Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Bounded Synthesis of Petri Games with True Concurrency Firing Semantics

submitted by
Niklas Oliver Metzger

submitted on
October 20th, 2017

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 are an extension of Petri nets and synthesize causality-based distributed systems. A Petri game is a multiplayer game between the environment and the system, which has the goal to not let the environment reach bad situations. One way to find winning strategies for the system is based on bounded synthesis. In bounded synthesis, we incrementally increase the size of the strategy and can focus on small ones, which are more efficient to implement.

To solve these games, we encode the behaviour of the game and the existence of a winning strategy as a quantified boolean formula (QBF). Since the search space grows exponentially, we want to reduce the size of the encoded formula. The sequential encoding describes the flow of the Petri game by interleaving the fired transitions and checking the winning conditions for a given bound.

In this bachelor's thesis, we introduce a true concurrent firing semantics on the transitions of the Petri game and extend the existing encoding to the new semantics, such that multiple transitions can be fired in one step of the encoding. With that, we can potentially speed-up the solving of the QBF, while not losing expressiveness. We prove the correctness of our approach and implement the result. A benchmark evaluation against the sequential encoding is provided.

Acknowledgement

I am very grateful to Prof. Bernd Finkbeiner for giving me the opportunity to work on such an interesting topic, fulfilling my research interests. I want to thank my advisor Jesko Hecking-Harbusch for the guidance and help to complete this thesis. The meetings and discussions were always joyful and refreshing. Moreover, I would like to thank Prof. Bernd Finkbeiner and Prof. Holger Hermanns for reviewing this thesis. Furthermore, I also place on record, my sincere thank you to my family and friends for their ongoing support.

Contents

1 Introduction

Correctness is a mandatory property of real life systems. Systems that are not dependable constitute a risk and therefore have to be recognized before their launch. Multiple theoretical attempts in proving this correctness are developed and become an incrementing part of computer science. Two fundamental ways of proving correctness are the common areas of research. On the one hand, there is verification. Verification takes a specification that expresses a mandatory behaviour and analyzes an existing implementation of the system, deciding if that implementation satisfies the specification. With that strategy, finding a false behaviour in the implementation always leads to debugging and verifying the correctness again. On the other hand is synthesis, which provides a correct per definition implementation to a given specification, if one exists. Since synthesis creates a correct behaviour for a system, the implementation step can easily be done by software, which frees the programmer from needless work.

Bounded synthesis [10] sets a bound on the size of the evaluated implementations and increases the bound if no specification satisfying implementation can be found. With this approach we can focus the search on small implementations but lose recognition of a non synthesizeable system.

Petri nets [16] model the interaction of processes in distributed systems by defining the concurrent behaviour of process representing tokens in the net. Tokens move through the net and are able to synchronize with each other or progress locally without interacting with other processes. The possible behaviour of tokens defines the flow of a net.

Petri games [9] are a game based extension of Petri nets, introducing system players and environment players as tokens in the net. The system tokens are controllable and react to the behaviour of the non-deterministic environment players. The system wins the game if no bad situation in the Petri game can be reached and the environment wins the game if the tokens are able to force a bad situation. The information contained by a token is its causal past in the net, i.e. the previous flow by this token. Petri games provide infinite flows in finite Petri games and contain loops of the flow. To maintain causal past, the Petri game is unfolded after every loop such that a distinct history of the tokens can be extracted. To maintain the finiteness of the Petri game, we can bound the unfolding and define the unfolding depth of the Petri game. Two synchronizing tokens share their previous flow in the Petri game and have the knowledge of both flows afterwards. Based on this knowledge, the system players try to avoid bad situation in the Petri game. The *synthesis* problem of Petri games is the existence and automatic construction of a winning strategy for the system players.

Bounded synthesis of Petri games [6] is the encoding of a bounded winning strategy for a Petri game to a quantified boolean formula (QBF). We bound the number of transitions fired in one run to n and the unfolding of the Petri game to b, which copies places and their following transitions with respect to the bound. Limiting the number of transitions corresponds to a finite flow of the game. To symbolize an infinite flow with a finite number of transitions, we have to find a loop in the bounded unfolding within at most n transitions fired. Finding no winning strategy leads to increasing the bound of n and b.

The bounded synthesis approach is implemented as a prototype tool [7]. The bounded winning strategy for a given Petri game and its bounds n and b is encoded to a *qcir* [15] file and solved by a QBF solver. In case of finding a winning strategy, which corresponds to SAT, the winning strategy for the game is extracted.

This bachelor's thesis presents a true concurrency firing semantics for Petri games. The new semantics allows multiple transitions to fire in one iteration step and maintains the flow of individual tokens in the underlying Petri net. The goal of the semantics is to decrease the necessary bound n to find a winning strategy of a Petri game. We define sets of transitions that are dependent of each other and define the true concurrent flow based on these sets. We introduce an encoding of the firing semantics to QBF, substituting the encoding of the flow in the existing formula and adjusting the bounded synthesis approach to true concurrent firing. We prove the correctness of the encoding and the true concurrency semantics by proving the equivalence of the token's causal past in both firing semantics.

The new formula is implemented as an extension to the prototype tool. We present a new benchmark family and provide results of the benchmark families for Petri games [7], comparing both approaches.

This thesis is structured as follows. A general overview of bounded synthesis, Petri nets, Petri games, and QBF is given in Section 2. We introduce the bounded synthesis of Petri games approach in Section 3 and present the true concurrent firing semantics in Section 4 . We extend the bounded synthesis approach to the new semantics in Section 5 and prove the completeness of the new formula. We proof the correctness of the true concurrency firing semantics in Section 6. The implementation of the new approach is shown in Section 7. Section 8 presents the benchmark results of our new approach compared to the sequential approach. Related work is discussed in Section 9 and we conclude this thesis in Section 10.

2 Background

In this chapter we introduce bounded synthesis and the basic concepts of Petri nets. Petri games, which extend Petri nets, are introduced in Section 2.3, followed by a short overview of quantified boolean formulas. The combination of all introduced concepts are explored in Section 3.

2.1 Bounded Synthesis

As mentioned before, *synthesis* constructs a per definition correct system and frees the programmer from implementation work. The reason why *verification* is a much more investigated approach is runtime. Pnueli and Rosner proved, that *distributed systems are hard to synthesize* [17] and it is known, that the general synthesis problem is 2EXPTIME-complete and even worse for distributed systems. It actually is undecidable for simple distributed architectures [10]. As opposed to this, verification is in PSPACE, both, in size of the specification and in size of the system. This yields to only considering verification for correctness analysis. The reason why synthesis gets more attention these days is the research of finding subcategories of synthesis, that can be solved in smaller complexities. Pnueli and Rosners conclusion is based on a worst case approach of synthesis, which is not necessary for all synthesis problems. The synthesis of a restricted Petri game is EXPTIME-complete [9], proven by Finkbeiner and Olderog, which is one example for a smaller upper bound of the complexity.

One reason for the complexity of the synthesis problem is the fact, that the *size of the implementation* cannot be restricted as in verification. Therefore, huge implementations are synthesized as well as small implementations, leading to memory and runtime problems. To solve this issue, Finkbeiner and Schewe introduced the *bounded synthesis* [10]. The key idea of bounded synthesis is the following: We add a new parameter to the problem, which bounds the size of the considered implementations. If a solution is found, the problem is solved, if not, the parameter for the implementation size is increased and the solving is started again. The obvious drawback of this approach is the loss of decidability. If a specification is not satisfiable, we will not recognize it and run out of memory or in a timeout, affected by increasing the bound after every failed synthesis. We can focus the search on small implementations, which is an advantage for runtime and also practical aspects, e.g. minimizing the required communication flow of processes in a distributed system.

2.2 Petri Nets

Petri nets [16] are a graphical model to symbolize interaction between independent components of distributed systems. The components can synchronize with each other which defines the flow of a Petri net. A Petri net consists of *places* and *transitions*, which are circles and squares respectively. Places can have *tokens* and each token represents one process of the whole system. In this thesis we only consider safe Petri nets, which bound the maximal number of tokens per place to one. Transitions connect places by incoming arrows from places and outgoing arrows to places. A transition is *enabled* if all places with an arrow to the transition possess a token. If a transition is enabled, it can be *fired.* Firing a transition deletes all tokens in the places preceding the transition, and adds one token to the places after the transition. The sequential firing of transitions defines the flow of the Petri net.

Definition 2.1 (Petri Nets)**.**
A Petri net $\mathscr{N}$ is a 4-tuple $(\mathscr{P}, \mathscr{T}, \mathscr{F}, In)$, where:

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

The set $\mathscr{P}$ and the set $\mathscr{T}$ are disjoint. The pair of the form $(p, t) \in \mathscr{F}$ with $p \in \mathscr{P}$ and $t \in \mathscr{T}$ means that there is an arrow from p to t, whereas $(t, p) \in \mathscr{F}$ symbolizes an arrow from t to p. The flow relation describes the possible motion of tokens in the net.

The initial marking In is the state of the Petri net before the first transition is fired. For all places $p \in In$, one token resides in the place, for all places $p' \notin In$, no token is possessed by the place. From the initial marking ongoing, all possible distributions of tokens on places can be described as *markings* $M_i \subseteq \mathscr{P}$. The index i is the number of transitions fired before this marking was reached, starting from M_0, which is exactly In. Every place in a set M_i contains a token, all other places are empty. We call a sequence of transitions that are fired sequentially to reach the marking M_i a (sequential) *run*: $\pi = \langle t_1, ..., t_i \rangle$. All runs from In to M_i is the set $\{\langle t_1, ..., t_i \rangle | In \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_i} M_i\}$. The set of reachable markings $\mathcal{R}(\mathscr{N})$ of $\mathscr{N}$ are all markings that can be created by firing transitions, starting from In.

2.2.1 Presets and Postsets

The ongoing topic of this thesis yields the definition of the following sets: The preset *pre* of a transition t, denoted by ${}^\bullet t$, is defined by $pre(t) = \{p \in \mathscr{P} \mid (p, t) \in \mathscr{F}\}$. This corresponds to the set of places that point to the transition. According to the previous definition, we declare the postset *post* of a transition t with $post(t) = \{p \in \mathscr{P} \mid (t, p) \in \mathscr{F}\}$ and denote it by $t^\bullet$. The postset contains all places the transition t points to.

The expressivism of preset and postset can be extended to places, wherefore we use the definitions $pre(p) = \{t \in \mathscr{T} \mid (t, p) \in \mathscr{F}\}$ and $post(p) = \{t \in \mathscr{T} \mid (p, t) \in \mathscr{F}\}$ for the *preset* and *postset* respectively. The notations ${}^\bullet p$ and $p^\bullet$ are used equivalently to ${}^\bullet t$ and $t^\bullet$.

2.2.2 Firing Transitions

The flow relation defines which condition has to be fulfilled to fire a transition. For a pair of the form $(\mathscr{P}, \mathscr{T})$, the place has to contain a token to enable the transition, which will be consumed by firing, whereas the form $(\mathscr{T}, \mathscr{P})$ defines that if the transition is fired, a token is produced in this place. We say that a transition is enabled if all places in the *preset* of the transition contain a token. An enabled transition can be fired, which consumes all tokens in the preset and adds one token to all places in the postset of the transition. In case of safe nets, exactly one token resides in ${}^\bullet t$, the preset of the transition. It is also certain for safe nets, that before firing an enabled transition, no place in the postset of t contains a token.

If for a transition t holds, that $|{}^\bullet t| \geq 1$, we say that all tokens in ${}^\bullet t$ synchronize by firing t.

2.2.3 Petri Net Example

The Petri net in Figure 1 represents a simple net with an finite flow. In Figure 1a the initial state of the net is pictured. The initial marking is $\{A, B, C\}$ we have 5 places $\{A, B, C, D, E\}$ and two transitions $\{tAB, tAC\}$. The flow relation is defined by $\{(B, tAB), (A, tAB), (A, tAC), (C, tAC), (tAB, D),$ $(tAC, E)\}$. Both transitions, tAB and tAC are enabled and either one can be fired. One can see the enabledness by looking at the presets ${}^\bullet tAB$, which is $\{B, A\}$ and ${}^\bullet tAC$, with $\{A, C\}$. All places in the presets possess a token. We have two possible flows of this net corresponding to the choice of A, deciding which transition in its postset $A^\bullet = \{tAB, tAC\}$ will be fired. In Figure 1b the processes residing in A and C synchronized in transition tAC and communicated. The tokens in place A and C are consumed and the token in E was produced. After tAC was fired, no transition is enabled and therefore

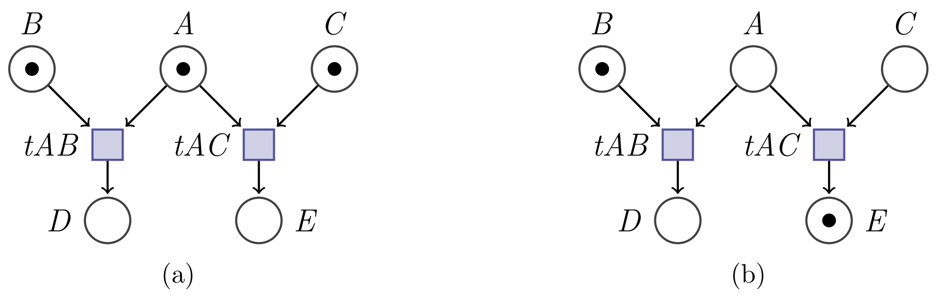

Figure 1: A Petri net modelling a choice of communication. Pictured at the initial marking with two enabled transitions in 1a and after firing transition tAC in 1b.

the game terminated. Note that the selection of enabled transitions to be fired is non-deterministic.

2.3 Petri Games

Games are often used to describe the interaction of a computer system with its environment. Since games can be won or lost, the contestants, i.e. the system and environment both try to win. Winning conditions for games are specifications that enforce correct behaviour of the system, leading the environment player to *act* in a way that these specifications are invalidated and the system to *react* to the environment. By analyzing the realizability of the game, one can synthesize a winning strategy for a player. In this chapter, we will take a closer look at Petri games and their winning strategies.

2.3.1 Definition of Petri Games

Petri games [9] as introduced by Finkbeiner and Olderog are a multiplayer game based extension to Petri nets [16]. They combine the distributed characteristics of Petri nets with a game and enable the possibility of modelling a real life system that can be synthesized.

Every Petri game has an underlying Petri net defining the flow and the reachable markings. To represent a game between the system and the environment, the places of the net are divided into system- and environment places, $\mathscr{P}_S$ (grey) and $\mathscr{P}_E$ (white) respectively. If a system place contains a token, it is a system token, otherwise an environment token. While having full control over the decisions the system makes, we cannot control the behaviour of the environment and therefore have to be able to react to all possible decisions. Modelling this in a Petri game, we can always decide which transition will be fired starting from only system places, and in contrast, environment tokens can choose between all enabled transitions in their places postset. The behaviour of transitions with environment and system places in their presets is described in Section 2.3.3. We only consider finite Petri games, declaring $|\mathscr{P}|$ and $|\mathscr{T}|$ to be finite.

Definition 2.2 (Petri Games)**.**
A Petri game $\mathscr{G}$ is a 6-tuple $(\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, In, \mathscr{B})$, where:

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

The game $\mathscr{G}$ has the underlying Petri net $\mathscr{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In)$, with $\mathscr{P} = \mathscr{P}_S \cup \mathscr{P}_E$.

The definitions for transitions $\mathscr{T}$, the flow relation $\mathscr{F}$ and the initial marking In are equivalent to the ones in Petri nets.

