Reachability Switching Games

John Fearnley[1], Martin Gairing[1], Matthias Mnich[2], and Rahul Savani[1]

1 **University of Liverpool, UK**
{john.fearnley, gairing, rahul.savani}@liverpool.ac.uk
2 **Universität Bonn, Germany and Maastricht University, The Netherlands**
mmnich@uni-bonn.de

Abstract

In this paper, we study the problem of deciding the winner of reachability switching games. These games provide deterministic analogues of Markovian systems. We study zero-, one-, and two-player variants of these games. We show that the zero-player case is NL-hard, the one-player case is NP-complete, and that the two-player case is PSPACE-hard and in EXPTIME. In the one- and two-player cases, the problem of determining the winner of a switching game turns out to be much harder than the problem of determining the winner of a Markovian game. We also study the structure of winning strategies in these games, and in particular we show that both players in a two-player reachability switching game require exponential memory.

Keywords and phrases Deterministic Random Walks, Model Checking, Reachability, Simple Stochastic Game, Switching Systems

1 Introduction

Probabilistic model checking is an important topic in the field of formal verification. Like any model checking problem, it asks us to check that a given system satisfies a logical specification. In the probabilistic setting, the system itself makes use of probability, either in the form of an explicit use of randomization, or because the system interacts with a randomized environment. Probabilistic model checking is now a mature topic, with tools like PRISM [15] providing an accessible interface to the research that has taken place.

In this paper, we approach this topic from a different view. Prior work has studied *deterministic* random systems, which attempt to replicate the properties of a random system in a deterministic way. A *switching system* (also known as a Propp machine) does this by replacing the nodes of a Markov chain with *switching nodes*. Each switching node maintains an ordered queue over its outgoing edges. When the system arrives at the node, it is sent along the first edge in this queue, and that edge is then sent to the back of the queue. In this way, the switching node ensures that, after a large number of visits, the system uses each outgoing edge a roughly equal number of times. This mimics a Markovian node with a uniform distribution over its outgoing edges, since such a node also ensures a fairness property over its outgoing edges, in expectation.

There has been much work that studies how well switching systems achieve their goal of simulating a Markov chain, which we will discuss in more detail in the related work section. However, in this paper, we study the question *how hard is it to model check switching systems?* We already have a good knowledge about the complexity of model checking Markovian systems, but how does this change when we instead use switching nodes?

There are good reasons why a designer may want to implement a system using switching nodes. Firstly, true randomness is actually quite expensive, requiring specialist hardware to implement. Most systems actually use pseudorandom generators as their source of randomness, but these generators add complexity to the system. For example, the Mersenne

Twister, used as the standard generator in Python, requires an extra 2.5 kilobytes of internal state. This complicates the program, and hence makes the model checking task much harder. By comparison, a switching system provides a much cheaper implementation, so long as the designer is willing to accept deterministic randomness. Another reason why a system designer may use switching nodes is that they naturally satisfy fairness properties. In fact, they do this better than random systems, which can only provide fairness in expectation.

Our contribution. In this paper, we initiate the study of model checking in switching systems. We focus on *reachability* problems, one of the simplest model checking tasks. This corresponds to determining the winner of a *two-player reachability switching game*. We study zero-, one-, and two-player variants of these games, which correspond to switching versions of Markov chains, Markov decision processes [18], and simple stochastic games [3], respectively.

The main message of the paper is that deciding reachability in one- and two-player switching games is much harder than deciding reachability in Markovian systems. Our results are summarised in the table below.

	Markovian	Switching
0-player	$\texttt{PL}$-complete[1]	$\texttt{NL}$-hard; in $\texttt{PLS}$, in $\texttt{NP} \cap \texttt{coNP}$
1-player	$\texttt{P}$-complete	$\texttt{NP}$-complete
2-player	$\texttt{NP} \cap \texttt{coNP}$ [2]	$\texttt{PSPACE}$-hard; in $\texttt{EXPTIME}$

The $\texttt{PLS}$ and $\texttt{NP} \cap \texttt{coNP}$ upper bounds for the 0-player case were shown before [8, 13], but all other upper and lower bounds for switching games we show for the first time in this paper.

We also investigate the properties of winning strategies in these games. For the one-player case, we show that the reachability player can win using a *marginal* strategy, which simply counts the number of times that each edge has been used. For the two-player case, we show that both players can win using exponential memory, and also that both players *require* exponential memory in order to win.

Related work. Our work was directly inspired by the work of Dohrau, Gärtner, Kohler, Matousek, and Welzl [8]. They studied zero-player reachability switching games and showed that the associated decision problem is in $\texttt{NP} \cap \texttt{coNP}$. More recently, it was shown that the zero-player problem is in $\texttt{PLS}$ [13]. The contribution of our work is to study these questions in the one- and two-player settings.

Switching games are part of a research thread at the intersection of computer science and physics. This thread has studied zero-player switching systems, also known as *deterministic random walks*, *rotor-router walks*, the *Eulerian walkers model* [17] and *Propp machines* [4–7, 11, 12]. Propp machines have been studied in the context of derandomizing algorithms and pseudorandom simulation, and in particular have received a lot of attention in the context of load balancing [1, 9]. However, most work on Propp machines has focused on how well

[1] $\texttt{PL}$, or probabilistic $\texttt{L}$, is the class of languages recognizable by a polynomial time logarithmic space randomized machine with probability $> 1/2$.
[2] It is a long-standing open problem whether we can solve these problems in $\texttt{P}$.

multi-token switching systems simulated Markov chains. The idea of studying *single-token* reachability questions should be credited to the work of Dohrau at al. [8] mentioned above.

