, the goal is to synthesise a Petri net model of the dining philosophers that does not have deadlocks nor starvation. This will be done without dictating how these goals are achieved so that a suitable algorithm has to be derived by the synthesis procedure.

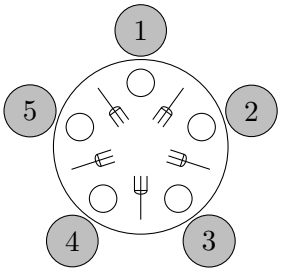


Figure 11.1.: Five dining philosophers at a table.

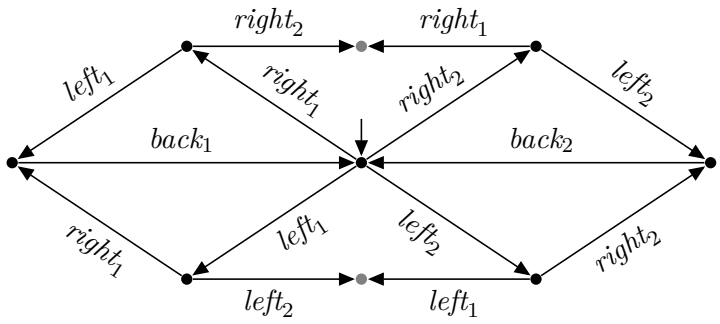


Figure 11.2.: Two philosophers modelled as an lts. Deadlock states are shown in grey.

For our model, we assume that there are n philosophers. The three possible actions of the i-th philosopher are represented by events: The philosopher can grab his left fork, which we model with the event $left_i$, he can grab his right fork modelled by $right_i$, and he can put both forks back on the table by executing the event $back_i$. The philosophers are sitting in clockwise ascending order, e.g. philosopher 4 is to the left of philosopher 3.

11.1. Modelling as a Labelled Transition System

We begin by examining how this problem can be modelled as an lts. The lts in Figure 11.2 visualises the behaviour of $n = 2$ philosophers. We can see that initially either philosopher can grab either fork. When only one fork is remaining, only the corresponding events stay enabled. When both forks are in use, there is either a deadlock, or there is one philosopher who can eat and then put the forks back on the table.

There are two deadlocks in this system. They are shown in grey in Figure 11.2. One is reached when both philosophers grab their right fork and the other deadlock occurs when they each grab their left fork. Removing both of these states results in an lts without deadlocks.

Starvation is harder to handle. In the lts as it is specified, it is possible that just one philosopher eats all the time, for example, by doing the sequence $left_1\, right_1\, back_1$ in a loop. If we wanted to remove this starvation sequence, the result would be that the philosophers take turns in eating, i.e. after philosopher 1 has eaten, philosopher 2 must eat next. Specifying this in the lts would require to split the lts from Figure 11.2 into two parts, one part for each philosopher, by removing the behaviour of the other philosopher. These parts could then be connected appropriately, e.g. when philosopher 1 finished eating, the corresponding edge would be redirected to the initial state of the other lts instead, because philosopher 2 may now eat.

Clearly, this is non-trivial. If we wanted to do the same task for $n > 2$ philosophers, more work would be necessary and we would not be able to easily reuse solutions from n philo-

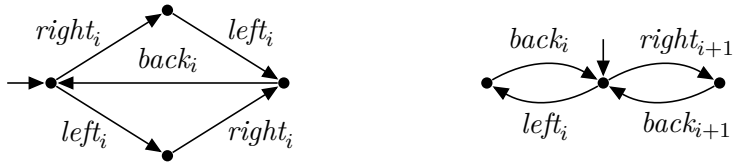


Figure 11.3.: A model of a single philosopher (left) and a model of a single fork (right).

sophers for $n+1$ philosophers. Moreover, with n philosophers, the lts has at least $4^{\lfloor \frac{n}{2} \rfloor}$ reachable states[1]. So, our approach does not scale to larger number of philosophers.

11.2. Modelling as a Distributed System in the Modal μ-calculus

In this section we model the dining philosophers with a modal specification, specifically a formula of the modal μ-calculus. At first, modalities do not make this problem any easier than the previous approach for an lts as specification. However, the modal μ-calculus has a conjunction operator that allows us to combine multiple partial specifications. Thus, we will exploit the distributed nature of the system and create a separate specification for each philosopher and each fork. These specifications are then later combined into the final specification.

The operation of a single philosopher is only loosely influenced by the actions of other philosophers. On a large table, the actions of philosophers that are at opposite sides of the table have no immediate consequences for each other. The only information that a single philosopher needs is the availability of his left and right fork. In fact, a single philosopher has just the four states that are visualised in the lts on the left side of Figure 11.3: Initially a philosopher is thinking. If he becomes hungry, he grabs his two neighbouring forks in arbitrary order. When he has both forks he can start eating, and, after eating, he puts them back and starts thinking again. The lts is closely related to the informal specification of the problem from the beginning of this chapter, while the lts from Figure 11.2 is not.

Figure 11.3 also shows the model of a single fork: Either the philosopher i grabs this fork as his left fork, or the philosopher $i+1$ grabs this fork as his right fork. In both cases the fork becomes unavailable until the philosopher puts it back. Here, $i+1$ is understood as wrapping around, i.e. the philosopher to the left of fork n is the first philosopher and $right_{n+1}$ is interpreted as $right_1$.

These models only consider the local events and ignore non-local events. For a single philosopher these are the actions that he executes on his neighbouring forks, while for

[1] Each philosopher has four states: thinking, eating, and having only one of the two forks. When only allowing every second philosopher to act, the given number of states is reached.

a single fork these are the actions done by the neighbouring philosophers. Therefore, an event $right_i$ will change the local state of the i-th philosopher so that he knows that he has another fork, but a far away event like $right_{i+100}$ does not change his state. The actions on the opposite side of the table do not matter for the local state. Thus, problems of arbitrary size can easily be specified by combining the specifications of n philosophers and forks conjunctively.

If we wanted to add the event $right_{i+100}$ to the specification for the i-th philosopher from Figure 11.3, the event should not modify the state, so edges with this event form a loop at every state. The philosopher i does not influence permissibility of the event, so this corresponds to a may edge in an mts. So formally we interpret Figure 11.3 as an mts, amended so that each non-local event forms a may loop around every state, i.e. does not lead to another state. In [BFLV16], this operation is called an *alphabet extension*. This intuitive description on how non-local events are handled and how they are locally ignored will be formalised next.

11.2.1. The Hiding Operator

We develop a function *hide* that modifies a specification as outlined above: Non-local events are ignored. The function will be defined for the modal μ-calculus since it is the most expressive specification language that we consider. The function will translate a given formula β into another formula $hide(\beta)$.

To do this, we first have to define which events are local. In the example from the previous section, the local events of the i-th philosopher were $\{right_i, left_i, back_i\}$, and the i-th fork had $\{left_i, back_i, right_{i+1}, back_{i+1}\}$ as its local events. These are the events that appear in Figure 11.3. For an arbitrary formula, we will use the events appearing in its modalities as its local alphabet.

Definition 11.2.1 (Local alphabet)**.** *The* local alphabet $\Sigma(\beta)$ *of a formula* β *of the modal* μ*-calculus is defined recursively via syntactic induction as follows, where* $X \in \mathsf{Var}$ *is an arbitrary variable,* $a \in \Sigma$ *is an event, and* β_1 *and* β_2 *are subformulas.*

- $\Sigma(\texttt{true}) = \Sigma(\texttt{false}) = \Sigma(X) = \emptyset$,
- $\Sigma(\nu X.\beta_1) = \Sigma(\mu X.\beta_1) = \Sigma(\beta_1)$,
- $\Sigma(\beta_1 \wedge \beta_2) = \Sigma(\beta_1 \vee \beta_2) = \Sigma(\beta_1) \cup \Sigma(\beta_2)$, *and*
- $\Sigma(\langle a \rangle \beta_1) = \Sigma([a]\beta_1) = \Sigma(\beta_1) \cup \{a\}$.

