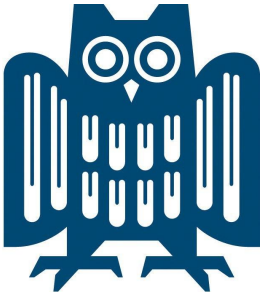
Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

Generating Concurrency-preserving Petri Games



submitted by
Sanny Alexander Schmitt

submitted on
January 17th, 2019

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Jesko Hecking-Harbusch, M.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr.-Ing. Holger Hermanns

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Petri games represent a multiplayer game model used to synthesize distributed systems. In these games, each token either belongs to the team of system players or to the team of environment players, and moves independently between the places of an underlying Petri net. While doing so, each player collects local information in form of his causal past. This information is exchanged between two or more players whenever they synchronize on joint transitions.

During the course of a game, the numbers of players might change due to the generation or consumption of a player. Petri games that do not change the number of players are called *concurreny-preserving*. Because Petri games are used to model and synthesize distributed systems, it is desirable to know the maximal number of processes a priori. Furthermore, there exists algorithms based on Petri games that require the number of players to stay the same.

In this thesis, we present an algorithm that makes Petri games concurrency-preserving if the Petri game matches a certain form. Furthermore, the transformed game preserves the flow and causality of the original game. It is also shown that the winning strategy for the extended game is the same as in the original game.

Acknowledgement

I am very grateful to Prof. Bernd Finkbeiner for offering me this interesting and challenging topic. I also want to thank my advisor Jesko Hecking-Harbusch for his help, constructive criticism, and guidance during completing this thesis. Furthermore, I would like to thank Prof. Bernd Finkbeiner and Prof. Holger Hermanns for reviewing this thesis. Moreover, I also place on record, my sincere thank you to my family and friends for their ongoing support. Especially, I want to thank Niklas, Rafael, and Tom for proofreading the thesis.

Contents

1 Introduction

Errors in safety-critical systems like autonomous vehicles can be extremely expensive, e.g., if the recognition of such an error delays the market launch, or even pose a threat to human lives if the error remains undetected. In computer science, there exist two major approaches to prove the absence of errors in such a system. On the one hand, there is *verification*. A verification tool takes as input a specification that describes the desired behaviour and checks whether a given implementation satisfies the specification. If the tool can find incorrect behaviour in the implementation, one has to debug the system and check for correctness again. This procedure repeats itself until the implementation is correct and poses a lot of work to the programmer. On the other hand, there is *synthesis*. A synthesis tool checks whether a given specification is satisfiable. If this is the case, it automatically constructs a per definition correct implementation. Although this frees the programmer from additional work, synthesis is much harder to solve than verification.

In this thesis, we are particularly interested in the synthesis of distributed systems, i.e., systems that consist of multiple components. For example, consider a computer program that at first spawns several threads. After spawning, each thread works on a given task. While processing the given task, every thread progresses with its own speed and completely independent of the other threads. Only after finishing the task, all threads synchronize and the program terminates.

A model for the interaction between multiple independent components in a distributed system are *Petri nets* [5]. In a Petri net, each component is represented as a token and can either progress locally or by synchronizing with other components.

Petri games [2] extend Petri nets by partitioning the places into the system places and environment places. Depending on the type of the place a token currently resides on, it either represents a system player or an environment player. Information is gathered locally by each player in form of her causal past and only exchanged between multiple players by synchronizing on a joint transition. Every local environment player represents a source of nondetemernism, i.e., the environment is uncontrollable. System players are controlled by the global system strategy. The strategy decides which transitions should be taken based on the causal past of the involved system players.

Besides partitioning the places into system and environment places, a Petri game also marks certain places as bad. The goal of the system players is to avoid such bad places, whereas the environment tries to reach these places in order to win the game. A strategy for the system players that

avoids bad places against every possible behaviour of the environment is called winning and corresponds to a correct implementation. The synthesis problem for Petri games asks for the existence of a winning strategy for the system players. If such a strategy exists, it automatically constructs one.

During the course of a game, it is possible that players are *generated* or *consumed*. For example, the spawning and terminating of a thread can be modelled as generated and consumed players, respectively. Games that do not change the number of players are called *concurrency-preserving*. As Petri games are used to model distributed systems, it is desirable that the number of needed components is known a priori. Furthermore, there exist algorithms, e.g., the extraction of a finite strategy representation for Petri games with one system and a bounded number of environment players [1], that require games where the underlying net is concurrency-preserving.

To this end, the main contribution of this thesis consitutes an algorithm that makes a given Petri game concurrency-preserving. We start the challenge of generating concurrency-preserving Petri games with a small example and provide a procedure that can transform it in the desired form. Afterwards, the procedure is applied to more complex examples such that it has to be extended in order to satisfy certain properties. This method is repeated until the algorithm can handle all kinds of Petri games.

The main problem while constructing the algorithm was to preserve the flow and causality of the original game. Moreover, there exist certain types of Petri games that have to be preprocessed such that we can apply the algorithm to it. In the end, we were able to satisfy and prove the desired properties. Furthermore, it is shown that the system players have a winning strategy in the extended game if and only if they have one in the original game.

The remainder of this thesis is structured as follows. In Section 2 2, a general overview of Petri nets and Petri games is given. Section 3 provides an algorithm that makes Petri games concurrency-preserving and defines the desired properties of the algorithm. Proofs for these properties are presented in Section 4. Related work is discussed in Section 5 and we conclude this thesis in Section 6.

2 Background

In this section, we introduce the basic definitions of Petri nets. Especially the terms of generating, consuming, and concurrency-preserving transitions are of importance for the remainder of this thesis. Moreover, Petri games are presented as an extension of Petri nets to games.

2.1 Petri Nets

Petri nets [5] constitute a graphical model describing the behaviour of independent components in distributed systems. They consist of *places* and *transitions*, illustrated by circles and squares, respectively. Places can contain *tokens*, which represent different parts of the system. In this thesis, we only consider *safe* Petri nets, i.e., nets where every place holds at most one token at a time. Tokens can move through the Petri net by using transitions that are connected to places by arrows, which are defined by the *flow relation*. A transition is *enabled* if all preceding places hold a token. When *firing* an enabled transition, all tokens of the preceding places are removed and one token is added to every place after the transition.

Definition 2.1. *(Petri Net)*
A Petri net $\mathcal{N}$ *is a tuple* $(\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$*, where*

- $\mathcal{P}$ *is a finite non-empty set of* places*,*
- $\mathcal{T}$ *is a finite non-empty set of* transitions*,*
- $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ *is the* flow relation*,*
- $In \subseteq \mathcal{P}$ *is the* initial marking*.*

We require that the set of places and the set of transitions are disjoint. Pairs of the form $(p, t) \in \mathcal{F}$ represent that p precedes t, denoted by an arrow from p to t, whereas pairs of the form $(t, p) \in \mathcal{F}$ represent that p follows t, denoted by an arrow from t to p. Places that are part of the *initial marking* are marked by a dot. By convention, we refer to the components of a Petri net $\mathcal{N}$ by $\mathcal{P}, \mathcal{T}, \mathcal{F}$ and In, and similarly for nets with names $\mathcal{N}_1$, $\mathcal{N}^U$, $\mathcal{N}^\sigma$, etc.

A *marking* $\mathcal{M} \subseteq \mathcal{P}$ of $\mathcal{N}$ is a finite set of places that describes the current state of the system by listing all places that currently hold a token. For a transition t, we define the *preset* by $pre(t) = \{p \in \mathcal{P} \mid (p, t) \in \mathcal{F}\}$ and the *postset* by $post(t) = \{p \in \mathcal{P} \mid (t, p) \in \mathcal{F}\}$. They represent the places preceding and following the transition t, respectively. Analogously, we define the pre- and postset of a place p as $pre(p) = \{t \in \mathcal{T} \mid (t, p) \in \mathcal{F}\}$ and $post(p) = \{t \in \mathcal{T} \mid (p, t) \in \mathcal{F}\}$.

We say a transition t is *enabled* in a marking $\mathcal{M}$ if $pre(t) \subseteq \mathcal{M}$. By *firing* an enabled transition t, we obtain a new marking $\mathcal{M}' = \mathcal{M} \setminus pre(t) \cup post(t)$ by removing all tokens preceding t and adding tokens to the places following t. We write $\mathcal{M}\ [t\rangle\ \mathcal{M}'$ to denote that firing t in $\mathcal{M}$ leads to $\mathcal{M}'$. Note that firing a transition might change the number of tokens that are currently in the net. We say a transition t is *generating* if $|pre(t)| < |post(t)|$, *consuming* if $|pre(t)| > |post(t)|$, and *concurrency-preserving* if it is neither generating nor consuming. The set of *reachable markings* $\mathcal{R}(\mathcal{N})$ of a net $\mathcal{N}$ is defined by $\mathcal{R}(\mathcal{N}) = \{\mathcal{M} \mid \exists t_1, \ldots, t_n \in \mathcal{T}.\ In\ [t_1\rangle\ \mathcal{M}_1\ [t_2\rangle\ \ldots\ [t_n\rangle\ \mathcal{M}_n = \mathcal{M}\}$.

Petri nets provide a great model to show the causal dependencies between different events. In order to analyze these dependencies, we will first introduce notations for the different kinds of causal relationships. The symbol $<$ is used for the transitive closure of $\mathcal{F}$ and $\leq$ for its reflexive and transitive closure. We use infix notation when talking about the closures of $\mathcal{F}$, i.e., instead of writing $(x, y) \in <$ and $(x, y) \in \leq$, we will write $x < y$ and $x \leq y$, respectively. Also, we use the term *nodes* when refering to elements of $\mathcal{P} \cup \mathcal{T}$. Two nodes x and y are *causally related* if $x \leq y$ or $y \leq x$. If there exists a place p, which is different from x and y, such that the nodes x and y can be reached from p using different outgoing arcs, we say that x and y are *in conflict*, denoted by $x \sharp y$. We call x and y *concurrent* if they are neither causally related nor in conflict. The *causal past* of a node x is the set $past(x) = \{y \in \mathcal{P} \cup \mathcal{T} \mid y \leq x\}$.

We are now able to define *occurrence nets*, certain acyclic nets that explicitly show the causal relationships between nodes:

Definition 2.2. *(Occurrence Net)*
An occurrence net *is a Petri net $\mathcal{N}$, where*

- $\forall p \in \mathcal{P}.\ |pre(p)| \leq 1$*, i.e., each place has at most one incoming transition,*
- $In = \{p \in \mathcal{P} \mid pre(p) = \emptyset\}$*,*
- *the inverse flow relation $\mathcal{F}^{-1}$ is well-founded, i.e., starting from any node in $\mathcal{N}$ will eventually reach a place in the initial marking when following the flow relation backwards,*
- $\neg\exists t \in \mathcal{T}.\ t \sharp t$*, i.e., no transition is in self-conflict.*

Obviously, no place of an occurrence net is in conflict with itself either. Thus, because of the well-foundedness of $\mathcal{F}^{-1}$, each place in an occurrence net has a unique causal past.

A *homomorphism* from a Petri net $\mathcal{N}_1$ to a Petri net $\mathcal{N}_2$ is a function $h : \mathcal{P}_1 \cup \mathcal{T}_1 \to \mathcal{P}_2 \cup \mathcal{T}_2$ that maps places to places and transitions to transi-

tions, and preserves the pre- and postsets of transitions, i.e., for all $t \in \mathcal{T}_1$ it holds that $h(pre(t)) = pre(h(t))$ and $h(post(t)) = post(h(t))$. If additionally $h(In_1) = In_2$ holds, we call h an *initial homomorphism*.