Katz et al. [14] and Groote and Ploeger [10] considered *switching graphs*; these are graphs in which certain vertices (switches) have exactly one of their two outgoing edges activated. However, the activation of the alternate edge does not happen when a vertex is traversed by a run; this is the key difference to switching games in this paper. That model was also studied by others [10, 16, 19].

2 Preliminaries

A reachability switching game is defined by a tuple $(V, E, V_{\mathrm{R}}, V_{\mathrm{S}}, V_{\mathrm{Swi}}, \mathrm{Ord}, s, t)$, where (V, E) is a finite graph, and V_{R}, V_{S}, V_{Swi} partition V into *reachability vertices*, *safety vertices*, and *switching vertices*, respectively. The reachability vertices V_{R} are controlled by the *reachability player*, the safety vertices V_{S} are controlled by the *safety player*, and the switching vertices V_{Swi} are not controlled by either player, but instead follow a predefined "switching order". The function Ord defines this *switching order*: for each switching vertex $v \in V_{\mathrm{Swi}}$, we have that $\mathrm{Ord}(v) = \langle u_1, u_2, \ldots, u_k \rangle$ where the sequence is required to be a permutation over the vertices that have an incoming edge from v. The vertices $s, t \in V$ specify *source* and *target* vertices for the game.

A *state* of the game is defined by a tuple (v, C), where v is a vertex in V, and $\mathrm{C} : V_{\mathrm{Swi}} \to \mathbb{N}$ is a function that assigns a number to each switching vertex, which represents how far that vertex has progressed through its switching order. Hence, it is required that $\mathrm{C}(u) \leq |\,\mathrm{Ord}(v)| - 1$, since the counts specify an index to the sequence $\mathrm{Ord}(v)$.

When the game is at a state (v, C) with $v \in V_{\mathrm{R}}$ or $v \in V_{\mathrm{S}}$, then the respective player chooses an outgoing edge at v, and the count function does not change. For states (v, C) with $v \in V_{\mathrm{Swi}}$, the successor state is determined by the count function. More specifically, we define $\mathrm{Upd}(\mathrm{C}, v) : V_{\mathrm{Swi}} \to \mathbb{N}$ so that for each $u \in V_{\mathrm{Swi}}$ we have

$$\mathrm{Upd}(\mathrm{C}, v)(u) = \begin{cases} (\mathrm{C}(u) + 1) \bmod |\,\mathrm{Ord}(u)| & \text{if } v = u, \\ \mathrm{C}(u) & \text{otherwise} \end{cases} .$$

This function updates the count at v by 1, and wraps around to 0 if the number is larger than the number of outgoing edges of v. Then, the successor state of (v, C), denoted as $\mathrm{Succ}(v, \mathrm{C})$ is $(u, \mathrm{Upd}(\mathrm{C}, v))$, where u is the element at position $C(v)$ in $\mathrm{Ord}(v)$.

A *play* of the game is a (potentially infinite) sequence of states $(v_1, C_1), (v_2, C_2), \ldots$ with the following properties:

1. $v_1 = s$ and $C_1(v) = 0$ for all $v \in V_{\mathrm{Swi}}$;
2. If $v_i \in V_{\mathrm{R}}$ or $v_i \in V_{\mathrm{S}}$ then $(v_i, v_{i+1}) \in E$ and $C_i = C_{i+1}$;
3. If $v_i \in V_{\mathrm{Swi}}$ then $(v_{i+1}, C_{i+1}) = \mathrm{Succ}(v_i, C_i)$;
4. If the play is finite, then the final state (v_n, C_n) must either satisfy $v_n = t$, or v_n must have no outgoing edges.

A play is *winning for the reachability player* if the play is finite and the final state is at the target vertex t. A (deterministic, history dependent) *strategy for the reachability player* is a function that maps a play prefix $(v_1, C_1), (v_2, C_2), \ldots, (v_k, C_1)$, to an outgoing edge of v_k. A play $(v_1, C_1), (v_2, C_2), \ldots$ is *consistent* with a strategy if, whenever $v_i \in V_{\mathrm{R}}$, we have that v_{i+1} is the vertex chosen by the strategy. A strategy is *winning for the reachability player* if every play that is consistent with the strategy is winning for the reachability player. Strategies for the safety player are defined analogously.

3 One-player reachability switching games

In this section we consider one-player reachability switching games, i.e., games with $V_S = \emptyset$.

3.1 Containment in NP

We show that deciding whether the reachability player wins a one-player reachability switching game is in NP. The proof uses *controlled switching flows*. These extend the idea of switching flows, which were used in [8] to show containment of the zero-player reachability problem in $\mathtt{NP} \cap \mathtt{coNP}$.

Controlled switching flow. A *flow* is a function $F : E \to \mathbb{N}$ that assigns a natural number to each edge in the game. For each vertex v, we define

$$\mathrm{Bal}(F, v) = \sum_{(v,u)\in E} F(v, u) - \sum_{(w,v)\in E} F(w, v)$$

to be the difference between the outgoing and incoming flow at v.

A flow F is a *controlled switching flow* if it satisfies the following constraints:

- The source vertex s satisfies $\mathrm{Bal}(F, s) = 1$
- The target vertex t satisfies $\mathrm{Bal}(F, t) = -1$
- Every vertex v other than s or t satisfies $\mathrm{Bal}(F, v) = 0$
- Let $v \in V_{\mathrm{Swi}}$ be a switching node and $\mathrm{Ord}(v) = \langle u_1, u_2, \ldots, u_k \rangle$. There exists a constant c and an index $i \leq k$ such that
  - $F(v, u_j) = c + 1$ for all $j < i$.
  - $F(v, u_j) = c$ for all $j \geq i$.