The winning condition of the game is a safety condition on the basis of the set $\mathscr{B}$ of *bad places*. The system wins a run of the game by avoiding the flow to reach a bad place $b \in \mathscr{B}$ and the run is lost for the system if at some point a token resides on a place b. The system wins the game by winning all runs of the game, the environment wins with a single winning run. Winning system strategies change the possible behaviour of system tokens by forbidding some transitions of the game, that include system places in their preset.

Petri games extend the information exchange model given by the underlying Petri net. As Petri nets illustrate the communication flow in a distributed system, Petri games even deploy causal memory. Every token knows its causal past, e.g. the transitions fired on its own previous flow. By synchronizing with other tokens, their complete *history* is exchanged and known by the tokens in the synchronizing transitions postset. One can say that either all information is shared between tokens, or none. Tokens can be understood as local players knowing their own history. Based on their causal past, they make decisions in the game directing the flow to their own benefit.

2.3.2 Unfoldings of Petri Games

We have to unfold Petri games and therefore their underlying net to represent causal past. Informally, if a token resides on a place with multiple incoming transitions, knowing the previous way through the Petri game is necessary and can't be derived from that one place. As a solution, we create a number of copies of the place according to the incoming flows.

Formally, we create an unfolding $\beta_U = (\mathscr{G}^U, \lambda)$ of a Petri game $\mathscr{G}$, where all causal memory contradictions of places in $\mathscr{G}$ have been solved by copying these places and their subsequent transitions and places. λ is a homomorphism from $\mathscr{G}^U$ to $\mathscr{G}$, mapping all places and transitions of $\mathscr{G}^U$ to their original ones in $\mathscr{G}$. There are two possible contradictions to causal memory, the one mentioned before and an incoming transition to a place in the initial marking. Reaching this place corresponds to starting an infinite flow all over again, which symbolizes a loop, losing the history of the previous run. As before, solving this creates copies of the places, implying copies of all following places and transitions. This can lead to infinite unfoldings for infinite flows.

Copying places includes duplicating their characteristics, e.g. bad, environment or system places. In the unfolding, the history of local players, i.e. tokens, is unambiguously replicable by firing previous transitions backwards. It is ensured that the local players contain the histories of all synchronized tokens and their own. The *initial markings* of $\mathscr{G}$ and $\mathscr{G}^U$ are the same and with λ one can easily derive $\mathscr{G}$ from $\mathscr{G}^U$.

A finite *sub-process* $\beta_{U'} = (\mathscr{G}^{U'}, \lambda)$ of an unfolding $\beta_U = (\mathscr{G}^U, \lambda)$ is created by deleting transitions and their subsequent places in $\mathscr{G}^U$. The homomorphism λ maps the remaining places and transitions to their corresponding ones in $\mathscr{G}$. A *sub-process* has to be finite in contrast to unfoldings that can be infinite. To still handle infinite unfoldings of loops, we stop this unfolding at some point and come back to the beginning of it.

2.3.3 Strategies

A *strategy* of an unfolded Petri game is a finite *sub-process* of the game. The strategy gets all the information of the game through the unfolding and therefore can replicate all causal dependencies of tokens. Strategies can only affect transitions including system places in their preset. The possible decisions a strategy can make are enabling a subset of possible transitions which then will be determined by the environment, blocking a transition, such that it will never be fired or not influencing the original flow of the unfolding.

A formal definition of a strategy is the following:
A *strategy* $\sigma = (\mathscr{G}^\sigma, \lambda^\sigma)$ for the system players is a sub-process for the unfolding $\beta_U = (\mathscr{G}^U, \lambda)$ of the underlying game $\mathscr{G}$, if the following conditions hold:

(S1) σ is deterministic in all places $p \in \mathscr{P}^\sigma_E$

(S2) $\forall t \in \mathscr{T}^U . {}^\bullet t \subseteq \mathscr{P}^U_E$ states that $t \in \mathscr{T}^\sigma$

(S3) $\forall t \in \mathscr{T}^U . t \notin \mathscr{T}^\sigma \Rightarrow \exists p \in {}^\bullet t \cap \mathscr{P}_S^\sigma . \forall t' \in p^\bullet . \ \lambda(t') = \lambda(t) \Rightarrow t' \notin \mathscr{T}^\sigma$

Condition S1 refers to deterministic firing of *system* place including preset transitions, restricting the local system player to not having a choice in firing. Following that, the behaviour of the system in the *sub-process* σ is equal in all possible flows of σ. Note, that there are still multiple possible flows for a strategy since *environment* decisions are *non-deterministic.*

S2 forbids the strategy to influence *environment* transitions, e.g. transitions containing only environment places in their preset. Modelling real-live systems, the environment must stay uncontrollable.

The third condition S3 states that if a transition t is forbidden by the strategy, then it is forbidden by a place p in the transitions preset, such that all instances t' of the transition t are forbidden.

A strategy satisfying the three conditions σ is *winning* for the system player, if no *bad place* $p \in \mathscr{B}$ is reachable, e.g. for all possible flows of the strategy's game $\mathscr{G}^\sigma$, no token ever resides on a bad place. The strategy is winning for the environment, if there exists a flow reaching a bad place, vice versa. Winning strategies are forced to be *deadlock-avoiding*, forbidding the system to stall the game-flow, which would lead to a false win of the game if the environment cannot reach a bad place without synchronizing with the system. To avoid this behaviour it is necessary that as long as a transition is enabled in the underlying unfolding of the strategy, a transition in the strategy has to be fired (this does not mean that every enabled transition in the underlying game has to be fired at some point, e.g. infinite safe circles that can be fired infinitely without firing other enabled transitions). Since the environment's goal is reaching a bad place, blocking the flow is not beneficial and therefore not considered. Unreachability of bad places describes a safety condition, formulating that in every possible marking of the game, none of the places belonging to markings are in $\mathscr{B}^\sigma$.

2.3.4 Bounded Unfoldings and Bounded Strategies

Deducing finite strategies from infinite unfoldings infers a problem. To solve this issue, we introduce a *bound* $b : \mathscr{P} \to \mathbb{N}$ which assigns a natural number to every place of the game. All places are copied as often as the assigned number distinguishes. A *b-bounded unfolding* of a game $\mathscr{G}$ is the pair $(\mathscr{G}^b, \lambda)$, such that the unfolding respects the flow of the game. To create finiteness, at some point of the unfolding loops are not unrolled any more, such that the flow continues infinitely but on a finite number of places. Notice that finding bounded unfoldings is far from trivial and not part of this thesis. In the following, we assume to have a bounded unfolding by construction.

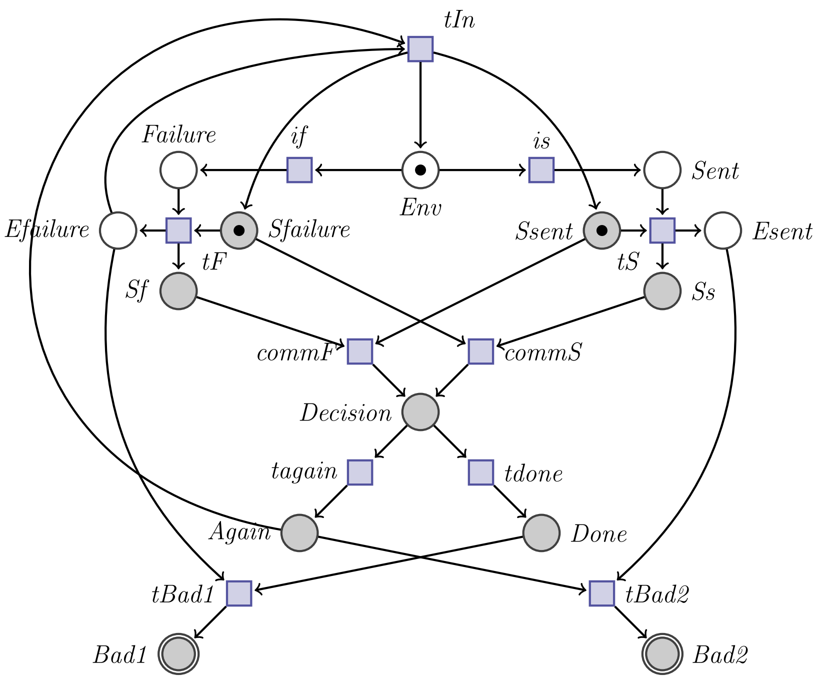

Figure 2: A Petri game simulating a simple communication protocol with one environment token and two system tokens in the initial marking.

A *b-bounded strategy* is a finite game $\mathscr{G}^f$ generated by restricting the flow of a *b-bounded unfolding* $\mathscr{G}^b$, meaning that $\mathscr{F}^f \subseteq \mathscr{F}^b$. All unreachable places and transitions are not included in the b-bounded strategy's places and transitions.

2.3.5 Petri Game Example

We now introduce our running example for this thesis in Figure 2. The Petri game models a simple communication protocol with one sender and one receiver. The only task of this protocol is sending an undefined message until it is received, and if not, repeat the sending. In the initial marking, there is one environment token in *Env* and two system tokens in *Sfailure* and *Ssent*. Two transitions are enabled, *if* and *is*, representing a failure in the message delivery and a sending completion respectively. After a failure, the system and the environment synchronize in *tF* and since the history of the consumed tokens are combined in the produced token, tokens in *Efailure* and *Sf* *know* the failed delivery. After synchronizing with the environment, the

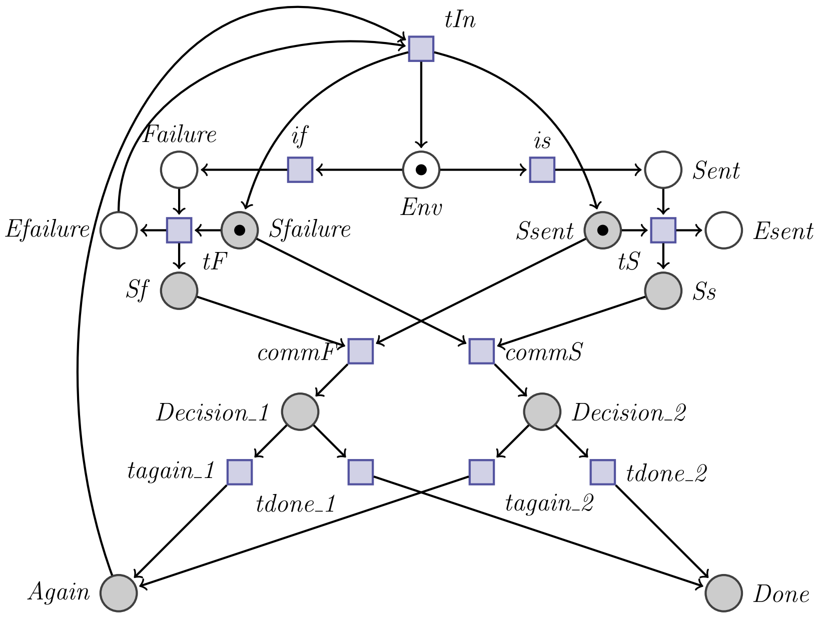

Figure 3: The bounded unfolding of the Petri game in Figure 2 with bound 2 for *Decision* and bound 1 for all other places.

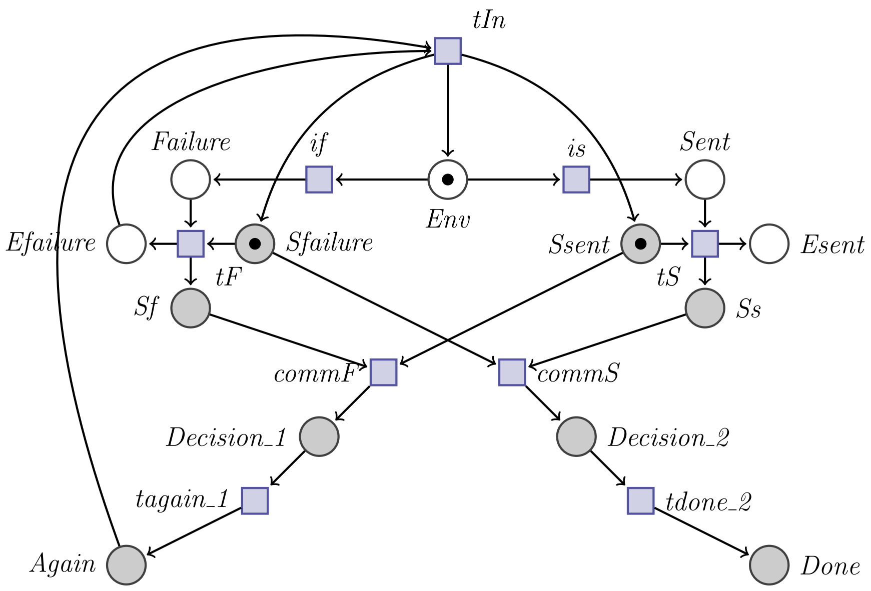

Figure 4: A bounded winning strategy for the system players for the Petri Game in Figure 2

system token in *Sf* has to communicate the failure with the local player in *Ssent* by firing *commF*. The new system token in *Decision* now can decide between starting again with *tagain* and terminating with *tdone*. The flow of successful sending corresponds to the flow described before but taking *is* first.

The bad places $\mathscr{B} = \{Bad1, Bad2\}$ are reached after the markings $(Esent, Again)$ and $(Efailure, Done)$. The transitions $tBad1$ and $tBad2$ each are the only firing option as soon as they are enabled. $Bad1$ corresponds to recognizing a failure although terminating and $Bad2$ intercepts the behaviour of delivering the message successfully and sending it again afterwards.

We cannot find a winning strategy for this Petri game. The system's interesting decision appears by reaching the place *Decision*. It must forbid firing either *tagain* or *tdone* to satisfy determinism. By deleting *tagain*, the bad place after $(Efailure, Done)$ is reachable and without *tdone*, the environment can win by reaching the bad place following $(Esent, Again)$. The token in *Decision* loses the information about the causal history of the game. The local system player cannot see which transition was fired, *commS* or *commF*, and therefore cannot find a winning strategy. To restore the causal history of the game, we have to unfold it.

The unfolding of the Petri game is displayed in Figure 3. It is a minimal bounded unfolding for the game, which corresponds to bound 2 for place *Decision* and bound 1 for all other places. Notice that transitions *tagain* and *tdone* are also copied. The flow of the unfolding changed in a way, that *Decision_1* is reached after having a failure and *Decision_2* is reached after a completed sending. There are now different *Decision* places representing different histories of the system token. Unfolding the game leads to the strategy depicted in Figure 4, which deletes *tdone_1* and *tagain_2*. Since we cannot reach *Done* after a failure and *Again* after sending, no bad place is reachable, the strategy is deterministic (since we deleted transitions causing non-determinism) and deadlock avoiding (the game only terminates naturally or runs infinitely) and therefore winning for the system. In the following we will refer on this example and extend it to multiple receivers.

2.4 Quantified Boolean Formulas

Many problems in computer science are encoded to boolean formulas. Since solvers for *satisfiability* gained performance in the last years (e.g.[13]) and varying solving strategies are developed, they are applicable for many tasks in practice. We provide a short overview over the main definitions of QBF.