Definition 2.3. *(Branching Process)*
An (initial) branching process *β of a Petri net $\mathcal{N}$ is a pair $(\mathcal{N}^U, \lambda)$, where $\mathcal{N}^U$ is an occurrence net and λ is an (initial) homomorphism from $\mathcal{N}^U$ to $\mathcal{N}$ such that*

$$\forall t_1, t_2 \in \mathcal{T}^U.\ pre(t_1) = pre(t_2) \wedge \lambda(t_1) = \lambda(t_2) \ \rightarrow\ t_1 = t_2.$$

Intuitively, a branching process (partially) describes the behaviour of a Petri net by copying places and transitions to illustrate the different possible causal pasts. Using the homomorphism λ, we can relate nodes of the occurrence net to the original node in $\mathcal{N}$. For every net, there exists a maximal branching process that describes the complete behaviour of the net. We call this maximal branching process the *unfolding*:

Definition 2.4. *(Unfolding)*
The unfolding *of a Petri net $\mathcal{N}$ is an initial branching process $(\mathcal{N}^U, \lambda)$ such that for every set $S \subseteq \mathcal{P}^U$ of pairwise concurrent places, we have*

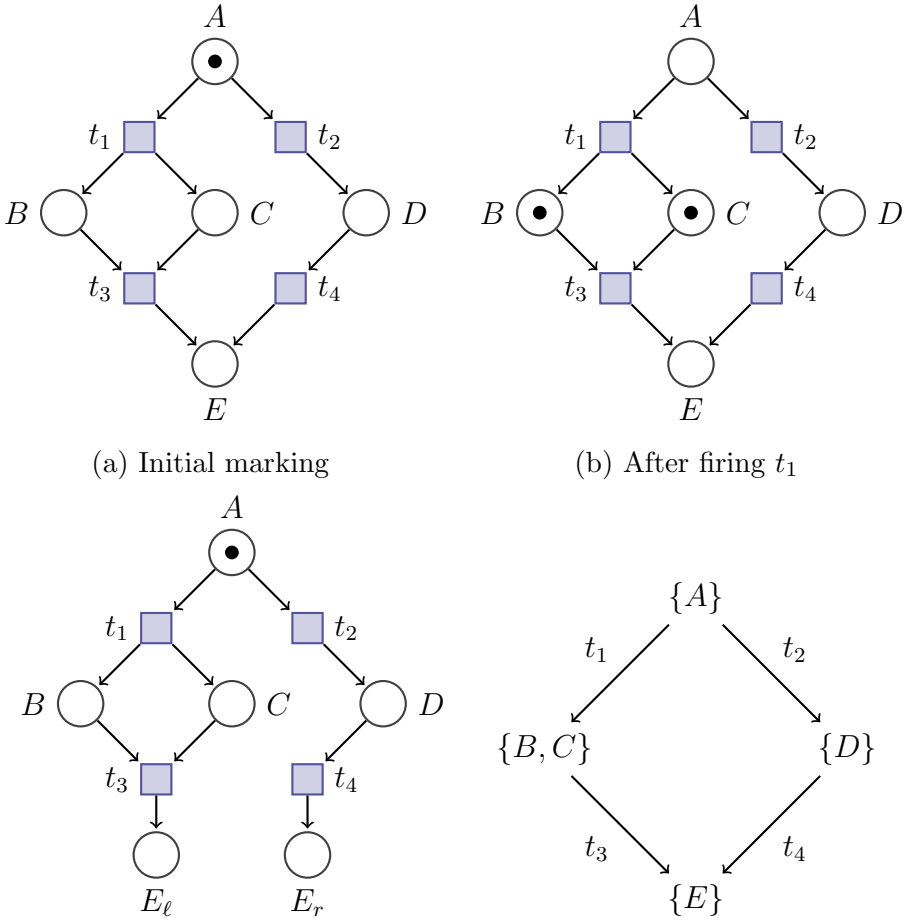
$$\forall t \in \mathcal{T}.\ \lambda(S) = pre(t) \ \rightarrow\ \exists t^U \in \mathcal{T}^U.\ pre(t^U) = S \wedge \lambda(t^U) = t.$$

Whenever a place in the net can be reached via different transitions, the unfolding copies the place and the following subnet, and redirects one incoming arrow of a transition in the place's preset to the copy. When there are cycles in the Petri net, i.e., a reachable sequence of markings that start and end in the same marking, the unfolding unrolls them. This leads to infinite unfoldings, even if the original net is finite.

In some scenarios, we are only interested in the reachable markings of a Petri net and how to reach them, and do not need the additional information of causality. To this end, we introduce *reachability graphs*, certain labeled directed graphs, where the vertices are all reachable markings and the edges represent the transition that needs to be fired in order to get from one marking to another.

Definition 2.5. *(Reachbility Graph)*
The reachability graph *of a Petri net $\mathcal{N}$ is a labeled directed graph $G = (V, E)$, where:*

- $V = \mathcal{R}(\mathcal{N})$,
- $E = \{(\mathcal{M}, t, \mathcal{M}') \mid \mathcal{M}\ [t\rangle\ \mathcal{M}'\}$.



(a) Initial marking

(b) After firing t_1

(c) Unfolding

(d) Reachability graph

Figure 1: A simple Petri net, its unfolding and its reachability graph.

2.2 Petri Net Example

Figure 1 shows an example of a simple Petri net. It consists of five places $\{A, B, C, D, E\}$ and four transitions $\{t_1, t_2, t_3, t_4\}$. The flow is given by $\{(A, t_1), (A, t_2), (B, t_3), (C, t_3), (D, t_4), (t_1, B), (t_1, C), (t_2, D), (t_3, E), (t_4, E)\}$. The initial marking is $\{A\}$ as depicted in Figure 1a. Both transitions in the postset of A, namely t_1 and t_2, are enabled, but either one of them can be fired. The firing of t_1, depicted in Figure 1b, results in the marking $\{B, C\}$ and generates a new token, whereas t_3 consumes a token. Both t_2 and t_4 do not change the number of tokens in the net, i.e., they are concurrency-preserving. Starting from the initial marking, we can reach the marking $\{E\}$ by either firing t_1 and t_3 or t_2 and t_4. When looking at the unfolding in

Figure 1c, we know that we reached E in the underlying net by firing t_1 and t_3 if there is a token in E_ℓ. If there resides a token in E_r, t_2 and t_4 have been fired. This difference cannot be seen in the reachability graph, as shown in Figure 1d.

2.3 Petri Games

Petri games [2] extend Petri nets to multiplayer games and are used for the synthesis of distributed systems. In these games, we divide the set of places into the *system places* (depicted in grey) and the *environment places* (depicted in white). If a token resides in a system place, it is part of the team of system players, otherwise it is part of the team of environment players. The goal is to find a strategy for the system players that is winning against every possible behaviour of the environment, where winning means that the system can avoid reaching certain *bad places*.

Definition 2.6. *(Petri Game)*
A Petri game $\mathcal{G}$ *is a tuple* $(\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, In, \mathcal{B})$*, where*

- $\mathcal{P}_S$ *is a finite set of places belonging to the* system*,*
- $\mathcal{P}_E$ *is a finite set of places belonging to the* environment*,*
- $\mathcal{B} \subseteq \mathcal{P}_S \cup \mathcal{P}_E$ *is a set of* bad places*.*

We call $\mathcal{N}^{\mathcal{G}} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$*, where* $\mathcal{P} = \mathcal{P}_S \cup \mathcal{P}_E$*, the* underlying net *of* $\mathcal{G}$*.*

We say a transition is *purely environmental* if there are only environment places in its preset. Otherwise, we call the transition a *system transition*.

Each player in a Petri game collects local information in form of her causal past when moving through the underlying net. This information is exchanged between different players whenever they synchronize on a joint transition, i.e., after firing a shared transition, all players taking part in it have the same knowledge about the game. When making decisions in the course of the game, each player can only use its own local information to choose her next move.

As mentioned earlier, we have to unfold Petri nets to explicitly display the causal past of places. Thus, we extend the notion of unfoldings to Petri games: the unfolding of a Petri game is the unfolding of the underlying net. Copying places in the unfolding of the underlying net includes copying their properties, e.g. system, environment, or bad places. From the unfolding of a game, we can construct a strategy for the system players by deleting system transitions and their subsequent nodes. Formally, this deleting is called a

sub-process $(\mathcal{G}^{U'}, \lambda)$ of an unfolding $(\mathcal{G}^{U}, \lambda)$, where λ is a homomorphism mapping nodes of the strategy to the underlying net.

Definition 2.7. *(Winning Strategy)*
A winning strategy *for the system players is a subprocess* $\sigma = (\mathcal{N}^\sigma, \lambda^\sigma)$ *of the unfolding* $\beta_U = (\mathcal{N}^U, \lambda)$ *of the underlying net of the game that satisfies the following conditions:*

(1) *For all* $\mathcal{M} \in \mathcal{R}(\mathcal{N}^\sigma)$*, there exists no place* $p \in \lambda(\mathcal{M})$ *such that* $p \in \mathcal{B}$*.*

(2) *For all* $p \in \mathcal{P}_S^\sigma$ *and all* $\mathcal{M} \in \mathcal{R}(\mathcal{N}^\sigma)$ *with* $p \in \mathcal{M}$*, there is at most one transition* $t \in post(p)$ *that is enabled in* $\mathcal{M}$*.*

(3) *For all* $\mathcal{M} \in \mathcal{R}(\mathcal{N}^\sigma)$*, we require that, if there is a transition* $t \in \mathcal{T}^U$ *that is enabled in* $\mathcal{M}$*, then there also is a transition* $t^\sigma \in \mathcal{T}^\sigma$ *that is enabled in* $\mathcal{M}$*.*

(4) *Let* $S \subseteq \mathcal{P}^\sigma$ *be a set of pairwise concurrent places and* $t \in \mathcal{T}$ *be a transition, where* $\lambda(S) = pre(t)$ *but there is no* $t^\sigma \in \mathcal{T}^\sigma$ *such that* $\lambda(t^\sigma) = t$ *and* $pre(t^\sigma) = S$*. Then, there exists a place* $p \in S \cap P_S^\sigma$ *such that* $t \notin \lambda(post(p))$*.*

In the definition of winning strategies, we denote the system places in the strategy by $\mathcal{P}_S^\sigma := \mathcal{P}^\sigma \cap \lambda^{-1}(\mathcal{P}_S)$ and the environment places in the strategy by $\mathcal{P}_E^\sigma := \mathcal{P}^\sigma \cap \lambda^{-1}(\mathcal{P}_E)$. The definition of strategies used in [2] only includes conditions (2) and (4). Using their definition, our strategies correspond to winning and deadlock-avoiding strategies. However, we are never interested in strategies that are not winning or deadlock-avoiding.

Condition (1) describes the *safety* condition, i.e., that the system must ensure that no player ever reaches a bad place.

Condition (2) forces the system players to be *deterministic*. For every reachable marking in the strategy, a system player must allow at most one transition in its postset. This does not forbid a system player from allowing multiple transitions in one place as long as they are not enabled in the same marking.

In order to satisfy condition (1), the system could simply refuse to allow any transition at all such that the game makes no progress and thus, possibly avoid bad places. However, condition (3) forces the system to *avoid deadlocks*. Whenever there is a transition enabled in a marking, there also must be an enabled transition in the strategy. Note that it is still possible that the game reaches a marking where there is no transition enabled at all. Furthermore, a system player can refuse to allow any transition in its postset as long as she knows that there are other players that will keep the game alive.

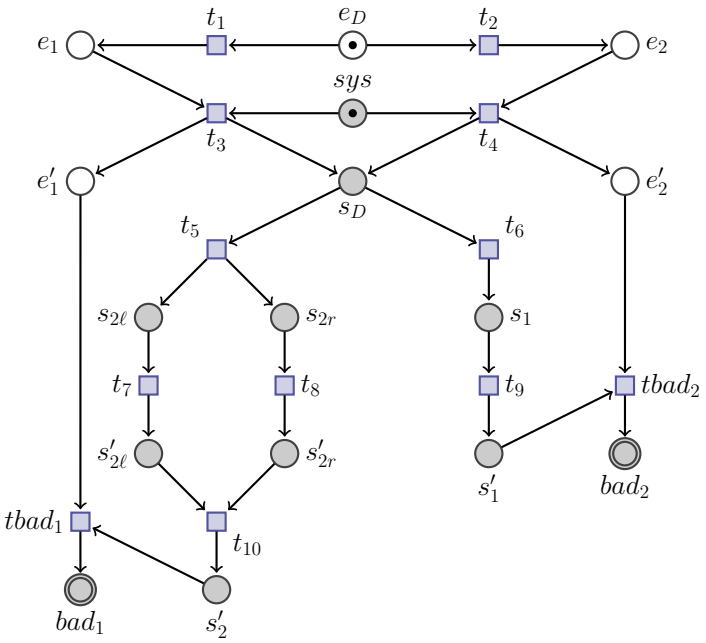


Figure 2: A simple Petri game modelling the decision between spawning one or two threads to work on a given task.

Lastly, condition (4), which is named *justified refusal*, intuitively means that system players influence the game by forbidding certain transitions in the postsets of their current places. Even if a transition is enabled in the underlying net, it will not be fired in the course of the game if a system player refuses to take part in it. Note that condition (4) only allows us to restrict system transitions. Furthermore, a system player can only refuse all transitions in the strategy with the same label or must allow all of them.