Next, formulas have to be translated. In the modal μ-calculus, $\langle a \rangle \beta$ expresses that an edge with event a must be possible next. We want $hide(\langle a \rangle \beta)$ to express that a is not only possible as the next event, but that it must also be possible after some sequence of non-local events. Here, the existence of one such sequence is enough since this is an

existential modality, but it is not allowed for a to never happen, for example, because the rest of the system does infinitely many steps.

Analogously, the universal modality $[a]\beta$ expresses that whenever a happens, afterwards β holds. If a does not happen, then this is fine as well. Thus, by duality, $\mathit{hide}([a]\beta)$ has to express that after all sequences of non-local events, if afterwards a is enabled, then β holds next. An infinite sequence of non-local events is allowed, too.

These ideas can be represented via fixed points as follows, where $\sigma = \Sigma \setminus \Sigma(\beta)$ is the set of non-local events:

Definition 11.2.2 (Hiding operator)**.** *Given a formula β of the modal μ-calculus with alphabet Σ, the* hiding operator *$\mathit{hide}(\beta)$ is defined as $\mathit{hide}(\beta) = h(\beta, \Sigma \setminus \Sigma(\beta))$, where $h(\beta, \sigma)$ is defined inductively as follows: The base cases are $h(\mathtt{true}, \sigma) = \mathtt{true}$, $h(\mathtt{false}, \sigma) = \mathtt{false}$, and $h(X, \sigma) = X$ for a variable $X \in \mathsf{Var}$. Inductively, define $h(\nu X.\beta, \sigma) = \nu X.h(\beta, \sigma)$, $h(\mu X.\beta, \sigma) = \mu X.h(\beta, \sigma)$, $h(\beta_1 \wedge \beta_2, \sigma) = h(\beta_1, \sigma) \wedge h(\beta_2, \sigma)$, and $h(\beta_1 \vee \beta_2, \sigma) = h(\beta_1, \sigma) \vee h(\beta_2, \sigma)$. Modalities introduce a fresh variable $X \in \mathsf{Var}$ as follows:*

$$h([a]\beta, \sigma) = \nu X.[a]h(\beta, \sigma) \wedge \bigwedge_{b \in \sigma} [b]X$$
$$h(\langle a \rangle \beta, \sigma) = \mu X.\langle a \rangle h(\beta, \sigma) \vee \bigvee_{b \in \sigma} \langle b \rangle X$$

It is worth noting that this definition is compatible with negation, which was defined as an abbreviation in the modal μ-calculus. Negation was introduced in Definition 7.2.5 via a number of dualities. It can easily be verified[2] that $\mathit{hide}(\neg\beta) = \neg\mathit{hide}(\beta)$, which means that the hiding operator is compatible with these dualities. In particular, this means that $\neg\mathit{hide}([a]\beta) = \mathit{hide}(\langle a \rangle \neg\beta)$.

As an example of this definition, consider the formula $\Phi = \langle a \rangle [a]\mathtt{false}$, which expresses (in a deterministic system) that the event a is possible exactly once. We construct $\mathit{hide}(\Phi)$ for the alphabet $\Sigma = \{a, b\}$. The local alphabet of Φ is $\Sigma(\Phi) = \{a\}$, thus $\mathit{hide}(\Phi)$ evaluates to $h(\Phi, \{b\})$. Next, the existential modality $\langle a \rangle \beta$ is replaced via $h(\langle a \rangle \beta, \sigma) = \mu X.\langle a \rangle h(\beta, \sigma) \vee \langle b \rangle X$. In the remaining formula, $\beta = [a]\mathtt{false}$ is replaced via $h([a]\mathtt{false}, \sigma) = \nu Y.[a]\mathtt{false} \wedge [b]Y$. Altogether we arrive at:

$$\mathit{hide}(\langle a \rangle [a]\mathtt{false}) = \mu X.\langle a \rangle (\nu Y.[a]\mathtt{false} \wedge [b]Y) \vee \langle b \rangle X$$

Informally, this formula expresses that the event a can occur at most once while event b is ignored. More formally, the outer fixed point expresses that there is a sequence of b's, after which a is possible and the inner fixed point holds. The inner fixed point forbids a along any sequence of b's.

[2] To actually prove this, formulas have to be considered up to renaming of bound variables, i.e. $\nu X.[a]X$ and $\nu Y.[a]Y$ are equivalent.

11.2.2. Concurrent Dining Philosophers in the Modal μ-Calculus

Figure 11.3 provided a specification for the behaviour of a single philosopher and a single fork. We can now understand this behaviour as an mts where every edge is a may edge, because this only specifies the allowed edges between states of the system, but is not meant to require these edges to be present. These mts can be translated into the modal μ-calculus via the construction from Section 7.3. This produces formulas $phil_i$ for specifying the behaviour of the i-th philosopher and $fork_i$ for specifying the behaviour of the i-th fork. The abbreviation $\not\to^{a,b}$ is used to express $[a]\texttt{false} \wedge [b]\texttt{false}$.

$$\begin{aligned} phil_i = \nu X. &\not\to^{back_i} \wedge [right_i](\not\to^{right_i, back_i} \wedge [left_i](\not\to^{left_i, right_i} \wedge [back_i]X)) \\ &\wedge [left_i](\not\to^{left_i, back_i} \wedge [right_i](\not\to^{left_i, right_i} \wedge [back_i]X)) \\ fork_i = \nu X. &\not\to^{back_i, back_{i+1}} \wedge [left_i](\not\to^{back_{i+1}, left_i, right_{i+1}} \wedge [back_i]X) \\ &\wedge [right_{i+1}](\not\to^{back_i, left_i, right_{i+1}} \wedge [back_{i+1}]X) \end{aligned}$$

Thus, the two philosophers that were specified in Figure 11.2 can alternatively be specified as $hide(phil_1) \wedge hide(phil_2) \wedge hide(fork_1) \wedge hide(fork_2)$. This specification can easily be extended to more philosophers, in contrast to the lts that was considered previously.

An additional requirement is deadlock-freedom. For this we use the formula Global(β) from Section 8.3.1, which expresses that β holds in all reachable states. This allows to express deadlock-freedom as $no\text{-}deadlock = \text{Global}(\bigvee_{a \in \Sigma} \langle a \rangle \texttt{true})$, i.e. in every reachable state there is at least one possible event.

Also, starvation should be forbidden. This can be done via the formula Eventually(a), which expresses that along every path, eventually $a \in \Sigma$ has to occur. This formula can be defined as $\text{Eventually}(a) = \mu X. \bigwedge_{b \in \Sigma \setminus \{a\}} [b]X$, i.e. there are no infinite[3] paths that do not contain a. This formula does not exclude starvation due to deadlocks, but this case was already handled via *no-deadlock*. Starvation-freedom for the i-th philosopher can now be expressed as $no\text{-}starvation_i = \text{Global}(\text{Eventually}(back_i))$, which means that in every state, eventually philosopher i will be done eating[4].

The formula Φ_n for modelling n dining philosophers without deadlock nor starvation combines all these individual parts:

$$\Phi_n = no\text{-}deadlock \wedge \bigwedge_{i=1}^{n} hide(phil_i) \wedge hide(fork_i) \wedge no\text{-}starvation_i$$

In Section 11.1, the behaviour of philosophers was specified as an lts. It was observed that this approach does not generalise to more philosophers. Also, the size of the lts grows exponentially in the number of philosophers. In contrast, the formula-based approach scales easily. The length of the formula grows linearly with the number of philosophers.

[3] This is forbidden by the least fixed point.

[4] This formula contains two fixed points that both follow most events. A more efficient, but less intuitive, representation would be $\nu G.\mu E.[back_i]G \wedge [left_i]E \wedge [right_i]E \wedge \bigwedge_{j \neq i}([left_j]E \wedge [right_j]E \wedge [back_j]E)$.

