On-the-Fly Synthesis for Strictly Alternating Games

Shyam Lal Karra, Kim G. Larsen, Marco Muñiz, and Jiří Srba

Aalborg University, Denmark
{shyamlal,kgl,muniz,srba}@cs.aau.dk

Abstract. We study two-player zero-sum infinite reachability games with strictly alternating moves of the players allowing us to model a race between the two opponents. We develop an algorithm for deciding the winner of the game and suggest a notion of alternating simulation in order to speed up the computation of the winning strategy. The theory is applied to Petri net games, where the strictly alternating games are in general undecidable. We consider soft bounds on Petri net places in order to achieve decidability and implement the algorithms in our prototype tool. Finally, we compare the performance of our approach with an algorithm proposed in the seminal work by Liu and Smolka for calculating the minimum fixed points on dependency graphs. The results show that using alternating simulation almost always improves the performance in time and space and with exponential gain in some examples. Moreover, we show that there are Petri net games where our algorithm with alternating simulation terminates, whereas the algorithm without the alternating simulation loops for any possible search order.

1 Introduction

An embedded controller often has to interact continuously with an external environment and make decisions in order for the overall system to evolve in a safe manner. Such systems may be seen as (alternating) games, where two players—the controller and the environment—race against each other in order to achieve their individual objectives. The environment and controller alternate in making moves: the environment makes a move and gives the turn to the controller who can correct the behaviour of the system and give the control back to the environment and so on. We consider zero-sum turn-based games where the objective of the controller is to reach a set of goal states, while the objective of the environment is to avoid these states. Winning such a game requires to synthesize a strategy for the moves of the controller, so that any run under this strategy leads to a goal state no matter the moves of the environment. We talk about *synthesizing a winning controller strategy.*

Consider the game in Fig. 1. The game has six configurations $\{s_0, \ldots, s_5\}$ and the solid edges indicate controller moves, while the dashed edges are environmental moves. The game starts at s_0 and the players alternate in taking turns, assuming that the controller has to make the first move. It has three choices, i.e.,

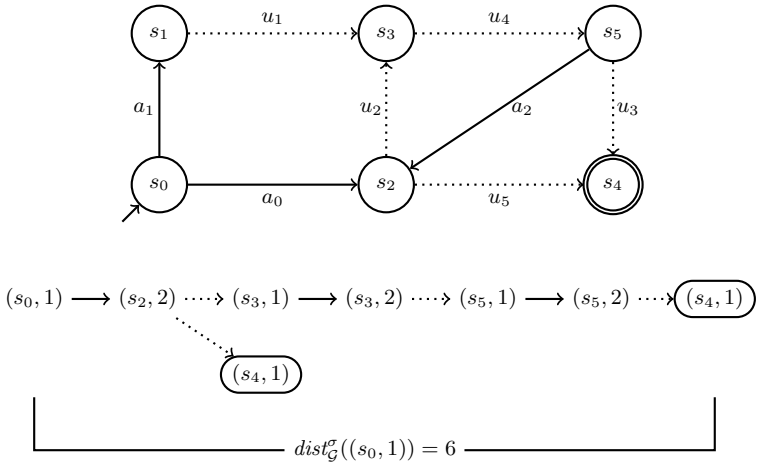


Fig. 1: A game graph and the distance of its winning strategy σ where $\sigma(s_0, 1) = a_0$, $\sigma(s_1, 1) = \sigma(s_2, 1) = \sigma(s_3, 1) = \sigma(s_4, 1) = \sigma(s_5, 1) = \epsilon_1$

move to s_1 or s_2 (shown by solid arrows) or stay in s_0 without moving. Assume that the controller chooses to move to s_2 and gives the control to environment. The environment now has the choice to go to s_3 or s_4 (which is a goal configuration that the environment tries to avoid). Observe that there can be states like s_5 from which both the environment and controller can make a move, based on whose turn it is. Also the game may have states like s_0 or s_3 where one of the players does not have any explicit move in which case they play an empty move and give the control to the other. The controller can play the empty move also in the situation where other controllable moves are enabled, whereas the environment is allowed to make the empty move only if no other enviromental moves are possible (in order to guarantee progress in the game). The goal of the controller is to reach s_4 and it has a winning strategy to achieve this goal as shown below the game graph (here the second component in the pair denotes the player who has the turn).

Petri nets [9] is a standard formalism that models the behaviour of a concurrent system. Petri net games represent the reactive interaction between a controller and an environment by distinguishing controllable transitions from uncontrollable ones. In order to account for both the adversarial behaviour of the environment and concurrency, one can model reactive systems as alternating games (as mentioned before) played on these nets where each transition is designated to either environment or controller and the goal of the controller is to eventually reach a particular marking of the net.

Contribution: We define the notion of alternating simulation and prove that if a configuration c' simulates c then c' is a winning configuration whenever c is a winning configuration. We then provide an on-the-fly algorithm which uses this alternating simulation relation and we prove that the algorithm is partially

correct. The termination of the algorithm is in general not guaranteed. However, for finite game graphs we have total correctness. We apply this algorithm to games played on Petri nets and prove that these games are in general undecidable and therefore we consider a subclass of these games where the places in nets have a bounded capacity, resulting in a game over a finite graph. As an important contribution, we propose an efficiently computable (linear in the size of the net) alternating simulation relation for Petri nets. We also show an example where this specific simulation relation allows for a termination on an infinite Petri net game, while the adaptation of Liu-Smolka algorithm from [2] does not terminate for all possible search strategies that explore the game graph. Finally, we demonstrate the practical usability of our approach on three case studies.

Related Work: The notion of *dependency graphs* and their use to compute fixed points was originally proposed in [7]. In [2,3] an adaptation of these fixed point computations has been extended to two player games with reachability objectives, however, without the requirement on the alternation of the player moves as in our work. Two player coverability games on Petri nets with strictly alternating semantics were presented in [10] and [1], but these games are restricted in the sense that they assume that the moves in the game are monotonic with respect to a partial order which makes the coverability problem decidable for the particular subclass (B-Petri games). Our games are more general and hence the coverability problem for our games played on Petri nets, unlike the games in [10], are undecidable. Our work introduces an on-the-fly algorithm closely related to the work in [3] which is an adaptation of the classical Liu Smolka algorithm [7], where the set of losing configurations is stored, which along with the use of our alternating simulation makes our algorithm more efficient with respect to both space and time consumption. Our main novelty is the introduction of alternating simulation as a part of our algorithm in order to speed up the computation of the fixed point and we present its syntactic approximation for the class of Petri net games with encouraging experimental results.

2 Alternating Games

We shall now introduce the notion of an alternating game where the players strictly alternate in their moves such that the first player tries to enforce some given reachability objective and the other player's aim is to prevent this from happening.

Definition 1. *A* Game graph *is a tuple* $\mathcal{G} = (S, Act_1, Act_2, \longrightarrow_1, \longrightarrow_2, Goal)$ *where:*

- S *is the set of states,*
- Act_i *is a finite set of Player-i actions where* $i \in \{1, 2\}$ *and* $Act_1 \cap Act_2 = \emptyset$*,*
- $\longrightarrow_i \subseteq S \times Act_i \times S$*,* $i \in \{1, 2\}$ *is the Player-i edge relation such that the relations* $\longrightarrow_i$ *are deterministic i.e. if* $(s, \alpha, s') \in \longrightarrow_i$ *and* $(s, \alpha, s'') \in \longrightarrow_i$ *then* $s' = s''$*, and*
- $Goal \subseteq S$ *is a set of goal states.*

If $(s, a, s') \in \longrightarrow_i$ then we write $s \xrightarrow{a}_i s'$ where $i \in \{1, 2\}$. Also, we write $s \longrightarrow_i$ if there exists an $s' \in S$ and $a \in Act_i$ such that $s \xrightarrow{a}_i s'$; otherwise we write $s \not\longrightarrow_i$. In the rest of this paper, we restrict ourselves to game graphs that are *finitely branching*, meaning that for each action a (of either player) there are only finitely many a-successors.

Alternating Semantics for Game Graphs Given a game graph $\mathcal{G} = (S, Act_1, Act_2, \longrightarrow_1, \longrightarrow_2, Goal)$, the set $C = (S \times \{1, 2\})$ is the set of configurations of the game graph and $C_1 = (S \times \{1\}), C_2 = (S \times \{2\})$ represent the set of configurations of *Player-1* and *Player-2* respectively, $\Longrightarrow \subseteq (C_1 \times (Act_1 \cup \{\epsilon_1\}) \times C_2) \cup (C_2 \times (Act_2 \cup \{\epsilon_2\}) \times C_1)$ is the set of transitions of the game such that

- if $(s, \alpha, s') \in \longrightarrow_i$ then $((s, i), \alpha, (s', 3 - i)) \in \Longrightarrow$
- for all configurations $(s, 1) \in C_1$ we have $((s, 1), \epsilon_1, (s, 2)) \in \Longrightarrow$
- for all configurations $(s, 2) \in C_2$ where $s \not\longrightarrow_2$ we have $((s, 2), \epsilon_2, (s, 1)) \in \Longrightarrow$

Apart from the actions available in Act_1 and Act_2, the controller and the environment have two empty actions ϵ_1 and ϵ_2, respectively. Given a state $s \in S$, when it is its turn, the controller or the environment can choose to take any action in Act_1 or Act_2 enabled at s respectively. While, the controller can always play ϵ_1 at a given state s, the environment can play ϵ_2 only when there are no actions in Act_2 enabled at s so that the environment can not delay or deadlock the game.

We write $(s, i) \overset{\alpha}{\Longrightarrow}_i (s', 3 - i)$ to denote $((s, i), \alpha, (s', 3 - i)) \in \Longrightarrow$ and $(s, i) \overset{\alpha}{\Longrightarrow}_i$ if there exists $s' \in S$ such that $(s, i) \overset{\alpha}{\Longrightarrow} (s', 3 - i)$. A configuration $c_g = (s_g, i)$ is called *goal configuration* if $s_g \in Goal$. By abuse of notation, for configuracion c we use $c \in Goal$ to denote that c is a goal configuration. A run in the game $\mathcal{G}$ starting at a configuration c_0 is an infinite sequence of configurations and actions $c_0, \alpha_0, c_1, \alpha_1, c_2, \alpha_2 \ldots$ where for every $j \in \mathbb{N}^0$, $c_j \overset{\alpha_j}{\Longrightarrow} c_{j+1}$. The set $Runs(c_0, \mathcal{G})$ denotes the set of all runs in the game $\mathcal{G}$ starting from configuration c_0. A run $c_0, \alpha_0, c_1, \alpha_1, c_2, \alpha_2 \ldots$ is *winning* if there exists j such that c_j is a goal configuration. The set of all winning runs starting from c_0 is denoted by $WinRuns(c_0, \mathcal{G})$. A *strategy* σ for *Player-1* where $\sigma : C_1 \to Act_1 \cup \{\epsilon_1\}$ is a function such that if $\sigma(c) = \alpha$ then $c \overset{\alpha}{\Longrightarrow}$. For a given strategy σ, and a configuration $c_0 = (s_0, i_0)$, we define $OutcomeRuns^{\sigma}_{\mathcal{G}}(c_0) = \{c_0, \alpha_0, c_1, \alpha_1 \ldots \in Runs(c_0, \mathcal{G}) \mid$ for every $k \in \mathbb{N}^0$, $\alpha_{2k+i_0-1} = \sigma(c_{2k+i_0-1})\}$. A strategy σ is *winning* at a configuration c if $OutcomeRuns^{\sigma}_{\mathcal{G}}(c) \subseteq WinRuns(c, \mathcal{G})$. A configuration c is a *winning configuration* if there is a winning strategy at c. A winning strategy for our running example is given in the caption of Figure 1.