2.4 Petri Game Example

Consider the Petri game in Figure 2. The goal of the system is to decide whether it needs to spawn one or two threads to process a given task. In the initial marking, there is an environment player in e_D and a system player in sys, and there are two transitions enabled, namely t_1 and t_2. At first, the environment player in e_D has to decide whether the system has to solve a simple task that only needs one thread, represented by taking t_1, or a hard task that needs two threads, represented by taking t_2. After the environment player's decision, she synchronizes with the system player in sys by taking t_3 or t_4, depending on the current place of the environment player. Now, because of her causal past, the system player in s_D knows the difficulty of the

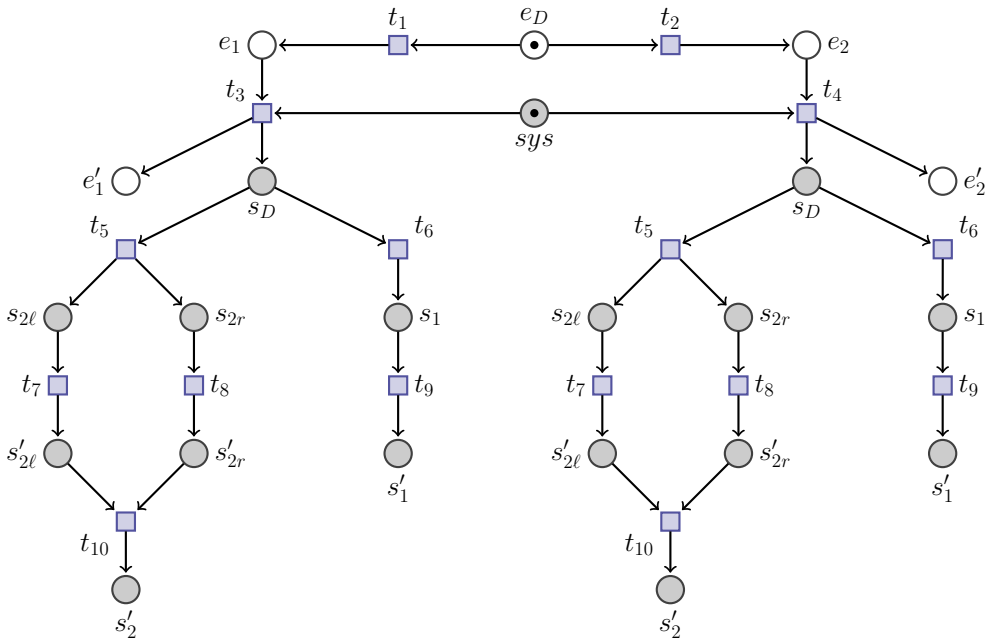


Figure 3: The unfolding of the Petri game in Figure 2. The labels of the places and transitions are the same as in the underlying game. Bad places are omitted for readability.

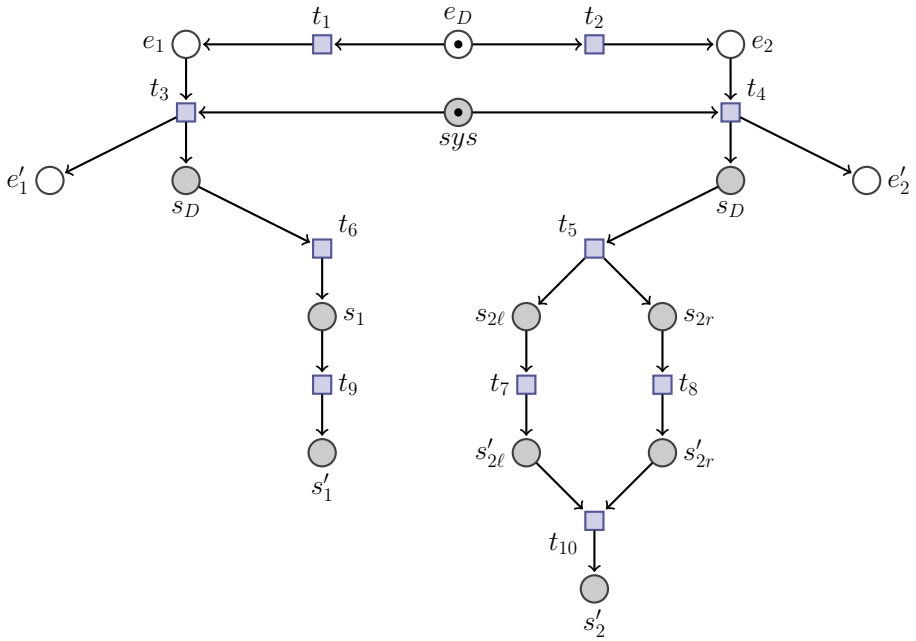


Figure 4: A winning strategy for the system players based on the unfolding from Figure 3 for the Petri game in Figure 2.

given task and has to decide between spawning one thread, i.e., firing transition t_6, or spawning two threads, i.e., firing transition t_7. After spawning, the thread (or threads) work on the task and finish in place s'_1 (or s'_2).

There are two bad places, bad_1 and bad_2, that are reached after the markings $\{e'_1, s'_2\}$ and $\{e'_2, s'_1\}$, respectively. bad_1 corresponds to the scenario where the system spawned one thread too much, whereas bad_2 represents the scenario that the system spawned one thread too little.

The unfolding of the game is given in Figure 3. Looking at the places labelled s_D, we directly can see whether the chosen task is an easy or a hard one, depending on the position of s_D. If it is the place in the postset of t_3, the task is easy, otherwise it is hard. Using this information, the system can easily construct the strategy depicted in Figure 4 by deleting t_5 on the left-hand side and t_6 on the right-hand side of the unfolding. It is now impossible to reach a bad marking when playing conforming to the strategy. Furthermore, there are no reachable markings where any system player enables more than one transition in its postset. As the game always terminates, it is also deadlock-avoiding. Thus, the strategy is winning for the system players.

3 Algorithm

In this section, we present the main achievement of this bachelor's thesis. We give an algorithm that makes safe Petri games concurrency-preserving. In Section 3.1, we explain the desired properties of the algorithm while making the given Petri game concurrency-preserving. Afterwards, we incremently construct our algorithm. We start with a small example of a Petri game and present a procedure to make it concurrency-preserving. Then, we apply this procedure to other Petri games and show that the resulting game violates one (or more) of the desired properties. Thus, we extend our algorithm such that the violated property is satisfied. We repeat this method until the algorithm can handle all kinds of Petri games. Proofs for the desired properties are provided in Section 4.

3.1 Goals of the Algorithm

The most obvious goal of our algorithm is, of course, to make Petri games concurrency-preserving. Although it is easy to bring a game in such a form by simply extending the pre- or postsets of transitions that are generating or consuming, doing so can create a Petri game that differs a lot from the original game.

When generating the concurrency-preserving game, we want to keep the original flow of the underlying net, i.e., we want that in both games the same sequences of transitions can be fired starting from the initial marking. Petri games are used to model the interaction and information flow in a distributed system, so restricting the firing sequences would not represent the intended behaviour of the original game.

Using the same motivation, we also want to preserve the causality of the original game. Altering the causality might result in giving a player more information than she should have according to the designer's intent. This invalid information could be used by the system players to falsely construct a winning strategy for a game they would actually lose.

The goal of Petri games is to automatically construct a strategy for the system players that is winning against every possible behaviour of the environment. Thus, we require that there is a winning strategy for the system in the game generated by the algorithm if and only if there is a winning strategy for the system in the original game.

Lastly, as we only consider safe Petri nets in this thesis, we want the underlying net of the resulting game to be safe too.

(a) Generating transition (b) Consuming transition

Figure 5: Two simple Petri nets before and after applying Algorithm 1.

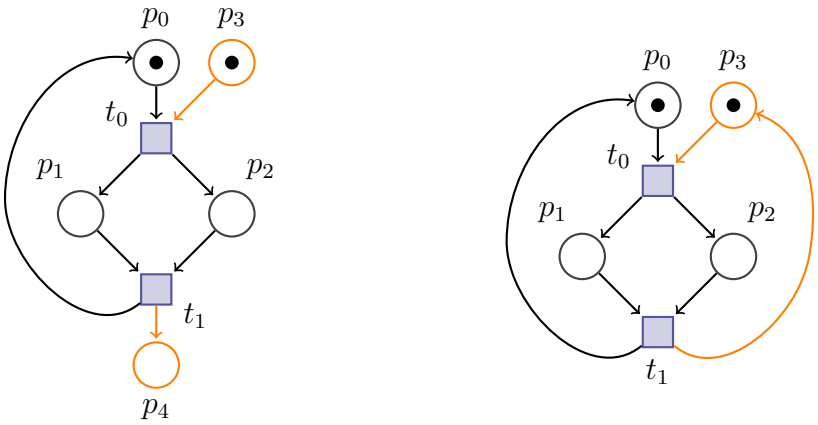
3.2 Sequential Petri Nets

In the remainder of this thesis, *solving* a game or net refers to making the (underlying) net concurrency-preserving. When illustrating a Petri net in a figure, the original net consists of all nodes except the orange ones, whereas its solution includes every node. Furthermore, we will only solve Petri nets at first. The extension to Petri games occurs in Section 3.9. Up until Section 3.7, we will ignore the extension of the initial marking and assume that the intermediate algorithms add the necessary tokens correctly.

Consider the Petri nets in Figure 5 as our first example. The net on the left-hand side consists of a transition with one place in its preset and two places in its postset, i.e., it is a *generating* transition, and the net on the right-hand side consists of a transition with two places in its preset and one place in its postset, i.e., it is a *consuming* transition. Solving these nets can be done by adding a fresh place to the preset of the generating transition and a fresh place to the postset of the consuming transition. In general, a sequential Petri net is solved by extending the pre- and postsets of all generating and consuming transitions until the sizes of the pre- and postset of a transition are equal:

Algorithm 1: Solving Sequential Petri Nets

```
1 Procedure MakeCP(𝒩):
2     for t ∈ 𝒯 do
3         if |pre(t)| < |post(t)| then                    // t generating
4             for i ∈ [0, |post(t)| − |pre(t)|] do
5                 add fresh place p to 𝒫 and (p, t) to ℱ
6         else if |post(t)| < |pre(t)| then               // t consuming
7             for i ∈ [0, |pre(t)| − |post(t)|] do
8                 add fresh place p to 𝒫 and (t, p) to ℱ
```



(a) Using Algorithm 1 (b) Using Algorithm 2

Figure 6: A Petri net that contains a single cycle. Using Algorithm 1 to solve it yields a net that terminates after firing t_0 and t_1, although the original net never terminates. Using Algorithm 2 solves this issue by redirecting the token consumed by t_1 into the fresh place in the preset of t_0.

3.3 Redirecting Tokens

So far, we only considered Petri nets with a finite flow, i.e., Petri nets that terminate after firing a finite number of transitions. An example of a net with an infinite flow is shown in Figure 6. When applying Algorithm 1 to the Petri net, we add the place p_3 to the preset of the generating transition t_0, and the place p_4 to the postset of the consuming transition t_1, illustrated in Figure 6a. Unfortunately, this changes the flow of the original net. Beginning in the initial marking $\{p_0\}$, we can fire t_0 and t_1, reach the initial marking again, and repeat the process infinitely often. The altered net, however, reaches the marking $\{p_0, p_4\}$ after firing t_0 and t_1 and terminates because t_0 is not enabled again. This can be avoided if our algorithm takes into account the possiblity of cycles in the net. Instead of creating a dead end for tokens that are no longer needed after firing a consuming transition, the token has to be moved to a subsequent and newly added place in the cycle that is in the preset of a generating transition. As cycles repeat markings at a certain point, the notion of a *subsequent* place might refer to a place in the following iteration of the cycle. From now on, we will use the term *redirecting* when moving a token into the preset of a generating transition. The solution of the net, shown in Figure 6b, corresponds to redirecting the consumed token of t_1 into p_3.

Formally, when talking about a cycle in a net, we refer to a cycle in

its reachability graph, i.e., a sequence of markings and transitions, where, beginning in the first marking of the sequence, firing the transitions in order leads to the first marking again. Note that this marking is the only marking that occurs twice in the cycle and does not necessarily has to be the initial marking of the net. As we are not interested in the markings of a cycle, we represent it by only listing its transitions and write it as $\langle t_0, \ldots, t_n \rangle$. For now, we assume that a transition can only occur in at most one cycle. Petri nets with transitions that can possibly be contained in multiple cycles will be discussed in Section 3.4.