2.4.1 Definition

Quantified boolean formulas (QBF) are an extension of boolean formulas with quantifiers. QBFs are propositional formulas over a finite set of variables X combined with the logical operations negation ($\neg$), and ($\wedge$), or ($\vee$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$). All transformations are adopted from propositional logic. The possible values for all variables, the domain, is $\mathbb{B} = \{0, 1\}$. The syntax of boolean formulas is extended to the universal quantifier ($\forall$) and the existential quantifier ($\exists$). The syntax of QBF is the following:

Definition 2.3. QBF Syntax

$$\begin{aligned} \phi \; &::= x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \mathit{true} \mid \mathit{false} \\ \Phi \; &::= \phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \exists x.\Phi \mid \forall x.\Phi \end{aligned}$$

We call Q an unspecified quantifier with $Q \in \{\forall, \exists\}$. We only consider formulas in *prenex* form $Q_1 x_1 ... Q_n x_n.\phi$, with $x \in X$. All variables of the form $Qx.\phi$ are called *bound*, all others are *free*. We use the quantification over sets of variables instead of quantification over variables with $\forall x_1.\forall x_2...\forall x_n.\phi = \forall X.\phi$ and $\exists x_1.\exists x_2...\exists x_n.\phi = \exists X.\phi$ with $X = \{x_1, ..., x_n\}$. The *assignment* of a variable set X is a function $\alpha : X \rightarrow \mathbb{B}$ mapping each variable of the set to 0 or 1, false or true respectively. A *set of assignments* of a set of variables X is represented by $\mathcal{A}(X)$. We describe the assignment x as the set of variables assigned to true with $x \subseteq X$. The satisfiability problem of QBF, i.e. the question if there exists a variable assignment validating to true, is PSPACE-complete [3] and is symbolized by $x \models \Phi$ with $x \subseteq X$ as assignment for the free variables of X.

To determine if $x \models \Phi$ or $x \not\models \Phi$ we introduce the semantics of QBF. We extend the known SAT semantics with quantifiers:

Definition 2.4. QBF Semantics
We extend the semantics of SAT with the definitions for quantifiers for an assignment $x \subseteq X$:

- For a QBF formula ϕ and a universal quantifier $\forall$ states:

$$x \models \forall x_1.\phi \Leftrightarrow x \cup x_1 \models \phi \wedge x \models \phi$$

- For a QBF formula ϕ and an existential quantifier $\exists$ states:

$$x \models \exists x_1.\phi \Leftrightarrow x \cup x_1 \models \phi \vee x \models \phi$$

2.4.2 Skolem Functions

Variables in a QBF formula can depend on the values of other variables. Exapmle 2.1 presents a formula, where the value of y is dependent on the value of x. We say that the dependency set of a existentially quantified variable y is the set of universally quantified variables X such that y is in the scope of X. We call this set $dep(y)$. In our example, y is in the scope of $\forall x$ and $dep(y) = \{x\}$. To solve the issue of dependent variable values, we introduce *Skolem functions*, that map assignments of dependencies of y to an assignment of y: $f_y = \mathcal{A}(dep(y)) \rightarrow \mathbb{B}$. Evaluating a QBF formula ϕ is equivalent to the existence of a Skolem function f_y for every existentially quantified variable such that $\{y \in Y | f_y(x \cap dep(y))\} \models \phi$ for all assignments $x \subseteq X$ with Y as the set of existentially quantified variables and X as the set of universally quantified variables.

2.4.3 QBF Example

We consider the following quantified boolean formula:

Example 2.1. Quantified Boolean Formula

$$\forall x.\exists y.(x \Rightarrow y) \wedge (x \Rightarrow \neg y)$$

The formula is UNSAT. In words, the formula describes a boolean assignment, that for all possible x we find a y, such that if x is true, y and not y hold. Because the formula includes a universal quantifier bounding x, we have to find a Skolem function for y that maps the assignment of x to the *assignment* of y. Since $x = 0$ sets both implications to true independently of y, this does not influence the choice of y. For $x = 1$, both implications depend on y, which leads to a contradiction.

Example 2.2. Quantified Boolean Formula

$$\exists x.\forall y.(x \Rightarrow y) \wedge (x \Rightarrow \neg y)$$

In the second example, the quantifiers are switched and the interpretation changed. We have to find an x, that satisfies the formula for all values of y, i.e. both implications evaluate to true. This is easily done by setting x to 0. The formula is *SAT*. Notice that no existentially quantified variable is in the scope of a universally quantified variable. Therefore no Skolem function is needed to solve the formula.

2.4.4 2-QBF

In this thesis we only consider *2-QBF* which bounds the number of quantifiers to 2. With that restriction, only formulas of the form $\forall X.\exists Y.\phi$ and $\exists X.\forall Y.\phi$ with X and Y as sets of variables are possible. QBF solvers specialized for *2-QBF* show high performances in solving formulas in comparison to general QBF solvers. Multiple problems can be formalized as *2-QBF* and do not need the full expressivity of *QBF* [1].

Finding a solution to a SAT formula can be very fast, whereas proving a formula to be UNSAT usually takes more time. In a naive thinking, all possible assignments have to be evaluated, and even though there are better algorithms implementing the QBF-problem, one can run in timeouts easily.

3 Bounded Synthesis of Petri Games

In this section, we introduce bounded synthesis for Petri games [6] which encodes the existence of a bounded winning strategy for a bounded unfolding of a Petri game to a quantified boolean formula.

3.1 Synthesizing Petri Games

We want to find one strategy for the system, that handles all possible flows of the environment such that we are *deadlock-avoiding*, *deterministic* and never reach a *bad place*. To encode this into QBF, we categorize the strategy of the system with V_S and a sequence of transitions as $V_{T,n}$. Since the limitation of the size of the implementation is one of the main features of bounded synthesis, the parameter n bounds the number of fired transitions to a natural number. This sets an upper bound to the size of the considered strategies. Finding one strategy V_S for all possible sequences of transitions $V_{T,n}$ that are part of the flow in the underlying Petri net correlates to synthesizing a winning strategy for a Petri game.

3.2 Formula

The previous introduced synthesis condition formulated as 2-QBF for a *b-bounded unfolding* of a Petri game results in the following formula:

$$\Phi_n = \exists V_S.\forall V_{T,n}.\phi_n,$$

where ϕ expresses that if V_S and $V_{T,n}$ describe a firing sequence of the underlying net $\mathscr{N}^b$, the sequence is winning for the system. V_S is defined as the set $V_S = \{(p, \lambda^b(t)) \mid p \in \mathscr{P}^b_S, t \in \mathscr{T}^b, p \in^\bullet t\}$, which consists of variables for all system places in presets of transitions. To restrict transitions, some variables in the set V_S will be set to 0, forbidding the transitions to be fired in the flow. $V_{T,n}$ also is a set containing one variable for each place in each iteration: $V_{T,n} = \{(p, i) \mid p \in \mathscr{P}^b, i \in \{1, ..., n\}\}$. All possible *runs* with n transitions fired are defined by quantifying universally over all possible $V_{T,n}$, not only the ones validating the flow of the Petri game. Setting a token on place p after firing three transitions corresponds to the tuple $(p, 4)$ and $\neg(p, 5)$ describes p to be empty after the next firing.

To describe ϕ in more detail one can say, if V_S and $V_{T.n}$ result in a sequence of the game, then the sequence is winning for the system players and by reaching the end of bound n, we have to recognize the n'th marking as a loop of the game:

$$\phi_n = \big(\bigwedge_{i\in\{1,\dots,n-1\}} \mathit{sequence}_i \Rightarrow \mathit{winning}_i\big) \wedge (\mathit{sequence}_n \Rightarrow \mathit{loop} \wedge \mathit{winning}_n)$$

To satisfy that the markings in $V_{T,n}$ match a firing sequence, $\mathit{sequence}_n$ is built iteratively as follows:

$$\mathit{sequence} = \mathit{initial} \wedge \bigwedge_{j\in\{1,\dots,i-1\}} flow_j$$

initial describes the first marking of $V_{T,n}$ to be the initial marking of the game and is the first condition of the formula. Intuitively for all places $p \in In, (p,1) = 1$ and for all other places $p' \notin In, (p',1) = 0$. $flow_i$ describes the enabledness of at least one transition and the firing of exactly one transition, defining the values of places from the i'th marking to the i+1'th marking. Notice that $\neg flow_i$ declares a *deadlock* in the flow.

$$\begin{aligned}
\mathit{initial} &= \big(\bigwedge_{p\in In^b} (p,1)\big) \wedge \big(\bigwedge_{p\in\mathscr{P}^b\setminus In^b} \neg(p,1)\big)\\
flow_i &= \bigvee_{t\in\mathscr{T}^b} \Big(\big(\bigwedge_{p\in{}^\bullet t} (p,i)\big) \wedge \big(\bigwedge_{p\in{}^\bullet t\cap\mathscr{P}^b_S} (p,\lambda(t))\big) \wedge \big(\bigwedge_{p\in t^\bullet} (p,i+1)\big)\\
&\quad \wedge \big(\bigwedge_{p\in\mathscr{P}^b\setminus({}^\bullet t\cup t^\bullet)} (p,i) \Leftrightarrow (p,i+1)\big) \wedge \big(\bigwedge_{p\in{}^\bullet t\setminus t^\bullet} \neg(p,i+1)\big)\Big)
\end{aligned}$$

initial describes the initial state of the game as mentioned before. Since we encode boolean formulas, (p,i) describes value 1 and $\neg(p,i)$ describes the value 0. The main goal of this thesis is to change the formula $flow_i$, so we take a closer look at it. The order of the enumeration is equivalent to the formula's conjunctions. For one transition, which is stated by the outermost disjunction indicating over all transitions, it has to state that:

1. All places in the preset of t are set to 1 in the i'th iteration.
2. If the transition contains system places in its preset, the strategy has to enable this transition.
3. In the i+1'th marking, all places in t's postset have to be set.
4. All places that are not part of ${}^\bullet t$ or $t^\bullet$ do not change their value from marking i to i+1.

5. The places in ${}^\bullet t$ that are not in $t^\bullet$ are set to 0.

The winning condition $\mathit{winning}_i$ ensures the absence of bad places, that deadlocks are terminating and that the behaviour of the strategy is deterministic:

$$\mathit{winning}_i = \mathit{nobadplace}_i \wedge \mathit{deadlocksterm}_i \wedge \mathit{deterministic}_i$$

If we reach $\mathit{sequence}_n$ without invalidating Φ_n, we considered n valid markings and did not reach a deadlock. We now have to show that we encountered a loop in the Petri game by finding a previous marking, that is equivalent to the current marking:

$$\mathit{loop} = \bigvee_{j,k \in \{1,\ldots,n-1\}, j<k} \Big(\bigwedge_{p \in \mathscr{P}^b} (p,j) \Leftrightarrow (p,k) \Big)$$

The definitions of all other needed formulas is straightforward. $\mathit{nobadplace}_i$ checks that no bad place possesses a token. $\mathit{deadlocksterm}_i$ defines that every occurring deadlock is caused by termination of the game, i.e. no transition is enabled. $\mathit{deterministic}_i$ ensures that the decisions of transitions by the strategy is deterministic and $\mathit{deadlock}_i$ encodes the occurence of a deadlock in the game:

$$\begin{aligned}
\mathit{nobadplace}_i &= \bigwedge_{p \in \mathscr{B}^b} \neg(p,i) \\
\mathit{deadlocksterm}_i &= \mathit{deadlock}_i \Rightarrow \mathit{terminating}_i \\
\mathit{terminating}_i &= \bigwedge_{t \in \mathscr{T}^b} \bigvee_{p \in {}^\bullet t} \neg(p.i) \\
\mathit{deterministic}_i &= \bigwedge_{t_1,t_2 \in \mathscr{T}^b, t_1 \neq t_2, {}^\bullet t_1 \cap {}^\bullet t_2 \cap \mathscr{P}^b_S \neq \emptyset} \Big(\bigvee_{p \in {}^\bullet t_1 \cup {}^\bullet t_2} \neg(p,i)) \vee \\
&\quad (\bigvee_{p \in {}^\bullet t_1 \cap \mathscr{P}^b_S} \neg(p, \lambda^b(t_1)) \vee \bigvee_{p \in {}^\bullet t_2 \cap \mathscr{P}^b_S} \neg(p, \lambda^b(t_2))) \Big) \\
\mathit{deadlock}_i &= \bigwedge_{t \in \mathscr{T}^b} \big(\bigvee_{p \in {}^\bullet t} \neg(p,i) \big) \vee \big(\bigvee_{p \in {}^\bullet t \cap \mathscr{P}^b_S} \neg(p, \lambda^b(t)) \big)
\end{aligned}$$

Notice that the formula has been slightly changed from the original paper, especially considering bad places instead of bad markings and the changes from [7].

4 True Concurrency Firing Semantics

In this section, we introduce the new concept of *true concurrency firing semantics* for Petri games. The approach is added to the formula of [6] and is implemented in Section 5 and Section 7. Section 6 provides a correctness proof for the presented concepts.

The standard firing semantics of Petri games implements an interleaving of all enabled transitions of the current marking of the game. These semantics are the coarsest possible considering the number of markings that have to be analyzed. Petri games depend on Petri nets, which deliver varying semantics including true concurrency approaches, e.g. *step semantics* and *pomset semantics* [18]. The goal of this thesis is to define similar true concurrency semantics for Petri games. Some transitions in Petri games do not depend on other transitions and could be fired *true concurrently*, allowing multiple enabled transitions to be fired from one marking M_i to the following marking M_{i+1}. Thereby we save the analysis of one marking, i.e. the marking after firing one transition and before firing the second enabled transition. This is especially interesting for the bounded synthesis approach in [6] since firing multiple transitions at once can reduce the required bound n of the formula and reduce the search space in terms of possible markings that have to be winning.

We introduce conditions for transitions that cannot be fired true concurrently, build sets of transitions based on this property and define the true concurrent flow with the constructed sets.

4.1 True Concurrency

The basic concept of our *true concurrency* approach is firing as many transitions as possible from one marking to the following. Since we want to reduce the possible markings of a Petri game, we have to enforce firing true concurrently such that whenever firing multiple transitions is possible, we prohibit sequential firing. Therefore, we have to take a closer look at the properties of Petri games.

The main feature of Petri games is causal past. The information exchange is based on synchronization of tokens sharing their previous flow and taking decisions based on that knowledge. Firing multiple transitions at once, the possibility of *waiting* for transitions to be enabled and having the complete choice of synchronization mustn't be lost. Therefore, we define a new condition for transitions to be enabled.

We start with the definition of *mutual exclusive transitions*:

Definition 4.1. Mutual Exclusive Transitions
A pair of transitions t' and t are called *mutual exclusive*, if their common preset contains at least one place, e.g. ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$

Two transitions that are *mutual exclusive* cannot be fired at once. Local players residing in their common preset have a choice in firing one of them. With *mutual exclusive transitions* we define the set of pairwise *mutual exclusive* transitions:

Definition 4.2. Common Preset Set
A non-empty set of pairwise *mutual exclusive* transitions $\mathscr{T}_M$ contains all mutual exclusive transitions for every element t in $\mathscr{T}_M$. We call this set *common preset (cp-) set.*

We state that all transitions belonging to the same cp-set *cannot* be fired true concurrently. We also say that transitions of the same cp-set are only allowed to be fired in a true concurrent flow if all transitions in $\mathscr{T}_M$ are enabled.

Definition 4.3. True Concurrent Enabledness
A cp-set $\mathscr{T}_M$ is *true concurrently* enabled (short enabled) if all transitions $t \in \mathscr{T}_M$ are enabled. We say a transition t' is true concurrently enabled, if the corresponding set with $t' \in \mathscr{T}_M$ is true concurrently enabled.

If a cp-set is true concurrently enabled, we fire it corresponding to the following definition:

Definition 4.4. Firing cp-Sets
A cp-set $\mathscr{T}_M = \{t_1, ..., t_m\}$ is fired by firing exaclty *one* transition t_i with $1 \leq i \leq m$.