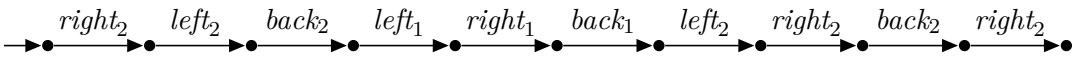


Figure 11.4.: An intermediate lts generated by the algorithm.

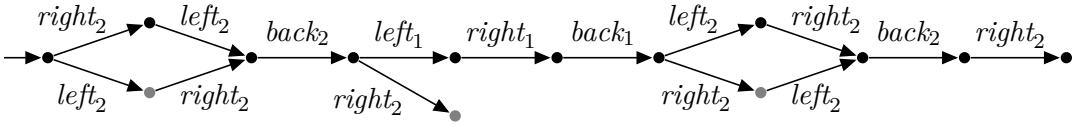


Figure 11.5.: The minimal over-approximation by the reachability graph of a 1-bounded Petri net of the lts from Figure 11.4. New states are shown in grey.

Because the size of the alphabet grows linearly with the number of philosophers, the full formula after expanding the *hide* function grows quadratically. Additionally, the formula only specifies that deadlocks and starvation are not allowed. It does not express how these goals shall be achieved and is thus more generic.

11.3. Finding Realisations

The formula Φ_2 can now be realised with a Petri net via the implementation that was presented in Chapter 10. Since this problem is only decidable for k-bounded Petri nets, we also have to provide a bound. We pick $k = 1$ for simplicity.

11.3.1. First Attempt at Finding Realisations

When trying to realise Φ_2 with a 1-bounded Petri net, the implementation quickly runs out of memory. Further examinations reveal that, for example[5], the lts from Figure 11.4 is generated. In this lts, first the second philosopher eats, then the first, and finally the second philosopher eats again. The first time he eats, the second philosopher grabs his right fork first. The second time he begins with his left fork. Clearly this lts is one of the many possibilities that are generated by the formula that forbids deadlocks, which means that this lts can be generated by repeatedly picking some event and using it to eliminate a deadlock state.

The minimal over-approximation by a 1-bounded Petri net reachability graph for this lts is shown in Figure 11.5. Because the forks were grabbed in both possible orders, the Petri net has to allow an arbitrary order of grabbing forks for the second philosopher. Specifically, this means that, for example, there is no 1-bounded Petri net place that does not allow the event $left_2$ in the initial marking and still has all the edges of the lts in its reachability graph.

[5]The exact problem depends on some non-deterministic choices in the implementation and so another lts could cause the same problem.

Next, the tableau that the algorithm constructed for the lts from Figure 11.4 has to be transferred to the lts from Figure 11.5. The formula from which the tableau is generated contains the term Global($\bigvee_{a\in\Sigma}\langle a\rangle\texttt{true}$) to express that there are no deadlocks. This formula expands to $\nu X.\bigvee_{a\in\Sigma}\langle a\rangle\texttt{true}\wedge\bigwedge_{a\in\Sigma}[a]X$, i.e. the full formula holds in a state if the inner disjunction holds and each successor fulfils the full formula. Because the formula has to hold in each successor, the newly added states in the lts from Figure 11.5 have to fulfil this as well. Also, recursively, each of their successors has to fulfil this formula. This means that even though the sequences $right_2 left_2$ and $left_2 right_2$ reach the same state, the inner formula is evaluated twice in the reached state, because the tableau method only considers the specific path that was used to reach a state and not other branches in the proof tree.

Thus, the inner disjunction is evaluated on all following states again. This disjunction contains one disjunct per event in the alphabet, thus the algorithm creates six branches for each following state. This happens eleven times, once for each reachable state, so there are $|\Sigma|^{11} = 6^{11} \approx 2^{28}$ tableaux[6] that need to be continued next.

Clearly, this lts results in too many possibilities, and it is just one specific situation where such a state explosion occurs. It is very likely that similar state explosions happen with different lts. Thus, the program will not finish its computation in a sensible time, nor within a realistic amount of space.

11.3.2. Evaluating Closed Formulas Only Once

The state space explosion in the previous section happens, because the inner part of Global($\bigvee_{a\in\Sigma}\langle a\rangle\texttt{true}$), that forbids deadlocks, is evaluated multiple times in the same state. This occurs because the state is reachable via different paths. However, the inner formula only requires some event to be enabled. This condition does not depend on the path that reached the current state. In fact, in the tableau method, only the evaluation of constant symbols is path dependent, because it ensures that a constant symbol is only expanded once per state.

This insight was used to change the implementation to avoid state explosion due to multiple paths reaching the same state. A tableau now also tracks for each state the closed formulas, which means formulas without constant symbols, that were already expanded in the state. This is used to compute the subtree for such a closed formula only once. If this does not result in any successful tableaux, the whole tableau is discarded. Thus, when a closed formula is seen for the second time in a state, it can simply be assumed to hold.

[6] There are actually more tableaux than this, since the last three states are reached via a total of four different paths, thus multiplying this number by $6^{3\cdot 3}$ and the inner branch that ends early in a deadlock adds another 6 possibilities.

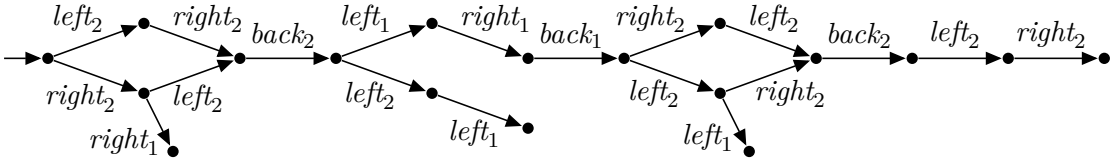


Figure 11.6.: Another intermediate lts generated by the algorithm.

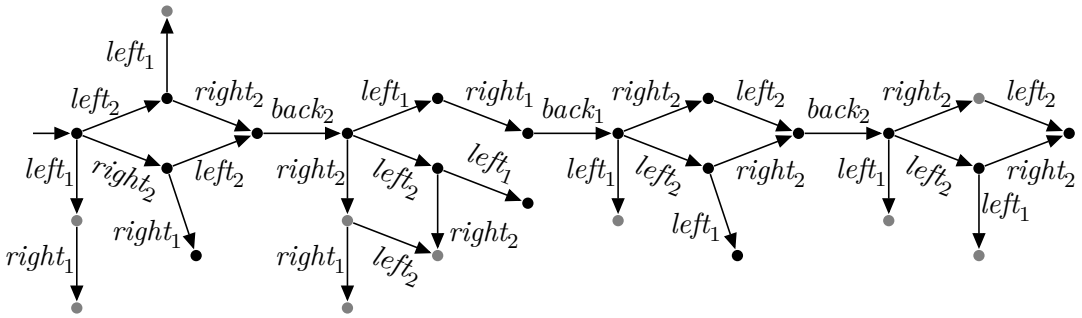


Figure 11.7.: The minimal over-approximation by the reachability graph of a 1-bounded Petri net of the lts from Figure 11.6. New states are shown in grey.

11.3.3. Second Attempt at Finding Realisations

Even with this improvement, the algorithm will still run out of memory. An example for this problem is shown in Figure 11.6. This lts is generated iteratively by the algorithm to realise the given formula. The next step is to minimally over-approximate this lts with the reachability graph of a Petri net. The resulting lts is shown in Figure 11.7. The over-approximation added ten new states—shown in gray—to the lts where the formula *no-deadlocks* has to be evaluated. This results in $|\Sigma|^{10} = 6^{10} \approx 2^{26}$ possible tableaux. Even when ignoring states which are not a deadlock, which the implementation does not do, there are seven states to consider, which leads to $6^7 \approx 2^{18}$ tableaux. Because the formula *no-deadlock* is also evaluated in states which already have an outgoing edge, in reality also non-existing deadlocks are eliminated.