After describing the process of redirecting tokens, we have to install it into the algorithm. To this end, we introduce a new procedure called `SolveCycle`. This procedure takes an argument *transitions* that contains the transitions of a cycle in the reachability graph. Additionally, the local variable *extendedpre* records all the places that have been added to the preset of a transition but have not been used for redirection yet. The set *extendedpost* does the same for places that have been added to a postset. To solve the cycle, we iterate over all its transitions and distinguish whether they are consuming or generating. If a transition is consuming, we add the correct amount of fresh places to its postset and *extendedpost*. In the case of a generating transition, we check whether there are places in *extendedpost*, i.e., if there are places of consuming transitions that can be used for redirection, and put them into the preset of the transition. If *extendedpost* is empty, we have to add a fresh place to the transitions preset and *extendedpre*. After finishing one iteration of the cycle, we have to connect the remaining elements of *extendedpre* and *extendedpost*. Thus, we take the first elements *prep* and *postp* from *extendedpre* and *extendedpost*, respectively, and combine them by replacing the flow $(t, postp)$ by $(t, prep)$, where t is the transition extended by *postp*. As there is no flow left that contains *postp*, we remove it.

The procedure `SolveCycle` can be found in Algorithm 2. Now, instead of iterating over all transitions, `MakeCP` iterates over all cycles in the reachability graph and makes them concurrency-preserving by calling `SolveCycle`. Afterwards, we handle the remaining transitions that are contained in no cycle like before to solve the complete net.

3.4 Reusing Added Places

Consider the Petri net depicted in Figure 7. Its reachability graph contains exactly two cycles, namely $C_1 = \langle t_2, t_3 \rangle$ and $C_2 = \langle t_0, t_1, t_2, t_4 \rangle$. Solving C_1 at first adds a new place p_8 to $pre(t_2)$ and $post(t_3)$. The problem that arises now is because of the fact that t_2 is contained in both cycles of the net. When trying to solve C_2 and iterating over its transitions, we add fresh places p_6,

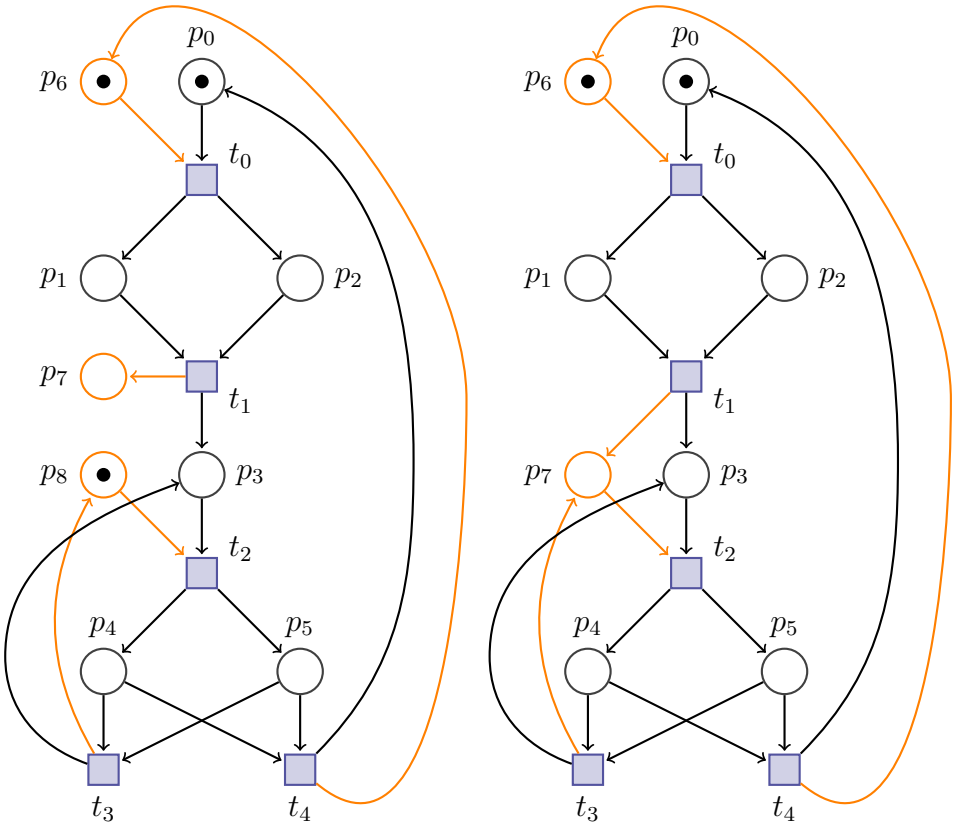
Algorithm 2: Solving Petri Nets by Redirecting Tokens

```
1  Procedure SolveCycle(𝒩, transitions):
2      extendedpre := ∅
3      extendedpost := ∅
4      for t ∈ transitions do
5          if |pre(t)| < |post(t)| then
6              for i ∈ [0, |post(t)| − |pre(t)|] do
7                  if extendedpost ≠ ∅ then
8                      remove newest p from extendedpost
9                  else
10                     add fresh place p to 𝒫 and extendedpre
11                 add (p, t) to ℱ
12         else if |post(t)| < |pre(t)| then
13             for i ∈ [0, |pre(t)| − |post(t)|] do
14                 add fresh place p to 𝒫 and extendedpost
15                 add (t, p) to ℱ
16     for prep ∈ extendedpre do
17         remove newest postp from extendedpost
18         t := transition extended by postp
19         replace (t, postp) by (t, prep) in ℱ
20         remove postp from 𝒫
```

p_7 and p_8 to $pre(t_0)$, $post(t_1)$ and $post(t_4)$, respectively. Because t_2 is already concurrency-preserving after solving C_1, it is not extended again. Due to this fact, there are two places in *extendedpost* and one place in *extendedpre* after the first **for**-loop in `SolveCycle`, so one place from *extendedpost*, namely p_7, is not used for redirection as there are not enough places in *extendedpre*. This scenario is displayed in Figure 7a.

The problem described above can be solved by using added places in multiple cycles. Instead of ignoring an already extended transition when solving another cycle that contains it, we reuse the places that were added to this transition. An application of reusing can be seen in Figure 7b, where p_7 now is contained in both $post(t_1)$ and $post(t_3)$.

To recognize added places, we define the *extended preset* and *extended postset* of a transition t as $expre(t) := \{p \in pre(t) \mid p \text{ added in } \texttt{SolveCycle}\}$ and $expost(t) := \{p \in post(t) \mid p \text{ added in } \texttt{SolveCycle}\}$, respectively. Now, before comparing the sizes of the pre- and postset, we check whether a transition has been extended in a previous call of `SolveCycle`. If t has an extended



(a) Without reusing places (b) With reusing places

Figure 7: A Petri net that contains two cycles sharing t_2. Using places only once for redirection results in a dead end in $post(t_1)$ after firing the sequence $\langle t_0, t_1, t_2, t_4, t_0, t_1 \rangle$ and makes it unsafe. Reusing places for redirection solves this issue by merging p_7 and p_8.

postset, we add all places of $expost(t)$ to *extendedpost* such that we can reuse them for redirection. In the case of an extended preset, we check whether there are places in *extendedpost*. If *extendedpost* is empty, we add the place of $expre(t)$ to *extendedpre* and use it for redirection after iterating over the cycle once. Otherwise, we have to merge the newest element of it with a place in $expre(t)$. Merging simply combines two places by moving the flows from one place to the other. As a place might extend multiple transitions in both its pre- and postset, we have to move the flows from and to all of this transitions. If the two given places are equal, there is nothing to merge.

Algorithm 3: Merging Two Places

```
1  Procedure Merge(prep, postp):
2      if prep = postp then
3          return
4      for ext ∈ pre(postp) do
5          replace (ext, postp) by (ext, prep) in F
6      for ext ∈ post(postp) do
7          replace (postp, ext) by (prep, ext) in F
8      remove postp from P
```

Like before, we have to combine the places that have not been used for redirection in the first iteration. This is done by replacing lines 18-20 in Algorithm 2 with a call to `Merge` with *prep* and *postp*. Additionally, we have to replace line 5 in Algorithm 2 to implement the reusing of places:

Algorithm 4: Extension for Algorithm 2

```
1  if expost(t) ≠ ∅ then
2      extendedpost := extendedpost ∪ expost(t)
3  else if expre(t) ≠ ∅ then
4      for prep ∈ expre(t) do
5          if extendedpost ≠ ∅ then
6              remove newest postp from extendedpost
7              Merge(prep, postp)
8          else
9              add prep to extendedpre
10 else if |pre(t)| < |post(t)| then
       // ...
```

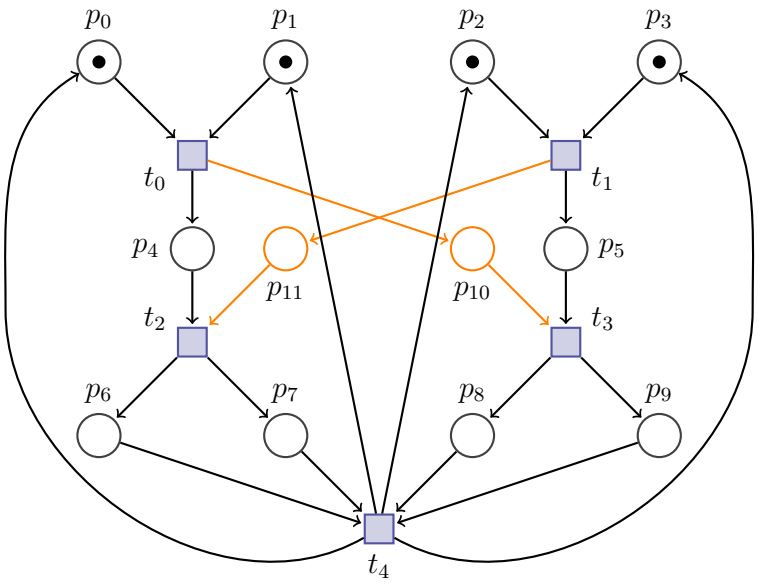
3.5 Preserving Causality

The reachability graph of the net shown in Figure 8 contains four different cycles: $\langle t_0, t_1, t_2, t_3, t_4\rangle$, $\langle t_0, t_1, t_3, t_2, t_4\rangle$, $\langle t_1, t_0, t_2, t_3, t_4\rangle$, and $\langle t_1, t_0, t_3, t_2, t_4\rangle$. As they only differ in the possible interleavings of the transitions t_0, t_1, t_2, and t_3, we would like to treat them as the same cycle, i.e., we only solve one of them. Assume a call to `SolveCycle` with $\langle t_0, t_1, t_2, t_3\rangle$. When reaching t_2 in the first **for**-loop, *extendedpost* contains two places, p_{10} and p_{11}. As p_{11} is the newer place, it is added to $pre(t_2)$. After extending t_2, p_{10} is added to $pre(t_3)$. The resulting net is depicted in Figure 8a.

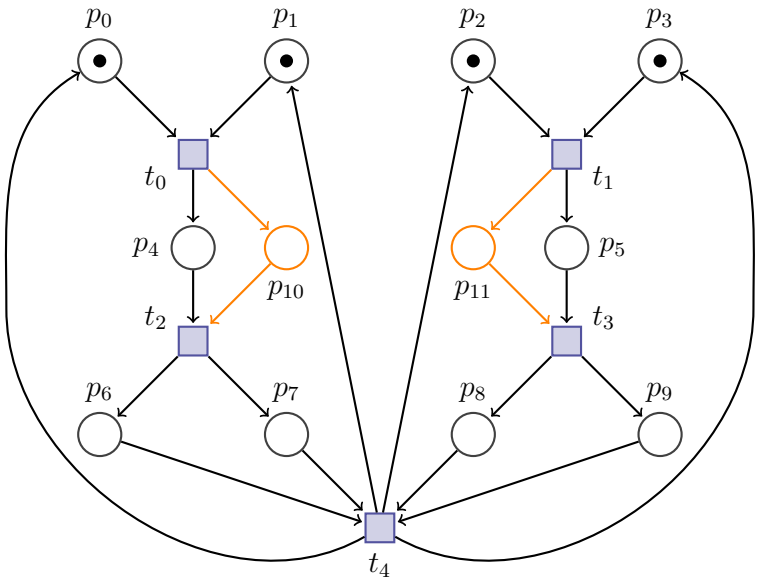
The problem with this solution is that it alters both the flow and causality of the original net. After firing t_0, the transition t_2 should be enabled but is not because there is no token in p_{11}. Even worse, if we also fire t_2 and t_3 afterwards, the tokens residing in the places p_8 and p_9 know of the firing of t_0 because of their causal past. In the original net, this information would not be delivered until the synchronization of all tokens when firing t_4.