By waiting for all transitions to be enabled, we ensure that despite firing transitions true concurrently, we do not reduce the possibilities of firing for local players. Figure 5a shows a Petri game with one environment player and one system player. The local player residing in C can either fire tC or wait for tBC to be enabled and fire tBC. If we realize *true concurrency* and would enforce all "only" *enabled* transitions to fire at once, the token in C would fire tC, since tC is enabled and tBC is not. The environment token would fire tB true concurrently to tC. Enabling transition tBC would not be possible in this flow since the marking $\{B, C\}$ is unreachable. To avoid this behaviour, transitions have to be *true concurrently enabled*, forbidding C to fire until tBC and tC are enabled. This Petri game consists of two *cp-sets* $\mathscr{T}_{M,1} = \{tB\}$ and $\mathscr{T}_{M,2} = \{tBC, tC\}$, having tBC as a mutual exclusive transition of

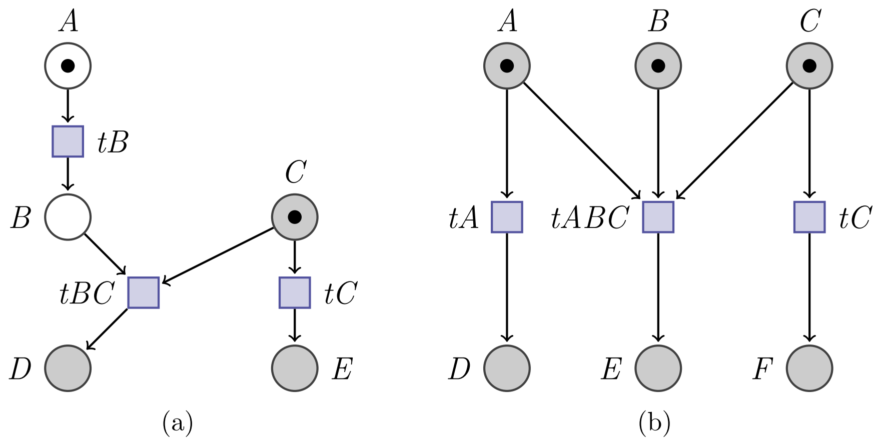

Figure 5: Two Petri games to illustrate the set enabledness in Figure 5a and the firing of a cp-set in 5b.

tC and no mutual exclusive transition to tB. tB is a local transition whose *true concurrency enabledness* is identical to the common *enabledness*.

Notice that not all transitions of a cp-set have to be pairwise *mutual exclusive*, it suffices that there is one mutual exclusive partner in the set. The Petri game in Figure 5b has only one cp-set with all three transitions $\{tA, tABC, tC\}$. tA and tC are not *mutually exclusive* but they belong to the same set since they both are *mutually exclusive* to $tABC$. The transition set is enabled and we can fire it by firing *one* of the three transitions. Transition tA and tC could possibly be fired at once, their common preset is empty. We limit the firing of a cp-set to the previous Definition 4.4, which does not include firing tA and tC in one step. This restriction is necessary concerning the implementation structure and we leave this feature open for future work.

We extend the definition of the $^\bullet$ operator to synchronization sets with $^\bullet\mathscr{T}_M = \bigcup_{t\in\mathscr{T}_M} pre(t)$ and $\mathscr{T}_M{}^\bullet = \bigcup_{t\in\mathscr{T}_M} post(t)$ and call it the preset and postset of the cp-sets.

4.2 True Concurrent Flow

With the definitions of *cp-sets*, *true concurrent enabledness* and *firing cp-sets*, we are able to define the true concurrent flow in the following.

To enforce a true concurrent flow we change the flow of transitions from firing one of the *enabled* transitions to firing *all* of the *true concurrently enabled sets*, demanding that the reachable markings of the true concurrent flow is not exceeding the sequential flows reachable markings. To define the true concurrent flow, we add the superset of all possible cp-sets to the definition of Petri games.

Definition 4.5. Set of cp-Sets
The set of all possible cp-sets of a Petri game $\mathcal{G}$ is identified with $\mathcal{C}_{\mathcal{T}}$. All cp-sets $\mathcal{T}_{M,i} \in \mathcal{C}_{\mathcal{T}}$ with $0 \leq i \leq m$ and $m = |\mathcal{C}_{\mathcal{T}}|$ are mutual exclusive.

As all possible sets in $\mathcal{C}_{\mathcal{T}}$ are mutual exclusive, every transition of $\mathcal{T}$ belongs to exactly one cp-set. We can therefore say that $|\mathcal{T}|$ is greater or equal to $|\mathcal{C}_{\mathcal{T}}|$, since the special case of $|\mathcal{T}| = |\mathcal{C}_{\mathcal{T}}|$ corresponds to a bijection from transitions to cp-sets. It is a special case because local players cannot synchronize with each other and share their causal past which does not correlate to playing a game in the first place. We finally define the true concurrent flow with $\mathcal{C}_{\mathcal{T}}$:

Definition 4.6. True Concurrent Flow
For a Petri game $\mathcal{G}$ and its set of cp-sets $\mathcal{C}_{\mathcal{T}}$ the *true concurrent flow* is the following:
As long as at least one set $\mathcal{T}_M \in \mathcal{C}_{\mathcal{T}}$ is *true concurrently enabled*, all true concurrently enabled sets of $\mathcal{C}_{\mathcal{T}}$ are fired, otherwise *one enabled* transition $t \in \mathcal{T}$ is fired.

Having no cp-set true concurrent enabled can happen in some Petri games and is handled through a corner case, which fires one transition sequentially. Even if the sequential flow of a game is similar to its true concurrent flow, not being able to fire a set of mutual exclusive transitions is an exception. We clarify this with an example game in Section 4.3. cp-sets can easily be defined for *bounded unfoldings* of an unfolding β_U with their superset $\mathcal{C}_{\mathcal{T}^b}$ building the sets of mutually exclusive transitions after copying the original transitions.

4.3 Example

The Petri game in Figure 6 has the same behaviour as the Petri game in Figure 2 but implementing one sender and two receivers. It therefore has two instances of the original game and new communication transitions between the games. Bad situations are the combination of bad places in the sub games. We have to avoid reaching *Done* as long as at least one of the sub games has a token in $Efailure$ or $Efailure'$. We left out the reinitializing transitions starting from *Again* firing to the *initial marking* from all possible markings after synchronization of both sub games and also the transitions to *bad places* for readability.

The initial marking enables two transition sets true concurrently, $\{if, is\}$ and $\{if', is'\}$. Since we have at least one set enabled, we fire all enabled sets. We fire two out of the four enabled transitions which leads to the next marking. We assume that if and if' were fired in the first iteration and the tokens in *Env* and *Env'* are in $Failure$ and $Failure'$ now. The current marking $\{Failure, Failure', Sfailure, Sfailure', Ssent, Ssent'\}$ has no transition set enabled. For example, *commS* and *tF* are part of the same cp-set and *commS* is not enabled. The same holds for their counterparts in the second sub game. We solve this situation with the corner case of the definition and fire transitions sequentially until we reach *Decision* or *Decision'*. As soon as *Decision* is part of the current marking, we fire *tagain* or *tdone* immediately as they are true concurrently enabled. When both sub games reach *Result* and *Result'* they can synchronize based on their causal past over $tAgain$ or $tDone$.

We can only fire the transitions from the initial marking true concurrently in this Petri game since we never reach a marking where two transition sets are enabled, except the initial marking. In contrast to the sequential flow, the number of sequences to reach the initial marking again and therefore find a loop is decreased by one, since we fire two transitions in one sequence. The difference in needed sequences to solve the game will increase by one for each added receiver comparing the sequential flow and the true concurrent flow.

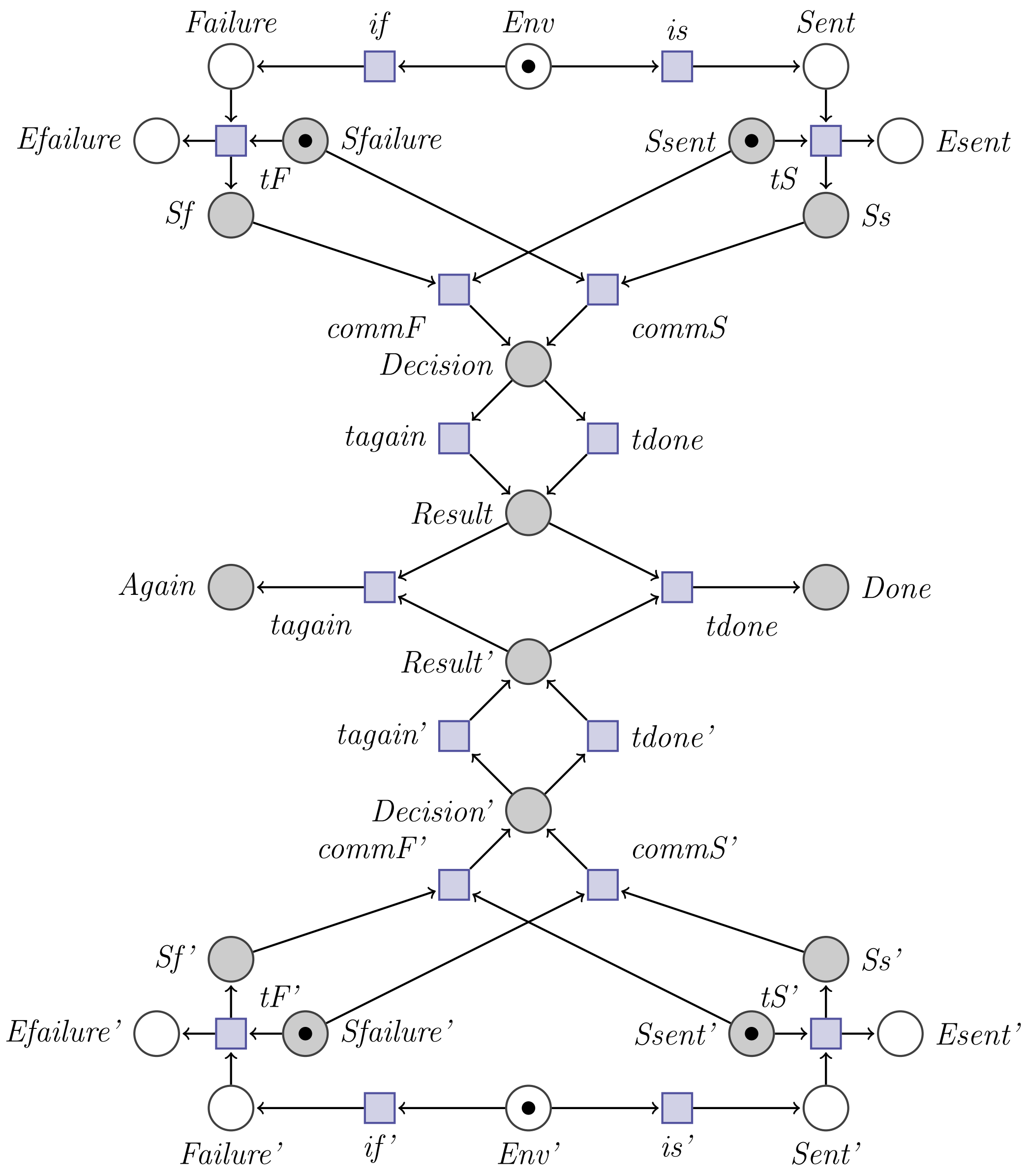

Figure 6: The example Petri game with two receivers and one sender. Bad places and the reset of the initial marking are omitted due to readability.

5 Bounded Synthesis with True Concurrency Firing Semantics

In this Section we extend the approach of Section 3 to our new *true concurrency firing semantics* of Section 4.1. We therefore present the new formula and argue its completeness afterwards.

5.1 True Concurrency to QBF

We present the formula for the true concurrent flow that substitutes the $flow_i$ formula in the sequential encoding. To construct a solvable formula, we have to ensure that all places in all possible situations are well defined and their value is unambiguous. To deliver this completeness, we develop an iterative structure of the formula where not all sub formulas define all possible values of places, but the composition of the sub formulas delivers a distinct definition for each place.

The part of the formula in Section 3.2 that describes the *sequential* flow of the game is formulated in $flow_i$. The intuition of $flow_i$ is choosing one of the *enabled* transitions, deleting the tokens in its preset, adding tokens in its postset and ensure the rest of the game stays the same. We replace this part of the formula with the *true concurrent* flow and start with the differentiation of true concurrent firing or sequential firing.

5.1.1 Flow

The new $flow_i$ defines the frame for true concurrency. The index i represents the current iteration of the game and therefore the number of markings we passed up to this situation:

$$\begin{aligned} flow_i = & \big(\neg \bigvee_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}}} is\_enabled_{i,\mathscr{T}_{M,j}} \wedge fire\_one\_transition_{i,\mathscr{T}^b}\big) \\ & \vee \big(\bigvee_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}}} is\_enabled_{i,\mathscr{T}_{M,j}} \wedge \bigwedge_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}^b}} decision_{i,\mathscr{T}_{M,j}}\big) \end{aligned}$$

We differentiate between two main cases. Either we have no true concurrency enabled set $\mathscr{T}_M$ or we have at least one concurrently enabled set: $\neg \bigvee_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}}} is\_enabled_{i,\mathscr{T}_{M,j}}$ and the negation of it. The first case is the sequential case and we solve it with $fire\_one\_transition_{i,\mathscr{T}^b}$, which behaves like the $flow_i$ of the encoding in Section 3 only defined on a set of transitions, formulated in Section 5.1.4. The second case leads to the $decision_{i,\mathscr{T}_M}$ sub formula which makes the decision of firing the cp-set or not, depending on its enabledness. It is defined in Section 5.1.3.

5.1.2 True Concurrency Enabled

To ensure enabledness on sets, we introduce the following formula:

$$is\_enabled_{i,\mathscr{T}_{M,j}} = \bigwedge_{p \in {}^\bullet\mathscr{T}_{M,j}} (p,i) \wedge \bigvee_{t \in \mathscr{T}_{M,j}} \bigwedge_{p \in {}^\bullet t \cap \mathscr{P}^b_S} (p, \lambda^b(t))$$

The left part of the formula pledges that all places in the *preset* of the true concurrency set possess a token in the i'th iteration. On the right-hand side we observe the behaviour of the strategy for all transitions in the respecting set. We enforce that at least one transition is also allowed by the *strategy* and can be fired. Therefore all system places in its preset have to be set by the strategy. Otherwise we would consider a set as enabled even if the strategy forbids firing all transitions of the set. Notice that one strategy enabled transition suffices and that for transition sets with at least one environment transition the right part of the formula evaluates to true since an empty conjunction is equivalent to true.

5.1.3 Decision

The $decision_{i,\mathscr{T}_{M,j}}$ sub formula is part of the the right-hand side of the newly introduced $flow_i$. At this point of the formula we know that we have at least one true concurrently enabled cp-set and therefore have to fire all possible sets.

$$decision_{i,\mathscr{T}_{M,j}} = (fire_{i,\mathscr{T}_{M,j}}) \vee (dont\_fire_{i,\mathscr{T}_{M,j}})$$

We only have two options for the corresponding set $\mathscr{T}_{M,j}$. Either we fire it or we do not fire it. The decision is based on the *enabledness* of the set. If a set is enabled, $fire_{i,\mathscr{T}_{M,j}}$ has to evaluate to true and $dont\_fire_{i,\mathscr{T}_{M,j}}$ otherwise. $fire_{i,\mathscr{T}_{M,j}}$ and $dont\_fire_{i,\mathscr{T}_{M,j}}$ are mutual exclusive and it is guaranteed that one of them evaluates to true in the case that we fire concurrently, i.e. at least one of the cp-sets is enabled.