Thus, while evaluating closed formulas only once per state helps, it is not enough to find realisations. The problem is that the algorithm works greedily: All deadlocks are eliminated independently and at the same time. A good approach to handle this problem would be to make the algorithm less greedy, i.e. to eliminate one deadlock, check the repercussions of the newly added edges, and then continue with other deadlocks. For the tableau method, this means that only one former leaf of the tableau should be continued in each iteration instead of all leaves at the same time. Also, information from other parts of the tableau could be used so that no attempt is made to remove the deadlock by enabling $left_1$ when the formula $[left_1]\texttt{false}$ also has to hold in the current state.

11.3.4. Finding Deadlock-Free Realisations of Formulas

The optimisations proposed in the last section cannot easily be added to the current implementation. Since the state explosion problem stems from the branching caused by the disjunctions inside the formula *no-deadlock*, we take a different approach that works without this formula:

When $\Phi \wedge$ *no-deadlock* should be realised, i.e. a deadlock-free realisation of Φ is wanted, we can just realise Φ. The resulting lts may contain deadlocks. One such state with a deadlock is picked and the deadlock is eliminated by adding a new state that is reached from the deadlock state. This is done $|\Sigma|$ times, once for each event in the alphabet. Next, finding realisations is continued with this lts and the corresponding tableau. If a realisation of Φ is found that has no deadlocks, then this lts also satisfies $\Phi \wedge$ *no-deadlock*. If all branches fail to produce results, then we can be sure that no realisations without deadlocks exist. This approach avoids the state space explosion problem by only eliminating a single deadlock in each iteration. If an event is not allowed by the specification in the current state, this will be noticed immediately and this unhelpful branch of the computation is discarded.

11.3.5. Third Attempt at Finding Realisations

The modified algorithm was implemented in the module `deadlock_free_realise_pn`. It works in the same way as `realise_pn`, but when a realisation containing a deadlock is found, this deadlock is eliminated as explained above and the search for realisations continues. The modified algorithm can now easily handle the given formula and finds 32 realisations in about 20 seconds.

Since the problem is symmetric in the sense that all left and right events can be swapped with each other, and the identities of philosophers can be swapped, the input formula was extended with the conjunct $\langle \mathit{left}_1 \rangle \texttt{true}$, which enforces that the first philosopher has to begin by grabbing his left fork. This addition eliminates some solutions that are identical up to the renaming that was just explained.

The implementation now finds six minimal realisations[7] of the formula in six seconds. Each realisation consists in the philosophers taking turns in eating, with the first philosopher beginning. The differences are in whether the philosophers have to begin grabbing their left or right fork, or if they can grab their forks in arbitrary order. These are three possibilities per philosopher, so nine possibilities in total. However, the additional formula $\langle \mathit{left}_1 \rangle \texttt{true}$ means that the first philosopher cannot begin by grabbing his right fork, so only six realisations remain.

Three philosophers still cannot be handled within a reasonable amount of memory.

[7] Actually, eight realisations are found, but three realisations are isomorphic to each other.

11.4. Conclusion

In this chapter we modelled the dining philosophers problem in the modal μ-calculus. To exploit the distributed nature of this problem, a generic hiding operator was defined. This operator can be used to model parts of a system while ignoring other parts. We used this operator to model the behaviour of individual philosophers and the forks between them. These individual specifications were then combined conjunctively to produce the final specification. We also modelled the problem as an lts. While the lts grows exponentially in the number of philosophers, the formula only has quadratic growth.

The complete specification for two dining philosopher was then given to the implementation that was presented in the previous chapter. The task was to find 1-bounded Petri net realisations. However, the formula that should forbid deadlocks caused a state space explosion problem, and the program ran out of available computer memory. To overcome this problem, several improvements were proposed and some of them implemented:

The evaluation of a closed formula does not depend on its context. Thus, the implementation was changed to evaluate such a formula only once per state. Still, the greedy approach for handling all deadlocks at once caused problems. One approach to overcome this could be to make the algorithm less greedy, which means that instead of calculating complete tableaux in each step, only one branch of the proof tree could be computed. While this approach sounds promising, it is not easily implementable in the existing program. Thus, a different approach was followed that works specifically for finding realisations without deadlocks and works by eliminating one deadlock at a time instead of all of them at once.

More optimisations are possible for the algorithm. While the implementation already defers handling of disjunctions so that the two branches created by a disjunction do not compute the same, unrelated subtree, more optimisations in the construction of proof trees should be possible. Another idea is to integrate minimal over-approximations more closely into the algorithm. Right now it is only used as a subroutine. In Chapter 5, an algorithm for computing this minimal over-approximation was presented that applied an expansion operator until a fixed point is reached. This expansion operator was based on the set of unsolvable separation problems. If the specification requires some event to be enabled in a given state, then the corresponding event/state separation problem could just be considered to be unsolvable. This saves the time needed to check the solvability of the separation problem. Also, the current implementation only saves an lts. Regions that were found for a previous lts could possibly be transferred to later lts, which would again save some computations. Finally, a single over-approximation step will likely still require more time than a single tableau-based extension step. Thus, it might be worthwhile to add multiple tableau-based extensions before a single over-approximation is performed. Alternatively, instead of computing the full over-approximation, it is also possible to only do a single step in the computation of the over-approximation. This would interleave the over-approximation algorithm with the algorithm for finding realisations.

All of these proposed heuristics only improve the computing time of the algorithm. However, it was seen that memory usage is actually the limiting factor, at least for this case study. Thus, most of these optimisations were not pursued further.

With the implemented improvements it was possible to realise the dining philosopher's problem for two philosophers with a 1-bounded Petri net, while avoiding deadlocks and starvation. However, the implementation still requires too much memory for three philosophers, while Petri net synthesis from lts is possible for this problem. Still, this case study shows some of the possibilities of using modal specifications for Petri net synthesis, especially for distributed and concurrent systems.

12. Conclusion

In this thesis, Petri net synthesis for subclasses of nets and for modal specifications was studied.

In the first part, we have introduced a generic algorithm that supports targeting Petri net synthesis into a given combination of subclasses, such as plain and pure Petri nets. To deal with unsolvable inputs, an algorithm for minimal over-approximation was introduced. This algorithm works for some of the subclasses that were previously handled, while for others the minimal over-approximation is not necessarily a finite lts. The over-approximation is based on the synthesis algorithm, which fails if some separation problems are unsolvable. The information about unsolvable separation problems is used to merge states and add new outgoing edges to the current lts. These algorithms were implemented in the tool APT.

The second part of the document investigated synthesis from modal specifications, namely modal transition systems, the modal μ-calculus and a subset of the μ-calculus, which is called the conjunctive ν-calculus. The ν-calculus and modal transition systems are equally expressive and we have shown via a reduction from two-counter machines for both specification languages that Petri net synthesis is undecidable. Next, we introduced an algorithm for synthesising k-bounded Petri nets from the full modal μ-calculus, where $k \in \mathbb{N}$ is given *a priori*, showing that this restriction makes the problem decidable even for the more expressive modal μ-calculus. The algorithm extends its current lts by the behaviour required by the given specification. To ensure solvability by a Petri net, the minimal over-approximation from the first part of this thesis is used. All subclasses supported by the minimal over-approximation can be combined with k-boundedness and are supported for synthesis from modal specifications. This synthesis algorithm was implemented as an extension to APT and used for a case study of the dining philosophers problem. The modal μ-calculus allowed to express this problem succinctly, but the implementation ran into state space explosion when realising the resulting formula.

To summarise, synthesising pure and unbounded Petri nets from modal specifications is undecidable [Feu05b]. We have shown that this problem stays undecidable for bounded Petri nets and for pure and bounded Petri nets, but becomes decidable for k-bounded Petri nets and combinations of k-boundedness and some other subclasses.

The relation of this document with the author's publications is as follows: Chapter 4 is based on [BS15; Sch16b], except for Section 4.1.2, which appeared in [SS17] and was created together with Valentin Spreckels. Chapter 5 is based on [Sch18], with most ideas about lts homomorphisms from [SW17] written together with Harro Wimmel.