The first three constraints ensure that F is actually a flow from s to t, while the final constraint ensures that the flow respects the switching order at each switching node. Note that there are no constraints on how the flow is split at the nodes in V_R.

Marginal strategies. A *marginal* strategy for the reachability player is defined by a function $M : E \to \mathbb{N}$, which assigns a target number to each outgoing edge of the vertices in V_R. The strategy ensures that each edge e is used no more than $M(e)$ times. That is, when the play arrives at a vertex $v \in V_R$, the strategy checks how many times each outgoing edge of v has been used so far, and selects an outgoing edge e that has been used strictly less than $M(e)$ times. If there is no such edge, then the strategy is undefined.

Observe that a controlled switching flow defines a marginal strategy for the reachability player. We prove that this strategy always reaches the target.

▶ **Lemma 1.** *If a one-player reachability switching game has a controlled switching flow F, then the corresponding marginal strategy is winning for the reachability player.*

Proof. The proof will be by induction on the total amount of flow in F, which is defined as $\sum_{e\in E} F(e)$.

The base case is $\sum_{e\in E} F(e) = 1$. The requirements of a controlled switching flow imply that $F(s, t) = 1$, and all other edges have no flow at all. If $s \in V_R$, then the corresponding marginal strategy is required to choose the edge (s, t), and thus it is a winning strategy. If $s \in V_{\mathrm{Swi}}$, then the balance requirement of a controlled switching flow ensures that t is the first vertex in $\mathrm{Ord}(s)$, so the switching node will move to t, and the reachability player will win the game.

There are two cases to consider for the inductive step. First, assume that $\sum_{e\in E} F(e) = i$, and that $s \in V_R$. Let (s, v) be the outgoing edge chosen by the marginal strategy (this can be any node that satisfies $F(s, v) > 0$). If G denotes the current game, then we can create a new switching game G', which is identical to G, but where v is the designated starting node. Moreover, we can create a controlled switching flow F' for G' by setting $F'(s, v) = F(s, v) - 1$ and leaving all other flow values unchanged. Observe that all properties of a controlled switching flow continue to hold for F'. Since $\sum_{e \in E} F'(e) = i - 1$, the inductive hypothesis implies that the marginal strategy that corresponds to F' (which is consistent with the marginal strategy for F) is winning for the reachability player.

The second case for the inductive step is when $\sum_{e \in E} F(e) = i$ and $s \in V_{\text{Swi}}$. Let (s, v) be the first edge in $\text{Ord}(s)$, which is the edge that the switching node will use. Again we can define a new game G' where the starting node is v, and in which $\text{Ord}(s)$ has been rotated so that v appears at the end of the sequence. We can define a controlled switching flow F' for G' where $F'(s, v) = F(s, v) - 1$ and all other flow values are unchanged. Observe that F' satisfies all conditions of a controlled switching flow, and in particular that rotating $\text{Ord}(s)$ allows s to continue to satisfy the balance constraint on its outgoing edges. Again, since $\sum_{e \in E} F'(e) = i - 1$, the marginal strategy corresponding to F' (which is identical to the marginal strategy for F) is winning for the reachability player. ◀

In the other direction, if the reachability player has a winning strategy for the game, then we can show that there exists a controlled switching flow.

▶ **Lemma 2.** *If the reachability player has a winning strategy for a one-player reachability switching game, then that game has a controlled switching flow of bounded size.*

Proof. Let $v_1, v_2, \dots, v_k$ be the play that is produced when the reachability player uses his winning strategy. We may assume that during the play no state (v, C) is repeated. This is without loss of generality, since if the safety player can force a loop in the state space than she could force to stay in this loop forever and thus the reachability player would not have a winning strategy. Thus, if the play visits a fixed vertex multiple times then for each visit the switch configuration C must be unique. It follows that each vertex is visited at most n^n times. Define the flow F so that $F(e)$ is the number of times e is used by the play. Since each vertex is visited at most n^n times, we have $F(e) \leq n^n$ for all e. We claim that F is a controlled switching flow. In particular, since the play is a path through the graph starting at s and ending at t, we will have $\text{Bal}(F, s) = 1$ and $\text{Bal}(F, t) = -1$, and we will have $\text{Bal}(F, v) = 0$ for every vertex v other than s and t. Moreover, it is not difficult to verify that the balance constraint will be satisfied for every vertex $v \in V_{\text{Swi}}$. ◀

Combing the two previous lemmas yields the following corollary.

▶ **Corollary 3.** *If the reachability player has a winning strategy for a one-player reachability switching game, then he also has a marginal winning strategy.*

Finally, we can show that solving a one-player reachability switching game is in `NP`.

▶ **Theorem 4.** *Deciding the winner of a one-player reachability switching game is in* `NP`.