5.1.4 Fire

Firing a transition set means firing one of the transitions of the set. Since firing true concurrently is only possible if all transitions are enabled and at least one is also allowed by the strategy, we have to determine if the current transition set is enabled:

$$fire_{i,\mathscr{T}_{M,j}} = is\_enabled_{i,\mathscr{T}_{M,j}} \wedge fire\_one\_transition_{i,\mathscr{T}_{M,j}}$$

To fire a transition set, we state that the transition set has to be enabled and that *only one* transition of the set is fired. $fire\_one\_transition_{i,S}$ is also part of the sequential flow of the formula, we only added the parameter S which corresponds to a set of transitions.

$$
\begin{aligned}
fire\_one\_transition_{i,S} = \bigvee_{t \in S} \Big(& \big(\bigwedge_{p \in {}^\bullet t} (p,i) \big) \wedge \big(\bigwedge_{p \in {}^\bullet t \cap \mathscr{P}^b_S} (p, \lambda(t)) \big) \\
& \wedge \big(\bigwedge_{p \in t^\bullet} (p, i+1) \big) \wedge \big(\bigwedge_{p \in {}^\bullet S \setminus ({}^\bullet t \cup t^\bullet)} (p,i) \Leftrightarrow (p,i+1) \big) \\
& \wedge \big(\bigwedge_{p \in {}^\bullet t \setminus t^\bullet} \neg (p, i+1) \big)
\end{aligned}
$$

Informally, this formula chooses one transition to be fired out of a set S respecting the strategy's behaviour. The part of the formula has only slightly changed to the $flow_i$ of Section 3.2, introducing a general form for sets of transitions. It is a *disjunction* quantifying over all transitions in the set S and choosing one to fire and all others to not fire. Therefore we check for enabledness, the strategy's choice of firing the transition or not, we reorder the related tokens and retain all other token possessing places.

5.1.5 Do Not Fire

$decision_{i,\mathscr{T}_{M,j}}$ enforces to fire *or* not fire. Since one of the sub formulas has to evaluate to true, we need a case that handles a *not* concurrently enabled set:

$$
\begin{aligned}
dont\_fire_{i,\mathscr{T}_{M,j}} = \neg is\_enabled_{i,\mathscr{T}_{M,j}} \wedge & \bigwedge_{p \in {}^\bullet \mathscr{T}_{M,j}} (p,i) \Rightarrow (p, i+1) \\
\wedge \bigwedge_{p \in \mathscr{T}_{M,j}{}^\bullet} \big(& \bigvee_{\mathscr{T}_{M,l} \in \mathscr{C}_\mathscr{G}, p \in \mathscr{T}_{M,l}{}^\bullet, \mathscr{T}_{M,j} \neq \mathscr{T}_{M,l}} is\_enabled_{i,\mathscr{T}_{M,l}} \\
& \vee \neg (p,i) \Rightarrow \neg (p, i+1) \big)
\end{aligned}
$$

The condition to not fire a set is that it is *not* enabled. This is implemented by $\neg is\_enabled_{i,\mathscr{T}_{M,j}}$ of $dont\_fire_{i,\mathscr{T}_{M,j}}$. Not firing the transitions means that the places in the preset are not changed by that transition. All pre places that possess a token keep it in the next iteration step: $\bigwedge_{p \in {}^\bullet i, \mathscr{T}_{M,j}} (p,i) \Rightarrow (p, i+1)$. We cannot make an assessment on empty places. Their preceding transitions could be fired and therefore they are defined by the transition that has these places in their postset.

The second big *conjunction* in the formula defines the post places of the transitions. For all places of the postset, the inner *disjunction* has to hold.

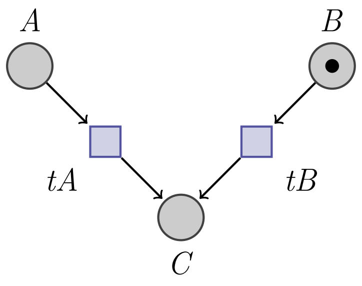

Figure 7: A Petri game with two cp-sets with a common postset.

We cannot simply say that all post places stay the same from i to $i+1$ since the preset of post places, which are transitions, can possibly be part of another transition set $\mathscr{T}_{M,j} \neq \mathscr{T}_{M,l}$ which could be fired and therefore changes the value of its post places. In Figure 7 this behaviour is illustrated in an example. The cp-sets $\{tA\}$ and $\{tB\}$ both have C as their post place. Transition set $\{tB\}$ is enabled whereas $\{tA\}$ is not, meaning that $dont\_fire$ has to evaluate to true for $\{tA\}$. We have to check if another transition set in the preset of C is enabled, which is found with $\{tB\}$. Therefore $dont\_fire$ cannot say if C will possess a token in the next iteration, but firing the enabled set $\{tB\}$ will define the state of this place. This behaviour occurs because the postsets of cp-sets are not mutual exclusive. We defined the cp-sets only on their presets. If we find a different and enabled set, we can rely on it to be fired and define this place, if not, we have to ensure that this place does not have a token in the $i+1$'th iteration if it did not possess one in the i'th marking: $\bigvee_{\mathscr{T}_{M,l,i}\in\mathscr{C}_{\mathscr{G}},p\in\mathscr{T}_{M,l,i}^{\bullet},\mathscr{T}_{M,j,i}\neq\mathscr{T}_{M,l,i}} is\_enabled_{i,\mathscr{T}_{M,l}} \vee \neg(p,i) \Rightarrow \neg(p,i+1)$. If a post place is set, we cannot assume its state in the next marking. It will then be defined by the transition set it is preceding by firing it or not firing the set. The complete new formula is shown in Figure 8 and has to be substituted as the $flow_i$ of Section 3.2.

5.2 Completeness

Implementing the formula requires the completeness of the introduced formula. To ensure this property and the absence of contradictions in the formula, we prove that every place in every iteration is well-defined. We assume the underlying formula as provided [6] to be complete.

As mentioned before, the sub formulas can only provide distinct definitions of places by combining each other. We state the following:

$$
\begin{aligned}
flow_i &= \big(\neg \bigvee_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}^b}} is\_enabled_{i,\mathscr{T}_{M,j}} \wedge fire\_one\_transition_{i,\mathscr{T}^b}\big) \\
&\quad \vee \big(\bigvee_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}}} is\_enabled_{i,\mathscr{T}_{M,j}} \wedge \bigwedge_{\mathscr{T}_{M,j} \in \mathscr{C}_{\mathscr{G}}} decision_{i,\mathscr{T}_{M,j}}\big) \\
is\_enabled_{i,\mathscr{T}_{M,j}} &= \bigwedge_{p \in {}^\bullet\mathscr{T}_{M,j}} (p,i) \wedge \bigvee_{t \in \mathscr{T}_{M,j}} \bigwedge_{p \in {}^\bullet\mathscr{T}_{M,j} \cap \mathscr{P}^b_S} (p, \lambda^b(t)) \\
decision_{i,\mathscr{T}_{M,j}} &= (fire_{i,\mathscr{T}_{M,j}}) \vee (dont\_fire_{i,\mathscr{T}_{M,j}}) \\
fire_{i,\mathscr{T}_{M,j}} &= is\_enabled_{i,\mathscr{T}_{M,j}} \wedge fire\_one\_transition_{t \in \mathscr{T}_{M,j}} \\
fire\_one\_transition_{i,S} &= \bigvee_{t \in S} \Big(\big(\bigwedge_{p \in {}^\bullet t} (p,i)\big) \wedge \big(\bigwedge_{p \in {}^\bullet t \cap \mathscr{P}^b_S} (p, \lambda(t))\big) \\
&\quad \wedge \big(\bigwedge_{p \in t^\bullet} (p, i+1)\big) \wedge \big(\bigwedge_{p \in {}^\bullet S \setminus ({}^\bullet t \cup t^\bullet)} (p,i) \Leftrightarrow (p,i+1)\big) \\
&\quad \wedge \big(\bigwedge_{p \in {}^\bullet t \setminus t^\bullet} \neg(p, i+1)\big) \\
dont\_fire_{i,\mathscr{T}_{M,j}} &= \neg is\_enabled_{i,\mathscr{T}_{M,j}} \wedge \bigwedge_{p \in {}^\bullet\mathscr{T}_{M,j}} (p,i) \Rightarrow (p,i+1) \\
&\quad \wedge \bigwedge_{p \in \mathscr{T}_{M,j}{}^\bullet} \big(\bigvee_{\mathscr{T}_{M,l} \in \mathscr{C}_{\mathscr{G}}, p \in \mathscr{T}_{M,l}{}^\bullet, \mathscr{T}_{M,j} \neq \mathscr{T}_{M,l}} is\_enabled_{i,\mathscr{T}_{M,l}} \\
&\qquad \vee \neg(p,i) \Rightarrow \neg(p,i+1)\big)
\end{aligned}
$$

Figure 8: The complete new part of the formula implementing true concurrency semantics to the bounded synthesis of Petri games approach [6]

Theorem 5.1. Completeness
An encoding of a Petri game is complete, if for every iteration step i to $i+1$, every place in the Petri game has a well-defined value in $i+1$ if it has a well-defined value in i.

Proof. We prove Theorem 5.1 with a direct proof and case distinction over the formula.

We consider iteration i as the current iteration of the flow and $i+1$ as the next iteration step. We show that for every cp-set $\mathscr{T}_{M,j}$ and every place $p \in {}^\bullet\mathscr{T}_{M,j} \cup \mathscr{T}_{M,j}^\bullet$, the formula defines the place's state in the next step distinctly.

The case distinction is based on the two branches of the true concurrent flow definition:

1. **No cp-set is enabled:**
In this case, the definition of the places is exactly the definition of the sequential flow. Since we assume the sequential approach to be complete, all places are well defined in iteration $i+1$.

2. **At least one cp-set is enabled:**
Following the formula, we have to differentiate for every cp-set if it is enabled or not:

    - **$\mathscr{T}_{M,j}$ is enabled:**
    Since the cp-set is enabled, we know that it has to be fired. The formula $fire\_one\_transition_{i,\mathscr{T}_{M,j}}$ does not consist of implications but only of statements and equivalences, defining all places in its preset and postset. The preset of the one fired transition is set to zero, the postset to one. All other places that are not included in the fired transitions preset or postset stay equivalent in $i+1$ and therefore all places in ${}^\bullet\mathscr{T}_{M,j} \cup \mathscr{T}_{M,j}^\bullet$ are well defined.

    - **$\mathscr{T}_{M,j}$ is not enabled:**
    If the cp-set is not enabled and we fire true concurrently, the formula $dont\_fire_{i,\mathscr{T}_{M,j}}$ has to evaluate to true. The sub formula defines a place in the preset of the cp-set only if the place is set in i. The places in the postset are only defined if no other transition set in their preset is enabled and if it is not set in iteration i. If one of the transition sets in the preset of these places is enabled, it is well defined arguing with the previous step of the proof. To show that all other places are also well defined we consider a second cp-set.

Let $\mathscr{T}_{M,l}$ be a different cp-set and $\mathscr{T}_{M,l}^{\bullet} \cap {}^{\bullet}\mathscr{T}_{M,j} \neq \emptyset$. If $\mathscr{T}_{M,l}$ is enabled in iteration i and fires, its post places are distinctly defined and therefore the places $p \in \mathscr{T}_{M,l}^{\bullet} \cap {}^{\bullet}\mathscr{T}_{M,j}$ too. If $\mathscr{T}_{M,l}$ is not enabled, the formula of $dont\_fire_{i,\mathscr{T}_{M,l}}$ defines exactly the counter case of the places in $\mathscr{T}_{M,l}^{\bullet} \cap {}^{\bullet}\mathscr{T}_{M,j}$ that are not defined by $dont\_fire_{i,\mathscr{T}_{M,j}}$, which is $\neg(p,i)$. With that, all cases are handled and the pre places $p \in {}^{\bullet}\mathscr{T}_{M,j}$ are well defined.

Let $\mathscr{T}_{M,k}$ be a different cp-set and ${}^{\bullet}\mathscr{T}_{M,k} \cap \mathscr{T}_{M,j}^{\bullet} \neq \emptyset$. If $\mathscr{T}_{M,k}$ is enabled in iteration i and fires, its pre places are distinctly defined and therefore the places in $p \in {}^{\bullet}\mathscr{T}_{M,k} \cap \mathscr{T}_{M,j}^{\bullet}$ too. If $\mathscr{T}_{M,k}$ is not enabled, the formula $dont\_fire_{i,\mathscr{T}_{M,k}}$ defines exactly the counter case of the places $p \in \mathscr{T}_{M,k}^{\bullet} \cap {}^{\bullet}\mathscr{T}_{M,j}$ that are not defined by $dont\_fire_{i,\mathscr{T}_{M,j}}$, which is (p,i). With that, all cases are handled and the post places $p \in \mathscr{T}_{M,j}^{\bullet}$ are well defined.

All sub formulas that define the places in $i+1$ corresponding to iteration i define the places distinctly and are not contradictory. □

We presented a proof for the completeness of the formula and can prove the correctness of the true concurrency firing semantics. The implementation of the formula is presented in Section 7.

6 Proof of Correctness

In this Section we prove the correctness of the true concurrency firing semantics introduced in Section 4.3 for infinite unfoldings of Petri games. We then extend the correctness proof to bounded unfoldings to justify our true concurrency approach.

6.1 Runs

We prove the correctness of the newly introduced true concurrent flow definition by showing the equivalence of true concurrent and sequential flow on Petri games. Informally, we have to show that we can simulate every behaviour of tokens in the Petri game defined by the sequential flow with the true concurrent flow and vice versa. To formalize the behaviour of tokens in an unfolded game, we recap the definition of a sequential finite run.

Definition 6.1. Sequential Run
A *sequential* finite (initial) run $\pi = \langle t_1, ..., t_i \rangle$ with $In \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_i} M_i$ is a sequence of transitions that are fired in an infinite unfolding of a Petri game. We call M_π the reached marking of the run π. For the places in the visited markings of the run, it holds that:
$\forall p \in In, M_1, ..., M_i.\ |pre(p)| \leq 1 \wedge |post(p)| \leq 1.$

We can describe the reachable markings of an unfolding by the set of all possible runs of the underlying net. We can represent runs as Petri games, where all transitions that are not part of the run and unreachable places are deleted from the underlying Petri net. To define the reachable markings of a true concurrent flow, we introduce true concurrent runs:

Definition 6.2. True Concurrent Run
A *true concurrent* finite (initial) run
$\pi_C = \big\langle \{t_{1,1}, ..., t_{n_1,1}\}, ..., \{t_{1,i}, ..., t_{n_i,i}\} \big\rangle$ with $In \xrightarrow{\{t_{1,1},...,t_{n_1,1}\}} M_1 \xrightarrow{\{t_{1,2},...,t_{n_2,2}\}}$ $... \xrightarrow{\{t_{1,i},...,t_{n_i,i}\}} M_i$ and all $n_1, ..., n_i$ are greater than 0 is a sequence of sets of transitions that are fired in an infinite unfolding of a Petri game. We call M_{π_C} the reached marking of the run π_C. For the places in the visited markings of the run, it holds that:
$\forall p \in In, M_1, ..., M_i.\ |pre(p)| \leq 1 \wedge |post(p)| \leq 1.$

We changed the definition of sequential runs in a way that between two markings a set of transitions is fired. If every set of π_C has only one element, it represents a sequential run. Notice that the sets of the tuple π_C do not provide any order of transition. The true concurrent run only defines an order of transition sets. We present the definition of sequential reordered runs to categorize sequential runs of a Petri game:

Definition 6.3. Reordered Runs
A *reordered* run π_R of a Petri game $\mathscr{G}$ for a sequential run π of $\mathscr{G}$ is a sequential run that consists of the same transitions as the run π and the same reached marking $M_\pi = M_{\pi_R}$, but provides a permutation in the order of the transitions. The number of transitions is equal in π and π_R and every transition is uniquely identified.