Some parts of the description of the implementation in Chapter 6 were also published in [BS15], which was written together with Eike Best, but a substantial part of this chapter is new. Chapter 8 is loosely based on [Sch16a], but the result was lifted from bounded Petri nets to pure and bounded Petri nets. Similarly, Chapter 9 is loosely based on [SW17], which is joint work with Harro Wimmel, but the construction that was originally defined for disjunctive modal transition systems was reformulated for the more expressive modal μ-calculus. The description of the implementation of this algorithm in Chapter 10 and the case study in Chapter 11 were not published before.

The author also participated in the investigation of Petri net solvability of binary words, i.e. transition systems with an alphabet of size two and at most one outgoing edge in every state [BESW16]. The main result of [BBSS17] is a list of necessary conditions for an lts to be solvable by plain, pure, and 1-bounded Petri nets. An efficient Petri net synthesis algorithm targeting the subclass of place-output-nonbranching Petri nets, also known as *choice-free*, is investigated in [BDS18]. In [BS17], incrementality of Petri net synthesis in a process discovery context was investigated. Its main contribution is an algorithm to efficiently incorporate new information, i.e. newly observed behaviour, into a synthesis algorithm without redoing all calculations from scratch. Solvability of separation problems was characterised geometrically in [BDS17; SW18]. This can hopefully be used to estimate the size of a minimal over-approximation, as mentioned in Section 5.6. Factorisation of lts to speed up Petri net synthesis was investigated in [DS18]. A result from this paper was used in Section 8.3.4 to encode two-counter machines.

There are still many open problems in the context of this thesis. We investigated Petri net solvability for subclasses of nets, but the complexity of this problem remains open. There are polynomial algorithms for general Petri net synthesis [BBD95], but for plain, pure, and 1-bounded Petri nets the problem is NP-complete [BBD97]. Thus, the exact complexity can vary wildly depending on the specific targeted subclass. Also, the complexity of both the minimal over-approximation and the realisation of modal specifications are unknown.

Another interesting problem is maximal under-approximation of lts. However, while a unique minimal over-approximation exists, the maximal under-approximation is not unique. Also, it is not clear how such under-approximations could be computed.

The approach for realising formulas of the modal μ-calculus can also be applied to other specification languages. It merely needs a mechanism to amend an lts so that it becomes an implementation. For example, monadic second order logic (mso) is more expressive than the μ-calculus [JW96] and allows to require equality of states. This could be handled by merging states, similarly to how states are merged in the minimal over-approximation. However, local model checking seems not to be investigated for mso, so a new approach for identifying missing edges is needed. Still, this thesis could serve as a base for finding Petri net realisations from monadic second order logic.

Bibliography

[Aal16] Wil M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016. ISBN: 978-3-662-49850-7. DOI: 10.1007/978-3-662-49851-4.

[AD93] André Arnold and Anne Dicky. 'Equivalences and Preorders of Transition Systems'. In: *MFCS 1993*. Ed. by Andrzej M. Borzyszkowski and Stefan Sokolowski. Vol. 711. LNCS. Springer, 1993, pp. 20–31. DOI: 10.1007/3-540-57182-5_2.

[AM98] N. Alon and T.H. Marshall. 'Homomorphisms of Edge-Colored Graphs and Coxeter Groups'. In: *Journal of Algebraic Combinatorics* 8.1 (1998), pp. 5–13. ISSN: 1572-9192. DOI: 10.1023/A:1008647514949.

[AN01] André Arnold and Damian Niwiński. *Rudiments of μ-calculus*. North Holland, 2001. ISBN: 978-0-444-50620-7.

[Ant+08] Adam Antonik, Michael Huth, Kim G. Larsen, Ulrik Nyman and Andrzej Wasowski. '20 Years of Modal and Mixed Specifications'. In: *Bulletin of the EATCS* 95 (2008), pp. 94–129.

[Arn94] André Arnold. *Finite transition systems - semantics of communicating systems*. Prentice Hall international series in computer science. Prentice Hall, 1994. ISBN: 978-0-13-092990-7.

[BBD15] Eric Badouel, Luca Bernardinello and Philippe Darondeau. *Petri Net Synthesis*. Texts in Theoretical Computer Science. Springer, 2015. ISBN: 978-3-662-47966-7. DOI: 10.1007/978-3-662-47967-4.

[BBD95] Eric Badouel, Luca Bernardinello and Philippe Darondeau. 'Polynomial Algorithms for the Synthesis of Bounded Nets'. In: *TAPSOFT'95*. Ed. by Peter D. Mosses, Mogens Nielsen and Michael I. Schwartzbach. Vol. 915. LNCS. Springer, 1995, pp. 364–378. DOI: 10.1007/3-540-59293-8_207.

[BBD97] Eric Badouel, Luca Bernardinello and Philippe Darondeau. 'The Synthesis Problem for Elementary Net Systems is NP-Complete'. In: *Theoretical Computer Science* 186.1-2 (1997), pp. 107–134. DOI: 10.1016/S0304-3975(96)00219-8.

[BBSS17] Kamila Barylska, Eike Best, Uli Schlachter and Valentin Spreckels. 'Properties of Plain, Pure, and Safe Petri Nets'. In: *Trans. Petri Nets and Other Models of Concurrency XII*. LNCS 10470 (2017). Ed. by Maciej Koutny, Jetty Kleijn and Wojciech Penczek, pp. 1–18. DOI: 10.1007/978-3-662-55862-1_1.

[BCD02] Eric Badouel, Benoît Caillaud and Philippe Darondeau. 'Distributing Finite Automata Through Petri Net Synthesis'. In: *Formal Aspects of Computing* 13.6 (2002), pp. 447–470. DOI: 10.1007/s001650200022.

[BD04] Eric Badouel and Philippe Darondeau. 'The synthesis of Petri nets from path-automatic specifications'. In: *Information and Computation* 193.2 (2004), pp. 117–135. DOI: 10.1016/j.ic.2004.04.004.

[BD11] Eike Best and Philippe Darondeau. 'Petri Net Distributability'. In: *PSI 2011*. Ed. by Edmund M. Clarke, Irina Virbitskaite and Andrei Voronkov. Vol. 7162. LNCS. Springer, 2011, pp. 1–18. DOI: 10.1007/978-3-642-29709-0_1.

[BD14] Eike Best and Raymond R. Devillers. 'Characterisation of the State Spaces of Live and Bounded Marked Graph Petri Nets'. In: *LATA 2014*. Ed. by Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez and Bianca Truthe. Vol. 8370. LNCS. Springer, 2014, pp. 161–172. DOI: 10.1007/978-3-319-04921-2_13.

[BD15a] Eike Best and Raymond R. Devillers. 'State space axioms for T-systems'. In: *Acta Inf.* 52.2-3 (2015), pp. 133–152. DOI: 10.1007/s00236-015-0219-0.

[BD15b] Eike Best and Raymond R. Devillers. 'Synthesis of Bounded Choice-Free Petri Nets'. In: *CONCUR 2015*. Ed. by Luca Aceto and David de Frutos-Escrig. Vol. 42. LIPIcs. Schloss Dagstuhl, 2015, pp. 128–141. DOI: 10.4230/LIPIcs.CONCUR.2015.128.

[BD96] Eric Badouel and Philippe Darondeau. 'Theory of Regions'. In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. LNCS. Springer, 1996, pp. 529–586. DOI: 10.1007/3-540-65306-6_22.

[BDS17] Eike Best, Raymond R. Devillers and Uli Schlachter. 'A Graph-Theoretical Characterisation of State Separation'. In: *SOFSEM 2017*. Ed. by Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey and Tiziana Margaria. Vol. 10139. LNCS. Springer, 2017, pp. 163–175. DOI: 10.1007/978-3-319-51963-0_13.

[BDS18] Eike Best, Raymond R. Devillers and Uli Schlachter. 'Bounded choice-free Petri net synthesis: algorithmic issues'. In: *Acta Inf.* 55.7 (2018), pp. 575–611. DOI: 10.1007/s00236-017-0310-9.