Proof. By Lemmas 1 and 2, the reachability player can win if and only if the game has a controlled switching flow of bounded size. Moreover, we can guess a flow, and check whether it satisfies the requirements of a controlled switching flow in polynomial time. ◀

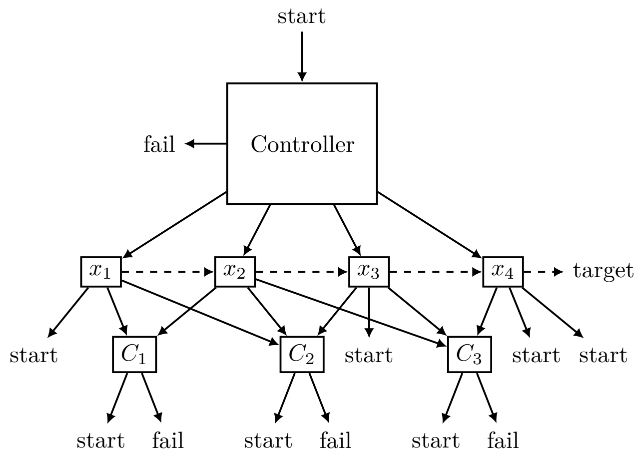

Figure 1 High-level overview of our construction for one player for the example formula $C_1 \wedge C_2 \wedge C_3 = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$. Note that the negations of variables in the formula are not relevant for this high-level view; they will feature in the clause gadgets as explained below. The edges for the variable phase are solid, and the edges for the subsequent verification phase are dashed.

3.2 NP-hardness

In this section we show that deciding the winner of a one-player reachability switching game is NP-hard. We will do so by reducing from 3SAT. Throughout this section, we will refer to a 3SAT instance with variables $x_1, x_2, \ldots, x_n$, and clauses $C_1, C_2, \ldots, C_m$. It is well-known [20, Thm. 2.1] that 3SAT remains NP-hard even if all clauses contain at most three variables, and all variables appear in at most three clauses. We make this assumption during our reduction.

Overview. At a high level, the idea behind the construction is that the reachability player will be asked to assign values to each variable. Each variable x_i will have a corresponding vertex that will be visited three times during the game. Each time this vertex is visited, the reachability player will be asked to assign a value to x_i in a particular clause C_j. If the player chooses an assignment that *does not* satisfy C_j, then the game records this by incrementing a counter. If the counter corresponding to any clause C_j is incremented to three (or two if the clause only has two variables), then the reachability player immediately loses, since the chosen assignment fails to satisfy C_j.

The problem with the idea presented so far is that there is no mechanism to ensure that the reachability player chooses a consistent assignment to the same variable. Since each variable x_i is visited three times, there is nothing to stop the reachability player from choosing contradictory assignments to x_i on each visit. To address this, the game also counts how many times each assignment is chosen for x_i. At the end of the game, if the reachability player has not already lost by failing to satisfy the formula, the game is configured so that the target is only reachable if the reachability player chose a consistent assignment.

A high-level overview of the construction for an example formula is given in Fig. 1.

The control gadget. The sequencing in the construction is determined by the control gadget, which is shown in Fig. 2. In our diagramming notation, square vertices belong to the reachability player. Circle vertices are switching nodes, and the switching order of each switching vertex is labelled on its outgoing edges. Our diagrams also include *counting*

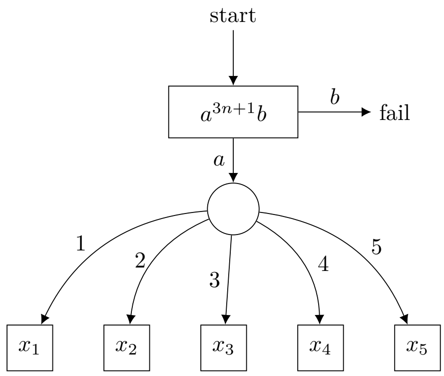

Figure 2 The control gadget.

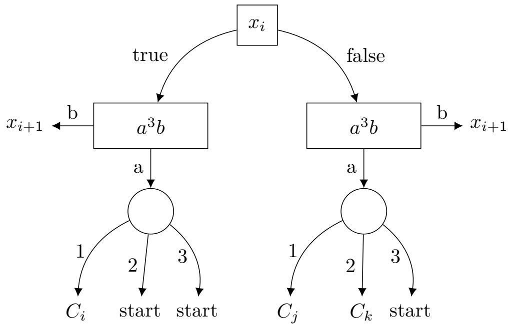

Figure 3 A variable gadget.

gadgets, which are represented as non-square rectangles that have labelled output edges. The counting gadget is labelled by a sequence over these outputs, with the idea being that if the play repeatedly reaches the gadget, then the corresponding output sequence will be produced. In this example the gadget is labelled by $a^{3n+1}b$, which means the first $3n + 1$ times the gadget is used the token will be moved along the a edge, and the $3n + 2$nd time the gadget is used the token will be moved along the b edge. This gadget can be easily implemented by a switching node that has $3n+2$ outgoing edges, the first $3n+1$ of which go to a, while the $3n + 2$nd edge goes to b. We use gadgets in place of this because it simplifies our diagrams.

The control gadget has two phases. In the *variable phase*, each variable gadget, represented by the vertices x_1 through x_n is used exactly 3 times, and thus overall the gadget will be used $3n$ times. This is accomplished by a switching node that ensures that each variable is used 3 times. After each variable gadget has been visited 3 times, the control gadget then sends the token to the x_1 variable gadget for the *verification phase* of the game. In this phase, the reachability player must prove that he gave consistent assignments to all variables. If the control state is visited $3n + 2$ times, then the token will be moved to the *fail* vertex. This vertex has no outgoing edges, and thus is losing for the reachability player.

The variable gadgets. Each variable x_i is represented by a variable gadget, which is shown in Figure 3. This gadget will be visited 3 times in total during the variable phase, and each time the reachability player must choose either the true or false edges at the vertex x_i. In either case, the token will then pass through a counting gadget, and then move to a switching vertex which either moves the token to a clause gadget, or back to the start vertex.