Given a configuration $c = (s, i)$ and one of its *winning* strategies σ, we define the quantity $dist^{\sigma}_{\mathcal{G}}(c) := \max(\{n \mid c_0, \alpha_0 \ldots c_n \ldots \in OutcomeRuns^{\sigma}_{\mathcal{G}}(c)$ with $c = c_0$ and $c_n \in Goal$ while $c_i \notin Goal$ for all $i < n\})$. The quantity represents the *distance* to the goal state, meaning that during any play by the given strategy there is a guarantee that the goal state is reached within that many steps. Due

to our assumption on finite branching of our game graph, we get that distance is well defined.

Lemma 1. *Let σ be a winning strategy for a configuration c. The distance function $dist^{\sigma}_{\mathcal{G}}(c)$ is well defined.*

Proof. Should $dist^{\sigma}_{\mathcal{G}}(c)$ not be well defined, meaning that the maximum does not exist, then necessarilty the set $T = \{c_0, \alpha_0 \dots c_n \mid c_0, \alpha_0 \dots c_n \dots \in OutcomeRuns^{\sigma}_{\mathcal{G}}(c)$ where $c = c_0$ and $c_n \in Goal$ with $c_i \notin Goal$ for all $i < n\}$ induces an infinite tree. By definition the game graph $\mathcal{G}$ is a deterministic transition system with finite number of actions and hence the branching factor of $\mathcal{G}$ and consequently of T is finite. By Köning's Lemma the infinite tree T must contain an infinite path without any goal configuration, contradicting the fact that σ is a winning strategy. □

Consider again the game graph shown in Figure 1 where solid arrows indicate the transitions of *Player-1* while the dotted arrows indicate those of *Player-2*. A possible tree of all runs under a (winning) strategy σ is depicted in the figure and its distance is 6. We can also notice that the distance strictly decreases once *Player-1* performs a move according to the winning strategy σ, as well as by any possible move of *Player-2*. We can now observe a simple fact about alternating games.

Lemma 2. *If $(s, 2)$ is a winning configuration in the game $\mathcal{G}$, then $(s, 1)$ is also a winning configuration.*

Proof. Since $(s_0, 1) \stackrel{\epsilon_1}{\Longrightarrow} (s_0, 2)$ and there is a winning strategy from $(s_0, 2)$, we can conclude $(s_0, 1)$ is also a winning configuration. □

We shall be interested in finding an efficient algorithm for deciding the following problem.

Definition 2 (Reachability Control Problem). *For a given game $\mathcal{G}$ and a configuration (s, i), the* reachability control problem *is to decide if (s, i) is a winning configuration.*

3 Alternating Simulation and On-the-Fly Algorithm

We shall now present the notion of alternating simulation that will be used as the main component of our on-the-fly algorithm for determining the winner in the alternating game. Let $\mathcal{G} = (S, Act_1, Act_2, \longrightarrow_1, \longrightarrow_2, Goal)$ be a game graph and let us adopt the notation used in the previous section.

Definition 3. *A reflexive binary relation $\preceq \subseteq C_1 \times C_1 \cup C_2 \times C_2$ is an alternating simulation relation iff whenever $(s_1, i) \preceq (s_2, i)$ then*

- *if $s_1 \in Goal$ then $s_2 \in Goal$,*
- *if $(s_1, 1) \stackrel{a}{\Longrightarrow}_1 (s'_1, 2)$ then $(s_2, 1) \stackrel{a}{\Longrightarrow}_1 (s'_2, 2)$ such that $(s'_1, 2) \preceq (s'_2, 2)$, and*

– *if* $(s_2, 2) \overset{u}{\Longrightarrow}_2 (s_2', 1)$ *then* $(s_1, 2) \overset{u}{\Longrightarrow}_2 (s_1', 1)$ *such that* $(s_1', 1) \preceq (s_2', 1)$.

An important property of alternating simulation is that it preserves winning strategies as stated in the following theorem.

Theorem 1. *Let* (s, i) *be a winning configuration and* $(s, i) \preceq (s', i)$ *then* (s', i) *is also a winning configuration.*

Proof. By induction on k we shall prove the following claim: if $(s, i) \preceq (s', i)$ and (s, i) is a winning configuration with $dist^{\sigma}_{\mathcal{G}}((s, i)) \leq k$ then (s', i) is also a winning configuration.

Base case ($k = 0$): Necessarily (s, i) is a goal state and by the definition of alternating simulation (s', i) is also a goal configuration and hence the claim trivially holds.

Induction step ($k > 0$): Case $i = 1$. Let σ be a winning strategy for $(s, 1)$ such that $(s, 1) \overset{\sigma((s,1))}{\Longrightarrow}_1 (s_1, 2)$. We define a winning strategy σ' for $(s', 1)$ by $\sigma'((s', 1)) = \sigma((s, 1))$. By the property of alternating simulation we get that $(s', 1) \overset{\sigma((s,1))}{\Longrightarrow}_1 (s_1', 2)$ such that $(s_1, 2) \preceq (s_1', 2)$ and $dist^{\sigma}_{\mathcal{G}}((s_1, 2)) < k$. Hence we can apply induction hypothesis and claim that $(s_1', 2)$ has a winning strategy which implies $(s', 1)$ is a winning configuration as well. Case $i = 2$. If $(s', 2) \overset{u}{\Longrightarrow}_2 (s_1', 1)$ for some $u \in Act_2$ then by the property of alternating simulation also $(s, 2) \overset{u}{\Longrightarrow}_2 (s_1, 1)$ such that $(s_1, 1) \preceq (s_1', 1)$ and $dist^{\sigma}_{\mathcal{G}}((s_1, 2)) < k$. By the induction hypothesis $(s_1', 1)$ is a winning configuration and so is $(s', 2)$. □

As a direct corollary of this result, we also know that if $(s, i) \preceq (s', i)$ and (s', i) is not a winning configuration then (s, i) cannot be a winning configuration either. Before we present our on-the-fly algorithm, we need to settle some notation. Given a configuration $c = (s, i)$ we define

– $Succ(c) = \{c' \mid c \Longrightarrow c'\}$,
– $MaxSucc(c) \subseteq Succ(c)$ is a set of states such that for all $c' \in Succ(c)$ there exists a $c'' \in MaxSucc(c)$ such that $c' \preceq c''$, and
– $MinSucc(c) \subseteq Succ(c)$ is a set of states such that for all $c' \in Succ(c)$ there exists a $c'' \in MinSucc(c)$ such that $c'' \preceq c'$.

Remark 1. There can be many candidate sets of states that satisfy the definition of $MaxSucc(c)$ (and $MinSucc(c)$). In the rest of the paper, we assume that there is a way to fix one among these candidates and the theory proposed here works for any such candidate.

Given a game graph $\mathcal{G}$, we define a subgraph of $\mathcal{G}$ denoted by $\mathcal{G}'$, where for all *Player-1* sucessors we only keep the maximum ones and for every *Player-2* succesors we preserve only the minimum ones. In the following lemma we show that in order to solve the reachability analysis problem on $\mathcal{G}$, it is sufficient to solve it on $\mathcal{G}'$.

Definition 4. *Given a game graph* $\mathcal{G} = (S, Act_1, Act_2, \longrightarrow_1, \longrightarrow_2, Goal)$*, we define a pruned game graph* $\mathcal{G}' = (S, Act_1, Act_2, \longrightarrow'_1, \longrightarrow'_2, Goal)$ *such that* $\longrightarrow'_1 = \{(s_1, 1, s_2) \mid (s_1, 1, s_2) \in \longrightarrow_1 \text{ where } (s_2, 2) \in MaxSucc(s_1, 1)\}$ *and* $\longrightarrow'_2 = \{(s_2, 2, s_1) \mid (s_2, 2, s_1) \in \longrightarrow_2 \text{ where } (s_1, 1) \in MinSucc(s_2, 2)\}$.

Lemma 3. *Given a game graph* $\mathcal{G}$*, a configuration* (s, i) *is winning in* $\mathcal{G}$ *iff* (s, i) *is winning in* $\mathcal{G}'$.

Proof. We prove the case for $i = 1$ (the argument for $i = 2$ is similar).

" $\Longrightarrow$ " : Let $(s, 1)$ be winning in $\mathcal{G}$ under a winning strategy σ. We define a new strategy σ' in $\mathcal{G}'$ such that for every $(s_1, 1)$ where $\sigma((s_1, 1)) = (s'_1, 2)$ we define $\sigma'((s_1, 1)) = (s''_1, 2)$ for some $(s''_1, 2) \in MaxSucc(s_1, 1)$ such that $(s'_1, 2) \preceq (s''_1, 2)$. By induction on $dist^{\sigma}_{\mathcal{G}}((s, 1))$ we prove that if $(s, 1)$ is winning in $\mathcal{G}$ then it is winning also in $\mathcal{G}'$. Base case ($dist^{\sigma}_{\mathcal{G}}((s, 1)) = 0$) clearly holds as $(s, 1)$ is a goal configuration. Let $dist^{\sigma}_{\mathcal{G}}((s, 1)) = k$ where $k > 0$ and let $\sigma((s, 1)) = (s', 2)$ and $\sigma'((s, 1)) = (s'', 2)$. Clearly, $dist^{\sigma}_{\mathcal{G}}((s', 2)) < k$ and by induction hypothesis $(s', 2)$ is winning also in $\mathcal{G}'$. Because $(s', 2) \preceq (s'', 2)$ we get by Theorem 1 that $(s'', 2)$ is also winning and $dist^{\sigma}_{\mathcal{G}}((s'', 2)) < k$. Since $s \xrightarrow{\sigma'(s,1)}{}'_1 s''$ we get that σ' is a winning strategy for $(s, 1)$ in $\mathcal{G}'$.

" $\Longleftarrow$ " : Let $(s, 1)$ be winning in $\mathcal{G}'$ by strategy σ'. We show that $(s, 1)$ is also winning in $\mathcal{G}$ under the same strategy σ'. We prove this fact by induction on $dist^{\sigma'}_{\mathcal{G}'}((s, 1))$. If $dist^{\sigma'}_{\mathcal{G}'}((s, 1)) = 0$ then $(s, 1)$ is a goal configuration and the claim follows. Let $dist^{\sigma'}_{\mathcal{G}'}((s, 1)) = k$ where $k > 0$ and let $\sigma'(s, 1) = (s'', 2)$. Clearly $dist^{\sigma'}_{\mathcal{G}'}((s'', 2)) < k$. So by induction hypothesis, $(s'', 2)$ is a winning configuration in $\mathcal{G}$. Since $s \xrightarrow{\sigma'(s,1)}{}_1 s''$ we get that σ' is a winning strategy for $(s, 1)$ in $\mathcal{G}$. □

We can now present an algorithm for the reachability problem in alternating games which takes some alternating simulation relation as a parameter, along with the game graph and the initial configuration. In particular, if we initialize $\preceq$ to the identity relation, the algorithm results essentially (modulo the additional *Lose* set construction) as an adaption of Liu-Smolka fixed point computation as given in [3]. Our aim is though to employ some more interesting alternating simulation that is fast to compute (preferably in syntax-driven manner as we demonstrate in the next section for the case of Petri net games). Our algorithm uses this alternating simulation at multiple instances which improves the efficiency of deciding the winning and losing status of configurations without having to explore unnecesary state space.