The reordering relation is an equivalence relation whose proof is trivial and left out in this thesis. Reordering the transitions of a run such that the behaviour of tokens does not change and the semantics of the Petri game is still satisfied gives us the possibility to modify the firing order of transitions that are enabled at the same marking. To deploy reordered runs in our correctness proof of the true concurrency flow, we state the following:

Theorem 6.1. Substituting Reordered Runs
A run π and its reordered run π_R of a Petri game $\mathscr{G}$ simulate the same behaviour of individual tokens of the game and can be *substituted* by each other without changing the causal history of the tokens in the game.

Proof. We prove the Theorem 6.1 with a direct proof.

We show that two runs π and π_R that are reordered runs to each other simulate the same behaviour in the unfolding of a game. Therefore, the history of the tokens in the reached markings M_π and M_{π_R} has to be equal in both markings. All transitions in both runs are the same. With that, we cannot have different histories by firing different transitions. We know from [9] that the condition $|post(p)| \leq 1$ holds for all places p in the Petri game symbolizing the run. With this condition we can reconstruct the distinct flow of tokens by firing transitions backwards until the initial marking is reached. An unambiguous causal past is possessed by every token. Since markings are sets of tokens, we know that the history of markings understood as the combined history of its places is distinct.

By reaching the same marking with the same transitions fired, we then know that π and π_R simulate the same behaviour of tokens in the Petri game and can therefore be substituted by each other. $\square$

6.2 Equivalence of Runs

To prove the equivalence of the true concurrent flow and the sequential flow, we state the following:

Theorem 6.2. Equivalence
For every *true concurrent* run of a Petri game $\mathscr{G}$, there exists a *transformed sequential* run of the game. For every *sequential* run of the game, there exists a reordered sequential run that can be transformed to a *true concurrent* run.

We define a syntactic transformation from true concurrent runs to sequential runs and vice versa to be able to establish our equivalence theorem in Theorem 6.2. We prove the correctness and state the underlying conditions of the transformation in Proof 6.2.

Definition 6.4. Transforming Runs
A true concurrent run π_C can be transformed to a sequential run π and vice versa in the following way: $\big\langle\{t_{1,1}, ..., t_{n_1,1}\}, ..., \{t_{1,i}, ..., t_{n_i,i}\}\big\rangle$ is transformed to $\langle t_{1,1}, ..., t_{n_1,1}, ..., t_{1,i}, ..., t_{n_i,i}\rangle$ and both reached markings are the same. Transforming a true concurrent run to a sequential run is done by firing all transitions in the transition sets sequentially in the order of the tuple π. For transforming a sequential run to a concurrent run, we require $t_{1,i}, ..., t_{n_i,i}$ to be enabled at marking M_{i-1}. For t_i which is enabled at M_{i-1} and t_{i+1} which is true concurrently enabled at M_{i-1}, we require that t_{i+1} is before t_i in the tuple for all consecutive transitions of the run.

With the transformation of runs we can show the equivalence of sequential and true concurrent flow of the Petri game.

Proof. We prove the Theorem 6.2 and with it, that we can simulate true concurrent flow sequentially and vice versa.

We claim that for all true concurrent runs π_C there exists a sequential run π we can transform π_C into and for all sequential runs π there exists a reordered sequential run π_R that can be transformed to a true concurrent run π_C. We leave out the proof that transformed runs have the same behaviour in the game since it is straightforward. Since we use true concurrent runs, we consider the information of the corresponding cp-sets to be computed beforehand.

$'' \Rightarrow''$ $\forall \pi_C \exists \pi$ **that can be transformed from** π_C

We use the construction given in Definition 6.4. We know that if the construction is correct and universally applicable, then we find a simulating run by transforming π_C to the run π. The transformation is based on firing the transitions of a set in π_C sequentially, so we show that this is possible and does not invalidate the semantics of a Petri game, which is the behaviour of the tokens in the game.
Since all transitions in one set of the tuple corresponding to π_C are in different transition sets by definition, firing them in any order will not change the enabledness of each other. After firing the set corresponding to the i'th element of the tuple with j transitions sequentially, we know that the marking M_{i+j} in the sequential run transformed up to this point enables all transitions in the following set of the true concurrent run and we can fire them again. With that, the transformation is correct and delivers a sequential run for every true concurrent run.

$'' \Leftarrow''$ **$\forall\pi\exists\pi_R$ that can be transformed to run π_C.**

We reorder the transitions adjusting it to the true concurrent flow semantics of Section 4.1. With that we create a reordered run that can be directly transformed to a true concurrent run π_C.

Step 1 We start at t_1. For all pairs of transitions t_i, t_{i+1} and corresponding markings M_{i-1}, M_i, if M_{i-1} enables t_i sequentially and the cp-set of t_{i+1} true concurrently, we switch the order of transition t_{i-1} and t_i. We apply this reordering until no transitions can be switched.

Step 2 (a) If t_i is true concurrently enabled at M_{i-1}, we take all following j transitions where the corresponding cp-set is enabled at M_{i-1}, until a only sequentially enabled transition is found. With all found transitions we build the set $\{t_{1,i}, ..., t_{j,i}\}$ and continue at t_{j-1} and M_j again with (a).
(b) If t_i is not true concurrently enabled at M_{i-1}, we build the one elementary set $\{t_i\}$ and go to $t_i + 1$ starting step 2 again.

The transformed reordered tuple with sets is exactly the transformed true concurrent run of π. The transformation steps simulate the true concurrent flow, since true concurrently enabled sets are fired first and all simultaneously enabled sets are fired in one step. This is exactly the definition of the true concurrency flow in Definition 4.6.

Since both directions have been proven, we can say that for an arbitrary Petri game the sequential flow and the true concurrent flow simulate the same behaviour of tokens in the game. They can be substituted without losing information of the game.

□

With the definition of reordered runs, we find equivalence classes of runs that deliver the same history of tokens and can therefore be reduced to one of the runs of each class. We restrict the possible runs and therefore the *reachable* markings of the game with the true concurrent flow without invalidating the behaviour of tokens in the Petri game. Notice that true concurrent firing semantics is only applicable to Petri games that define their winning condition by *bad places* and not *bad markings*. Since we reduce the number of reachable markings, we could skip bad markings and therefore invalidate the game.

The proof shows the correctness of the introduced true concurrency firing semantics depending on the sequential firing semantics. Our proof is based on infinite unfoldings which are not applicable for our implementation. We therefore have to define runs for *bounded unfoldings*.

6.3 Bounded runs

If we consider *bounded unfoldings* of a Petri game, at some point in the flow we reach a visited marking again. As explained before, we lose the causal history of the tokens in this marking since we cannot differentiate between both visits. This behaviour is observed by a loop or by reaching the initial marking again which is a special case of a loop since we reach a marking that has been visited before.

Runs as defined beforehand implicitly avoid losing the causal history. Since we define the postset and preset of places to a maximum of one transition, even reaching the same marking again, which corresponds to a loop, has a distinct causal past. Whenever a previously visited marking occurs, all following transitions are repetitions of previous sub tuples of the run to the repeating marking. With that intuition we define *bounded runs*:

Definition 6.5. Bounded Runs
For a bounded unfolding of a Petri game, a *bounded* sequential run $\pi_B = \langle t_1, ..., t_i \rangle$ with $In \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_i} M_i$ is a sequence of transitions that are fired, where either $\exists!\ M_j.\ 0 \leq j \leq i-1.\ M_j = M_i$ and no other markings are equal or all markings are unique.

The definition of bounded *true concurrent* runs is synchronous to Definition 6.5 and Definition 6.2. With the newly introduced *bounded runs*, we avoid multiple occurences of transitions in runs and can apply Proof 6.1 and Proof 6.2 to these runs and bounded unfoldings. The correctness of the true concurrent flow for bounded unfoldings justifies the implementation in Section 5.

We proved the correctness of our true concurrent firing approach on infinite unfoldings and bounded unfoldings of Petri games. Since the initial Petri game is equivalent to its unfolding with bound 1 for all places, we can apply our true concurrent flow also to non unfolded games.

7 Implementation

In this section we introduce our implementation of the formula in Section 5 as an extension to the existing prototype implementation [7] of the formula in Section 3.2.

7.1 Tools

The following tools are used to implement the *true concurrency firing semantics*.

7.1.1 Bounded Synthesis Prototype

The existing prototype Java implementation mentioned in [7] takes a Petri game as *apt* file which is delivered by the *APT* tool [2] and two bounds n and b as input and delivers a bounded winning strategy if one exists for the given bounds. As before, n is the number of transitions fired in one *run* of the game and b represents the bound of the unfolding. The bounded winning strategy of the Petri game is encoded to the formula in Section 3.2 which is then solved by the QBF-solver QuAbS [19]. The encoded formula is written to a *qcir* [15] file that can be parsed by the solver. In case of *SAT*, the variable assignment is converted into a Petri game, which is the winning strategy, and extracted as a *dot* [11] file.

7.1.2 QuAbS

The QBF-solver used to find a variable assignment that evaluates the formula to *SAT* is *QuAbS* [19]. It is a counter example guided abstraction refinement (*CEGAR*) [14, 12] based solver that is not restricted to conjunctive normal form. *CEGAR* is a solver strategy that learns provided counterexamples for an assignment evaluating to *UNSAT* and uses this information for the following solving attempts. The solver tries to find a variable assignment for the previously constructed formula and returns the solving assignment or *UNSAT*.

7.2 True Concurrency Firing Semantics Implementation

Our implementation is based on the bounded synthesis prototype implementation in [7]. We introduced a new solver option for the true concurrent flow and changed the existing methods to encode the flow of a given Petri game as a true concurrent flow. We will present the idea and structure of the implementation and show an example encoding.
The first step of the implementation is the construction of the cp-sets. We figured out that building these sets corresponds to a fixed point iteration. Finding all transitions of a set is computed by following the transitions that are already in the set to their pre places and adding all transitions of these pre place's postset to the set until it is not changing any more. To be able to encode the flow, which is a *disjunction* over the sets in $\mathscr{C}_{\mathscr{G}^b}$ of the given Petri game, we *preprocess* the unfolding of the game with the fixed point iteration. To minimize the size of the implementation, we also save the postsets and presets of the cp-sets and the presets of all *places*. We need these presets to be able to handle $dont\_fire$ and check if another transition set is true concurrently enabled or not. After calculating all necessary sets, the implementation of the formula is based on the *fields* shown in Figure 9.

setlist is a list of all cp-sets of the game. *postlist* and *prelist* are the respective postsets and presets to the sets in *setlist*. The indexes of the sets coincide in all lists. *transitionmap* is a map starting from a transition to the index of the belonging cp-set in the *setlist*. With that, we can easily find the cp-set to a given transition.

The implementation of the formula is based on the sets and accords to the structure of the formula in Section 5. We iterate n times over all sets in *setlist* and go through the formula, having a method for every sub formula. We unroll the quantifications of conjunctions and disjunctions and write the properties of places to the *qcir* file. Notice that the formula of the flow encodes only the game's places, not its transitions. It forces them to possess a token or be empty in each iteration depending on their state in the previous iteration.

Example 7.1.
We present an example for the encoding of the game in Figure 7. Figure 10 displays only the flow from iteration 1 to 2. We leave out the transformation from places and iteration `(p,i)` to a unique literal and revert the order of the QBF-literals for readability. In the figure, `##` starts and ends a comment, `(A,1)` means that place A possesses a token in iteration 1 and the system place B allows transition tB with `(B,tB)`. We left out the encoding

```java
protected Map<Transition, Integer> transitionmap;
protected List<Set<Place>> postlist;
protected List<Set<Place>> prelist;
protected List<Set<Transition>> setlist;
```

Figure 9: The necessary preprocessed sets to encode the true concurrent flow of the Petri game.

of *decision* for the set with transition tB, which is synchronous to *decision* for the set $\{tA\}$. Whenever a sub formula needs one of the missing literals, we add it in the explanation. In the following we present an overview of the encoding.

The formula is assembled as follows, categorized by the respective literals:

51-17 $flow$
The $flow$ encoding is the composition of all sub formulas. Literal `51` is the outer disjunction of the formula, `50` the left-hand side and `49` the right-hand side.

48-46 $Fire\_one\_transition$
$Fire\_one\_transition$ consists of firing tB in `47` *or* firing tA in `46`.

45-24 *decision*
The *decision* conjunction *of all transition sets* is in literal `45`. Literal `44` encodes *decision* of the transition set with tA and literal `34` represents *decision* with transition set $\{tB\}$. We left out `34` - `24` since it is synchronous to decision of the set $\{tA\}$.

44-35 *decision with* $\{tA\}$
decision with set $\{tA\}$, which is either enabledness of the set and firing in `42` or the set is not enabled and not fired in `43`.

43,41-38 $dont\_fire$
We check for not enabledness of the transition set with tA in `22` and $dont\_fire$ the set starting with `41`. `40` checks if a transition in the preset of the postplaces, that is not part of the considered cp-set, is enabled. `29` is the implication of `-(C,1)` $\Rightarrow$ `-(C,2)` from $dont\_fire$. Notice that the implication is transformed to a disjunction. The behaviour of the preplaces is encoded in `39` and `38`.

42,37-35 $fire\_one\_transition$
These literals encode the firing of transition set $\{tA\}$. As in $dont\_fire$, we check for the enabledness with literal `22`. We choose one of the transitions in `36` and fire the transition in `35`. Since we only have transition tA in the set, `36` and `35` only consider one transition.

34-24 *decision with* $\{tB\}$
This part is synchronous to literals `44-35` only encoded for transition set $\{tB\}$ and not shown in Figure 10.

23-17 *disjunction* $is\_enabled$
`23` is the disjunctions over all cp-sets checking if at least one is enabled.

22-20 $is\_enabled$
These literals encode the enabledness of $\{tA\}$. `22` checks for the preplaces, `21` encodes that one transition has to be enabled by the strategy and literal `20` encodes that the system token in A enables transition tA.

19-17 $is\_enabled$
The encoding of enabledness of $\{tB\}$ is analogous to `22-20`.

7.3 Implementation Details

Implementing the formula leads to finding special cases and improvements which are shown in Section 7.3.1. Since the performance of a QBF-solver depends among others on the number of variables, we have to take a closer look at minimizing the formula, which is presented in Section 7.3.2.