It can be seen that the gadget is divided into two almost identical branches. One corresponds to a true assignment to x_i, and the other to a false assignment to x_i. The clause gadgets are divided between the two branches of the gadget. In particular, a clause appears on a branch if and only if the choice made by the reachability player *fails* to satisfy the clause. So, the clauses in which x_i appears positively appear on the false branch of the gadget, while the clauses in which x_i appears negatively appear on the true branch.

The switching vertices each have exactly three outgoing edges. These edges use an arbitrary order over the clauses assigned to the branch. If there are fewer than 3 clauses on a particular branch, the remaining edges of the switching node go back to the start vertex. Note that this means that a variable can be involved with fewer than three clauses.

The counting gadgets will be used during the verification phase of the game, in which the variable player must prove that he has chosen consistent assignments to each of the variables. Once each variable gadget has been used 3 times, the token will be moved to x_1 by the control gadget. If the reachability player has used the same branch three times, then he can choose that branch, and move to x_2, which again has the same property. So, if the reachability player gives a consistent assignment to all variables, he can eventually move to x_n, and then on to x_{n+1}, which is the target vertex of the game. Since, as we will show, there is no other way of reaching x_{n+1}, this ensures that the reachability player must give consistent assignments to the variables in order to win the game.

The clause gadgets. Each clause C_j is represented by a clause gadget, an example of

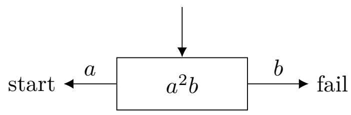

Figure 4 A gadget for a clause with three variables

which is shown in Figure 4. The gadget counts how many variables have failed to satisfy the corresponding clause. If the number of times the gadget is visited is equal to the number of variables involved with the clause, then the game moves to the fail vertex, and the reachability player immediately loses. In all other cases, the token moves back to the start vertex.

Correctness. The following pair of lemmas show that the reachability player wins the one-player reachability switching game if and only if the 3SAT instance is satisfiable.

▶ **Lemma 5.** *If there is a satisfying assignment to the 3SAT formula, then the reachability player can win the one-player reachability switching game.*

Proof. The strategy for the reachability player is as follows: at each variable vertex x_i, choose the branch that corresponds to the value of x_i in the satisfying assignment. We argue that this is a winning strategy. First note that the game cannot be lost in a clause gadget during the variable phase. Since the assignment is satisfying, the play cannot visit a clause gadget more than twice (or more than once if the clause only has two variables), and therefore the edges from the counting gadgets to the fail vertex cannot be used. Hence, the game will eventually reach the verification phase. At this point, since the strategy always chooses the same branch, the play will pass through x_1, x_2, …, x_n, and then arrive x_{n+1}. Since this is the target, the reachability player wins the game. ◀

▶ **Lemma 6.** *If the reachability player wins the one-player reachability switching game, then there is a satisfying assignment of the 3SAT formula.*

Proof. We begin by arguing that, if the reachability player wins the game, then he must have chosen the same branch at every visit to every variable gadget. This holds because x_{n+1} can only be reached by ensuring that each variable has a branch that is visited at least 3 times. The control gadget causes the reachability player to immediately lose the game if it is visited $3n + 2$ times. Thus, the reachability player must win the game after passing through the control gadget exactly $3n + 1$ times. The only way to do this is to ensure that each variable has a branch that is visited exactly 3 times during the variable phase.

Thus, given a winning strategy for the game, we can extract a consistent assignment to the variables in the 3SAT instance. Since the game was won, we know that the game did not end in a clause gadget, and therefore under this assignment every clause has at least one literal that is true. Thus, the assignment satisfies the 3SAT instance. ◀

Hence, we have the following theorem.

▶ **Theorem 7.** *Deciding the winner of a one-player reachability switching game is* `NP`*-hard.*

4 Two-player reachability switching games

4.1 Containment in `EXPTIME`

We first observe that solving a reachability switching game lies in `EXPTIME`. This follows from the fact that the game can be simulated by an *alternating Turing machine*, which is a machine that has both non-deterministic and universal control states. It has been shown that `APSPACE` =`EXPTIME` [2], which means that if we can devise an algorithm that runs in polynomial space on an alternating Turing machine, then we can obtain an algorithm that runs in exponential time on a deterministic Turing machine.

It is straightforward to implement a reachability switching game on an alternating Turing machine. The machine simulates a run of the game. It starts by placing a token on the staring state. It then simulates each step of the game. When the token arrives at a vertex belonging to the reachability player, it uses existential non-determinism to choose a move for that player. When the token arrives at a vertex belonging to the safety player, it uses universal non-determinism to choose a move for that player. The moves at the switching nodes are simulated by remembering the current switch configuration, which can be done in polynomial space. The machine accepts if and only if the game arrives at the target state.

This machine uses polynomial space, because it needs to remember the switch configuration. Thus, we have the following theorem.

▶ **Theorem 8.** *Deciding the winner of a reachability switching game is in* `EXPTIME`*.*

4.2 `PSPACE`-hardness

We show that deciding the winner of a two-player reachability switching games is `PSPACE`-hard, by reducing *true quantified boolean formula* (TQBF), the canonical `PSPACE`-complete problem, to our problem. Throughout this section we will refer to a TQBF instance $\exists x_1 \forall x_2 \dots \exists x_{n-1} \forall x_n \cdot \phi(x_1, x_2, \dots, x_n)$, where ϕ denotes a boolean formula given in negation normal form, which requires that negations are only applied to variables, and not sub-formulas. The problem is to decide whether this formula is true.