The algorithm uses the following data structures:

- W is the set of edges (in the alternating semantics of game graph) that are *waiting* to be processed,
- *Disc* is the set of already *discovered* configurations,
- *Win* and *Lose* are the sets of currently known *winning* resp. *losing* configurations, and
- D is a *dependency* function that to each configuration assigns the set of edges to be reinserted to the waiting set W whenever the configuration is moved to the set *Win* or *Lose* by the help functions AddToWin resp. AddToLose.

Algorithm 1 Game graph algorithm

Input Game graph $\mathcal{G} = (S, Act_1, Act_2, \longrightarrow_1, \longrightarrow_2, G)$, alternating simulation $\preceq$ and the initial configuration (s_0, i_0)

Output true if (s_0, i_0) is a winning configuration, false otherwise

1: $W = \emptyset$; $Disc := \{(s_0, i_0)\}$; $Lose := \emptyset$; $Win := \emptyset$; $D((s_0, i_0)) = \emptyset$;
2: If $s_0 \in G$ then AddToWin$((s_0, i_0))$;
3: AddSuccessors(W, s_0, i_0);
4: **while** (
5: $\nexists(s_0', i_0).(s_0', i_0) \preceq (s_0, i_0) \wedge (s_0', i_0) \in Win$ **and**
6: $\nexists(s_0', i_0).(s_0, i_0) \preceq (s_0', i_0) \wedge (s_0', i_0) \in Lose$ **and**
7: $W \neq \emptyset$
8:) **do**
9: Pick $(s_1, i, s_2) \in W$; $W := W \setminus \{(s_1, i, s_2)\}$;
10: **if** $(s_1, i) \in Win \cup Lose$ **then** Continue at line 4;
11: **if** (
12: $\exists s_1'.\ (s_1, i) \preceq (s_1', i) \wedge (s_1', i) \in Lose$ **or**
13: $i = 1 \wedge \forall s_3.s_1 \Longrightarrow_1 s_3$ implies $\exists s_3'.(s_3, 2) \preceq (s_3', 2) \wedge (s_3', 2) \in Lose$ **or**
14: $i = 2 \wedge \exists s_3, s_3'.(s_3, 1) \preceq (s_3', 1) \wedge s_1 \Longrightarrow_2 s_3 \wedge (s_3', 1) \in Lose$ **or**
15: $i = 2 \wedge \exists s_1'.\ (s_1, 1) \preceq (s_1', 1) \wedge (s_1', 1) \in Lose$ **or**
16: $s_1 \not\longrightarrow_1 \wedge s_1 \not\longrightarrow_2$
17:) **then**
18: AddToLose(W, s_1, i); Continue at line 4;
19: **end if**
20: **if** (
21: $\exists s_1'.\ (s_1', i) \preceq (s_1, i) \wedge (s_1', i) \in Win$ **or**
22: $i = 1 \wedge \exists s_3, s_3'.(s_1, 1) \Longrightarrow_1 (s_3, 2) \wedge (s_3', 2) \preceq (s_3, 2) \wedge (s_3', 2) \in Win$ **or**
23: $i = 2 \wedge \forall s_3.(s_1, 2) \Longrightarrow_2 (s_3, 1)$ implies $\exists s_3'.(s_3', 1) \preceq (s_3, 1) \wedge (s_3', 1) \in Win$
24:) **then**
25: AddToWin(W, s_1, i); Continue at line 4;
26: **end if**
27: **if** $(s_2, 3 - i) \notin Win \cup Lose$ **then**
28: **if** $(s_2, 3 - i) \in Disc$ **then**
29: $D((s_2, 3 - i)) := D((s_2, 3 - i)) \cup \{(s_1, i, s_2)\}$;
30: **else**
31: $Disc := Disc \cup \{(s_2, 3 - i)\}$;
32: $D((s_2, 3 - i)) := \{(s_1, i, s_2)\}$;
33: **if** $s_2 \in G$ **then**
34: AddToWin$(W, s_2, 3 - i)$;
35: **else**
36: AddSuccessors$(W, s_2, 3 - i)$;
37: **end if**
38: **end if**
39: **end if**
40: **end while**
41: **if** $\exists s_0'.\ s_0' \preceq s_0 \wedge (s_0', i_0) \in Win$ **then return** true
42: **else return** false
43: **end if**

Algorithm 2 Helper procedures for Algorithm 1

```
1: procedure AddSuccessors(W, s, i)
2:     if (i = 1) then
3:         L := {(s, i, s') | such that (s', 3 − i) ∈ MaxSucc(s,i)};
4:     end if
5:     if (i = 2) then
6:         L := {(s, i, s') | such that (s', 3 − i) ∈ MinSucc(s,i)};
7:     end if
8:     W := W ∪ L
9: end procedure
10: procedure AddToWin(W, s, i)
11:     Win := Win ∪ {(s, i)};
12:     W := W ∪ D(s, i) \ {(s'', 3 − i, s) | (s'', 3 − i) ∈ (Win ∪ Lose)};
13: end procedure
14: procedure AddToLose(W, s, i)
15:     Lose := Lose ∪ {(s, i)};
16:     W := W ∪ D(s, i) \ {(s'', 3 − i, s) | (s'', 3 − i) ∈ (Win ∪ Lose)};
17: end procedure
```

As long as the waiting set W is nonempty and no conclusion about the initial configuration can be drawn, we remove an edge from the waiting set and check whether the source configuration of the edge can be added to the losing or winning set. After this, the target configuration of the edge is explored. If it is already discovered, we only update the dependencies. Otherwise, we also check if it is a goal configuration (and call AddToWin if this is the case), or we add the outgoing edges from the configuration to the waiting set.

In order to argue about the correctness of the algorithm, we introduce a number of loop invariants for the while loop in Algorithm 1 in order to argue that if the algorithm terminates then $(s_0, turn)$ is winning if and only if $(s_0, turn) \in Win$.

Lemma 4. *Loop Invariant 1 for Algorithm 1: If* $(s, i) \in Win$ *and* $(s, i) \preceq (s', i)$ *then* (s', i) *is a winning configuration.*

Proof Sketch. Initially, $Win = \emptyset$ and the invariant holds trivially before the loop is entered. Let us assume that the invariant holds before we execute the body of the while loop and we want to argue that after the body is executed the invariant still holds. During the current iteration, a new element can be added to the set *Win* at lines 25 or 34. If line 25 is executed then at least one of the conditions at lines 21, 22, 23 must hold. We argue in each of these cases, along with the cases in which line 34 is executed, that the invariant continues to hold after executing the call to the function AddToWin. □

Lemma 5. *Loop Invariant 2 for Algorithm 1: If* $(s, i) \in Lose$ *and* $s' \preceq s$ *then* (s', i) *is not a winning configuration.*

Proof Sketch. Initially $Lose = \emptyset$ and loop invariant 2 holds trivially before the while loop at line 4 is entered. Similarly to the previous lemma, we argue that

in all the cases where the function AddToLose is called, the invariant continues to hold after the call as well. □

The next invariant is essential for the correctness proof.

Lemma 6. *Loop Invariant 3 for Algorithm 1: During the execution of the algorithm for any* $(s, j) \in Disc \setminus (Win \cup Lose)$ *invariantly holds*

(a) if $j = 1$ *then for every* $(s', 2) \in MaxSucc(s, 1)$
- ***I.*** $(s, 1, s') \in W$ *or*
- ***II.*** $(s, 1, s') \in D(s', 2)$ *and* $(s', 2) \in Disc \setminus Win$ *or*
- ***III.*** $(s', 2) \in Lose$

(b) if $j = 2$ *then for every* $(s', 1) \in MinSucc(s, 2)$
- ***I.*** $(s, 2, s') \in W$ *or*
- ***II.*** $(s, 2, s') \in D(s', 1)$ *and* $(s', 1) \in Disc \setminus Lose$ *or*
- ***III.*** $(s', 1) \in Win$.

This third invariant claims that for any discovered but yet undetermined configuration (s, j) and for any of its outgoing edges (maximum ones in case it is *Player-1* move and minimum ones for *Player-2* moves), the edge is either on the waiting list, or the target configuration is determined as winning or losing, or otherwise the edge is in the dependecy set of the target configuration, so that in case the target configuration is later on determined as winning or losing, the edge gets reinserted to the waiting set and the information can possibly propagate to the parent configuration of the edge. This invariant is crucial in the proof of partial correctness of the algorithm which is given below.

Theorem 2. *If the algorithm terminates and returns true then the configuration* (s_0, i) *is winning and if the algorithm upon termination returns false then the configuration* (s_0, i) *is losing.*

Proof. Upon termination, if the algorithm returns true (this can happen only at the Line 41), then it means there exists a $s'_0 \in S$ such that $(s'_0, i) \in Win$ and $(s'_0, i) \preceq (s_0, i)$. From Lemma 4, we can deduce that (s_0, i) is winning. On the other hand if the algorithm returns false, then there are two cases.

Case 1: There exists a $s'_0 \in S$ such that $(s_0, i) \preceq (s'_0, i)$ and $(s'_0, i) \in Lose$. From Lemma 5, we can deduce that (s_0, i) does not have a winning strategy.

Case 2: $W = \emptyset$ and there exists no $s'_0 \in S$ such that $(s'_0, i) \in Win$ and $(s'_0, i) \preceq (s_0, i)$ and there exists no $s'_0 \in S$ such that $(s'_0, i) \in Lose$ and $(s_0, i) \preceq (s'_0, i)$. We prove that (s_0, i) is not winning for *Player-1* by contradiction. Let (s_0, i) be a winning configuration. Let σ be a *Player-1* winning strategy. Let $L = Disc \setminus (Win \cup Lose)$. Now we make the following two observations.

From Lemma 6, we can deduce that for all $(s, 1) \in L$, if $(s, 1) \Longrightarrow_1 (s_1, 2)$ then there exists an s'_1 such that $(s_1, 2) \preceq (s'_1, 2)$ and $(s'_1, 2) \in L$ or $(s'_1, 2) \in Lose$. Also, we can deduce that the case that for all $(s_2, 1) \in Succ(s, 2)$ we have $(s_2, 1) \in Win$ is not possible as follows. Among all $(s_2, 1) \in Succ(s, 2)$ consider the last $(s_2, 1)$ entered the set *Win*. During this process in the call to function AddToWin, the edge $(s, 2, s_2)$ is added to the waiting list. Since by the end

of the algorithm, this edge is processed and it would have resulted in adding $(s, 2)$ to *Win* because of the condition at line 23. But this is a contradiction as $(s, 2) \in L$ and $L \cap \mathit{Win} = \emptyset$. Hence for all $(s, 2) \in L$, there exists an s_2 such that $(s, 2) \Longrightarrow_2 (s_2, 1)$ and $(s_2, 1) \in L$.