To solve this issue, redirected tokens must come from the postset of a transition that lies in the causal past of the currently considered transition. In the example, this means that p_{10} must be in $pre(t_2)$, and p_{11} must be in $pre(t_3)$, as shown in Figure 8b. Implementing this in `SolveCycle` is quite simple. Instead of checking whether *extendedpost* is non-empty, we must check if it contains a place *postp* such that there is a transition $pret \in pre(postp)$ that lies in the causal past of the currently considered transition t.

Causal relationships are only defined for occurrence nets. As there are possibly multiple different nodes in the unfolding of a net $\mathcal{N}$ that correspond to the same node in $\mathcal{N}$, it can be difficult to translate a marking of $\mathcal{N}$ into a marking of its unfolding. Thus, instead of using the complete unfolding to illustrate the causality, we only construct a subnet of it. Beginning from the first marking of the given cycle, we fire the cycle twice. The subnet is constructed by adding (copies of) the fired transitions and the places lying in their postsets. Assuming that a transition occurs only once in the same cycle (cf. Section 3.6), it now occurs exactly twice in the constructed occurrence net. For a transition t, we name its first and second occurrence t^1 and t^2, respectively. When redirecting tokens in the first **for**-loop, we check for a transition that lies in the causal past of t^1 because we have not finished the first iteration of the cycle yet. In the second **for**-loop, we investigate the causal past of t^2. If a place p is in *extendedpost*, the operation $ext(p)$ corresponds to the extended transition in $pre(p)$, otherwise it corresponds to the extended transition in $post(p)$. This transition is unique in the cycle that is currently considered. Note that we do not provide a procedure that constructs the described net.

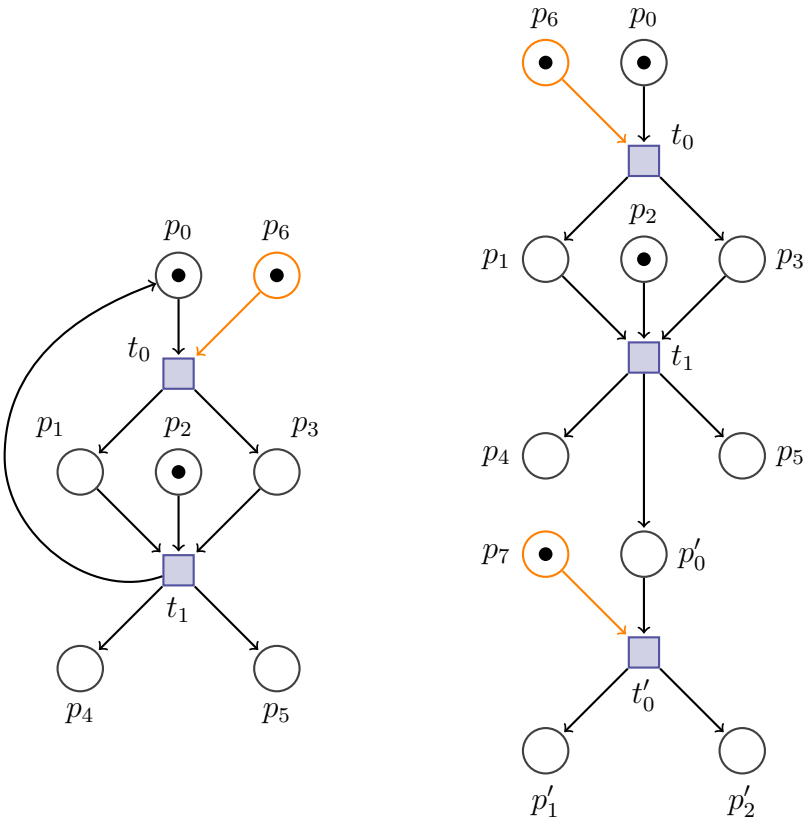


(a) Ignoring causal past



(b) Regarding causal past

Figure 8: A Petri net consisting of two concurrent parts that synchronize repeatedly by using transition t_4. Depending on the interleaving of the transitions t_0, t_1, t_2, and t_3, ignoring the causal past might destroy the flow and causality of the original net.



(a) Before unrolling (b) After unrolling

Figure 9: A Petri net where the generating transition t_0 can be fired exactly twice. Applying `MakeCP` extends $pre(t_0)$ with only one place, so t_0 can only be fired once in the extended net. Unrolling the relevant part copies t_0 and its subsequent nodes such that `MakeCP` provides a correct solution.

3.6 Preprocessing

Consider the Petri net illustrated in Figure 9. It contains one generating transition, t_0, one concurrency-preserving transition, t_1, and no cycle. Solving the net adds a fresh place p_6 to $pre(t_0)$ as usual (see Figure 9a). After firing t_0 and t_1, the marking in both nets is $\{p_0, p_4, p_5\}$. While t_0 can be fired another time in the original net, this is not the case for the extended net because there is no token in p_6. Since there are no consuming transitions, redirecting a token into p_6 is not an option to solve this problem. Neither is putting a second token in p_6 because we only consider safe Petri nets. However, to display that t_0 can be fired twice, we can copy it and its subsequent nodes after encountering it for the second time, as shown in Figure 9b. Now,

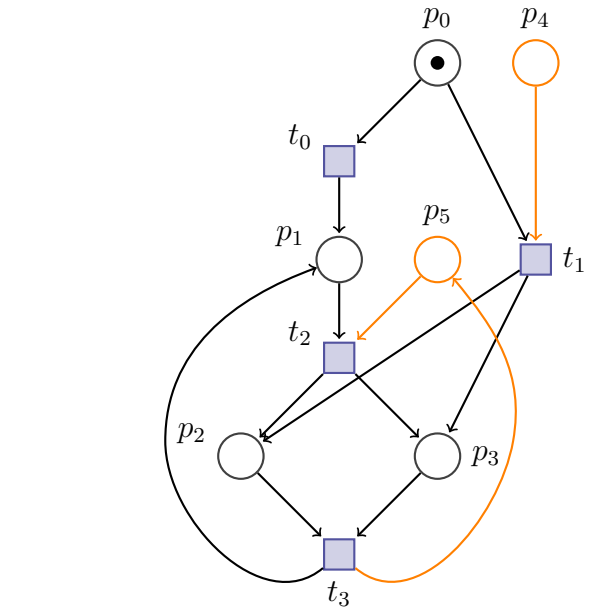
the first and second firing of t_0 in the original net corresponds to the firing of the different transitions t_0 and t_0' in the unrolled version, respectively. As t_0 and t_0' represent two different generating transitions, they both can be extended in their presets, resulting in a correct solution.

In general, when a transition t can be fired $k < \infty$ times in a net, we unroll the relevant part every time we encounter it after the first firing. We do the same with transitions that need to be fired k times to complete a cycle. An example for such a cycle would be an extension of the net in Figure 9a with a transition t_2, where $pre(t_2) = \{p_1, p_3, p_4, p_5\}$ and $post(t_2) = \{p_0\}$, resulting in a cycle $\langle t_0, t_1, t_0, t_2 \rangle$. Note that we do not give a formal algorithm to unroll the cases described above. We simply assume that the input for `MakeCP` is already converted into this form by a preprocessing algorithm.

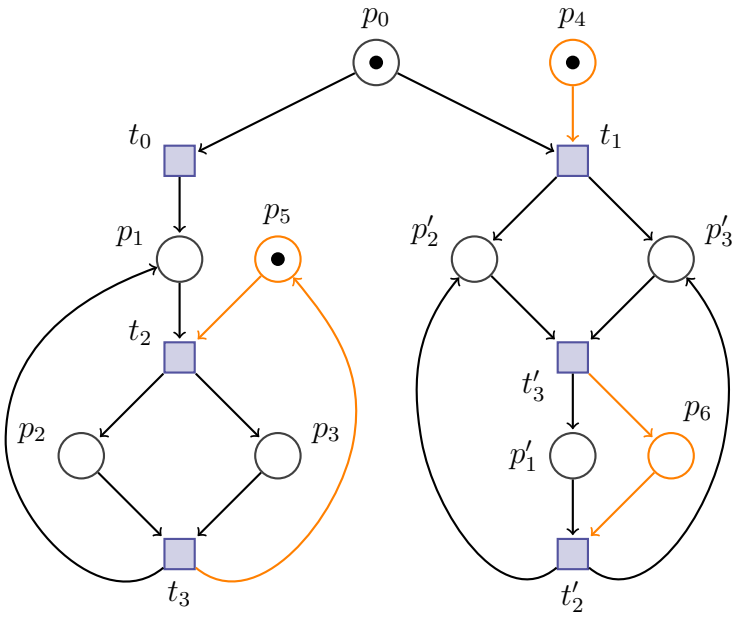
Another case that needs some preprocessing is depicted in Figure 10. The shown Petri net contains one cycle, namely $\langle t_2, t_3 \rangle$. Running the algorithm on it adds the places p_4 and p_5, shown in Figure 10a. When the cycle is entered via t_0, there must be a token in p_5 in order to be able to fire t_2. Also, there must be a token in p_4 in order to be able to fire t_1 from the initial marking. However, when entering the cycle via t_1, firing t_3 puts a second token into p_5. Thus, the extended net is not safe anymore. To solve this issue, we split the two entry points of $\langle t_2, t_3 \rangle$ such that there are now two different cycles. The splitted net is displayed in Figure 10b. $\langle t_2, t_3 \rangle$ represents the entering via t_0, whereas $\langle t_3', t_2' \rangle$ represents the entering via t_1. Containing two different cycles now, we can finally solve the net.

In general, if there exists a cycle in a Petri net that can be entered via k different submarkings, we split the net such that there arise k different cycles, each representing one of the entry points. The term submarking refers to the markings that only consist of places that are part of the pre- and postsets of the transitions in the cycle. The different positions of tokens outside of this submarking do not affect the cycle and can be ignored. Note again that we do not provide a formal algorithm for this preprocessing and assume the given net to already be in the desired form.

Assume a Petri net that contains two cycles with a shared transition. Further assume that there exist different paths in the net such that the representing nodes of the cycle in the unfolding can be in conflict or concurrent. Applying the algorithm to the net adds a fresh place that is now contained in both cycles. However, as the cycles might be executed concurrently, it is possible that both cycles need the token that is residing in the newly added place. If one of the cycles fires the transition with the added place in its preset, the other cycle has to stop until the places is filled again. This alters the flow of the original net. To avoid this kind of problem, we assume that cycles in a net are either in conflict or concurrent, but never both at the same



(a) Before splitting



(b) After unrolling

Figure 10: A Petri net that contains one cycle with two different entry points. For the given extension, it is not possible to find an initial marking such that it preserves the original flow and stays safe. Splitting the different entries makes it possible to solve the independent parts.

time. If they are in conflict, only one cycle can be active at a time, so only one cycle needs the added token. Otherwise, the cycles are concurrent and do not share any places.

When solving different cycles with shared transitions, the number of redirectable tokens in *extendedpost* while encountering a shared transition must not always be the same. However, as we will see in Section 3.7, this number determines whether a newly added place must be part of the initial marking or not. Hence, dealing with different numbers prevents us from finding a correct initial marking. To solve this issue, we split different cycles with shared transitions, where the number of redirectable tokens might differ, such that the shared transitions are different transitions now.

3.7 Extending the Initial Marking

After extending the net with places and describing the needed preprocessing steps, we now describe how the initial marking has to be extended. We start by looking at transitions that do not occur in any cycles. Obviously, only added places in the preset of generating transitions have to be added to the initial marking. Adding a place in the postset of a consuming transition to the initial marking would result in an unsafe net when firing said transition. Thus, when solving the remaining transitions in `MakeCP` after handling all cycles, we add all places that have to be added to presets of generating transitions to the initial marking.

Places that are added by our algorithm to the preset of a generating transition that is contained in a cycle are always also in the postset of a consuming transition. From Section 3.6, we know that every cycle has exactly one entry point. Beginning from this submarking, a place p in the preset of a generating transition has to be added to the initial marking if it has not been added to the preset of a consuming transition yet. This corresponds to the case where there is no suitable place for redirection in *extendedpost*. If it would be added to the initial marking although it could already be used for redirection, firing $ext(p)$ would put a second token into p and make the net unsafe.

Consider again the Petri net in Figure 7 with the cycles $C_1 = \langle t_2, t_3 \rangle$ and $C_2 = \langle t_0, t_1, t_2, t_4 \rangle$. Solving C_1 at first adds p_7 to the initial marking. Afterwards, solving C_2 also adds p_6 to the initial marking. Unfortunately, when firing t_0 und t_1 in the extended net now, there are two tokens in p_7, making the net unsafe. This can be avoided if we remove the token residing in p_7 while solving C_2. In general, if we encounter an already extended generating transition such that the place p in its preset can be used for redirection, we remove p from the initial marking.

