Summary This thesis presents a way to decide the synthesis problem for a given Game Labeled Kripke Structure and a negation-free CTL formula. By restricting the logic to CTL$^-$, we are furthermore able to synthesize all winning strategies. Finally, we show how to extract a winning strategy. We do this in 4 steps: first, we reduce the synthesis problem to computing the minimum fixed-point on a dependency graph, then we convert that dependency graph to a meta-dependency graph, then we construct the most permissive strategy from the meta-dependency graph, and finally, we extract a strategy from the most permissive strategy.

We reduce the synthesis problem for Game Labeled Kripke Structures and negation-free CTL formulae to computing the minimum fixed-point on dependency graphs by giving encoding rules for all possible variations of CTL formulae. After defining the encoding rules, we can construct a dependency graph from a Game Labeled Kripke Structure GS and a CTL formula φ by starting with the pair consisting of the initial state in GS and φ. On this constructed dependency graph, we can compute the minimum fixed-point assignment in linear time.

Calculating the minimum fixed-point of this first dependency graph is not enough to determine if there exists a strategy or not. The first dependency graph in itself does not have the capabilities to determine whether or not it contains a strategy conflict. To accommodate this, we transform the dependency graph into disjunctive normal form by grouping configurations into clauses. Since we group configurations into meta-configurations, these resulting graphs are aptly named Meta-dependency graphs. With some minor fixes to the resulting meta-dependency graph, we can determine whether or not a meta-configuration requires a strategy conflict to propagate one and reject those configurations. Rejecting these meta-configurations, along with some other minor changes, we can determine whether or not a strategy exists for a Game Labeled Kripke Structures with a negation-free CTL formula.

We construct a most Permissive strategy (MPS) by presenting a synthesis algorithm, such that given an MDG as input, constructs the MPS for a particular $Computationtreelogic(CTL)/CTL^-$ formula together with a Game Labelled Kripke Structure (GLKS) GS. This step alone only returns a strategy automaton that contains all winning strategies. We extract a strategy from the MPS. We call this strategy a deterministic instance of the MPS. This is done by using a technique known as alternating reachability.

Finally, we present PetriGAAL, a tool that implements all algorithms in the framework mentioned above. In addition, PetriGAAL is also able to represent all steps in the process graphically.

Most Permissive Strategies for Games with Negation-free CTL

Bogi Napoleon Wennerström, Sigmundur Vang, and Beinir Ragnuson

Department of Computer Science, Aalborg University,
9220 Aalborg Øst, Denmark

Abstract. We investigate the *Synthesis Problem* in a time-branching setting and show that it is PSPACE-hard. We encode a GLKS together with a negation-free CTL formula as a labeled dependency graph, then calculate the minimum fixed-point. This method is successful in deciding problems like equivalence checking and model checking. However, since synthesis in a time-branching setting is not compositional in the structure of the formula, this method alone is not enough. We solve the *Synthesis Problem* by translating the resulting labeled dependency graph into a Meta dependency graph, which is in disjunctive normal form. To find a strategy, we first construct the most permissive strategy, a sound and complete family of strategies (all strategies in the family are winning strategies and all winning strategies are in the family). Next, we show how to extract a strategy from the most permissive instance. Finally, we implement all algorithms in the tool PetriGAAL, along with support for visualizing all graphical structures contained herein.

1 Introduction

In computer science, formal verification has the distinct role of ensuring that a system accommodates a given system specification [3]. Given a system model M and a system specification φ, formally we say that $M \models \varphi$ whenever the model M satisfies the specification φ. We use property logic, such as *LTL*, *CTL*, or *ATL* [3] [2] to describe the systems behavior and modeling languages, such as *Petri Nets* and *Kripke Structures* [19] [20], to represent the systems model. There are several approaches with regards to formal verification, these approaches can be seen as verification problems.

We consider the two formal verification problems, namely *Synthesis Decision Problem* and the *Synthesis problem*. The *Synthesis Decision Problem*, states: given a specification φ and system model M, does there exist a strategy σ such that $M \models \varphi$? Where the *Synthesis problem* extends the *Synthesis Decision Problem*, by not only asking whether there exists a σ such that $M \models \varphi$, but seeks to finding this strategy.

When reasoning about system models, we consider them as either being open or closed. The distinction between these terms is that an open system's behavior is determined by the interaction between the system and its environment, over which the system has no control over, while the state of the system solely decides the behavior of a closed system. When examining open systems, we can define the interaction between the system and its environment as an adversarial two-player game. The system's role is to achieve its objective (satisfying a property), where the environment tries its best to destroy the system's efforts to do so [16].

The notion that synthesis of open systems correlates to the objective of constructing a winning strategy in a game between a system and its environment was first introduced in [16]. Since then, extensive work has been done with regards to synthesis of open systems.