Overview. We will implement the TQBF formula as a game between the reachability player and the safety player. This game will have two phases. In the *quantifier phase*, the two players assign values to their variables in the order specified by the quantifiers. In the *formula phase*, the two players determine whether ϕ is satisfied by these assignments by playing the standard model-checking game for propositional logic. The target state of the game is reached if and only if the model checking game determines that the formula is satisfied. This high-level view of our construction is depicted in Fig. 5.

The quantifier phase. Each variable in the TQBF formula will be represented by an *initialization gadget*. The initialization gadget for an existentially quantified variable is shown in Fig. 6. The gadget for a universally quantified variable is almost identical, but the state d_i is instead controlled by the safety player.

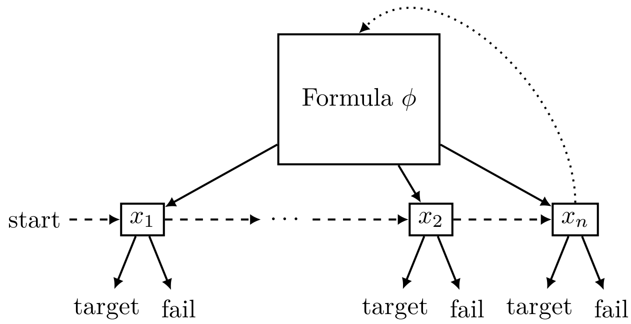

Figure 5 High-level overview of our construction for two players. The dashed lines between variables are part of the first, quantifier phase ; the dotted line from variable x_n to the Formula is the transition between phases, and the solid edges are part of the second, formula phase.

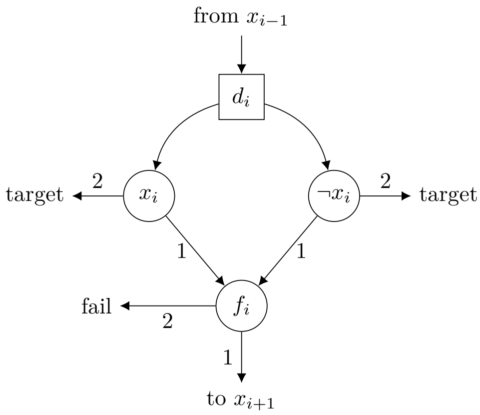

Figure 6 The initialization gadget for an existentially quantified variable x_i.

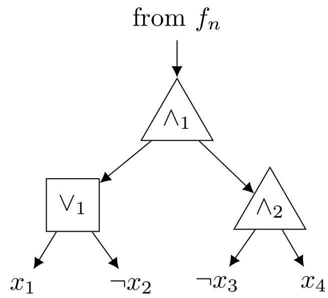

Figure 7 The formula phase game for the formula $(x_1 \vee \neg x_2) \wedge \neg x_3 \wedge x_4$.

During the quantifier phase, the game will start at d_1, and then pass through the gadgets for each of the variables in sequence. In each gadget, the controller of d_i must move to either x_i or $\neg x_i$. In either case, the corresponding switching node moves the token to f_i, which then subsequently moves the token on to the gadget for x_{i+1}.

The important property to note here is that once the player has made a choice, any subsequent visit to x_i or $\neg x_i$ will end the game. Suppose that the controller of d_i chooses to move to x_i. If the token ever arrives at x_i a second time, then the switching node will move to the target vertex and the reachability player will immediately win the game. If the token ever arrives at $\neg x_i$ the token will move to f_i and then on to the fail vertex, and the Safety player will immediately win the game. The same property holds symmetrically if the controller of d_i chooses $\neg x_i$ instead. In this way, the controller of d_i selects an assignment to x_i. Hence, the reachability player assigns values to the existentially quantified variables, and the safety player assigns values to the universally quantified variables.

The formula phase. Once the quantifier phase has ended, the game then moves into the formula phase. In this phase the two players play a game to determine whether ϕ is satisfied by the assignments to the variables. This is the standard model checking game for first order logic. The players play a game on the parse tree of the formula, starting from the root. The reachability player controls the $\vee$ nodes, while the safety player controls the $\wedge$ nodes (recall that the game is in negation normal form, so there are no internal $\neg$ nodes.) Each leaf is either a variable or its negation, which in our game are represented by the x_i and $\neg x_i$ nodes in the initialization gadgets. An example of this game is shown in Figure 7. In our diagramming notation, nodes controlled by the safety player are represented by triangles.

Intuitively, if ϕ is satisfied by the assignment to x_1 through x_n, then no matter what the safety player does, the reachability player should be able to reach a leaf node corresponding to a true assignment, and as we discussed earlier, he will then immediately win the game. Conversely, if ϕ is not satisfied by the assignment, then no matter what the reachability player does, the safety player can reach a leaf corresponding to a false assignment, and then immediately win the game.

▶ **Lemma 9.** *The reachability player wins if and only if the QBF formula is true.*

The proof can be found in Appendix A. Thus, we have shown the following theorem.

▶ **Theorem 10.** *Deciding the winner of a reachability switching game is* `PSPACE`*-hard.*

Note that all runs of the game have polynomial length, a property that is not shared by all reachability switching games. This gives us the following corollary.

▶ **Corollary 11.** *Deciding the winner of a polynomial-length reachability switching game is* `PSPACE`*-complete.*

The proof, which contains the argument for containment in `PSPACE`, is in Appendix A.