[Ben+13] Nikola Benes, Benoît Delahaye, Uli Fahrenberg, Jan Kretínský and Axel Legay. 'Hennessy-Milner Logic with Greatest Fixed Points as a Complete Behavioural Specification Theory'. In: *CONCUR 2013*. Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. LNCS. Springer, 2013, pp. 76–90. DOI: 10.1007/978-3-642-40184-8_7.

[BESW16] Eike Best, Evgeny Erofeev, Uli Schlachter and Harro Wimmel. 'Characterising Petri Net Solvable Binary Words'. In: *PETRI NETS 2016*. Ed. by Fabrice Kordon and Daniel Moldt. Vol. 9698. LNCS. Springer, 2016, pp. 39–58. DOI: 10.1007/978-3-319-39086-4_4.

[BFLV16] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen and Walter Vogler. 'Nondeterministic Modal Interfaces'. In: *Theoretical Computer Science* 642 (2016), pp. 24–53. DOI: 10.1016/j.tcs.2016.06.011.

[BKLS09] Nikola Benes, Jan Kretínský, Kim Larsen and Jirí Srba. 'On determinism in modal transition systems'. In: *Theoretical Computer Science* 410.41 (2009), pp. 4026–4043. DOI: 10.1016/j.tcs.2009.06.009.

[BL92] Gérard Boudol and Kim Guldstrand Larsen. 'Graphical Versus Logical Specifications'. In: *Theoretical Computer Science* 106.1 (1992), pp. 3–20. DOI: 10.1016/0304-3975(92)90276-L.

[Bor+13] Dennis Borde, Sören Dierkes, Raffaela Ferrari, Manuel Gieseking, Vincent Göbel, Renke Grunwald, Björn von der Linde, Daniel Lückehe, Uli Schlachter, Chris Schierholz, Maike Schwammberger and Valentin Spreckels. *Projektgruppe APT: Analyse von Petri-Netzen und Transitionssystemen*. Final report of a project assignment. Mar. 2013. URL: https://github.com/CvO-Theory/apt/blob/master/doc/APT.pdf.

[Bri16] Hadrien Bride. 'Verifying Modal Specifications of Workflow Nets: using Constraint Solving and Reduction Methods'. PhD thesis. University of Franche-Comté, Besançon, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01514168.

[Bru97] Glenn Bruns. 'An Industrial Application of Modal Process Logic'. In: *Science of Computer Programming* 29.1-2 (1997), pp. 3–22. DOI: 10.1016/S0167-6423(96)00027-5.

[BS15] Eike Best and Uli Schlachter. 'Analysis of Petri Nets and Transition Systems'. In: *ICE 2015*. Ed. by Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente and Hugo Torres Vieira. Vol. 189. EPTCS. 2015, pp. 53–67. DOI: 10.4204/EPTCS.189.6.

[BS17] Eric Badouel and Uli Schlachter. 'Incremental Process Discovery using Petri Net Synthesis'. In: *Fundam. Inform.* 154.1-4 (2017), pp. 1–13. DOI: 10.3233/FI-2017-1548.

[BST10] Clark Barrett, Aaron Stump and Cesare Tinelli. 'The SMT-LIB Standard: Version 2.0'. In: *SMT 2010*. Ed. by A. Gupta and D. Kroening. 2010.

[BW15] Julian Bradfield and Igor Walukiewicz. 'The mu-calculus and model-checking'. In: *Handbook of Model Checking*. Ed. by H. Veith E. Clarke T. Henzinger. Springer-Verlag, 2015. URL: http://www.labri.fr/perso/igw/Papers/igw-mu.pdf.

[Cai99] Benoît Caillaud. *Synet: A Synthesizer of Distributable Bounded Petri-Nets from Finite Automata*. 1999. URL: https://www.irisa.fr/s4/tools/synet/.

[Car+08] Josep Carmona, Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno and Alexandre Yakovlev. 'A Symbolic Algorithm for the Synthesis of Bounded Petri Nets'. In: *PETRI NETS 2008*. Ed. by Kees M. van Hee and Rüdiger Valk. Vol. 5062. LNCS. Springer, 2008, pp. 92–111. DOI: 10.1007/978-3-540-68746-7_10.

[Car12] Josep Carmona. 'The Label Splitting Problem'. In: *Trans. Petri Nets and Other Models of Concurrency*. LNCS 7400 (2012). Ed. by Kurt Jensen, Wil M. P. van der Aalst, Marco Ajmone Marsan, Giuliana Franceschinis, Jetty Kleijn and Lars Michael Kristensen, pp. 1–23. DOI: 10.1007/978-3-642-35179-2_1.

[CCK08] Josep Carmona, Jordi Cortadella and Michael Kishinevsky. 'A Region-Based Algorithm for Discovering Petri Nets from Event Logs'. In: *BPM 2008*. Ed. by Marlon Dumas, Manfred Reichert and Ming-Chien Shan. Vol. 5240. LNCS. Springer, 2008, pp. 358–373. DOI: 10.1007/978-3-540-85758-7_26.

[CCK09] Josep Carmona, Jordi Cortadella and Michael Kishinevsky. 'Genet: A Tool for the Synthesis and Mining of Petri Nets'. In: *ACSD 2009*. IEEE Computer Society, 2009, pp. 181–185. DOI: 10.1109/ACSD.2009.6.

[CCK10] Josep Carmona, Jordi Cortadella and Michael Kishinevsky. 'New Region-Based Algorithms for Deriving Bounded Petri Nets'. In: *IEEE Trans. Computers* 59.3 (2010), pp. 371–384. DOI: 10.1109/TC.2009.131.

[CGR11] Sjoerd Cranen, Jan Friso Groote and Michel A. Reniers. 'A linear translation from CTL* to the first-order modal μ -calculus'. In: *Theoretical Computer Science* 412.28 (2011), pp. 3129–3139. DOI: 10.1016/j.tcs.2011.02.034.

[CGS07] Maria Paola Cabasino, Alessandro Giua and Carla Seatzu. 'Identification of Petri Nets from Knowledge of Their Language'. In: *Discrete Event Dynamic Systems* 17.4 (2007), pp. 447–474. DOI: 10.1007/s10626-007-0025-0.

[CHN13] Jürgen Christ, Jochen Hoenicke and Alexander Nutz. 'Proof Tree Preserving Interpolation'. In: *TACAS 2013*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. LNCS. Springer, 2013, pp. 124–138. DOI: 10.1007/978-3-642-36742-7_9.

[CKLY95] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno and Alexandre Yakovlev. 'Synthesizing Petri nets from state-based models'. In: *ICCAD 1995*. Ed. by Richard L. Rudell. IEEE, 1995, pp. 164–171. DOI: 10.1109/ICCAD.1995.480008.

[CKLY98] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno and Alexandre Yakovlev. 'Deriving Petri Nets for Finite Transition Systems'. In: *IEEE Trans. Computers* 47.8 (1998), pp. 859–882. DOI: 10.1109/12.707587.

[Cor+97] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno and Alex Yakovlev. 'Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers'. In: *IEICE Trans. Inf. and Syst.* E80-D.3 (1997), pp. 315–325.

[Cor+99] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor and Alexandre Yakovlev. *Petrify: a tool for synthesis of Petri Nets and asynchronous circuits*. 1999. URL: http://www.cs.upc.edu/~jordicf/petrify/.

[Dar00] Philippe Darondeau. 'Region Based Synthesis of P/T-Nets and Its Potential Applications'. In: *PETRI NETS 2000*. Ed. by Mogens Nielsen and Dan Simpson. Vol. 1825. LNCS. Springer, 2000, pp. 16–23. DOI: 10.1007/3-540-44988-4_2.

[Dar01] Philippe Darondeau. 'On the Petri net realization of context-free graphs'. In: *Theoretical Computer Science* 258.1-2 (2001), pp. 573–598. DOI: 10.1016/S0304-3975(00)00162-6.