7.3.1 Formula Extensions

A Petri game and its bounded unfolding can have an infinite flow and therefore transitions that restore the initial marking. Some Petri games have finite flows or infinite flows that never reach the initial marking again and we have to be able to solve these games as well. By implementing the formula, we figured out that a place that has no incoming arrows has to be encoded differently to places with presets. $dont\_fire$ only defines the places of a cp-set's preset if they possess a token in the i'th iteration. As explained in Section 5.2, if it is empty, *decision* of the preceding cp-set will define the state of these places. We fix this issue by adding the following conjunction to the

```
# Flow #
51 = or(49,50)  # Outer disjunction of Flow #
50 = and(48,-23) # LHS of Flow, -23 is "no set enabled", 48 is "fire one transition" #
49 = and(23,45) # RHS of FLow, 23 is "one set enabled", 45 is decision #

# Fire one Transition sequential #
48 = or(46,47) # "Fire one Transition sequential" #

# Fire transition tB #
47 = and((B,tB),(B,1),-(B,2),(C,2),37,38) # Fire transition tB #

# Fire transition tA #
46 = and((A,tA),(A,1),-(A,2),(C,2),27,28) # Fire transition tA #

# Decision of all transition sets #
45 = and(34,44)   # All transition sets have to make the decision #

# Decision of set with tA #
44 = or(42,43) # Fire or not fire set with tA #

# Fire transition set with tA #
43 = and(-22,41) # tA set is not enabled, and dont_fire #

# Dont fire transition set with tA #
42 = and(37,22) # tA set is enabled and fire #

# Don_fire set with tA #
41 = and(38,39,40) # dont_fire of set with tA #
40 = or(19,29) # set with tB is enabled or -(C,1) implicates -(C,2) #
39 = or(-(A,2),(A,1)) # Pre places of the set with tA stay the same #
38 = or(-(A,1),(A,2)) # Pre places of the set with tA stay the same #

# Fire set with tA #
37 = and(36,22) # set tA is enabled #
36 = or(35) # one transition of set with tA is fired #
35 = and((A,tA),(A,1),-(A,2),(C,2)) # fire set with tA #

/* Decision for set with transition tB is left out */

# One set is enabled #
23 = or(19,22) # At least one cp-set is enabled #

# set with tA is enabled #
22 = and((A,1),21) # isEnabled of set with tA #
21 = or(20) # One transition of the set has to be strategy enabled #
20 = and((A,tA)) # The strategy enables tA #

# set with tB is enabled #
19 = and(18,(B,1)) # isEnabled of set with tB #
18 = or(17) # One transition of the set has to be strategy enabled #
17 = and(B,tB) # The strategy enables tB #
```

Figure 10: The true concurrency encoding for the flow and one iteration of the Petri game in Figure 7.

formula of $dont\_fire$:

$$\bigwedge_{p \in {}^\bullet\mathcal{T}_{M,j}, {}^\bullet p \, = \, \emptyset} (p, i) \Leftarrow (p, i+1)$$

Combined with the previous conjunction on preplaces, we force that the possession of tokens of these places is equivalent in the iterations i and $i+1$. Notice that this definition is necessary because the true concurrent flow cannot define *all other* places that are not included in a firing transition by staying equivalent, which is done by the sequential encoding. While solving the formula, we have no notion of transitions that will be fired since we consider every cp-set individually.

7.3.2 Minimization

Since the formula in Section 3.2 is based on a disjunction quantifying over all transitions, the first true concurrent approach was also quantifying over all transitions individually. This lead to a blow-up in the number of variables in the *qcir* file, since the presets and postsets of transition sets were defined for every transition, even if another transition in the same cp-set defined the same. To give an intuition in the order of magnitude, the encoded winning strategy of the *alarm system* presented in [7] has 6.119.049 variables by quantifying over every transition and 18.897 variables in the updated version. The QBF solver that usually eliminates multiple definitions of the same literals was not able to delete the variables because of the structure of the formula. We defined the pre places and post places for every transition, which lead to different occurrences of semantically identical literals. This is the main reason to construct cp-sets besides defining the true concurrency firing semantics.

Minimization is also possible considering *is_enabled*. This sub formula is used by $flow$, $fire$ and $dont\_fire$ and therefore replicated for every function call. We improved our implementation by saving the literals of *is_enabled* for every transition set and iteration to use the same literal whenever we need the formula.

Regarding $fire\_one\_transition$, we can also differentiate between two cases. Called by the left-hand side of flow, we have to check the enabledness for every transition. If called by $fire$, we can assume, that all pre places possess a token and leave out that part of the formula for performance improvements, which is $\bigwedge_{p \in {}^\bullet t}(p, i)$. Note that we still have to enforce that the strategy enables the transition such that we still need the following part of enabledness: $\bigwedge_{p \in {}^\bullet t \cap \mathcal{P}_S^b}(p, \lambda(t))$.

8 Results

In this section we present our experimental and theoretical results of our bounded synthesis approach in Section 5. We start with benchmark evaluations of the *sequential* and *true concurrent* approach and give an overview of our experience depending different designs of Petri games.

8.1 Experimental Results

We compare the prototype implementation [7] of bounded synthesis for Petri games with sequential firing semantics to our extension of the implementation with the true concurrency firing semantics. We use the benchmarks families of [7] and extend it with our running example.

8.1.1 Benchmark Families

Figure 11 corresponds to the following benchmarks. *CP* is our running example in Figure 2. For reasons of completeness, we recap the models provided as benchmarks.

CP: *Communication Protocol. Parameters:* m receivers.
We have m different receivers and 1 sender. The goal is that all receivers get the message. As long as one did not receive the message, we have to resend it. If all messages are delivered, the system has to terminate.

AS: *Alarm System. Parameters:* m locations.
Ther are m monitored locations that can be intruded by a burglar. All locations can communicate and have to inform each other about the location of the burglary. Also false alarms have to be avoided.

CM: *Concurrent Machines. Parameters:* m machines and k orders.
There are k orders that have to be processed by m machines where each machine can only process one order. The goal is to process all orders even if one machine is disabled.

SR: *Self-reconfiguring Robots. Parameters:* m robots with m tools each and k destroyed tools in total.
There are m different robots with m tools being able to equip one at a time. The environment can destroy k tools and the robots have to maintain processing material with the m tools by reconfiguring themselves.

JP: *Jop Processing. Parameters: m* processors.
A job has to be processed by a subset of m processes, which is chosen by the environment, in ascending order.

DW: *Document Workflow. m* clerks.
m clerks have to affirm or reject a document. The environment decides which clerk obtains the document first. DWs requires all clerks to endorse the document.

8.1.2 Comparison

We compare the two implementations in Figure 11. All tests are executed on a i7-2700K CPU with 3.50GHz and 32GB RAM. The timeout for each run is 1800 seconds. For every benchmark (*Ben.*) we show the parameters (*Par.*), the iteration bound n and the bound of the unfolding b. The configurations are run on the same bounds for both approaches, *sequential* and *true concurrent*. We show the *result*, the *time* needed in seconds and the number of variables in the *qcir* file (*var.*). We focus on the size of the bound that is needed to obtain *SAT* as result, the duration time for both approaches to deliver a result and the direct performance comparison solving with identical bounds.

The goal of true concurrency firing semantics is to decrease the size of the bound n. We decreased bound n by one for *CP* with parameter 2 and also by one for the benchmark *AS*. Since bounded synthesis depends on increasing bounds after not being able to find a suiting result, we compare the performance of the *last bound* with *UNSAT* and the *first bound* with *SAT*. The highest *UNSAT* bound for sequential and *CP* is $n = 11$, needing 25.2 seconds to compute the result. As opposed to this, the true concurrent approach with $n = 10$ as highest *UNSAT* bound only needs 19.9 seconds to terminate. Finding *SAT* with bound $n = 12$ and $n = 11$ for sequential and true concurrent respectively is done in 11.7 seconds and 9.7 seconds. In both comparisons, the true concurrent approach performs better. If we consider the complete bounded synthesis approach that would start at a lower bound and whose termination time is the addition of all runs until a *SAT* result is found, the difference is even higher. The results of *AS* display similar dependencies, terminating with 27.7 seconds as the highest *UNSAT* for sequential and 21.0 seconds for true concurrent. Finding *SAT* performs with the lower bound in 21.3 seconds and for the sequential approach 22 seconds. The differences are smaller than the ones for *CP*, but the proposition is the same. If we compare the bounds where both approaches evaluate to the same result, we cannot make a clear statement on which implementation performs better. For example, *CP* with parameter 2 and bounds $n = 12$ and $b = 2$ terminates faster for the true concurrent approach, but *CP* with parameter 2 and bounds $n = 10$ and $b = 2$ shows a contrary performance. Comparing the performance depending on *SAT* and *UNSAT*, we also have multiple results that do not enable us to make a clear statement.

Decreasing the needed bound is not possible considering the benchmark families *CM, SR, JP, DW, DWs.* Reasons for the requiring of the same bound as the sequential flow are presented in Section 8.2 and depend on the design of the constructed Petri games. We use these results to determine the general usability of our new true concurrent firing semantics. Figure 12 shows the time needed by the sequential approach compared to the time needed by the true concurrent approach. A red dot over the black line shows that the sequential approach terminated faster, whereas a red dot under the line represents a better performance by the true concurrent approach. As we pointed out before, the truth values of the runs with both approaches are always the same for the chosen benchmarks. We can see that we do not have big differences in performance an we *cannot* say that the sequential approach solves Petri games with the same bound faster than the true concurrent approach. It depends on the Petri game that has to be solved, and even on its bounds n and b, and a general assumption cannot be made. For Petri games that can be solved in less than 10 seconds by the sequential approach, we do not see a major difference between both approaches. For benchmark runs that need more solving time, the approach performing better changes depending on the Petri game, but we can see a slight tendency for the sequential approach to be faster. By evaluating all benchmark runs, we see that the sequential approach was about 10% faster in all runs combined, which is a small trade off to the possibility of finding the same winning strategies with lower bounds for the flow of the game. Notice that not all runs are plotted in the graph for readability.

The presented approach is an extension of the sequential encoding. We substitute the $flow$ by a larger formula that has a more complex structure. Since the substitution consists of the old formula and adds restrictions, it is obvious that the size of the formula, i.e. the number of variables, increases. The graphical representation of the difference in variable numbers is shown in Figure 13. The number of runs are ordered by the number of variables constructed by the sequential encoding. We see that the true concurrent approach encodes in all runs a slightly larger formula. After comparing the benchmark results we can estimate a linear increase of the size of the formula and therefore we assume that we do not change the complexity of the problem. Despite having a larger quantified boolean formula to solve, as we see in Figure 12 in some Petri games the formula can be solved faster. Since

				sequential			*true concurrent*		
Ben.	*Par.*	*n*	*b*	*result*	*time*	*var.*	*result*	*time*	*var.*
CP	1	5	2	UNSAT	**5.6**	975	UNSAT	5.7	1604
	1	6	2	SAT	5.7	1269	SAT	5.7	2085
	1	7	2	SAT	5.9	1596	SAT	5.9	2611
	2	10	2	UNSAT	**13.3**	9499	UNSAT	19.9	15116
	2	11	2	UNSAT	25.2	11807	**SAT**	**9.7**	19048
	2	12	2	SAT	11.7	13330	SAT	**11.1**	21647
AS	1	5	2	UNSAT	**14.0**	8661	UNSAT	21.0	11510
	1	6	2	UNSAT	27.7	10583	**SAT**	**21.3**	14424
	1	7	2	SAT	**22.**	12566	SAT	44.3	17511
	1	8	2	SAT	**31.6**	14610	SAT	47.3	20771
CM	2/1	5	3	UNSAT	6.	2056	UNSAT	**4.9**	2829
	2/1	6	3	SAT	6.	2583	SAT	**5.4**	3589
	3/1	5	3	UNSAT	**12.8**	5935	UNSAT	19.1	7512
	3/1	6	3	SAT	**7.3**	7318	SAT	8.2	9379
		...			...			...	
	8/1	5	3	-	timeout	47328	-	timeout	53681
	8/1	6	3	SAT	115.	57365	SAT	**67.3**	65586
SR	2/1	5	2	UNSAT	**7.3**	3831	UNSAT	7.5	5136
	2/1	6	2	SAT	7.7	4743	**SAT**	8.4	6494
	3/1	6	2	UNSAT	**801.6**	40691	UNSAT	1458.4	46332
	3/1	7	2	SAT	**28.**	47789	SAT	149.1	55032
JP	2	6	3	UNSAT	11.2	3601	UNSAT	**9.5**	5747
	2	7	3	SAT	**8.9**	4359	SAT	12.8	7042
	3	7	4	UNSAT	**1004.**	33929	UNSAT	1502.5	40542
	3	8	4	-	timeout	39102	-	timeout	47062
DW	1	7	1	UNSAT	5.6	914	UNSAT	5.6	1461
	1	8	1	SAT	5.6	1144	SAT	**5.5**	1782
	2	9	1	UNSAT	5.7	2157	UNSAT	5.7	3206
	2	10	1	SAT	**5.8**	2591	SAT	5.9	3771
		...			...			...	
	9	23	1	UNSAT	44.8	41098	UNSAT	**41.3**	50141
	9	24	1	SAT	28.4	48095	SAT	**19.3**	57960
DWs	1	4	1	UNSAT	**5.3**	307	UNSAT	5.5	494
	1	5	1	SAT	5.6	440	SAT	5.6	689
		...			...			...	
	8	18	1	UNSAT	65.	29602	UNSAT	**61.4**	36726
	8	19	1	SAT	121.9.	32794	SAT	**122.1**	40337
	9	20	1	UNSAT	421.6	40571	UNSAT	**401.5**	49502
	9	21	1	SAT	**1752.9**	44520	-	timeout	53921

Figure 11: Benchmark results of sequential flow and true concurrent flow.

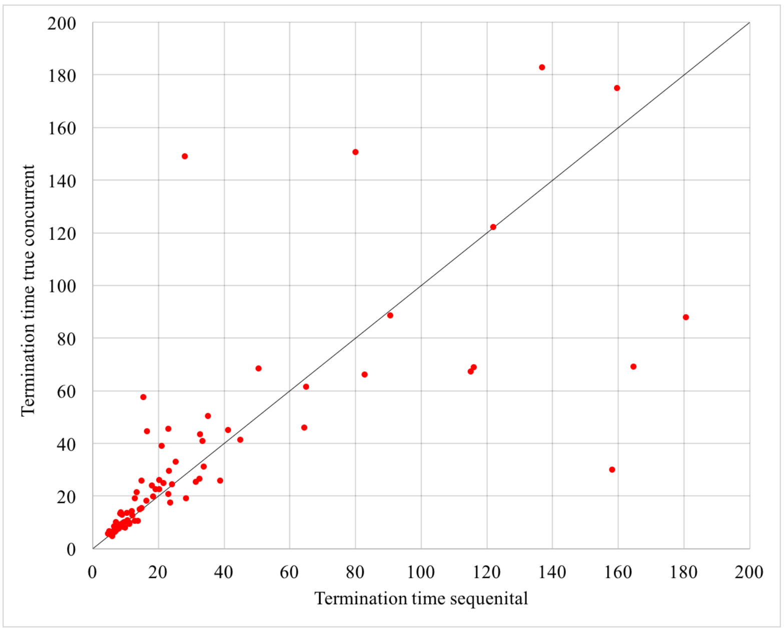

Figure 12: Termination time of the benchmarks *CM, SR, JP, DW, DWs* with sequential firing and true concurrent firing dependent on each other.

the constructed formula is composed of sub formulas that always invalidate their counterpart by evaluating to *true*, it is likely that the QuAbs solver benefits of its structure.

One can see that for games that do not benefit from the true concurrent encoding, the performance of the sequential approach is in general slightly faster than the true concurrent approach, which changes depending on the Petri game to be solved. For Petri games that do benefit, the true concurrent encoding is faster for all benchmarks presented.

8.2 Petri Game Design

Based on the experimental results and the proof of correctness of the true concurrency firing semantics, we present theoretical statements we can make on the applicability of true concurrency on Petri games.