In the other case, if we solve C_2 at first, we also add p_6 and p_7 to the initial marking. p_7 cannot be added to In although it cannot be used for redirection directly when solving C_1. To solve this problem, we only add such a place to the initial marking if it has not been extended before. Note that there might be cases where a generating transition appears in two (or more) different cycles and (some of) the places in its preset must indeed be added to the initial marking. Although we do not add these places to In when solving the second cycle, we already did so while solving the first one.

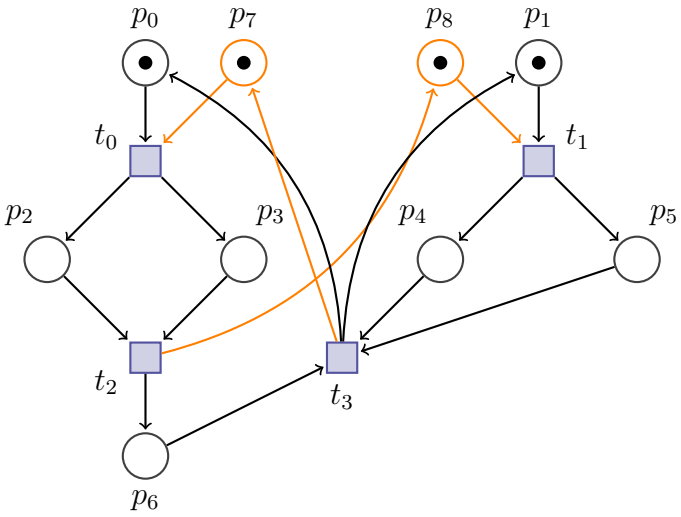
3.8 Correct Order of a Cycle

Consider the Petri net depicted in Figure 11. It contains one cycle that can occur in different interleavings. Assume a call to `SolveCycle` with $\langle t_0, t_1, t_2, t_3 \rangle$. Algorithm 5 returns the solution depicted in Figure 11a. Firing t_0 and t_2 or t_1 puts a second token into the place p_7 or p_8, respectively, and makes the extended net unsafe. Furthermore, it alters the causality of the original net. After firing t_0,t_2, and t_1, the tokens residing in p_4 and p_5 know of the firing of t_0 and t_2 although they should not. This problem arises due to the fact that we are not able to detect that the left-hand and right-hand side of the Petri net progress concurrently up until they synchronize via firing t_3.

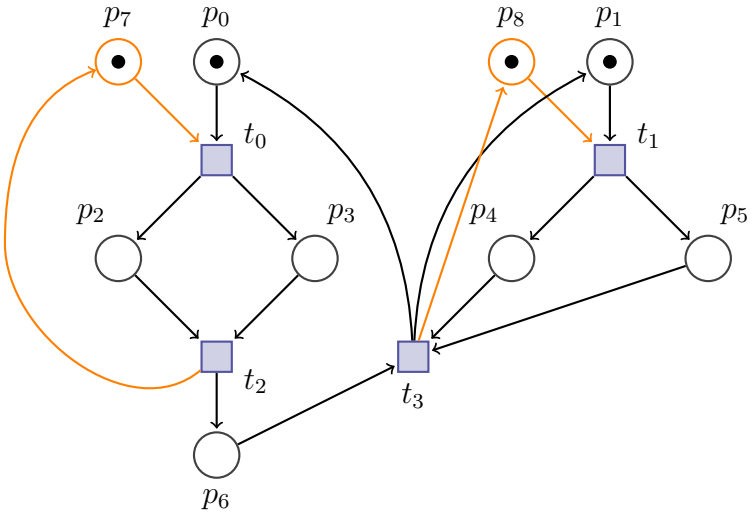
Solving another interleaving of the same cycle, namely $\langle t_0, t_2, t_1, t_3 \rangle$, is illustrated in Figure 11b. In this case, the algorithm is able to redirect the tokens into parts of the net they originated from, i.e., the token in p_7 never enters the right-hand side of the net, whereas the token in p_8 never enters the left-hand side. As already mentioned, we are not able to detect these independent parts of the net. Therefore, we must assume that the cycle given to `SolveCycle` is always in an order such that the algorithm can redirect tokens into their respective parts in the net.

3.9 From Petri Nets to Petri Games

The extension of our algorithm from Petri nets to Petri games is quite simple. When constructing a winning strategy for the system players, we are only allowed to omit system transitions in the unfolding. To avoid that the system players in the extended game could prevent generating transitions that were actually purely environmental from firing, the added places belong to the environment. Thus, these transitions stay purely environmental in the extended game. Also, we extend system transitions with environment places. Adding system places to these transitions would not change the existence of a winning strategy. However, for the sake of simplicity, we only add environment places, as seen in Algorithms 5 and 6.



(a) Solving $\langle t_0, t_1, t_2, t_3 \rangle$



(b) Solving $\langle t_0, t_2, t_1, t_3 \rangle$

Figure 11: A Petri net that contains one cycle with different interleavings. Depending on the interleaving yields a wrong or correct solution.

Algorithm 5: Making Cycles Concurrency-preserving

```
1  Procedure SolveCycle(𝒢, transitions):
2      extendedpre := ∅
3      extendedpost := ∅
4      for t ∈ transitions do
5          if expost(t) ≠ ∅ then
6              extendedpost := extendedpost ∪ expost(t)
7          else if expre(t) ≠ ∅ then
8              for prep ∈ expre(t) do
9                  if ∃postp ∈ extendedpost s.t. ext(postp)¹ ≤ t¹ then
10                     remove newest postp from extendedpost
11                     remove prep from In
12                     Merge(prep, postp)
13                 else
14                     add prep to extendedpre
15         else if |pre(t)| < |post(t)| then
16             for i ∈ [0, |post(t)| − |pre(t)|] do
17                 if ∃postp ∈ extendedpost s.t. ext(postp)¹ ≤ t¹ then
18                     remove newest postp from extendedpost
19                 else
20                     add fresh place p to 𝒫_E, In and extendedpre
21                 add (p, t) to ℱ
22         else if |post(t)| < |pre(t)| then
23             for i ∈ [0, |pre(t)| − |post(t)|] do
24                 add fresh place p to 𝒫_E and extendedpost
25                 add (t, p) to ℱ
26     for prep ∈ extendedpre do
27         remove newest postp from extendedpost such that
             ext(postp)¹ ≤ ext(prep)²
28         Merge(prep, postp)
```

Algorithm 6: Making Petri Games Conurrency-Preserving

```
1  Procedure MakeCP(𝒢):
2      for cycle ∈ ReachabilityGraph(𝒩^𝒢) do
3          SolveCycle(𝒢, cycle)
4      for t ∈ 𝒯 such that t is in no cycle do
5          if |pre(t)| < |post(t)| then
6              for i ∈ [0, |post(t)| − |pre(t)|] do
7                  add fresh place p to 𝒫_E and In
8                  add (p, t) to ℱ
9          else if |post(t)| < |pre(t)| then
10             for i ∈ [0, |pre(t)| − |post(t)|] do
11                 add fresh place p to 𝒫_E to 𝒫 and (t, p) to ℱ
```

4 Proof of Correctness

This section provides a correctness proof for Algorithm 6 presented in Section 3. First, we will prove that the algorithm is well defined and terminates for every Petri game with an underlying net that is in the form described in Section 3.6. Afterwards, we will prove the desired properties mentioned in Section 3.1.

Before giving proofs, we define the *extended game* $\widehat{\mathcal{G}}$ for a game $\mathcal{G}$, where

- $\widehat{\mathcal{P}_S} = \mathcal{P}_S$,
- $\widehat{\mathcal{P}_E} = \mathcal{P}_E \cup \mathcal{P}^A$,
- $\widehat{\mathcal{T}} = \mathcal{T}$,
- $\widehat{\mathcal{F}} = \mathcal{F} \cup \mathcal{F}^A$,
- $\widehat{In} = In \cup In^A$,
- $\widehat{\mathcal{B}} = \mathcal{B}$.

The sets $\mathcal{P}^A$, $\mathcal{F}^A$, and In^A represent the places and flows that were added to the corresponding sets of the original game. We use the notation $\widehat{x}$ for a node $x \in \mathcal{P} \cup \mathcal{T}$ whenever we want to refer to the corresponding part in the extended game.

4.1 Well-definedness and Termination

The only operation in the Algorithm that is possibly not well defined is the check for an element in *extendedpost* in line 27 such that the extended consuming transition lies in the causal past of another generating transition in the same cycle where some place in the preset still needs redirection. We show that this operation always works by arguing over the number of consumed and generated tokens during a cycle, and the fact that there always exists a consuming transition before the second occurence of another generating transition in the constructed occurrence net.

Lemma 4.1. Consider lines 26-28 in Algorithm 5. For every place $prep \in$ *extendedpre*, there exists a place $postp \in$ *extendedpost* such that $ext(postp)^1 \leq ext(prep)^2$.

Proof. Beginning from the first marking of a cycle, firing all transitions contained in the cycle in the correct order leads to the first marking again. Therefore, the number of tokens before and after firing the transitions is the same, i.e., the amount of consumed and generated tokens must be the same.

Thus, if it was not possible to redirect any token in the **for**-loop starting in line 4, the number of elements in *extendedpre* and *extendedpost* is the same after exiting the loop. As redirecting tokens removes exactly one place out of both of these sets, they always have the same size before line 26, no matter how many redirections took place before.

Because markings repeat themselves after finishing a cycle and presets of fireable transitions constitute a submarking, there must exist a transition sequence leading from the first to the second occurence of the preset of a fireable transition. Generating transitions in this sequence can receive tokens from producing transitions lying in their causal past with respect to this sequence. As the transition sequence represents a cycle, the first occurence of a generating transition always is in the causal past of the second occurence of a producing transition such that the tokens between these transitions can be correctly redirected. □

Proving termination follows by the fact that every **for**-loop terminates.

Lemma 4.2. Algorithm 6 terminates for every Petri game $\mathcal{G}$.

Proof. We first show that a call to `SolveCycle` terminates. Since every cycle consists of a finite number of transitions, the **for**-loop starting in line 4 terminates if every iteration terminates. This is indeed the case because all other **for**-loops starting in lines 8, 15, and 22 terminate due to the fact that pre- and postsets are finite. Because of this finiteness, we only add a finite number of places to *extendedpre*, so the **for**-loop starting in line 26 terminates too.

As every call to `SolveCycle` terminates and a reachability graph only contains a finite number of cycles, the **for**-loop starting in line 1 of `MakeCP` terminates. Further, a Petri net only contains a finite number of transitions and every transition has a finite pre- and postset. Therefore, the **for**-loop starting in line 4 terminates too. □

4.2 Preserving the Concurrency

The main goal of this thesis is to provide an algorithm that makes a Petri game concurrency-preserving. This is done for transitions in cycles in Algorithm 5. Transitions that do not occur in any cycle are extended in Algorithm 6. As every generating or consuming transition is extended exactly once by a set of places in the size of the difference between the sizes of pre- and postset, every transition in the extended game is concurrency-preserving.

Lemma 4.3. Every transition $\widehat{t} \in \widehat{\mathcal{T}}$ is concurrency-preserving.

Proof. By case distinction based on t. If t is concurrency-preserving in $\mathcal{N}^{\mathcal{G}}$, we have $|pre(t)| = |post(t)|$. As Algorithm 6 does not extend t, we also have $pre(\widehat{t}) = pre(t)$ and $post(\widehat{t}) = post(t)$. Thus, $|pre(\widehat{t})| = |post(\widehat{t})|$.

Assume t is generating in $\mathcal{N}^{\mathcal{G}}$, i.e., $|pre(t)| < |post(t)|$. If t is contained in a cycle, its preset is extended in lines 16-21 of `SolveCycle`, otherwise its preset is extended in lines 6-8 of `MakeCP`. In both cases, the number of added places is $|post(t)| - |pre(t)|$. Thus, we have $|pre(\widehat{t})| = |pre(t)| + |post(t)| - |pre(t)| = |post(t)| = |post(\widehat{t})|$.

Assume t is consuming in $\mathcal{N}^{\mathcal{G}}$, i.e., $|post(t)| < |pre(t)|$. If t is contained in a cycle, its postset is exended in lines 23-25 of `SolveCycle`, otherwise its postset is extended in lines 10-11 of `MakeCP`. In both cases, the number of added places is $|pre(t)| - |post(t)|$. Thus, we have $|post(\widehat{t})| = |post(t)| + |pre(t)| - |post(t)| = |pre(t)| = |pre(\widehat{t})|$. □

4.3 Preserving the Flow

We prove that every sequence of transitions that can be fired in the original net can also be fired in the extended net. Further, the reached marking in the extended net is a superset of the marking in the original net.