[Dar03] Philippe Darondeau. 'Unbounded Petri Net Synthesis'. In: *ACPN 2003*. Ed. by Jörg Desel, Wolfgang Reisig and Grzegorz Rozenberg. Vol. 3098. LNCS. Springer, 2003, pp. 413–438. DOI: 10.1007/978-3-540-27755-2_11.

[Dar05] Philippe Darondeau. 'Distributed implementations of Ramadge-Wonham supervisory control with Petri nets'. In: *IEEE CDC-ECC 2005*. Ed. by Eduardo Camacho. IEEE, 2005, pp. 2107–2112. DOI: 10.1109/CDC.2005.1582472.

[Dar98] Philippe Darondeau. 'Deriving Unbounded Petri Nets from Formal Languages'. In: *CONCUR '98*. Ed. by Davide Sangiorgi and Robert de Simone. Vol. 1466. LNCS. Springer, 1998, pp. 533–548. DOI: 10.1007/BFb0055646.

[Dev18] Raymond R. Devillers. 'Factorisation of transition systems'. In: *Acta Inf.* 55.4 (2018), pp. 339–362. DOI: 10.1007/s00236-017-0300-y.

[Dij71] Edsger W. Dijkstra. 'Hierarchical Ordering of Sequential Processes'. In: *Acta Inf.* 1 (1971), pp. 115–138. DOI: 10.1007/BF00289519.

[DR96] Jörg Desel and Wolfgang Reisig. 'The Synthesis Problem of Petri Nets'. In: *Acta Inf.* 33.4 (1996), pp. 297–315. DOI: 10.1007/s002360050046.

[DS18] Raymond R. Devillers and Uli Schlachter. 'Factorisation of Petri Net Solvable Transition Systems'. In: *PETRI NETS 2018*. Ed. by Victor Khomenko and Olivier H. Roux. Vol. 10877. LNCS. Springer, 2018, pp. 82–98. DOI: 10.1007/978-3-319-91268-4_5.

[EHH12] Dorsaf Elhog-Benzina, Serge Haddad and Rolf Hennicker. 'Refinement and Asynchronous Composition of Modal Petri Nets'. In: *Trans. Petri Nets and Other Models of Concurrency V*. LNCS 6900 (2012). Ed. by Kurt Jensen, Susanna Donatelli and Jetty Kleijn, pp. 96–120. DOI: 10.1007/978-3-642-29072-5_4.

[ER90a] Andrzej Ehrenfeucht and Grzegorz Rozenberg. 'Partial (Set) 2-Structures. Part I: Basic Notions and the Representation Problem'. In: *Acta Inf.* 27.4 (1990), pp. 315–342. DOI: 10.1007/BF00264611.

[ER90b] Andrzej Ehrenfeucht and Grzegorz Rozenberg. 'Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems'. In: *Acta Inf.* 27.4 (1990), pp. 343–368. DOI: 10.1007/BF00264612.

[Ero18] Evgeny Erofeev. 'Characterisation of a Class of Petri Net Solvable Transition Systems'. PhD thesis. Carl von Ossietzky Universität Oldenburg, 2018.

[Esp94] Javier Esparza. 'On the Decidability of Model Checking for Several μ-calculi and Petri Nets'. In: *CAAP 1994*. Ed. by Sophie Tison. Vol. 787. LNCS. Springer, 1994, pp. 115–129. DOI: 10.1007/BFb0017477.

[Esp97] Javier Esparza. 'Decidability of Model Checking for Infinite-State Concurrent Systems'. In: *Acta Inf.* 34.2 (1997), pp. 85–107. DOI: 10.1007/s002360050074.

[EW17] Evgeny Erofeev and Harro Wimmel. 'Reachability Graphs of Two-Transition Petri Nets'. In: *ACSD 2017*. Ed. by Wil M. P. van der Aalst, Robin Bergenthum and Josep Carmona. Vol. 1847. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 39–54. URL: http://ceur-ws.org/Vol-1847/paper03.pdf.

[Fah+09] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer and Karsten Wolf. 'Instantaneous Soundness Checking of Industrial Business Process Models'. In: *BPM 2009*. Ed. by Umeshwar Dayal, Johann Eder, Jana Koehler and Hajo A. Reijers. Vol. 5701. LNCS. Springer, 2009, pp. 278–293. DOI: 10.1007/978-3-642-03848-8_19.

[Feu05a] Guillaume Feuillade. *Modal specifications are a syntactic fragment of the Mu-calculus*. Research Report RR-5612. INRIA, 2005, p. 17. URL: https://hal.inria.fr/inria-00070396.

[Feu05b] Guillaume Feuillade. 'Spécification logique de réseaux de Petri'. PhD thesis. Université de Rennes I, 2005. URL: http://www.irisa.fr/s4/download/papers/Feuillade-these2005.pdf.

[FLT14] Uli Fahrenberg, Axel Legay and Louis-Marie Traonouez. 'Structural Refinement for the Modal nu-Calculus'. In: *ICTAC 2014*. Ed. by Gabriel Ciobanu and Dominique Méry. Vol. 8687. LNCS. Springer, 2014, pp. 169–187. DOI: 10.1007/978-3-319-10882-7_11.

[FP07] Guillaume Feuillade and Sophie Pinchinat. 'Modal Specifications for the Control Theory of Discrete Event Systems'. In: *DEDS* 17.2 (2007), pp. 211–232. DOI: 10.1007/s10626-006-0008-6.

[GGS11] Rob J. van Glabbeek, Ursula Goltz and Jens-Wolfhard Schicke. 'On Causal Semantics of Petri Nets'. In: *CONCUR 2011*. Ed. by Joost-Pieter Katoen and Barbara König. Vol. 6901. LNCS. Springer, 2011, pp. 43–59. DOI: 10.1007/978-3-642-23217-6_4.

[GGS13] Rob J. van Glabbeek, Ursula Goltz and Jens-Wolfhard Schicke-Uffmann. 'On Characterising Distributability'. In: *Logical Methods in Computer Science* 9.3 (2013). DOI: 10.2168/LMCS-9(3:17)2013.

[GN00] Emden R. Gansner and Stephen C. North. 'An open graph visualization system and its applications to software engineering'. In: *Softw., Pract. Exper.* 30.11 (2000), pp. 1203–1233. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

[HDK15] Thomas Hujsa, Jean-Marc Delosme and Alix Munier Kordon. 'On the Reversibility of Live Equal-Conflict Petri Nets'. In: *PETRI NETS 2015*. Ed. by Raymond R. Devillers and Antti Valmari. Vol. 9115. LNCS. Springer, 2015, pp. 234–253. DOI: 10.1007/978-3-319-19488-2_12.

[HM80] Matthew Hennessy and Robin Milner. 'On Observing Nondeterminism and Concurrency'. In: *ICALP 1980*. Ed. by J. W. de Bakker and Jan van Leeuwen. Vol. 85. LNCS. Springer, 1980, pp. 299–309. DOI: 10.1007/3-540-10003-2_79.

[HM85] Matthew Hennessy and Robin Milner. 'Algebraic Laws for Nondeterminism and Concurrency'. In: *J. ACM* 32.1 (1985), pp. 137–161. DOI: 10.1145/2455.2460.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 1st. Reading, Mass: Addison-Wesley, 1979. ISBN: 0-201-02988-X.

[Jan01] Petr Jancar. 'Nonprimitive recursive complexity and undecidability for Petri net equivalences'. In: *Theoretical Computer Science* 256.1-2 (2001), pp. 23–30. DOI: 10.1016/S0304-3975(00)00100-6.

[JCL04] Li Jiao, To-Yat Cheung and Weiming Lu. 'On liveness and boundedness of asymmetric choice nets'. In: *Theoretical Computer Science* 311.1-3 (2004), pp. 165–197. DOI: 10.1016/S0304-3975(03)00359-1.