Given these two observations, and the definition of $\preceq$, one can show that *Player-2* can play the game such that for the resulting run $\rho \in \mathit{OutcomeRuns}^{\sigma}_{\mathcal{G}}(s_0)$ where $\rho = \langle (s_0, i), a_0, (s_1, 3-i), a_1 \ldots \rangle$, there exists a $\rho' = \langle (s'_0, i), b_0, (s'_1, 3-i), b_1 \ldots \rangle \in \mathit{Runs}(s_0, \mathcal{G})$ such that $s'_0 = s_0$, for all $k \in \mathbb{N}$, $(s_k, 2-((i+k)\%2)) \preceq (s'_k, 2-((i+k)\%2))$ and $(s'_k, 2-((i+k)\%2)) \in L$. In other words the configurations in the run ρ are all (one by one) simulated by the corresponding configurations in the run ρ' and the configurations from the run ρ' moreover all belong to the set L. Since σ is a winning strategy, there must exist an index $j \in \{0, 1, 2, \ldots\}$ such that $s_j \in G$. Since the set of goal states are upward closed, it also means $s'_j \in G$. But $L \cap \{(s, 1), (s, 2) \mid s \in G\} = \emptyset$ because the line 34 of the algorithm adds a configuration in $\{(s, 1), (s, 2) \mid s \in G\}$ to *Win* when it enters the set *Disc* for the first time. Hence our assumption that (s_0, i) has a winning strategy is contradicted. □

The algorithm does not necessarily terminate on general game graphs, however, termination is guaranteed on finite game graphs.

Theorem 3. *Algorithm 1 terminates on finite game graphs.*

Proof. In the following, we shall prove that any edge $e = (s_1, i, s_2)$ can be added to W at most twice and since there are only finitely many edges and every iteration of the while loop removes an edge from W (at line 9), W eventually becomes empty and the algorithm terminates on finite game graphs.

During the execution of while loop, an edge can only be added W through the call to functions AddSuccessors at line 36, AddToWin at lines 25, 34, or AddToLose at line 18. We shall show that these three functions can add an edge e atmost twice to waiting list W.

Let $e = (s_1, i, s_2)$ be an edge added to W in iteration k through a call to AddToWin at line 34. This implies that, during iteration k, the condition in line 27 is true. Hence $(s_2, 3-i) \notin \mathit{Win} \cup \mathit{Lose}$ before iteration k is executed and after line 34 is executed, $(s_2, 3-i) \in \mathit{Win} \cup \mathit{Lose}$ (hence the condition in line 27 is now false). So the call to function AddToWin at line 34 can not add e to W after iteration k.

Let $e = (s_1, i, s_2)$ be an edge added to W in iteration k through a call to AddToWin at line 25. This implies that, during iteration k, the condition in line 10 is not true. Hence $(s_2, 3-i) \notin \mathit{Win} \cup \mathit{Lose}$ before iteration k is executed and after line 25 is executed, $(s_2, 3-i) \in \mathit{Win} \cup \mathit{Lose}$ (hence the condition in line 10 is false). So the call to function AddToWin at line 25 can not add e to W after iteration k.

Similar to previous cases, we can argue that the call to AddToLose at line 18 can add e to W atmost once. Also it is easy to observe that once $(s_2, 3-i)$ is added to set $\mathit{Win} \cup \mathit{Lose}$ by any of the lines 25, 34, 18, the other two can not add e to W. So, all together, all these three lines can add e to W atmost once.

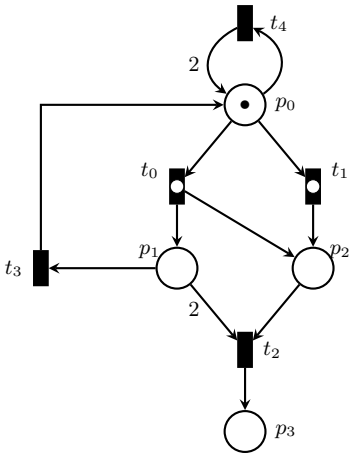


Fig. 2: A Petri net game

Now consider case when $e = (s_1, i, s_2)$ is added to W in iteration k through a call to AddSuccessors at line 36. This implies, during iteration k, the condition in the line 28 is not true. Hence $(s_1, i) \notin \mathit{Disc}$ before the iteration k is executed and after the line 31 is executed $(s_1, i) \in \mathit{Disc}$ by the end of iteration k and the condition in the line 28 is true . So the call to function AddSuccessors at line 36 can add e to W atmost once. In total, edge e can enter W atmost twice. □

4 Application to Petri Games

We are now ready to instantiate our general framework to the case of Petri net games where the transitions are partitioned between the controller and environment transitions and we moreover consider a soft bound for the number of tokens in places (truncating the number of tokens that exceed the bound).

Definition 5 (Petri Net Games). *A* Petri Net Game $\mathcal{N} = (P, T_1, T_2, W, B, \varphi)$ *is a tuple where*

- P *is a finite set of places,*
- T_1 *and* T_2 *are finite sets of transitions such that* $T_1 \cap T_2 = \emptyset$*,*
- $W : (P \times (T_1 \cup T_2)) \cup ((T_1 \cup T_2) \times P) \to \mathbb{N}^0$ *is the weight function,*
- $B : P \to \mathbb{N}^0 \cup \{\infty\}$ *is a function which assigns a (possible infinite) soft bound to every* $p \in P$*, and*
- φ *is a formula in* coverability logic *given by the grammar* $\varphi ::= \varphi_1 \wedge \varphi_2 \mid p \geq c$ *where* $p \in P$ *and* $c \in \mathbb{N}^0$*.*

A marking is a function $M : P \to \mathbb{N}^0$ *such that* $M(p) \leq B(p)$ *for all* $p \in P$*. The set* $\mathcal{M}(\mathcal{N})$ *is the set of all markings of* $\mathcal{N}$*.*

In Figure 2 we show an example of a Petri game where $P = \{p_0, \ldots, p_3\}$ are the places, there are three controller transitions $T_1 = \{t_2, t_3, t_4\}$ and two environment transitions (indicated with a circle inside) $T_2 = \{t_0, t_1\}$. The function W assigns each arc a weight of 1, except the arcs from p_1 to t_2 and t_4 to p_0 that have weight 2. The bound function B assigns every place ∞ and $\varphi ::= p_3 \geq 1$ requires that in the goal marking the place p_3 must have at least one token. The initial marking of the net is $\langle 1, 0, 0, 0 \rangle$ where the place p_0 has one token and the places p_1 to p_3 (in this order) have no tokens.

For the rest of this section, let $\mathcal{N} = (P, T_1, T_2, W, B, \varphi)$ be a fixed Petri net game. The satisfaction relation for a coverability formula φ in marking M is defined as expected:

- $M \models p \geq n$ iff $M(p) \geq n$ where $n \in \mathbb{N}^0$, and
- $M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$.

For a given formula φ, we now define the set of goal markings $\mathcal{M}_\varphi = \{M \in \mathcal{M}(\mathcal{N}) \mid M \models \varphi\}$.

Firing a transition $t \in T_1 \cup T_2$ from a marking M results in marking M' if for all $p \in P$ we have $M(p) \geq W((p,t))$ and

$$M'(p) = \begin{cases} M(p) - W((p,t)) + W((t,p)) & \text{if } M(p) - W((p,t)) + W((t,p)) \leq B(p) \\ B(p) & \text{otherwise.} \end{cases}$$

We denote this by $M \xrightarrow{t} M'$ and note that this is a standard Petri net firing transition in case the soft bound for each place is infinity, otherwise if the number of tokens in a place p of the resulting marking exceeds the integer bound $B(p)$ then it is truncated to $B(p)$.

Petri net game $\mathcal{N}$ induces a game graph $(\mathcal{M}(\mathcal{N}), T_1, T_2, \longrightarrow_1, \longrightarrow_2, \mathcal{M}_\varphi)$ where $\longrightarrow_i = \{(M, t, M') \mid M \xrightarrow{t} M' \text{ and } t \in T_i\}$ for $i \in \{1, 2\}$.

For example, the (infinite) game graph induced by Petri game from Figure 2 is shown in Figure 3a. Next we show that reachability control problem for game graphs induced by Petri games is in general undecidable.

Theorem 4. *The reachability control problem for Petri games is undecidable.*

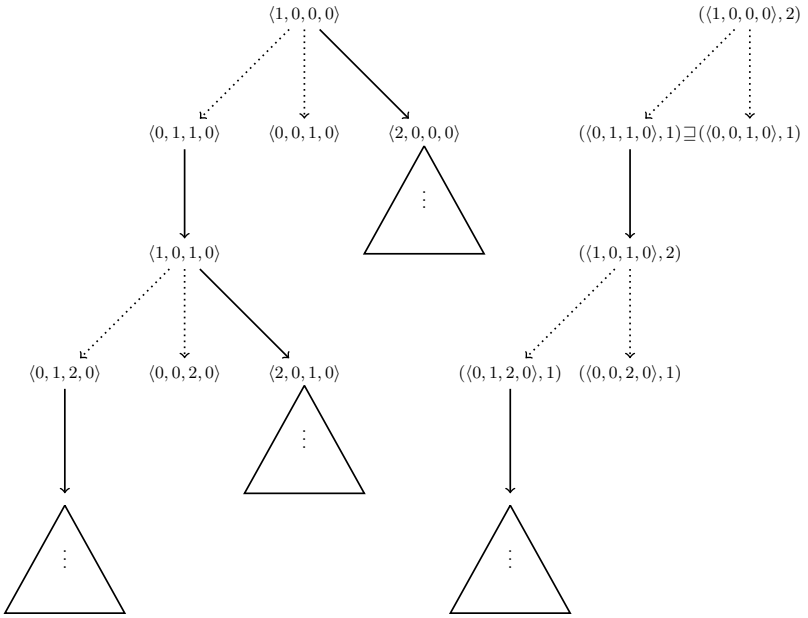
Proof. In order to show that reachability control problem for Petri games is undecidable, we reduce the undecidable problem of reachability for two counter Minsky machine [6] to the reachability control problem.

A Minsky Counter Machine with two non-negative counters c_1 and c_2 is a sequence of labelled instructions of the form:

$$1 : \mathit{instr}_1, 2 : \mathit{instr}_2, \ldots n : \text{HALT}$$

where for all $i \in \{1, 2, \ldots, n-1\}$ each instr_i is of the form

- $\mathit{instr}_i : c_r = c_r + 1$; goto j or
- instr_i : if $c_r = 0$ then goto j else $c_r = c_r - 1$ goto k



(a) Game graph for PN from Figure 2 (b) Alternating semantics for Figure 3a

Fig. 3: Game graph and its alternating semantics

for $j, k \in \{1, 2, \ldots, n\}$ and $r \in \{1, 2\}$.

A *computation* of a Minsky Counter Machine is a sequence of configurations (i, c_1, c_2) where $i \in \{1, 2 \ldots n\}$ is the label of instruction to be performed and c_1, c_2 are values of the counters. The starting configuration is $(1, 0, 0)$. The computation sequence is deterministic and determined by the instructions in the obvious way. The *halting problem* of a Minsky Counter Machine as to decide whether the computation of the machine ever halts (reaches the instruction with label n).