4.3 Memory requirements of winning strategies

In this section we will show that both players need exponentially many memory states to win a reachability switching game. We begin by giving a simple gadget that forces the

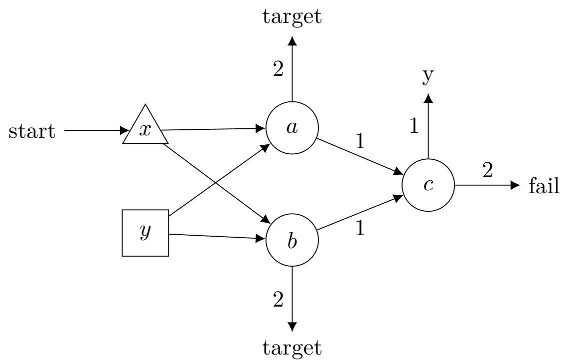

Figure 8 A reachability switching game in which the reachability player needs to use memory.

reachability player to use memory. The gadget is shown in Figure 8. The game starts by allowing the safety player to move the token from x to either a or b. Whatever the choice, the token then moves to c and then on to y. At this point, if the reachability player moves the token to the node chosen by the safety player, then the token will arrive at the target node and the reachability player will win. If the reachability player moves to the node not chosen by the safety player, the token will move to c for a second time, and then on to the fail vertex, which is losing for the reachability player. Thus, every winning strategy of the reachability player must remember the choice made by the safety player.

Observe that we can create a similar gadget that forces the safety player to uses memory, by swapping the two players. In this modified gadget, the safety player would have to chose the vertex *not chosen* by the reachability player. Thus, in a reachability switching game, winning strategies for both players need to use memory.

A memory lower bound. We can now use this gadget to show a lower bound on the amount of memory that is need to win a reachability switching game.

▶ **Lemma 12.** *In a reachability switching game, winning strategies for both players may need to use 2^n memory states, where n is the number of switching nodes.*

Corresponding upper bound. We can also show that exponential memory is sufficient in a two-player reachability switching game. We say that a strategy is a *switch configuration strategy* if it simply remembers the current switch configuration. Any such strategy uses at most exponentially many memory states. For games with binary switch nodes, these strategies use exactly 2^n memory states, where n is the number of switching nodes.

▶ **Lemma 13.** *In a reachability switching game, both players have winning switch configuration strategies.*

The proofs of Lemmas 12 and 13 can be found in Appendix A.

5 Zero-player reachability switching games

In this section we consider zero-player reachability switching games, i.e., games with $V_{\mathrm{R}} = V_{\mathrm{S}} = \emptyset$. As an initial hardness result for this case, we show that deciding the winner of a zero-player game is `NL`-hard. To do this, we reduce from the problem of deciding s-t connectivity in a directed graph.

The idea is to make every node in the graph a switching node. We then begin a walk from s. If, after $|V|$ steps we have not arrived at t, we go back to s and start again. The idea being that, if there is a path from s to t, then the switching nodes must eventually send the token along that path.

More formally, given a graph (V, E), we produce a zero-player reachability switching game played on $V \times V \cup \{\text{fin}\}$, where the second component of each state is considered to be a counter that counts up to $|V|$. Every vertex is a switching node, the start vertex is $(s, 0)$, and the target vertex is fin. Each vertex (v, k) with $v \neq t$ $k < |V|$ has outgoing edges to $(u, k+1)$ for each outgoing edge $(v, u) \in E$. Each vertex $(v, |V|)$ with $v \neq t$ has a single edge to $(s, 0)$. Every vertex (t, k) has a single outgoing edge to fin. Given (V, E), this game can be constructed in logarithmic space by looping over each element in $V \times V$ and producing the correct outgoing edges.

▶ **Theorem 14.** *Deciding the winner of a zero-player reachability switching game is NL-hard under logspace reductions.*

The proof can be found in Appendix B.

6 Further work

Many interesting open problems remain. For the zero-player case, there is an extremely large gap between the upper bounds of `NP` $\cap$ `coNP` and `PLS` and the easy lower bound of `NL` that we showed here. We conjecture that the problem is in fact P-complete, but despite much effort, we were unable to improve upon the upper or lower bounds.

For the one-player case we have shown tight bounds. For the two-player case we have shown a lower bound of PSPACE and an upper bounds of EXPTIME. We conjecture that the lower bound can be strengthened, since we did not make strong use of the memory requirements that we identified in Sect. 4.3.

Finally, here we studied the problem of reachability, which is one of the simplest model checking tasks. What is the complexity of model checking more complex specifications?

References

1 Hoda Akbari and Petra Berenbrink. Parallel rotor walks on finite graphs and applications in discrete load balancing. In *Proc. SPAA 2013*, pages 186–195, 2013.

2 Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. Assoc. Comput. Mach.*, 28(1):114–133, 1981.

3 Anne Condon. The complexity of stochastic games. *Inf. Comput.*, 96(2):203–224, 1992.

4 Joshua Cooper, Benjamin Doerr, Tobias Friedrich, and Joel Spencer. Deterministic random walks on regular trees. *Random Structures Algorithms*, 37(3):353–366, 2010.

5 Joshua Cooper, Benjamin Doerr, Joel Spencer, and Gábor Tardos. Deterministic random walks on the integers. *European J. Combin.*, 28(8):2072–2090, 2007.

6 Joshua N. Cooper and Joel Spencer. Simulating a random walk with constant error. *Combin. Probab. Comput.*, 15(6):815–822, 2006.

7 Benjamin Doerr and Tobias Friedrich. Deterministic random walks on the two-dimensional grid. *Combin. Probab. Comput.*, 18(1-2):123–144, 2009.

8 Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jirí Matousek, and Emo Welzl. ARRIVAL: A zero-player graph game in NP ∩ coNP. Technical report, 2016. https://arxiv.org/abs/1605.03546.