Lemma 4.4. For every transition sequence $\vec{t} = \langle t_1, \ldots, t_n \rangle$ and marking $\mathcal{M} \in \mathcal{R}(\mathcal{N}^{\mathcal{G}})$, there exists a marking $\widehat{\mathcal{M}} \in \mathcal{R}(\mathcal{N}^{\widehat{\mathcal{G}}})$ such that

$$In\ [t_1\rangle\ \ldots\ [t_n\rangle\ \mathcal{M} \implies \widehat{In}\ [\widehat{t_1}\rangle\ \ldots\ [\widehat{t_n}\rangle\ \widehat{\mathcal{M}} \wedge \mathcal{M} \subseteq \widehat{\mathcal{M}}.$$

Proof. By induction over the length of $\vec{t}$. If $\vec{t}$ is empty, there is no transition in $\vec{t}$ that can be fired and we have $\mathcal{M} = In$. Similarly, we have $\widehat{\mathcal{M}} = \widehat{In}$. By construction of $\widehat{In}$, we have $In \subseteq \widehat{In}$ and the claim holds.

Now assume the claim has already been established for a transition sequence $\vec{t} = \langle t_1, \ldots, t_n \rangle$ with $n \geq 1$, i.e., if $In\ [t_1\rangle\ \ldots\ [t_n\rangle\ \mathcal{M}$, there exists a marking $\widehat{\mathcal{M}} \in \mathcal{R}(\mathcal{N}^{\widehat{\mathcal{G}}})$ such that $\widehat{In}\ [\widehat{t_1}\rangle\ \ldots\ [\widehat{t_n}\rangle\ \widehat{\mathcal{M}}$ and $\mathcal{M} \subseteq \widehat{\mathcal{M}}$. We have to show that if $\mathcal{M}\ [t_{n+1}\rangle\ \mathcal{M}'$ for $t_{n+1} \in \mathcal{T}$ and $\mathcal{M}' \in \mathcal{R}(\mathcal{N}^{\mathcal{G}})$, there also exists a marking $\widehat{\mathcal{M}'} \in \mathcal{R}(\mathcal{N}^{\widehat{\mathcal{G}}})$ such that $\widehat{\mathcal{M}}\ [\widehat{t_{n+1}}\rangle\ \widehat{\mathcal{M}'}$ and $\mathcal{M}' \subseteq \widehat{\mathcal{M}'}$. We make a case distinction based on t_{n+1}:

If t_{n+1} is concurrency-preserving in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 neither extends $pre(t_{n+1})$ nor $post(t_{n+1})$. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1})$ and $post(\widehat{t_{n+1}}) = post(t_{n+1})$. Since $\mathcal{M}\ [t_{n+1}\rangle\ \mathcal{M}'$, we have $pre(t_{n+1}) \subseteq \mathcal{M}$ and $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. By induction hypothesis, we have $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, and thus $pre(\widehat{t_{n+1}}) \subseteq \widehat{\mathcal{M}}$. Therefore, we can fire $\widehat{t_{n+1}}$ in $\mathcal{N}^{\widehat{\mathcal{G}}}$, resulting in $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. Because $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, $pre(t_{n+1}) = pre(\widehat{t_{n+1}})$, and $post(t_{n+1}) = post(\widehat{t_{n+1}})$, we have $\mathcal{M}' \subseteq \widehat{\mathcal{M}'}$, proving the claim.

Assuming t_{n+1} is consuming in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 only extends $post(t_{n+1})$ with a set of places P^A. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1})$ and $post(\widehat{t_{n+1}}) = post(t_{n+1}) \cup P^A$. Since $\mathcal{M}\ [t_{n+1}\rangle\ \mathcal{M}'$, we have $pre(t_{n+1}) \subseteq \mathcal{M}$ and $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. By induction hypothesis, we have $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, and thus $pre(\widehat{t_{n+1}}) \subseteq \widehat{\mathcal{M}}$. Therefore, we can fire $\widehat{t_{n+1}}$ in $\mathcal{N}^{\widehat{\mathcal{G}}}$, resulting in $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. Because $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, $pre(t_{n+1}) = pre(\widehat{t_{n+1}})$, and $post(t_{n+1}) \subseteq post(\widehat{t_{n+1}})$, we have $\mathcal{M}' \subseteq \widehat{\mathcal{M}'}$, proving the claim.

If t_{n+1} is generating in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 only extends $pre(t_{n+1})$ with a set of places P^A. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1}) \cup P^A$ and $post(\widehat{t_{n+1}}) = post(t_{n+1})$. Since $\mathcal{M}\ [t_{n+1}\rangle\ \mathcal{M}'$, we have $pre(t_{n+1}) \subseteq \mathcal{M}$ and $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. By induction hypothesis, we also have $\mathcal{M} \subseteq \widehat{\mathcal{M}}$. In order show that $\widehat{t_{n+1}}$ is fireable in $\mathcal{N}^{\widehat{\mathcal{G}}}$, we have to show that $P^A \subseteq \widehat{\mathcal{M}}$:

- If t_{n+1} is not in any cycle, all places in P^A are added to $\widehat{In}$ in line 7 of `MakeCP`. As $\widehat{t_{n+1}}$ can only be fired once and no place in P^A is in $pre(\widehat{t})$ for $\widehat{t} \in \widehat{\mathcal{T}} \setminus \{\widehat{t_{n+1}}\}$, none of the transitions in $\vec{t}$ removed a token from P^A. Thus, we have $P^A \subseteq \widehat{\mathcal{M}}$.

- Assume t_{n+1} is part of a cycle. We know that the number of tokens k that can be redirected to the places in P^A when solving the first iteration of the cycle is always the same. This redirection happened by firing the transitions in the cycle that have to be fired before one can fire $\widehat{t_{n+1}}$. These transitions have to be part of $\vec{t}$ and are unique since every cycle has a unique entry point. By induction, all of the transitions could be fired in $\mathcal{N}^{\widehat{G}}$ and directed the tokens to the corresponding places. All other places that were not filled by redirection are contained in the initial marking. Because the transitions of two cycles cannot be in conflict and concurrent at the same time, a transition $\widehat{t'} \neq \widehat{t_{n+1}}$ that has one or more of the places in P^A in its preset must be in conflict with $\widehat{t_{n+1}}$. Thus, if there would be some token in P^A missing in $\widehat{\mathcal{M}}$, t' must have been fired in $\vec{t}$. As t' and t_{n+1} are in conflict, t_{n+1} would not be enabled in $\mathcal{M}$, contradicting the assumption. Hence, $P^A \subseteq \widehat{\mathcal{M}}$.

Firing $\widehat{t_{n+1}}$ in $\mathcal{N}^{\widehat{G}}$ results in $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. Although $pre(t_{n+1}) \subseteq pre(\widehat{t_{n+1}})$, we have $\mathcal{M} \setminus pre(t_{n+1}) \subseteq \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}})$ because the only difference between $pre(t_{n+1})$ and $pre(\widehat{t_{n+1}})$ are the places P^A, which are not contained in $\mathcal{M}$. Since $post(t_{n+1}) = post(\widehat{t_{n+1}})$, we have $\mathcal{M}' \subseteq \widehat{\mathcal{M}'}$. □

For the converse direction, we also show that every transition sequence in the extended net can be fired in the original net too. Furthermore, the reached marking in the original net corresponds to the marking in the extended marking without the added places.

Lemma 4.5. For every transition sequence $\vec{t} = \langle \widehat{t_1}, \ldots, \widehat{t_n} \rangle$ and marking $\widehat{\mathcal{M}} \in \mathcal{R}(\mathcal{N}^{\widehat{\mathcal{G}}})$, there exists a marking $\mathcal{M} \in \mathcal{R}(\mathcal{N}^{\mathcal{G}})$ such that

$$\widehat{In}\ [\widehat{t_1}\rangle\ \ldots\ [\widehat{t_n}\rangle\ \widehat{\mathcal{M}} \implies In\ [t_1\rangle\ \ldots\ [t_n\rangle\ \mathcal{M} \wedge \mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}.$$

Proof. By induction over the length of $\vec{t}$. If $\vec{t}$ is empty, there is no transition in $\vec{t}$ that can be fired and we have $\widehat{\mathcal{M}} = \widehat{In}$. Similarly, we have $\mathcal{M} = In$. As $\widehat{In}$ is the extension of $In \subseteq \mathcal{P}$ with a set of places $P^A \subsetneq \mathcal{P}$, we have $In = \widehat{In} \cap \mathcal{P}$ and the claim holds.

Now assume the claim has already been established for a transition sequence $\vec{t} = \langle \widehat{t_1}, \ldots, \widehat{t_n} \rangle$ with $n \geq 1$, i.e., if $\widehat{In}\ [\widehat{t_1}\rangle\ \ldots\ [\widehat{t_n}\rangle\ \widehat{\mathcal{M}}$, there exists a marking $\mathcal{M} \in \mathcal{R}(\mathcal{N}^{\mathcal{G}})$ such that $In\ [t_1\rangle\ \ldots\ [t_n\rangle\ \mathcal{M}$ and $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$. We have to show that if $\widehat{\mathcal{M}}\ [\widehat{t_{n+1}}\rangle\ \widehat{\mathcal{M}'}$ for $\widehat{t_{n+1}} \in \widehat{\mathcal{T}}$ and $\widehat{\mathcal{M}'} \in \mathcal{R}(\mathcal{N}^{\widehat{G}})$, there also exists a marking $\mathcal{M}' \in \mathcal{R}(\mathcal{N}^{\mathcal{G}})$ such that $\mathcal{M}\ [t_{n+1}\rangle\ \mathcal{M}'$ and $\mathcal{M}' = \widehat{\mathcal{M}'} \cap \mathcal{P}$. We make a case distinction based on t_{n+1}:

If t_{n+1} is concurrency-preserving in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 neither extends $pre(t_{n+1})$ nor $post(t_{n+1})$. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1}) \subseteq \mathcal{P}$ and $post(\widehat{t_{n+1}}) = post(t_{n+1}) \subseteq \mathcal{P}$. Since $\widehat{\mathcal{M}}\ [\widehat{t_{n+1}}\rangle\ \widehat{\mathcal{M}'}$, we have $pre(\widehat{t_{n+1}}) \subseteq \widehat{\mathcal{M}}$ and $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. By induction hypothesis, we have $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$ and thus, because $pre(\widehat{t_{n+1}}) \subseteq \mathcal{P}$, also $pre(t_{n+1}) \subseteq \mathcal{M}$. Firing t_{n+1} in $\mathcal{N}^{\mathcal{G}}$ results in $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. Because $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$, $pre(t_{n+1}) = pre(\widehat{t_{n+1}})$, and $post(t_{n+1}) = post(\widehat{t_{n+1}})$, we have $\mathcal{M}' = \widehat{\mathcal{M}'} \cap \mathcal{P}$, proving the claim.

Assuming t_{n+1} is consuming in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 only extends $post(t_{n+1})$ with a set of places P^A. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1}) \subseteq \mathcal{P}$ and $post(\widehat{t_{n+1}}) = post(t_{n+1}) \cup P^A$. Since $\widehat{\mathcal{M}}\ [\widehat{t_{n+1}}\rangle\ \widehat{\mathcal{M}'}$, we have $pre(\widehat{t_{n+1}}) \subseteq \widehat{\mathcal{M}}$ and $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. By induction hypothesis, we have $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$ and thus, because $pre(\widehat{t_{n+1}}) \subseteq \mathcal{P}$, also $pre(t_{n+1}) \subseteq \mathcal{M}$. Firing t_{n+1} in $\mathcal{N}^{\mathcal{G}}$ results in $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. Because $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$ and $pre(t_{n+1}) = pre(\widehat{t_{n+1}})$, we have $\mathcal{M} \setminus pre(t_{n+1}) = \widehat{M} \setminus pre(\widehat{t_{n+1}}) \cap \mathcal{P}$. Because $P^A \subsetneq \mathcal{P}$ and $post(\widehat{t_{n+1}}) = post(t_{n+1}) \cup P^A$, we have $post(t_{n+1}) = post(\widehat{t_{n+1}}) \cap \mathcal{P}$ and thus, $\mathcal{M}' = \widehat{\mathcal{M}'} \cap \mathcal{P}$, proving the claim.