[JW96] David Janin and Igor Walukiewicz. 'On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic'. In: *CONCUR '96*. Ed. by Ugo Montanari and Vladimiro Sassone. Vol. 1119. LNCS. Springer, 1996, pp. 263–277. DOI: 10.1007/3-540-61604-7_60.

[Koz83] Dexter Kozen. 'Results on the Propositional mu-Calculus'. In: *Theoretical Computer Science* 27 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6.

[Kre17] Jan Kretínský. '30 Years of Modal Transition Systems: Survey of Extensions and Analysis'. In: *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay and Radu Mardare. Vol. 10460. LNCS. Springer, 2017, pp. 36–74. DOI: 10.1007/978-3-319-63121-9_3.

[Lar89] Kim Larsen. 'Modal Specifications'. In: *AVMFSS*. Ed. by Joseph Sifakis. Vol. 407. LNCS. Springer, 1989, pp. 232–246. DOI: 10.1007/3-540-52148-8_19.

[LMJ07] Robert Lorenz, Sebastian Mauser and Gabriel Juhás. 'How to synthesize nets from languages: a survey'. In: *WSC 2007*. Ed. by Shane G. Henderson, Bahar Biller, Ming-Hua Hsieh, John Shortle, Jeffrey D. Tew and Russell R. Barton. WSC, 2007, pp. 637–647. DOI: 10.1109/WSC.2007.4419657.

[LR78] Lawrence H. Landweber and Edward L. Robertson. 'Properties of Conflict-Free and Persistent Petri Nets'. In: *J. ACM* 25.3 (1978), pp. 352–364. DOI: 10.1145/322077.322079.

[LT88] Kim Larsen and Bent Thomsen. 'A Modal Process Logic'. In: *LICS 1988*. IEEE, 1988, pp. 203–210. DOI: 10.1109/LICS.1988.5119.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. 'Equation Solving Using Modal Transition Systems'. In: *LICS 1990*. IEEE Computer Society, 1990, pp. 108–117. DOI: 10.1109/LICS.1990.113738.

[Min67] Marvin Lee Minsky. *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967. ISBN: 0-13-165563-9.

[MSS99] Markus Müller-Olm, David A. Schmidt and Bernhard Steffen. 'Model-Checking: A Tutorial Introduction'. In: *SAS '99*. Ed. by Agostino Cortesi and Gilberto Filé. Vol. 1694. LNCS. Springer, 1999, pp. 330–354. DOI: 10.1007/3-540-48294-6_22.

[Muk92] Madhavan Mukund. 'Petri Nets and Step Transition Systems'. In: *International Journal of Foundations of Computer Science* 3.4 (1992), pp. 443–478. DOI: 10.1142/S0129054192000231.

[NRT92] Mogens Nielsen, Grzegorz Rozenberg and P. S. Thiagarajan. 'Elementary Transition Systems'. In: *Theoretical Computer Science* 96.1 (1992), pp. 3–33. DOI: 10.1016/0304-3975(92)90180-N.

[Pop94] Sally Popkorn. *First steps in modal logic*. Cambridge University Press, 1994, p. 314. ISBN: 0-521-46482-X.

[PR08] C. Adam Petri and W. Reisig. 'Petri net'. In: *Scholarpedia* 3.4 (2008). revision #91646, p. 6477. DOI: 10.4249/scholarpedia.6477.

[PWM03] J.W. Pinney, D.R. Westhead and G.A. McConkey. 'Petri Net representations in systems biology'. In: *Biochemical Society Transactions* 31.6 (2003), pp. 1513–1515. ISSN: 0300-5127. DOI: 10.1042/bst0311513.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*. Vol. 4. EATCS Monographs on Theoretical Computer Science. Springer, 1985. ISBN: 3-540-13723-8. DOI: 10.1007/978-3-642-69968-9.

[RTS97] Laura Recalde, Enrique Teruel and Manuel Silva. 'Improving the decision power of rank theorems'. In: *IEEE Trans. Syst. Man, Cybern. B, Cybern.* 4 (1997), pp. 3768–3773. ISSN: 1062-922X. DOI: 10.1109/ICSMC.1997.633256.

[Sat+15] Y. Sato, S. Masuda, Y. Someya, T. Tsujii and S. Watanabe. 'An fMRI analysis of the efficacy of Euler diagrams in logical reasoning'. In: *IEEE VL/HCC*. Ed. by Zhen Li, Claudia Ermel and Scott D. Fleming. IEEE, Oct. 2015, pp. 143–151. DOI: 10.1109/VLHCC.2015.7357209.

[Sch16a] Uli Schlachter. 'Bounded Petri Net Synthesis from Modal Transition Systems is Undecidable'. In: *CONCUR 2016*. Ed. by Josée Desharnais and Radha Jagadeesan. Vol. 59. LIPIcs. Schloss Dagstuhl, 2016, 15:1–15:14. DOI: 10.4230/LIPIcs.CONCUR.2016.15.

[Sch16b] Uli Schlachter. 'Petri Net Synthesis for Restricted Classes of Nets'. In: *PETRI NETS 2016*. Ed. by Fabrice Kordon and Daniel Moldt. Vol. 9698. LNCS. Springer, 2016, pp. 79–97. DOI: 10.1007/978-3-319-39086-4_6.

[Sch18] Uli Schlachter. 'Over-Approximative Petri Net Synthesis for Restricted Subclasses of Nets'. In: *LATA 2018*. Ed. by Shmuel Tomi Klein, Carlos Martín-Vide and Dana Shapira. Vol. 10792. LNCS. Springer, 2018, pp. 296–307. DOI: 10.1007/978-3-319-77313-1_23.

[SS17] Uli Schlachter and Valentin Spreckels. 'Synthesis of Labelled Transition Systems into Equal-Conflict Petri Nets'. In: *ATAED 2017*. Ed. by Wil M. P. van der Aalst, Robin Bergenthum and Josep Carmona. Vol. 1847. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 122–130. URL: http://ceur-ws.org/Vol-1847/paper09.pdf.

[SW17] Uli Schlachter and Harro Wimmel. 'k-Bounded Petri Net Synthesis from Modal Transition Systems'. In: *CONCUR 2017*. Ed. by Roland Meyer and Uwe Nestmann. Vol. 85. LIPIcs. Schloss Dagstuhl, 2017, 6:1–6:15. DOI: 10.4230/LIPIcs.CONCUR.2017.6.

[SW18] Uli Schlachter and Harro Wimmel. 'A Geometric Characterisation of Event/State Separation'. In: *PETRI NETS 2018*. Ed. by Victor Khomenko and Olivier H. Roux. Vol. 10877. LNCS. Springer, 2018, pp. 99–116. DOI: 10.1007/978-3-319-91268-4_6.

[SW89] Colin Stirling and David Walker. 'Local Model Checking in the Modal Mu-Calculus'. In: *TAPSOFT'89 (CAAP'89)*. Ed. by Josep Díaz and Fernando Orejas. Vol. 351. LNCS. Springer, 1989, pp. 369–383. DOI: 10.1007/3-540-50939-9_144.

[SW91] Colin Stirling and David Walker. 'Local Model Checking in the Modal mu-Calculus'. In: *Theoretical Computer Science* 89.1 (1991), pp. 161–177. DOI: 10.1016/0304-3975(90)90110-4.

[WDHS08] Jan Martijn E. M. van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens and Alexander Serebrenik. 'Process Discovery Using Integer Linear Programming'. In: *PETRI NETS 2008*. Ed. by Kees M. van Hee and Rüdiger Valk. Vol. 5062. LNCS. Springer, 2008, pp. 368–387. DOI: 10.1007/978-3-540-68746-7_24.

[Wol18] Karsten Wolf. 'Petri Net Synthesis with Union/Find'. In: *PETRI NETS 2018*. Ed. by Victor Khomenko and Olivier H. Roux. Vol. 10877. LNCS. Springer, 2018, pp. 60–81. DOI: 10.1007/978-3-319-91268-4_4.

Index