9 Tobias Friedrich, Martin Gairing, and Thomas Sauerwald. Quasirandom load balancing. *SIAM J. Comput.*, 41(4):747–771, 2012.

10 Jan Friso Groote and Bas Ploeger. Switching graphs. *International Journal of Foundations of Computer Science*, 20(05):869–886, 2009.

11 Alexander E Holroyd, Lionel Levine, Karola Mészáros, Yuyal Peres, James Propp, and David B Wilson. Chip-firing and rotor-routing on directed graphs. In *In and Out of Equilibrium 2*, pages 331–364. Springer, 2008.

12 Alexander E. Holroyd and James Propp. Rotor walks and Markov chains. In *Algorithmic probability and combinatorics*, volume 520 of *Contemp. Math.*, pages 105–126. Amer. Math. Soc., Providence, RI, 2010.

13 Karthik C. S. Did the train reach its destination: The complexity of finding a witness. *Inf. Process. Lett.*, 121:17–21, 2017.

14 Bastian Katz, Ignaz Rutter, and Gerhard Woeginger. An algorithmic study of switch graphs. *Acta Inform.*, 49(5):295–312, 2012.

15 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV 2011*, volume 6806 of *Lecture Notes Comput. Sci.*, 2011.

16 Christoph Meinel. Switching graphs and their complexity. In *Proc. MFCS 1989*, volume 379 of *Lecture Notes Comput. Sci.*, pages 350–359, 1989.

17 Vyatcheslav B. Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Phys. Rev. Lett.*, 77(25):5079, 1996.

18 Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

19 Klaus Reinhardt. The simple reachability problem in switch graphs. In *Proc. SOFSEM 2009*, volume 5404 of *Lecture Notes Comput. Sci.*, pages 461–472, 2009.

20 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Appl. Math.*, 8(1):85–89, 1984.

A Proofs for Section 4

Proof of Lemma 9.

Proof. If the QBF formula is true, then during the quantifier phase, no matter what assignments the safety player picks for the universally quantified variables, the reachability player can choose values for the existentially quantified variables in order to make ϕ true. Then, in the formula phase the reachability player has a strategy to ensure that he wins the game, by moving to a node x_i or $\neg x_i$ that was used during the quantifier phase.

Conversely, and symmetrically, if the QBF formula is false then the safety player can ensure that the assignment does not satisfy ϕ during the quantifier phase, and then ensure that the game moves to a node x_i or $\neg x_i$ that was not used during the quantifier phase. This ensures that the safety player wins the game. ◀

Proof of Corollary 11.

Proof. Hardness follows from Theorem 10. For containment, observe that the simulation by an alternating Turing machine described in Section 4.1 runs in polynomial time whenever the game terminates after a polynomial number of steps. Hence, we can use the fact that `AP` =`PSPACE` [2] to obtain a deterministic polynomial space algorithm for solving the problem. ◀

Proof of Lemma 12.

Proof. Consider a game with n copies of the memory gadget shown in Figure 8, but modified so that the following sequence of events occurs.

1. The safety player selects a or b in all gadgets, one at a time.
2. The safety player then moves the game to one of the y vertices in one of the gadgets.
3. The reachability player selects a or b as normal, and then either wins or loses the game.

The reachability player has an obvious winning strategy in this game, which is to remember the choices that the safety player made, and then choose the same vertex in the third step. Since the safety player makes n binary decisions, this strategy uses 2^n memory states.

On the other hand, if the reachability player uses a strategy σ with $k < 2^n$ memory states, then the safety player can win the game in the following way. There are 2^n different switch configurations that the safety player can create at the end of the first step of the game. By the pigeon-hole principle there exists two distinct configurations C_1 and C_2 that are mapped to the same memory state by σ. The safety player selects a gadget i that differs between C_1 and C_2, and determines whether σ selects a or b for gadget i. He then selects the configuration that that is consistent with the other option, so if σ chooses a the safety player chooses the configuration C_i that selects b. He then sets the gadgets according to C_i in step 1, and moves the game to gadget i in step 2. The reachability player will then select the vertex not chosen in step 1, he loses the game.

Finally, observe that we can obtain the same lower bound for the safety player by swapping the roles of both players in this game. ◀

Proof of Lemma 13.

Proof. Let $G = (V, E, V_{\mathrm{R}}, V_{\mathrm{S}}, V_{\mathrm{Swi}}, o, s, t)$ be a reachability switching game, and let $\mathcal{C}$ denote the set of all switch configurations in this game. Consider the "blown-up" reachability game G' played on $V \times \mathcal{C}$, where there are no switching nodes, but instead the successor of a vertex (v, C) with $v \in V_{\mathrm{Swi}}$ is determined by C. It is straightforward to show that the reachability player wins the game G' if and only if he wins the original game. Both players in a reachability game have positional winning strategies. Therefore, if a player can win in G', then he can also win in G using a switch configuration strategy that always plays according to the positional winning strategy in G'. ◀

B Proof for Section 5

Proof. We must argue that there is a path from s to t if and only if the zero-player reachability game eventually arrives at fin. By definition, if the game arrives at fin, then there must be a path from s to t, since the game only uses edges from the original graph.

For the other direction, suppose that there is a path from s to t, but the game never arrives at fin. By construction, if the game does not reach fin, then $(s, 0)$ is visited infinitely often. Since $(s, 0)$ is a switching state, we can then argue that the vertex $(v, 1)$ is visited infinitely often for every successor v of s. Carrying on this argument inductively allows us to conclude that if there is a path of length k from s to v, then the vertex (v, k) is visited infinitely often, which provides our contradiction. ◀