If t_{n+1} is generating in $\mathcal{N}^{\mathcal{G}}$, Algorithm 6 only extends $pre(t_{n+1})$ with a set of places P^A. Thus, we have $pre(\widehat{t_{n+1}}) = pre(t_{n+1}) \cup P^A$ and $post(\widehat{t_{n+1}}) = post(t_{n+1})$. Because $P^A \subsetneq \mathcal{P}$, $pre(t_{n+1}) = pre(\widehat{t_{n+1}}) \cap \mathcal{P}$ also holds. Since $\widehat{\mathcal{M}}\ [\widehat{t_{n+1}}\rangle\ \widehat{\mathcal{M}'}$, we have $pre(\widehat{t_{n+1}}) \subseteq \widehat{\mathcal{M}}$ and $\widehat{\mathcal{M}'} = \widehat{\mathcal{M}} \setminus pre(\widehat{t_{n+1}}) \cup post(\widehat{t_{n+1}})$. By induction hypothesis, we have $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$ and thus, because $pre(t_{n+1}) = pre(\widehat{t_{n+1}}) \cap \mathcal{P}$, also $pre(t_{n+1}) \subseteq \mathcal{M}$. Firing t_{n+1} in $\mathcal{N}^{\mathcal{G}}$ results in $\mathcal{M}' = \mathcal{M} \setminus pre(t_{n+1}) \cup post(t_{n+1})$. Because $\mathcal{M} = \widehat{\mathcal{M}} \cap \mathcal{P}$, $pre(t_{n+1}) = pre(\widehat{t_{n+1}}) \cap \mathcal{P}$, and $post(t_{n+1}) = post(\widehat{t_{n+1}})$, we have $\mathcal{M}' = \widehat{\mathcal{M}'} \cap \mathcal{P}$, proving the claim. □

4.4 Preserving the Causality

As shown in the previous section, the game created by the algorithm presented in this thesis preserves the flow of the original game. In addition, the marking reached in the extended net is the same as in the original net plus some added places that do not occur in $\mathcal{P}$. To show that the algorithm also preserves the causality, it suffices to show that the causal past of the additional places in the preset of a generating transition is a subset of the combined causal pasts of the other places in the same preset.

Lemma 4.6. Let $t \in \mathcal{T}^U$ be a consuming transition in the unfolding $(\mathcal{G}^U, \lambda)$ of $\mathcal{N}^{\mathcal{G}}$, i.e., the preset of its extended version $\widehat{t} \in \widehat{\mathcal{T}}^U$ is $pre(\widehat{t}) = pre(t) \cup P^A$ for a non-empty set of places $P^A \subsetneq \mathcal{P}^U$. For every place $\widehat{p} \in P^A$, we have

$$past(\widehat{p}) \subseteq \bigcup_{p \in pre(\widehat{t}) \cap \mathcal{P}} past(p).$$

Proof. By case distinction based on $\lambda(t)$. If $\lambda(t)$ is not in any cycle, we have $past(\widehat{p}) = \emptyset$ and the claim holds.

Else, $\lambda(t)$ is contained in a cycle. If $\lambda(\widehat{p}) \in \widehat{In}$ and $\lambda(\widehat{t})$ has not been fired before, we have $past(\widehat{p}) = \emptyset$ and the claim holds. Otherwise, there exists a producing transition $t' \in \mathcal{T}^U$ such that its extended version $\widehat{t'}$ lies in the preset of $\widehat{p}$ and therefore $\widehat{t'} \leq \widehat{t}$. Because of Lemma 4.5, we also have $t' \leq t$. If $t' \leq t$, there must exists a place $p' \in pre(t)$ such that $t' \leq p'$, i.e., p' knows everything that happened before firing t' and of the firing of t' itself. The same holds for the corresponding $\widehat{p'} \in pre(\widehat{t})$ and $\widehat{t'}$. Further, all places in $post(\widehat{t'})$ know this, and nothing more. Thus, we have $past(\widehat{p}) \subseteq past(\widehat{p'})$. As $\widehat{p'} \in pre(\widehat{t}) \cap \mathcal{P}$, the claim is proven. □

4.5 Existence of a Winning Strategy

Unfoldings represent the behaviours of a net and describe the possible causal relationships. Removing certain system transitions in the unfolding of a Petri game yields a strategy for the system players. As shown in the two previous sections, the extended game preserves both flow and causality of the original game. The unfolding of the extended game is the same as the one for the original game plus some places representing the places that were added in Algorithm 6. These places are all part of the environment, i.e., we cannot restrict more transitions as before because purely environmental transitions stay purely environmental in the extended game and cannot be deleted by the system strategy. Furthermore, no player in the extended game knows more than its counterpart in the original game. Thus, a winning strategy for

the system in the original game can easily be translated to a winning strategy for the system in the extended game by deleting the exact same transitions.

4.6 Safeness

Transitions are either contained in a cycle or not. If a transition is not in a cycle and not concurrency-preserving, we only extend its pre- or postset, but never both. Further, we only put the added places occuring in presets to the initial marking. Therefore, it is not possible that places added to such transitions contain more than one token. The safeness for places that are added to transitions contained in cycles follows directly from the assumption stated in Section 3.8.

5 Related Work

Deciding Petri games is related to the control problem for asynchronous automata. A (deterministic, finite) *asynchronous automaton*, as introduced by Zielonka in [6], is defined as a set of processes $\mathbb{P} = \{P_1, \ldots, P_n\}$, where each P_i consists of a finite set of states S_i, a finite alphabet Σ_i, and an initial state $s_i^0 \in S_i$. While we assume that the states of the different processes are disjoint, the same symbol can occur in multiple alphabets. Whenever a symbol a is read, all processes P_i with $a \in \Sigma_i$ synchronize on a and change their states, whereas all other processes remain in their current state. Similar to Petri games, where tokens exchange their causal past on joint transitions, all processes that share an action a exchange complete information about their causal past when a is read.

Given a partioning of the actions into a set of controllable system actions and a set of uncontrollable environment actions, the goal of the control problem is to synthesize a controller that can restrict certain system actions to satisfy a given specification. In the so-called *action-based* approach, studied in [3], the controller is another asynchronous automaton over the same alphabet. Based on the causal past of the participating processes, it can refuse to execute a certain action. This differs from our model of Petri games, where we synthesize controllers for certain tokens instead of transitions. When restricting the dependencies on the actions, the authors show that the action-based control problem is decidable.

Another variant of the control problem, that is more related to our model, is called *process-based*. In this setting, every process can decide, based on its own causal past, whether a certain action is enabled or not. Like before, we can only disable the controllable system actions, so environment actions can occur whenever the corresponding processes can execute the same environment action. This stands in contrast to Petri games, where we split the processes into controllable and uncontrollable parts instead of the transitions. The authors of [4] show that the process-based control problem is decidable for reachability objectives on asynchronous automata with a tree architecture, i.e. every process can only communicate with its parent, its children, and the environment.

6 Conclusion

This section provides a summary of the presented work. We recap the development of the algorithm and the proofs of its desired properties. Moreover, we present possible improvements in future work.

6.1 Summary

The starting point of our challenge to construct an algorithm that makes Petri games concurrency-preserving were two small examples of Petri nets consisting of only a generating and consuming transition, respectively. Solving these was simply done by extending their pre- or postset with fresh places such that their sizes are equal.

Petri games can possibly run forever. Therefore, the simple extensions of pre- and postset were not enough. It is required that the newly added places were filled with tokens multiple times. To this end, we analyzed cycles in the reachability graph, i.e., markings that repeat themselves when firing a certain sequence of transitions. Using these cycles, we were able to use added places and tokens repeatedly.

Different choices might lead to different cycles. If the different cycles share one or more transitions, it is required to put the same added place in the pre- and postsets of multiple transitions. However, this sharing of added places only works if the cycles are always in conflict and can not occur concurrently.

Another problem while transforming Petri games was to preserve the causality. Cycles can consist of the interleaving of multiple independent parts in the net and a concluding synchronization between all involved processes. Using only the transitions to solve cycles put tokens including their causal past from one part into another and altered the information flow. However, if we only use tokens in our causal past for redirection, we are able to keep tokens in their respective parts of the net.

Except from adding places and flows to the net, we discovered that it is required to preprocess some Petri nets before making them concurrency-preserving. This is the case for nets where the same transition can be fired more than once but not infinitely often or must be fired multiple times to complete a cycle. Preprocessing is also needed for nets where cycles can be entered via different markings. To express different entry points of a cycle and the multiple firing of transitions, we assume the given Petri net to be partially unfolded such that these properties are made explicit.

Using the fact that every cycle has a unique entry point, we are able to correctly extend the initial marking. Whenever there are not enough tokens

to fire a generating transition in the cycle, we add the needed number of places to the preset. This is possible because the number of available tokens before firing a transition is always the same. Places that are not part of the initial marking will contain a token when needed due to redirection. For generating transitions outside of cycles, we add all fresh places to the initial marking.

We proved the correctness of the algorithm by first showing that it indeed makes the given Petri game concurrency-preserving. Furthermore, we provide proofs that the extended game preserves the flow of the original game. This is done by showing that both games enable the same transition sequences. The reached marking of the extended game is the same as in the original game with some tokens in the newly added places.

Together with the fact that the added tokens have a smaller causal past than the other tokens in the preset of the same enabled transition, and that we only added environment places, we were able to show that we can use the same winning system strategy from the original game in the extended game.

6.2 Future Work

The definition of the term *concurrency-preserving* used in this thesis does not take into account the partitioning of the places in a Petri game. Therefore, we chose to extend the net only with environment places. Although this makes it easier for us to prove the existence of a winning strategy for the system players in the extended game, it contradicts the purpose of Petri games. The goal of Petri games is to find an implementation for a distributed systems. Each system player in the game represents a component of the system and we want to infer an implementation for each of them. Thus, it is counterintuitive if a system player can suddenly become a player for the environment, but it happens every time in the extended game when firing a transition that consumes a system token. Therefore, it is desirable to extend the game in such a way that players do not change their team during the course of the game. The challenge that arises with this problem is to prove the existence of a winning strategy. For example consider a cycle that consists of two transitions. The first transition t_0 has one system and one environment player in its preset and consumes the system player. The second transition t_1 produces this system player again. Using Algorithm 6 and taking into account the team of the consumed token adds a system place between the two transitions. However, this transforms the actually purely environmental transition t_1 into a system transition. Assuming there exists another purely environmental transition t_2 that can be fired instead of t_1, the newly added system player can refuse to take part in firing t_1, violating condition (4).

Algorithm 6 is only defined for Petri games that are of the form described in Section 3.6, i.e., it might be required to preprocess a given Petri game first before making it concurrency-preserving. Therefore, it is required to provide an algorithm that handles the described preprocessing. Transitions that can be fired multiple times but infinitely often could be detected by analyzing the reachability graph. The same holds for transitions that must be fired multiple times in order to complete a cycle. A cycle explicitly lists all transitions in the correct firing order, i.e., we just have to count the occurence of identical transitions. After identifying the transitions, one has to copy the corresponding subsequent nodes of the net. The detection of multiple entry points to a cycle or a different number of available redirectable tokens seems more challenging. Due to different interleavings and different paths leading to a cycle, it is hard to detect all of these cases in the reachability graph.

Another assumption regarding the given Petri game is stated in Section 3.8. For a given cycle that might occur with different interleavings, we assume that the input of `SolveCycle` is in an order that can be handled by our algorithm. Although we can detect whether a given cycle produces an unsafe net by analyzing the reachability graph of the subnet constituted by the cycle, it remains to show that such an order always exists. Further, it would be desirable to find this correct interleaving before solving interleavings that lead to false extensions.

When extending transitions that do not occur in cycles, we only use the added places once. Consider a net that alternately consumes and generates a token k times. Its initial marking consists of two places, i.e., the maximal number of tokens is 2. The algorithm presented in this thesis extends every transition in the net by one place and adds every place that is added to a preset to the initial marking. This leads to an extended net with $k + 1$ tokens. Instead of adding a new token for every of the generating transitions, we could redirect tokens like we do it in cycles and minimize the number of tokens in the net. Similarly, when there is a place such that there are two enabled transitions in its postset that generate one token, we add one place to each of these transitions. However, sharing added places between these transitions decreases the number of tokens again.

References

[1] Bernd Finkbeiner and Paul Gölz. Synthesis in distributed environments. *CoRR*, abs/1710.05368, 2017.

[2] Bernd Finkbeiner and Ernst-Rüdiger Olderog. Petri games: Synthesis of distributed systems with causal memory. *Information and Computation*, 253:181 – 203, 2017. GandALF 2014.

[3] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, pages 275–286, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[4] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Asynchronous games over tree architectures. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'13, pages 275–286, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13(1):85 – 108, 1981. Special Issue Semantics of Concurrent Computation.

[6] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 21(2):99–135, 1987.