Given a two counter machine, we construct a Petri game $\mathcal{N} = \langle P, T_1, T_2, W, B, \varphi \rangle$ where $P = \{p_1, \ldots, p_n, p_{c_1}, p_{c_2}\} \cup \{p_{i:c_r=0}, p_{i:c_r \neq 0} \mid i \in \{1 \ldots n\}\}$ is the set of places containing place for each of n instructions, a place for each counter and some helper places. The set of transitions is $T_1 = \{t_{i:1}, t_{i:2}, t_{i:3}, t_{i:4} \mid i \text{ is a decrement instruction}\} \cup \{t_i \mid i \text{ is an increment instruction }\}$ and $T_2 = \{t_{i:cheat} \mid i \text{ is a decrement transition }\}$. Figures 4a, 4b define the gadgets that are added for each increment and decrement instruction, respectively. The function B binds every place to 1, except for p_{c_1}, p_{c_2} each of which have a bound ∞. The formula to be satisfied by the goal marking is $\varphi = p_n \geq 1$, stating that a token is put to the place p_n. The initial marking contains just one token in the place p_1 and all other places are empty. We now show that the counter machine halts iff the controller has a winning strategy in the constructed game.

" $\Longrightarrow$ " : Assume that the Minsky machine halts. The controller's strategy is to faithfully simulate the counter machine, meaning that if the controller is in place p_i for a decrement instruction labelled with i, then it selects the transition $t_{i:1}$ in case the counter c_r is not empty and transition $t_{i:2}$ in case the counter is empty. For increment instructions there is only one choice for the controller. As the controller is playing failtfully, the transition $t_{i:cheat}$ is never enabled and hence the place p_n is eventually reached and the controller has a winning strategy.

" $\Longleftarrow$ " : Assume that the Minsky machine loops. We want to argue that there is no winning controller's strategy. For the contradiction assume that the controller can win the game also in this case. Clearly, playing faithfully as described in the previous direction will never place a token into the place p_n. Hence at some point the controller must cheat when executing the decrement instruction (no cheating is possible in increment instruction). There are two cheating scenarios. Either the plase p_{c_r} is empty and the controller fires $t_{i:1}$ which leads to a deadlock marking and clearly cannot reach the goal marking that marks the place p_n. The other cheating move is to play $t_{i:2}$ in case the place p_{c_r} is not empty. However, now in the next round the enviroment has the transition $t_{i:cheat}$ enabled and can deadlock the net by firing it. In any case, it is impossible to mark the place p_n, which contradicts the existence of a winning strategy for the controller. □

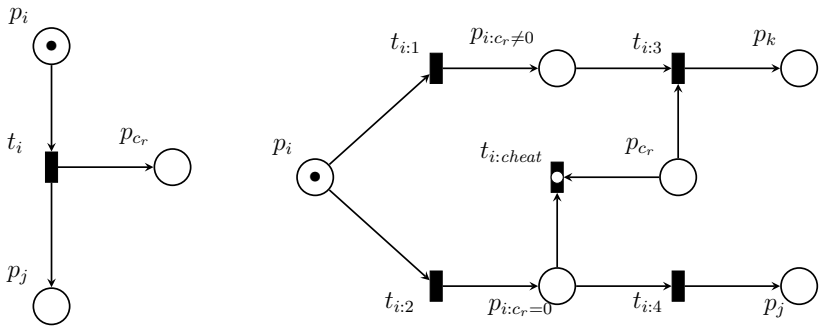
This means that running Algorithm 1 specialized to the case of Petri nets does not necessarity terminates on general unbounded Petri nets (nevertheless, if it terminates even for unbounded nets then it still provides a correct answer). In case a Petri net game is bounded, e.g. for each $p \in P$ we have $B(p) \neq \infty$, we can guarantee also termination of the algorithm.

Theorem 5. *The reachability control problem for bounded Petri net games is decidable.*

Proof. Theorem 2 guarantees the correctness of Algorithm 1 (where we assume that the alternating simulation relation is the identity relation). As the presence of soft bounds for each place makes the state space finite, we can apply Theorem 3 to conclude that Algorithm 1 terminates. Hence the total correctness of the algorithm is established. □

Clearly the empty alternating simulation that was used in the proof of the above theorem guarantees correctness of the algorithm, however, it is not necessarilty the most efficient one. Hence we shall now define a syntax-based and linear time computable alternating simulation for Petri net games. For a transition $t \in T_1 \cup T_2$, let ${}^\bullet t = \{p \in P \mid W(p,t) > 0\}$. We partition the places P in *equality* places P_e and *remaining* places P_r such that $P_e = \bigcup_{t \in T_2} {}^\bullet t$ and $P_r = P \setminus P_e$. Recall that a Petri net game $\mathcal{N}$ induces a game graph with configurations $C_i = \mathcal{M}(\mathcal{N}) \times \{i\}$ for $i \in \{1,2\}$.

Definition 6 (Alternating Simulation for Petri Net Games). *We define a relation $\sqsubseteq \subseteq C_1 \times C_1 \cup C_2 \times C_2$ on configurations such that $(M,i) \sqsubseteq (M',i)$ iff for all $p \in P_e$ we have $M(p) = M'(p)$ and for all $p \in P_r$ we have $M(p) \leq M'(p)$.*



(a) Increment instruction (b) Decrement instruction

Fig. 4: Petri game gadgets for wwo counter machine instructions

Theorem 6. *The relation $\sqsubseteq$ is an alternating simulation and it is computable in linear time.*

Proof. In order to prove that $\sqsubseteq$ is alternating simulation, let us assume that $(M_1, i) \sqsubseteq (M_1', i)$ and we need to prove the three conditions of the alternating simulation relation from Definition 3.

- If (M_1, i) is a goal configuration, then by definition since the set of goal markings are upward closed, (M_1', i) is also a goal configuration.
- If $(M_1, 1) \stackrel{t}{\Longrightarrow} (M_2, 2)$ where $t \in T_1$ then by definition of transition firing for every $p \in P$ holds that $M_1(p) \geq W(p, t)$. Since $(M_1, 1) \sqsubseteq (M_1', 1)$ then for every $p \in P$ holds $M_1'(p) \geq M_1(p)$ implying that $M_1'(p) \geq W(p, t)$. Hence t can be fired at M_1' and let $M_1' \stackrel{t}{\longrightarrow} M_2'$. It is easy to verify that $(M_2, 2) \sqsubseteq (M_2', 2)$.
- The third condition claiming that if $(M_1', 2) \stackrel{t}{\Longrightarrow} (M_2', 1)$ where $t \in T_2$ then $(M_1, 2) \stackrel{t}{\Longrightarrow} (M_2, 1)$ such that $(M_2, 1) \sqsubseteq (M_2', 1)$ follows straingforwardly as for all input places to the environment transition t we assume an equal number of tokens in both M_1 and M_1'.

We can conclude with the fact that $\sqsubseteq$ is an alternating simulation. The relation can be clearly decided in linear time as it requires only comparision of number of tokens for each place in the markings. □

Remark 2. In Figure 3*b*, at the initial configuration $c_0 = (\langle 1, 0, 0, 0\rangle, 2)$ *Player-2* has two transitions enabled. The two successor configurations to c_0 are $c_1 = (\langle 0, 1, 1, 0\rangle, 1)$ and $c_2 = (\langle 0, 0, 1, 0\rangle, 1)$ which are reached by firing the transitions t_0 and t_1 respectively. Since $c_1 \sqsupseteq c_2$, Algorithm 1 explores c_2 and ignores the successor configuration c_1. Hence our algorithm terminates on this game and correctly declares it as winning for *Player-2*.

We now point out the following facts about Algorithm 1 and how it is different from the classical Liu and Smolka's algorithm [7]. First of all, Liu and Smolka

Fire Alarm		Winning	Time (sec.)		States		Reduction in %	
Sensors	Channels		LS	ALT	LS	ALT	Time	States
2	2	true	1.5	0.2	116	19	87.5	83.6
3	2	true	30	0.7	434	56	97.6	87.1
4	2	false	351.7	1.1	934	60	99.7	93.6
5	2	false	2249.8	1.1	1663	63	99.9	96.2
6	2	false	8427.1	1.3	3121	64	99.9	98.0
7	2	false	T.O.	1.4	-	66	-	-
2	3	true	20.2	0.5	385	25	97.5	93.5
2	4	true	622.7	1.0	1233	31	99.8	97.5
2	5	true	10706.7	2.3	3564	37	99.9	98.9
2	6	true	T.O.	3.4	-	43	-	-

Table 1: Fire Alarm System

do not consider any alternating simulation and they do not use the set *Lose* in order to ensure early termination of the algorithm. As a result, Liu and Smolka's algorithm does not terminate (for any search strategy) on the Petri net game from Figure 2, whereas our algorithm always terminates (irrelevant of the search strategy) and provides a conclusive answer. This fact demonstrates that not only our approach is more efficient but it also terminates on a larger class of Petri net games than the previously studied approaches.

5 Implementation and Experimental Evaluation

We implemented both Algorithm 1 (referred to as ALT and using the alternating simulation on Petri nets given in Definition 6) and the classical Liu and Smolka algorithm (referred to as LS) in the prototype tool sAsEt [8]. We present three use cases where each of the experiments is run 20 times and the average of the time/space requirements is presented in the summary tables. The columns in the tables show the scaling, a Boolean value indicating whether the initial configuration has a winning strategy or not, the time requirements, number of explored states (the size of the set *Disc*) and a relative reduction of the running time and state space in percentage. The experiments are executed on AMD Opteron 6376 processor running at 2.3GHz with 10GB memory limit.

5.1 Fire Alarm Use Case

The German company SeCa GmbH produces among other products fire alarm systems. In [4] the formal verification of a wireless communication protocol for a fire alarm system is presented. The fire alarm system consists of a central unit and a number of sensors that communicate using a wireless Time Division Multiple Access protocol. We model a simplified version of the fire alarm system from [4] as a Petri net game and use our tool to guarantee a reliable message passing even

under interference. Table 1 shows that as the number of sensors and channels increases, we achieve an exponential speed up using ALT algorithm compared to LS. The reason is that the alternating simulation significantly prunes out the state space that is necessary to explore.

Model	Winning	Time (sec.)		States		Reduction in %	
		LS	ALT	LS	ALT	Time	States
$s_1a_6d_2w_2$	false	14.4	12.0	278	213	16.7	23.4
$s_1a_6d_2w_3$	false	220.9	101.0	734	527	54.3	28.2
$s_1a_6d_2w_4$	false	2107.7	620.1	1546	1027	70.6	33.6
$s_1a_6d_2w_5$	false	9968.1	2322.3	2642	1672	76.7	36.7
$s_1a_6d_2w_{10}$	true	7159.3	2710.0	2214	1744	62.1	21.2
$s_1a_6d_2w_{11}$	true	8242.0	2662.7	2226	1756	67.7	21.1
$s_1a_6d_2w_{12}$	true	7687.7	2867.2	2238	1768	62.7	21.0
$s_1a_6d_2w_{13}$	true	7819.7	2831.4	2250	1780	63.8	20.9
$s_1a_6d_2w_{14}$	true	7674.1	2960.3	2262	1792	61.4	20.8
$s_1a_6d_2w_{15}$	true	8113.8	2903.7	2274	1804	64.2	20.7
$s_1a_7d_2w_2$	false	35.3	25.8	382	286	26.9	25.1
$s_1a_7d_2w_3$	false	744.7	284.0	1114	783	61.9	29.7
$s_1a_7d_2w_4$	false	9400.1	2637.2	2604	1674	71.9	35.7
$s_1a_7d_2w_5$	false	T.O.	10477.3	T.O.	2961	-	-
$s_4a_6d_1w_2$	true	40.6	28.9	290	250	28.8	13.8
$s_4a_6d_1w_3$	true	60.7	45.3	326	286	25.4	12.3
$s_4a_6d_1w_4$	true	90.4	62.0	362	322	31.4	11.0
$s_4a_6d_1w_5$	true	122.9	93.9	398	358	23.6	10.1
$s_4a_6d_1w_6$	true	172.1	122.6	434	394	28.8	9.2
$s_4a_6d_1w_7$	true	197.7	163.6	470	430	17.2	8.5
$s_4a_6d_1w_8$	true	263.3	190.6	506	466	27.6	7.9
$s_2a_4d_1w_2$	true	1.7	1.9	90	82	-11.8	8.9
$s_2a_4d_1w_3$	true	2.9	3.1	110	102	-6.9	7.3
$s_2a_4d_1w_4$	true	4.3	4.7	130	122	-9.3	6.2
$s_2a_4d_1w_5$	true	6.3	6.4	150	142	-1.6	5.3
$s_2a_4d_1w_6$	true	9.5	8.3	170	162	12.6	4.7
$s_2a_4d_1w_7$	true	11.9	11.7	190	182	1.7	4.2
$s_2a_4d_1w_8$	true	16.4	15.7	210	202	4.3	3.8