Since we proved the correctness of our approach, we know that the true concurrent semantics never requires a higher bound to find a winning strategy than the sequential approach. In contrast, we can find solutions with lower bounds using the new implementation. Taking advantage of true concurrency highly depends on the design of the Petri game. We require multiple

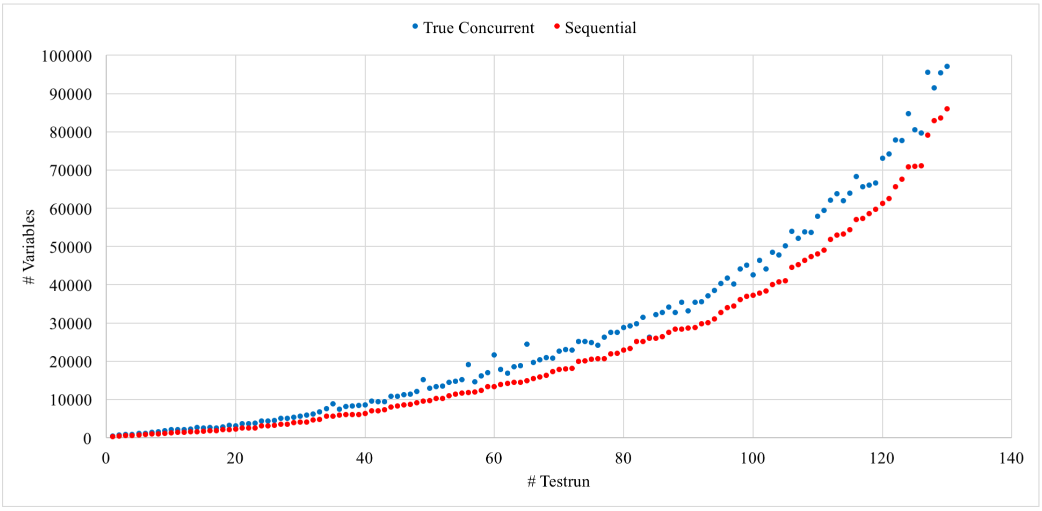

Figure 13: The number of variables in the *qcir* file of the sequential and true concurrent approach on a subset of all test runs sorted by the number of variables needed by the sequential run.

independent "decisions" at one marking. The benchmark family *AS* is implemented by a Petri game that has two independent system decisions before terminating, which can be handled in one iteration with the new approach. We are able to construct nets where multiple of these decisions are needed and every two cp-sets that can be fired true concurrently decrements the bound by one. In some cases, cp-sets containing only system transitions can be designed as a single cp-set combining all decisions in one. This increases the complexity of the design and is not necessary for some Petri games with the true concurrent implementation, since all the system decisions are made in one iteration step.

A different way to benefit from true concurrency is implemented by the *CP* benchmark. We scale up the parameter of the problem by adding sub games to the Petri game. E.g., if we have m as parameter, we consider m Petri games that are assembled as one Petri game, combining all decisions of the sub games and initializing all games again after a global decision. For *CP*, with m sub games we also consider m environment tokens to be in the game. Multiple environment token that fire from the same marking can be fired in one step, reducing the necessary bound to find a winning strategy. Multiple decisions of the environment can often also be reduced to one, maybe more complex, decision that is handled by the sub games afterwards. This cannot be done in general and if possible, the following system transitions are informed of all environment decisions. Constructing games this way is most

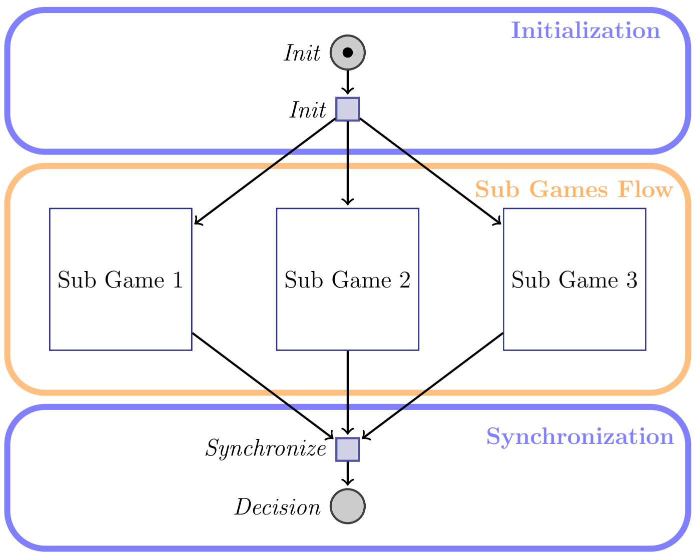

Figure 14: An overview of the sub game design of Petri games with 3 independent sub games.

likely to not benefit from our new approach. We show an overview of the sub game idea in Figure 14. The Petri game is assembled by “independent” sub Petri games, that can fire separately. After the flow of the sub games finished, the sub games synchronize, the outer Petri game collects all information and decides the next step afterwards. Without having the sub games to interact while firing, the flows can be computed true concurrently. Notice that the benefit is not as good as it seems. We encode the whole game as cp-sets and do not have a notion of sub games that can be fired true concurrently. The true concurrency is limited to markings where multiple of the sub games have true concurrently enabled cp-sets. We discuss this issue in more detail in Section 10.2.3

As presented before, we can decrease the bound in one of the previous benchmark families and in our newly introduced benchmark *CP*. One of the reasons we cannot decrease the size in the other benchmark families are limitations of the corresponding Petri games. The tool Adam [8] which is used to compare the bounded synthesis of Petri games prototype implementation results in [7], implements the bounded synthesis approach of [10], which is limited to only one environment token. The benchmarks we adopted from this paper cannot benefit from the *multiple environment token* statement per construction. By designing new benchmark families for future investigation of Petri games, one can consider distributing decisions of environment and system tokens to sub games, which would take advantage of true concurrency.

9 Related Work

In this section, we discuss related work of this thesis. We introduce a bounded synthesis approach for reactive systems and its implementation and a synthesis tool for Petri games.

Faymonville et al. present a bounded synthesis approach on linear temporal logic (LTL) [4]. The given *LTL* specification for a reactive system is transformed to a $\omega-$automaton satisfying the specification. The language of the automaton is then synthesized as a constraint system. Synthesizing creates a transition system that implements the synthesized LTL formula. The constraint system can be formalized as boolean formula (SAT), quantified boolean formula (QBF), and dependency quantified boolean formula (DQBF), which is then solved by a respective solver. The maximal size of the (transition) system is bounded for the synthesis approach and is changed if no system can be synthesized with the given bound. They present an evaluation of the different encodings and select QBF as the best performing encoding.

The encodings in [4] are synthesizing reactive systems, where the complexity of input and output in the system processes can be chosen. In contrast, Petri games model distributed systems and either complete information synchronization or no information synchronization of the simulated processes. The specification is delivered as LTL formula, whereas Petri games implicitly define such a specification with bad places. The limitations of the bounds are similar in both encodings, since the limitations of the size of the transition system corresponds to limiting the size of the winning strategy.

Faymonville et al. present *BoSy* which is an *experimentation framework for bounded synthesis* [5]. The tool implements the encodings of bounded synthesis in [4] and synthesizes transition systems for given LTL specifications. The tool is admissible for multiple inputs, i.e. LTL specifications, signatures and Mealy/More machines. The input is transformed to LTL and a representing automaton is constructed. Based on the automaton, an encoding of the constraint system is created in either SMT, SAT, QBF or DQBF. The tool outputs an implementation in case of finding a solution for the encoding or increases the bound in case of UNSAT. Using the QBF encoding, which performs best respected to the benchmark evaluations of [5], the formula can be solved by different QBF solvers including QuAbS which is also used to solve the encoding of this thesis.

The *BoSy* tool is a different implementation of bounded synthesis compared to out prototype implementation, but with a similar structure of solving the bounded synthesis problem. Whereas *BoSy* is also presented for benchmarking different solver approaches and encodings, the prototype implementation focuses on finding winning strategies of a Petri game without providing choices in different encodings.

Finkbeiner et al. present the tool ADAM which implements *causality-based synthesis for distributed systems.* The tool synthesizes a winning strategy for a Petri game with a safety objective and only one environment player. With this restriction, Petri games are EXPTIME-complete [9]. ADAM implements a reduction of a Petri game to a two-player game over finite graphs. The symbolic representation of the game is encoded to BDDs and solved afterwards. Input and output formats are the same as for our prototype implementation and we can evaluate both implementations with the same benchmarks limiting the number of environment players to one. Compared to our prototype implementation, finding a bound for the Petri game is not necessary and therefore also the reasoning of useful bounds. ADAM cannot focus on small implementations and can only solve a subset of Petri games. A more extensive evaluation of the ADAM tool and the prototype implementation of bounded synthesis [7], which is utilized for the implementation of the true concurrency semantics, can be found in [7].

10 Conclusion

In this section we summarize our presented work on true concurrency semantics of Petri games and its implementation as a prototype for bounded synthesis. We recall the main aspects of the semantics and the evaluated results. We then present possible extensions in future work.

10.1 Summary

We introduced a true concurrency firing semantics for Petri games that allows firing transitions true concurrently. The motivation of this thesis is the minimization of the bound of the flow needed to find a winning strategy with the bounded synthesis approach. Based on bounded synthesis, firing multiple transitions in one iteration of the flow reduces this bound every time it is possible. We force the flow of the Petri game to fire true concurrently as soon as it is possible, which guarantees that the new semantics are not increasing the number of possible runs of a Petri game.

We compute transition sets, called *cp-sets* that include transitions with pairwise common presets. These transitions are pairwise synchronizing transitions since the tokens in the preset of the constructed *cp-sets* share information by firing one of the transitions. Sharing token's past by synchronizing is the main aspect of Petri games and is therefore necessary for the solving of Petri games. Firing multiple transitions is only possible if all transitions in cp-sets are enabled. With this condition we ensure that we do not loose possible synchronizations by firing transitions before all transitions with common presets are enabled. Otherwise we would restrict the flow of the Petri game in its synchronization behaviour. The true concurrent flow is defined as firing *all enabled cp-sets* as long as at least one is enabled. If none is enabled, we fire *one enabled transition* sequentially in this iteration.

The bounded synthesis approach for Petri games with sequential firing semantics presents a 2-QBF formula that encodes a winning strategy of a game. We substitute the part of the formula describing the $flow$ of the game with a new formula for the true concurrent flow. We change the $flow$ from a disjunction quantifying over all transitions, which chooses one transition to fire, to a conjunction quantifying over all *cp-sets* of the game. The formula encodes the existence of at least one enabled cp-set and if a cp-set is fired or not. The remaining parts of the formula are not modified and inherited from the encoding with sequential firing.

We proved the correctness of the flow by building reordered equivalence classes of runs and proving that there exists one reordered run for every run, that can be translated to a true concurrent run. We also showed that every true concurrent run can be transformed to a sequential run and proved that the number of possible true concurrent runs of a Petri game is smaller or equal to the number of possible sequential runs of the same game. Proving the correctness justifies the implementation of the introduced formula.

We implemented the encoding of the QBF formula as an extension in Java to a prototype implementation of the existing sequential encoding. We ran benchmarks that showed that the performance of the new encoding is almost as fast as the sequential encoding and provides a decrease of the needed bound of the flow for some benchmark families. We increase the number of variables in the QBF file but cannot find a strict coherence between the size of the formula and the time needed to solve the formula.

We conclude with a new implementation of a true concurrent firing semantics for Petri Games that can be used in place of the sequential firing implementation without losing expressivism of the game. The performance is slightly worse but we gain minimization of the required bounds for the bounded synthesis of a winning strategy.

10.2 Future Work

In this section we present aspects for future work based on this thesis, which occurred while evaluating the introduced true concurrency firing approach. The possible extensions for the introduced semantics aims a higher reduction of the bound compared to the sequential flow.

10.2.1 Firing Multiple Transitions per CP-Set

As mentioned in Section 4.1, the current semantics cannot fire multiple transitions of one enabled cp-set, even if both could be fired independently. Figure 5b shows one possibility of transitions being elements of the same cp-set, which requires them to fire independently, but firing both true concurrently is not changing the flow of the game. Since only one is fired and in the next iteration the cp-set is not enabled, we are not able to fire the remaining enabled transition until no other cp-set is enabled. After firing the first transition, the second enabled transition is restricted to be fired sequentially. In this situation we lose a decrease of one depending the bound n and also prohibit possible true concurrent firing in the following flow, since the tokens residing in the preset of the enabled transition are blocked. This extension does not need huge structural changes of the formulas to be implemented but adds more sub formulas to the encoding, whereas the games benefiting of this change are only a small subset of Petri games.

10.2.2 Backwards Analysis

The introduced approach of true concurrent firing is based on an analysis of a *horizontal layer* of transitions. By computing the cp-sets, we search common transitions by going to post transitions and their pre transitions, which we then call *mutual exclusive* transitions. We decide firing and not firing by enabledness of the horizontal transitions connected with each other. Considering the search of true concurrently enabled transitions, this layer suffices to find all possible firing transitions. If no transition set is true concurrently enabled, the transition that is fired is chosen randomly without considering the global marking of the Petri game. Therefore, cp-sets that could be true concurrently enabled in a future marking can fire a single transition sequentially and avoid the true concurrent enabledness in future markings. Preferring transitions that can reach a cp-set and enable it true concurrently could be implemented by analysing the *vertical layer* of the game. Since the flow of a game is infinite in general, we would have to set a bound on the analysis depth going backwards in the game, otherwise, considering a bounded unfolding, we would always find a transition before the currently analysed cp-set. The size of the formula would increase again, depending on the bound, and an evaluation of performance would be necessary to find applicable bounds.

10.2.3 Independent Sub Games

We presented a construction of Petri games consisting of multiple sub games that can run independently in Section 8.2. Being able to fire transitions true concurrently even if they are not true concurrently enabled is possible if we fire transitions belonging to different sub games in one iteration step. This extension depends on the *backwards analysis* since we still have to figure out if no true concurrent enabled transition will be enabled in a future marking. We are also forced to handle the synchronization after completing the sub games and find a way to not invalidate the sequential flow of the game. Implementing this idea leads to constructing sets of cp-sets that can be fired true concurrently without restrictions to the game and relies to another preprocessing of the Petri game before creating the QBF formula. Notice that this is a contrary approach to the one introduced in this thesis, since we build sets on transitions that are not allowed to fire together.

References

[1] Valeriy Balabanov, Jie-Hong Roland Jiang, Christoph Scholl, Alan Mishchenko, and Robert K. Brayton. *2QBF: Challenges and Solutions*, pages 453–469. Springer International Publishing, Cham, 2016.

[2] Eike Best and Uli Schlachter. Analysis of petri nets and transition systems. In *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015.*, pages 53–67, 2015.

[3] Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In *Handbook of Satisfiability.* IOS Press, 2009.

[4] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. *Encodings of Bounded Synthesis*, pages 354–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[5] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. *BoSy: An Experimentation Framework for Bounded Synthesis*, pages 325–332. Springer International Publishing, Cham, 2017.

[6] Bernd Finkbeiner. Bounded synthesis for petri games. In *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pages 223–237, 2015.

[7] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Symbolic vs. bounded synthesis for petri games. In *SYNT 2017*, 2017.

[8] Bernd Finkbeiner, Manuel Gieseking, and Ernst-Rüdiger Olderog. *Adam: Causality-Based Synthesis of Distributed Systems*, pages 433–439. Springer International Publishing, Cham, 2015.

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

[10] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[11] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[12] Mikolás Janota. Abstraction-based algorithm for 2qbf. 2011.

[13] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. *Solving QBF with Counterexample Guided Refinement*, pages 114–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[14] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. *Solving QBF with Counterexample Guided Refinement*, pages 114–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf qbf solving with qcir.

[16] 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.

[17] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, SFCS '90, pages 746–757 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[18] Lutz Priese and Harro Wimmel. *Petri-Netze*. EXamen. press Series. Springer Berlin Heidelberg, 2008.

[19] Leander Tentrup. Solving QBF by abstraction. *CoRR*, abs/1604.06752, 2016.