Table 2: Student Teacher Use Case

5.2 Student-Teacher Scheduling Use Case

In [5] the student semester scheduling problem is modelled as a workflow net. We extend this Petri net into a game by introducing a teacher that can observe the student behaviour (work on assignments) and there is a race between the two

Model	Winning	Time (sec.)		States		Reduction in %	
		LS	ALT	LS	ALT	Time	States
m211	false	159.2	109.7	525	405	31.1	22.9
m222	false	562.1	298.7	798	579	46.9	27.4
m322	false	5354.2	1914.4	1674	1096	64.2	34.5
m332	false	9491.7	4057.6	2228	1419	57.3	36.3
m333	false	T.O.	8174.5	T.O.	1869	-	-
m2000	false	127.7	50.2	474	341	60.7	28.1
m2100	false	404.6	157.5	762	490	61.1	35.7
m2200	false	776.4	148.3	921	490	80.9	46.8

Table 3: Cat and Mice Use Case

players. In Table 2 we see the experimental results. Model instances are of the form $s_i a_j d_k w_l$ where i is the number of students, j is the number of assignments, k is the number of deliverables in each assignment and l is the number of weeks in a semester. We can observe a general trend that with a higher number of assignments ($j = 6$ and $j = 7$) and two deliverables per assignment ($d = 2$), the alternating simulation reduces a significant amount of the state space leading to considerable speed up. As the number of assignment and deliverables gets lower, there is less and less to save when using the ALT algorithm, ending in a situation where only a few percents of the state space can be reduced at the bottom of the table. This leaves us only with computational overhead for the additional checks related to alternating simulation in the ATL algorithm, however, the overhead is quite acceptable (typically less than 20% of the overall running time).

5.3 Cat and Mouse Use Case

This Petri net game consists of an arena of grid shape with number of mice starting at the first row. The mice try to reach the goal place at the other end of the arena without being caught while doing so by the cats (that can move only on one row of the arena). In Table 2 we consider arena of size 2×3 and 2×4 and the model instances show the initial mice distribution in the top row of the grid. The table contains only instances where the mice do not have a winning strategy and again shows an improvement both in the running time as well as the number of explored states. On the positive instances there was no significant difference between ALT and LS algorithms as the winning strategy for mice were relatively fast discovered by both approaches.

6 Conclusion

We presented an on-the-fly algorithm for solving strictly alternating games and introduced the notion of alternating simulation that allows us to significantly

speed up the strategy synthesis process. We formally proved the soundness of the method and instantiated it to the case of Petri net games where the problems are in general undecidable and hence the algorithm is not guaranteed to terminate. However, when using alternating simulation there are examples where it terminates for any search strategy, even though classical approaches like Liu and Smolka dependency graphs do not terminate at all. Finally, we demonstrated on examples of Petri net games with soft bounds on places (where the synthesis problem is decidable) the practical applicability of our approach. In future work, we can consider an extension of our framework to other types of formalism like concurrent automata or time Petri nets.

Acknowledgments. The work was suppored by the ERC Advanced Grant LASSO and DFF project QASNET.

References

1. Abdulla, P.A., Bouajjani, A., d'Orso, J.: Deciding monotonic games. In: Baaz, M., Makowsky, J.A. (eds.) Computer Science Logic. pp. 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
2. Cassez, F., David, A., Fleury, E., Larsen, K., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Proceedings of CONCUR 2005 – Concurrency Theory. pp. 66–80. IEEE Computer Society Press, United States (2005)
3. Dalsgaard, A.E., Enevoldsen, S., Fogh, P., Jensen, L.S., Jepsen, T.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Olesen, M.C., Pastva, S., Srba, J.: Extended dependency graphs and efficient distributed fixed-point computation. In: van der Aalst, W., Best, E. (eds.) Application and Theory of Petri Nets and Concurrency. pp. 139–158. Springer International Publishing, Cham (2017)
4. Feo-Arenis, S., Westphal, B., Dietsch, D., Muñiz, M., Andisha, S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal verification. Formal Aspects of Computing 28(3), 499–527 (May 2016), https://doi.org/10.1007/s00165-016-0365-3
5. Juhásova, A., Kazlov, I., Juhás, G., Molnár, L.: How to model curricula and learnflows by petri nets - a survey. In: 2016 International Conference on Emerging eLearning Technologies and Applications (ICETA). pp. 147–152 (Nov 2016)
6. L. Minsky, M.: Computation: Finite and infinite machines. The American Mathematical Monthly 75 (04 1968)
7. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) Automata, Languages and Programming. pp. 53–66. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
8. Muñiz, M.: Model checking for time division multiple access systems. Ph.D. thesis, Freiburg University (2015), https://doi.org/10.6094/UNIFR/10161
9. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Universität Hamburg (1962)
10. Raskin, J., Samuelides, M., Begin, L.V.: Games for counting abstractions. Electr. Notes Theor. Comput. Sci. 128(6), 69–85 (2005), https://doi.org/10.1016/j.entcs.2005.04.005

A Proofs

Proof. [**Lemma 4**] To start with, the set *Win* is empty. The *Loop Invariant 1* holds trivially in the start. In any iteration, there are three instances in which a state (s_1, i) can be added to the set *Win* by calling the function *AddToWin*. They are during the execution of lines 2, 25, 34. Lines 2 and 34 call *AddToWin* to add goal configurations, which are trivially winning, to the set *Win*. We shall prove that the call to *AddToWin* at line 25, also keeps the *Loop Invariant 1* true.

In the case of line 25, the configuration (s_1, i) is added to the set *Win* because of the following three cases. We show that in each of those cases the invariant remains true.

Consider any $s, s' \in S$ such that $(s, i) \preceq (s', i)$ and $(s, i) \in Win$. We consider the following cases

- Case $\exists s'_1.\ (s'_1, i) \preceq (s_1, i)$ and $(s'_1, i) \in Win$:
  In this case (s_1, i) is added to *Win* by the end of current iteration. Using the Loop invariant since $(s'_1, i) \in Win$ it has a winning strategy and using Theorem 1, we can conclude that (s_1, i) is also winning . Now since $(s, i) = (s_1, i) \preceq (s', i)$, by Theorem 1, we conclude (s', i) has a winning strategy. Hence Loop Invariant holds true after (s, i) added to Win in this case.
- Case $i = 1$ and $\exists s_3, s'_3.(s'_3, 2) \preceq (s_3, 2)$and $(s_1, 1) \Longrightarrow_1 (s_3, 2)$ and $(s'_3, 2) \in Win$ at line 22:
  In this case $(s_1, 1)$ is added to *Win* by the end of current iteration. Using the Loop invariant, since $(s'_3, 2) \in Win$, $(s'_3, 2)$ has a winning strategy and since $(s'_3, 2) \preceq (s_3, 2)$ by using Theorem 1, $(s_3, 2)$ has a winning strategy. Since $(s_1, 1) \Longrightarrow_1 (s_3, 2)$, *Player-1* can play the game to reach $(s_3, 2)$ and use the winning strategy of $(s_3, 2)$ to win the game. Hence $(s_1, 1)$ has a winning strategy. Now since $(s, i) = (s_1, 1) \preceq (s', i)$, by Theorem 1, we conclude (s', i) has a winning strategy. Hence Loop Invariant holds true after $(s_1, 1)$ added to Win in this case.
- Case $i = 2$ and $\forall s_3.(s_1, 2) \Longrightarrow_2 (s_3, 1)$ implies $\exists s'_3.(s'_3, 1) \preceq (s_3, 1)$ and $(s'_3, 1) \in Win$ at line 23:
  In this case $(s_1, 2)$ is added to *Win* by the end of current iteration. Using the Loop invariant, since $(s'_3, 1) \in Win$, $(s'_3, 1)$ has a winning strategy and since $(s'_3, 1) \preceq (s_3, 1)$, by Theorem 1, $(s_3, 1)$, has a winning strategy. So for any action the *Player-2* choses to play at configuration $(s, i) = (s_1, 2)$, the game reaches a configuration $(s_3, 1)$ from which *Player-1* has a winning strategy. Hence, we can deduce $(s, 2)$ is a winning configuration. Now consider any s' that such that $(s, i) \preceq (s', i)$. By using Theorem 1, we can conclude the configuration (s', i) also has a winning strategy Hence, the *Loop Invariant 1* holds true after $(s_1, 2)$ added to Win in this case.

□

Proof. [**Lemma 5**] To start with the set *Lose* is empty and hence the *Loop invariant 2* holds trivially. A tuple (s_1, i) can enter the set *Lose* during the execution of while loop in the algorithm only by the call to the function *AddToLose*

at the line 18. This call could have been made because of five cases. We shall prove the *Loop invariant 2* holds in all of the following five cases. We consider the following cases:

- Case $\exists s_1'.\ (s_1, i) \preceq (s_1', i)$ and $(s_1', i) \in \mathit{Lose}$ at Line 12: In this case (s_1, i) is added to *Lose* by the end of current iteration. Using the Loop invariant, since $(s_1', i) \in \mathit{Lose}$, (s_1', i) does not have a winning strategy and by using the contrapositive argument of Theorem 1 (which we refer to as a corollary of Theorem 1 from now on) we can conclude (s_1, i) also does not have a winning strategy. Hence, the *Loop Invariant 2* holds after $(s, i) = (s_1, i)$ is added to *Lose* in this case and using corollary of Theorem 1, we can conclude that for any (s_2', i), such that $(s_2', i) \preceq (s_1, i)$, is also not winning.
- Case $i = 2$ and $\exists s_3, s_3'.(s_3, 1) \preceq (s_3', 1)$ and $s_1 \Longrightarrow_2 s_3$ and $(s_3', 1) \in \mathit{Lose}$ at Line 14: In this case $(s_1, 2)$ is added to *Lose* by the end of current iteration. Using the Loop invariant, since $(s_3', 1) \in \mathit{Lose}$, $(s_3', 1)$ does not have a winning strategy and by using Lemma corollary of Theorem 1, we can conclude $(s_3, 1)$ also does not have a winning strategy. Also $(s_1, 2)$ is not a winning configuration as *Player-2* can take the transition $(s_1, 2) \Longrightarrow_2 (s_3, 1)$. By using Lemma corollary of Theorem 1, we can conclude, for any s' such that $(s, 2) = (s_1, 2) \preceq (s', 2)$, $(s', 2)$ is not a winning configuration.
- Case $i = 1$ and $\forall s_3.s_1 \Longrightarrow_1 s_3$ implies $\exists s_3'.(s_3, 2) \preceq (s_3', 2)$and $(s_3', 2) \in \mathit{Lose}$ at Line 13: In this case $(s_1, 1)$ is added to *Lose* by the end of current iteration. Using the Loop invariant, since $(s_3', 2) \in \mathit{Lose}$, $(s_3', 2)$ does not have a winning strategy and by using corollary of Theorem 1, we can conclude $(s_3, 2)$ also does not have a winning strategy. Also $(s_1, 1)$ is not a winning configuration as *Player-1*, no matter which transition he chooses, it does not result in a winning configuration. By using corollary of Theorem 1, we can conclude, for any s' such that $(s', 1) \preceq (s, 1) = (s_1, 1)$, $(s', 1)$ is not a winning configuration.
- Case $i = 2$ and $\exists s_1'.\ (s_1, 1) \preceq (s_1', 1)$ and $(s_1', 1) \in \mathit{Lose}$: In this case $(s_1, 2)$ is added to *Lose* by the end of current iteration. Using the Loop invariant, since $(s_1', 1) \in \mathit{Lose}$, $(s_1', 1)$ does not have a winning strategy. Now we can show, by proof of contradiction, that $(s_1, 2)$ is also not a winning configuration. Let $(s_1, 2)$ be a winning configuration. By Lemma 2, $(s_1, 1)$ is winning and by Theorem 1, $(s_1', 1)$ is a winning configuration. Hence it is a contradiction and $(s_1, 2)$ is not winning. By using corollary of Theorem 1, we can conclude for any s' such that $(s, 2) = (s_1, 2) \preceq (s', 2)$, $(s', 2)$ is not a winning configuration.
- Case $(s_1 \not\longrightarrow_1 \wedge s_1 \not\longrightarrow_2)$ at Line 16: In this case, since $s \notin G$ (If $s \in G$ no tuples of the form (s, i, s'), for any $s' \in S$, are added to W) and it has no transitions out of it. So the configuration (s, i) is not winning. Now consider any $s' \in S$ such that $s' \preceq s$. By using corollary of Theorem 1, we can conclude (s', i) is also losing. Hence, the *Loop Invariant 2* holds after (s, i) is added to *Lose* in this case.

□

Proof. [**Lemma 6**] Let $j = 1$. Before the while loop at the line 4 is entered, *Loop invariant a* holds because for every $(s, 1) \Longrightarrow_1 (s', 2)$, $(s, 1, s') \in W$ and *Loop invariant b* holds because $(s, 2) \notin Disc$ for any $s \in S$. Similarly we can show when $j = 2$, loop invariant 3 holds true before entering the while loop.

Now consider the case when the loop executed for a finite number of iterations. Let loop invariant 3 hold true until the previous iteration. Now we prove each part of Loop Invariant holds after executing current iteration. Let the tuple popped from W in the current iteration be $e = (s_1, i, s_2)$. Consider a configuration $(s, j) \in Disc \setminus (Win \cup Lose)$. If $j = 1$ then Loop Invariant a holds true by the end of previous iteration because for every $(s', 2) \in MaxSucc(s, 1)$ atleast one of the following three conditions **a-I**, **a-II**, **a-III** hold true. Similarly, if $j = 2$ then Loop Invariant b holds true by the end of previous iteration because for every $(s', 1) \in MinSucc(s, 1)$ atleast one of the following three conditions **b-I**, **b-II**, **b-III** hold true.

We shall prove in all the possible executions of the algorithm i.e when an iteration of the loop executes exactly, apart from the the previous lines, line 9 or lines $9, 10$, or lines $9, 18$, or lines $9, 25$, or lines $9, 29$, or lines $9, 31, 32, 34$, or lines $9, 31, 32, 36$, both Loop Invariants a and b continue to hold by the end of current iteration.

Line 9 **is executed**

Let the loop invariants a, b hold by the end of previous iteration. Now in the current iteration, if only the line 9 is executed, this means none of the conditions in the lines 10, 12 to 16, 21 to 23, 27 are true. In particular,

- if $i = 1$ then $(s_2, 2) \in Lose$ (since conditions in lines 27, 22 are not true $(s_2, 2) \in Win \cup Lose$ and $(s_2, 2) \notin Win$ respectively)
- if $i = 2$ then $(s_2, 1) \in Win$ (since conditions in lines 27, 15 are not true $(s_2, 1) \in Win \cup Lose$ and $(s_2, 1) \notin Lose$ respectively)

Now, during the current iteration, if $(s_1, 1, s_2)$ is removed in the line 9 the invariant continues to hold because $(s_2, 2) \in Lose$ i.e **a-III** holds true. Similarly if $(s_1, 2, s_2)$ is removed in the line 9 the invariant continues to hold because $(s_2, 1) \in Win$ i.e **b-III** holds true.

Lines $\{9, 10\}$ **are executed**

Loop Invariant a:

Let Loop invariant a be true by the end of previous iteration and $s = s_1, i = 1$ (the loop invariant holds trivially in the other cases). Although after executing the line 9, (s_1, i, s_2) is removed from W during the current iteration, since the condition in the line 10 is true, $(s, i) = (s_1, i) \in Win \cup Lose$ and hence *Loop invariant a* holds by the end of current iteration.

Loop Invariant b:

By following an argument similar to the case of *Loop Invariant a*, we can show *Loop invariant b* continues to hold true by the end of the current iteration.

Lines $\{9, 18\}$ **are executed**

Loop Invariant a:

$s_1 = s$ and $i = 1$:

In all the three cases **a-III**, **a-III**, **a-III**, since line 18 is executed, by the end of the current iteration $(s_1, 1) \in \mathit{Lose}$ and hence *Loop Invariant a* continues to hold.

$s_1 \neq s$ or $i \neq 1$:

In the case **a-I**, $(s, 1, s') \in W$ by the end of current iteration (because the edge removed from W during the current iteration is not $(s_1, 1, s_2)$ as either $s_1 \neq s$ or $i \neq 1$). Hence *Loop Invariant a* continues to hold. We can argue similarly in the cases **a-II**, **a-III**.

Loop Invariant b:

In the case where **b-II** is true by the end of previous iteration, if $s_1 = s'$ then the since $(s, 2, s') \in D(s', 1)$ (given by the **b-II** case of *Loop Invariant b*), $(s, 2, s')$ is added to W during the execution of function AddToLose() in Line 18. If $s_1 \neq s'$ then it continues to hold that $(s', 1) \in \mathit{Disc} \setminus \mathit{Lose}$ and $(s, 2, s') \in D(s', 1)$. Hence the *Loop Invariant b* holds by the end of current iteration. In all the other cases the *Loop Invariant b* trivially continues to hold by the end of current iteration.

Lines $\{9, 25\}$ **are executed**

Loop Invariant a:

$s_1 = s$ and $i = 1$:

In all the three cases **a-III**, **a-III**, **a-III**, since line 25 is executed, by the end of the current iteration $(s, 1) \in \mathit{Win}$ and hence *Loop Invariant a* continues to hold.

$s_1 \neq s$ and $i = 1$:

In all the three cases **a-I**, **a-II**, **a-III**, *Loop Invariant a* holds trivially.

$s_1 = s$ or $s_1 \neq s$ and $i = 2$:

In the case where **a-II** is true by the end of previous iteration, if $s_1 = s'$ then the since $(s, 1, s') \in D(s', 2)$ (given by the **a-II** case of *Loop Invariant a*), $(s, 1, s')$ is added to W during the execution of function AddToWin() in Line 25. Hence the *Loop Invariant a* holds. In all the other cases the *Loop Invariant a* trivially continues to hold by the end of current iteration.

Loop Invariant b:

$s_1 = s$ and $i = 2$:

Since line 25 is executed, by the end of the current iteration $(s, 2) \in \mathit{Win}$, so Loop Invariant b holds trivially.

$s_1 \neq s$ and $i = 2$

In all the three cases **b-I**, **b-II**, **b-III**, *Loop Invariant a* holds trivially.

$s_1 \neq s$ or $s_1 = s$ and $i = 1$:

In the case where **b-I** is true by the end of previous iteration, $(s, 2, s') \in W$ by the end of current iteration. Hence *Loop Invariant b* continues to hold.

In the two cases **b-II**, **b-III**, *Loop Invariant a* holds trivially.

Lines $\{9, 29\}$ **are executed**

Loop Invariant a:

$s_1 = s$ and $i = 1$:

Consider the case **a-I**. If $s_2 \neq s'$ then *Loop invariant a* continues to hold as $(s, 1, s') \in W$ at the end of current iteration. If $s_2 = s'$ then since the Line 29

is executed, the conditions in the Lines 27, 28 are true which implies $(s', 2) \in Disc \setminus Win$. And after the Line 29 is executed, $(s, 1, s') \in D(s', 2)$. Hence *Loop invariant a* holds by the end of current iteration in this case.

In the cases **a-II**, **a-III**, Loop Invariant *a* holds trivially.

$s_1 \neq s$ or $i \neq 1$:

In the case **a-I**, the *Loop Invariant a* holds as the tuple that is popped of the form $(s_1, 2, s_2)$. In the cases **a-II**, **a-III**, Loop Invariant *a* holds trivially.

Loop Invariant b:

$s_1 = s$ and $i = 2$:

Consider the case **b-I**. If $s_2 \neq s'$ then *Loop invariant b* continues to hold as $(s, 2, s') \in W$ at the end of current iteration. If $s_2 = s'$, then since 29 is executed, the conditions in the Lines 27, 28 are true which implies $(s', 1) \in Disc \setminus Lose$. And after the Line 29 is executed, $(s, 1, s') \in D(s', 2)$. Hence *Loop invariant b* holds by the end of current iteration in this case.

In the case **b-II**, *Loop invariant b* continues to hold trivially (after executing the line 29, $(s', 1) \in Disc \setminus Win$ by the end of the current iteration).

$s_1 \neq s$ or $i \neq 2$:

In the case **b-I**, *Loop invariant b* continues to hold because the tuple $(s, 2, s')$ is not popped in the current iteration. In the case **b-II**, *Loop invariant b* continues to hold trivially.

Lines $\{9, 34\}$ **are executed**

Loop Invariant a:

$s_1 = s$ and $i = 1$:

Consider the case **a-I**. If $s_2 \neq s'$ then *Loop invariant a* continues to hold as $(s, 1, s') \in W$ at the end of current iteration. If $s_2 = s'$ then since the line 34 is executed, (which means 31, 32 are also executed), $(s, 1, s') \in D(s', 2)$. Hence during the call to the function *AddToWin*, $(s, 1, s')$ is added to W. Hence *Loop invariant a* holds by the end of current iteration in this case.

In the case **a-II**, if $s_2 = s'$, then the argument is same as the corresponding case in **a-I**. If $s_2 \neq s'$ then *Loop Invariant a* continues to hold as $(s, 1, s') \in D(s', 2)$ and $(s', 2) \in Disc \setminus Win$ by the end of current iteration.

In the case **a-III**, *Loop Invariant a* holds trivially by the end of the current iteration.

$s_1 \neq s$ or $i \neq 1$:

In the case **a-I**, the *Loop invariant a* continues to hold as $(s, 1, s') \in W$ at the end of current iteration. We can trivially show that the *Loop Invariant a* holds in the rest of the cases.

Loop Invariant b:

$s_1 = s$ and $i = 2$:

In the case **b-I**, if $s_2 \neq s'$ then the *Loop Invariant b* holds as $(s, 2, s') \in W$ by the end of the current iteration. If $s_2 = s'$, then since a call is made to the function AddToWin() in the current iteration, $(s, 2, s')$ is added to W and the *Loop Invariant b* holds.

In the case **b-II**, if $s_2 \neq s'$ then Loop Invariant continues to hold as $((s, 2, s') \in D(s', 1)$ and $(s', 1) \in Disc \setminus Lose$ by the end of current iteration.

$s_1 \neq s$ or $i \neq 2$:

In the case **b-I**, the *Loop Invariant b* continues to hold by the end of current iteration because $(s, 2, s') \in W$ by the end of the current iteration.

In the case **b-II**, the argument is similar to the case when $s_1 = s$ and $i = 2$.

Lines $\{9, 36\}$ **are executed**

Loop Invariant a:

$s_1 = s$ and $i = 1$:

Consider the case **a-I**. If $s_2 \neq s'$ then *Loop invariant a* continues to hold as $(s, 1, s') \in W$ at the end of current iteration. If $s_2 = s'$, then since condition in line 27 is true and also lines 31, 32 are executed in the current iteration, $(s', 1) \in \mathit{Disc} \setminus \mathit{Win}$ and $(s, 1, s') \in D(s', 2)$. Hence *Loop invariant a* continues to hold.

In the cases **a-II**, **a-III**, the *Loop Invariant a* holds trivially as the Line 36 does not update *Win* or *Lose* sets.

$s_1 \neq s$ or $i \neq 1$:

In the case **a-I**, the *Loop invariant a* continues to hold as $(s, 1, s') \in W$ at the end of current iteration. We can trivially show that the *Loop Invariant a* holds in the rest of the cases.

Loop Invariant b:

$s_1 = s$ and $i = 2$:

Consider the case **b-I**. If $s_2 \neq s'$ then *Loop invariant b* continues to hold as $(s, 2, s') \in W$ at the end of current iteration. If $s_2 = s'$, then since condition in line 27 is true and also lines 31, 32 are executed in the current iteration, $(s', 1) \in \mathit{Disc} \setminus \mathit{Lose}$ and $(s, 1, s') \in D(s', 2)$. Hence *Loop invariant b* holds by the end of current iteration in this case.

In the case **b-II**, the *Loop Invariant b* holds true trivially.

$s_1 \neq s$ or $i \neq 2$:

In the case **b-I**, the *Loop Invariant b* continues to hold by the end of current iteration because $(s, 2, s') \in W$ by the end of the current iteration.

In the case **b-I**, the *Loop Invariant b* trivially holds by the end of current iteration.

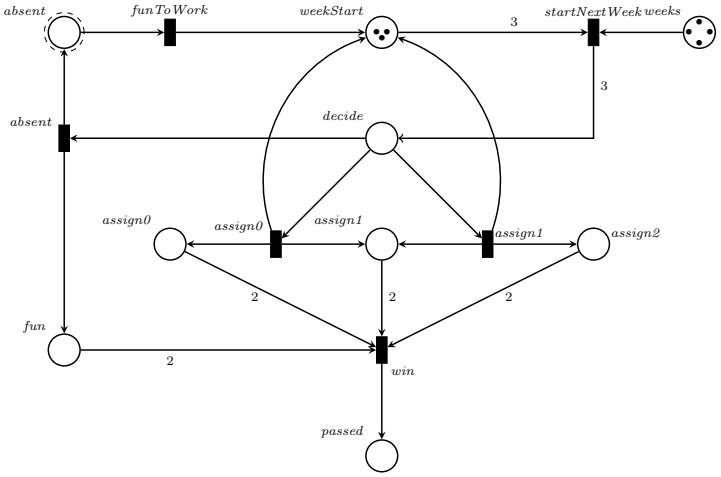
□

B Petri Net Use Cases

B.1 Student Teacher Net

Arena: The net shown in Fig 5 has two components. One for the Student and another for the Teacher. The component for the Student has *weekStart*, *weekEnd*, *weeks* places indicating start of a week,end end of a week and the number of weeks remaining for the completion of semester respectively. The places The transitions with dots belong to the cats.

All of the students complete the week at once before beginning the next week. Hence, the *startNextWeek* takes as many as number of students tokens(3) from *weekEnd* place and keeps them in *weekStart* place by decrementing number of tokens in *weeks* place by 1.



(a) Student

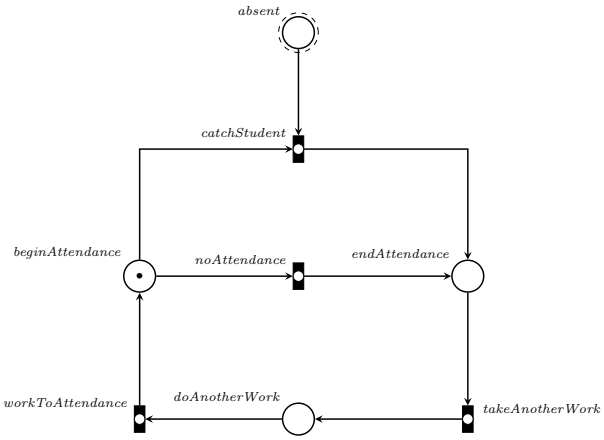


Fig. 5: Teacher

The soft bounds on the places are as follows

- On the week begin, week end , fun place the bound is the number of students
- On the number of weeks place the bound is the number of weeks
- On each assignment place the bound is the denomination of the assignments

Winnning Conditions: The goal of the Student is to place a token the place *Passed*.

B.2 Cat Mice Net

Arena: The net shown in Fig 6 has two components. One for the cat and another for the mouse. The cats are fixed in number and are indicated by the tokens in

the places in the last row ($c_0 \dots c_3$) and the mice to begin with are the places in the first row ($m_0 \dots m_3$) and they can reproduce.

Winnning Conditions: The goal of the mice is to place a token at the place *goal*.

B.3 Fire Alarm System

Arena Figure 7 gives a Petri game model for a simplified fire alarm system. Figure 7a shows a model for sensor 1. At the corresponding TDMA time window, sensor 1 will send a message to the central unit. Sensor 1 can choose to transmit its message among channels one, two, or both. After the message has been sent, it passes the token to the sensor number 2 in Figure 7b which operates similarly. Figure 7d shows the channel blocker which can block messages from the sensors. The place delay ensures that blocking a channel takes at least one turn. The messages are received by the central unit Figure 7c, if all messages are received, the central unit can take a transition leading to the place "goal".

Winning Conditions: The place "goal" has a token.

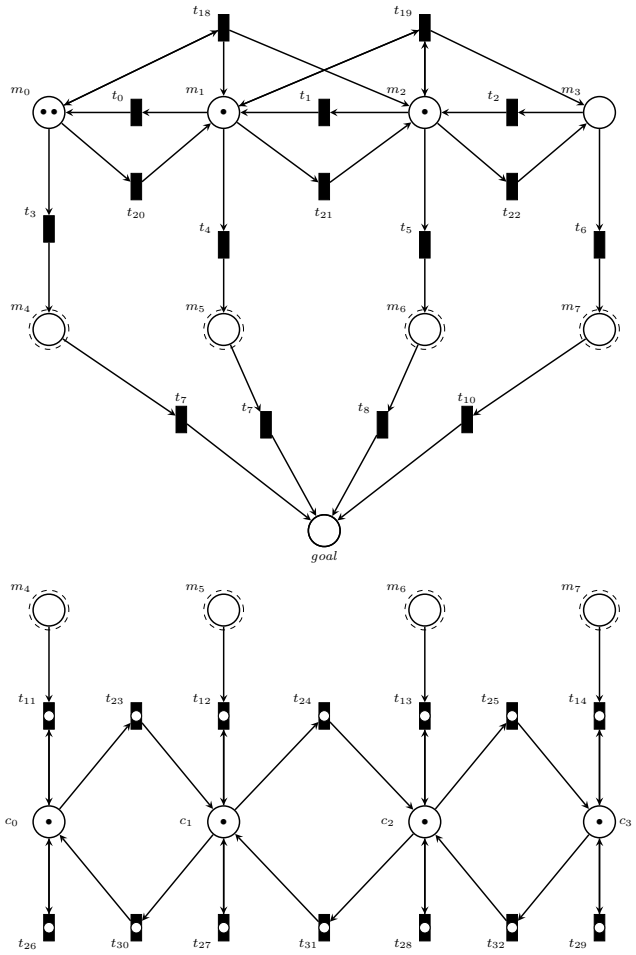
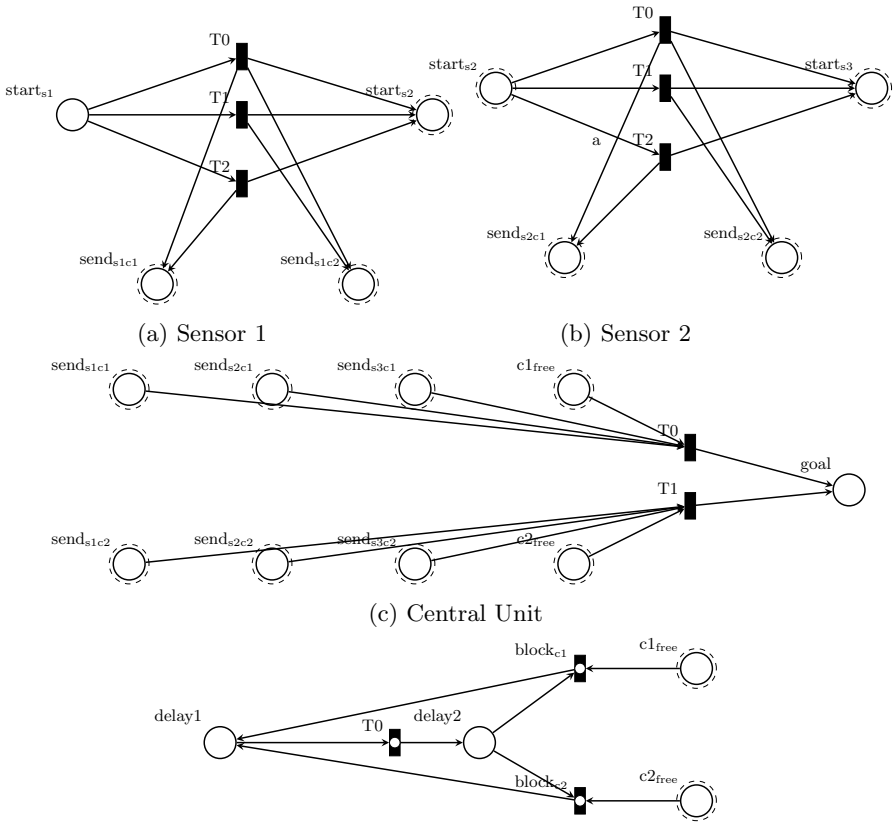


Fig. 6: Instance of Cat and Mice with 2,1,1,0 mice in the first row places and 1,1,1,1 cats in the last row places respectively. The dotted places are shared between cats and mice



(a) Sensor 1

(b) Sensor 2

(c) Central Unit

(d) Channel Blocker

Fig. 7: Fire Alarm System with 2 sensors. Places with dashed circles are shared places, we use shared places for readability. Solid transitions are controllable transitions, transitions with circles inside are uncontrollable transitions