In [10], Kaufmann et al. investigate both the model checking problem and the synthesis problem by presenting a recursive modal logic with semantics over nonnegative multi-weighted extension of Kripke structures, which they present in a game-theoretic setting. They tackled synthesizing strategies in branching time, which is problematic since synthesis in branching time is not compositional in the structure of the formula. They solved this difficulty by introducing a translation that transforms the weighted annotated formulas into disjunctive normal form. They mapped the synthesis problem to the problem of calculating the maximal fixed-point assignment of a dependency graph. Their solution shows that the model checking problem is EXPTIME-complete, and the synthesis problem is 2-EXPTIME and NEXPTIME-hard, in size of the Kripke Structure.

In [12], Kupferman et al. worked with the robust-model-checking problem, which entails that a model M robustly satisfies a property ψ if and only if for every open system M', that acts as an environment of M, satisfies ψ. The system models were modeled as non-deterministic Moore machines, where system properties were specified by the use of the Branching time logics CTL, CTL* [3] and μ-calculus [11]. They show that by using alternating tree automata that the robust model checking problem for CTL and μ-calculus is EXPTIME-complete, and for CTL* is 2EXPTIME-complete. Interestingly, they separated the time branching logic formulas into three categories: existential, universal, and a mixture of both. They demonstrated that these classes have different sensitivity to the robustness requirement. They show that formulas that are not mixed (Universal and Existential) the robust model checking problem is insensitive to non-deterministic environments.

Strategies in game theory and controller synthesis are often *irrevocable*. This entails that a controller can only commit to a single strategy for a given formula, in contract to *revocable* strategies which do not have this restriction (ATL uses *revocable* strategies) [1]. In [1], Ågotnes et al. study alternative variants of ATL where they consider using irrevocable strategies in contrast to the conventional revocable strategies in standard ATL.

Our Contribution. We present a framework, which aims to solve the *Synthesis Decision Problem* and the *Synthesis Problem* in a branching-time setting. We check the satisfiability for a given a negation-free CTL formula and a Game Labelled Kripke Structure by encoding them as labeled dependency graphs where we calculate the minimum-fixed point. This technique is based on the work in [13]. However, in a time-branching synthesis setting, this is only an over-approximation. This approach only gives definitive answers when a strategy does not exist (the formula does not hold). This problem arises from the fact that synthesis in a branching-time setting is not compositional in the structure of the formula [10].

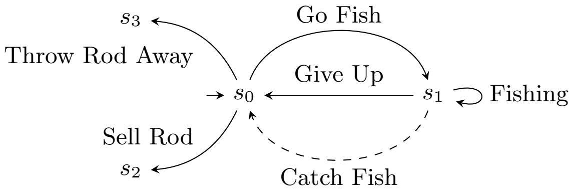


Fig. 1: A GLKS representing a fishing trip

Consider the the example in Figure 1, where we have a system, representing a fishing trip, together with the negation-free CTL formula $\varphi = (AF\, s_2) \wedge (AF\, s_1)$. The reader should note that given a GLKS, controller actions are depicted with solid lines, and environmental actions are depicted with dashed lines. For the formula to hold, we cannot sell the rod and then go fishing, as we do not have any rod to fish with. However if we evaluate the sub-formulas independently ($AF\, s_2$ and $AF\, s_1$), it is trivial to see that both formulas have a strategy, hence the entire formula φ should hold. However, this is not the case because the sequence for which the actions are taken matters. If we were to perform the action "Sell Rod" before taking action "Go Fish," the formula would intuitively never hold, as we have reached a deadlock, state s_2, before satisfying the entire formula. For the formula to hold, we must first take action "Go Fish," satisfying the right sub-formula and subsequently take action "Sell Rod," which satisfies the left sub-formula.

We tackle this problem by converting the labeled dependency graphs into so-called Meta dependency graphs, which are in disjunctive normal form. These Meta dependency graphs can handle the problem as mentioned earlier. However, this solution only solves the *Synthesis Decision Problem*, that is, does there exist a strategy. To solve the *Synthesis Problem* we introduce a restricted version of negation-free CTL, called CTL$^-$, and two synthesis algorithms. The first algorithm transforms a given Meta dependency graph into the most permissive strategy (strategy allowing all winning behavior). The second algorithm transforms the most permissive strategy into an instance of the most permissive strategy.

2 Preliminaries

In this section, we are going to introduce the formalisms and specification language which will serve as a foundation for the rest of the paper. The transition system on which we build upon throughout the article is a mixture between labelled transition systems and Kripke structures, called Labelled Kripke Structure.

Definition 1 (Labelled Kripke Structure). *A Labelled Kripke Structure (LKS) is a 6-tuple $KS = (S, s_0, Act, \rightarrow, AP, L)$ where*

- *S is a set of states,*
- *s_0 is the initial state,*
- *Act is a finite set of actions,*
- *$\rightarrow \subseteq S \times Act \times S$ is a deterministic transition relation, such that if $(s, \alpha, s') \in \rightarrow$ and $(s, \alpha, s'') \in \rightarrow$ then $s' = s''$,*
- *AP is a set of atomic propositions, and*
- *$L : S \rightarrow 2^{AP}$ is a labelling function.*

An LKS extends the Kripke structures found in [20], with the notion of labelled transitions, while reducing the set of initial states to a single initial state. We write path $s \xrightarrow{\alpha} s'$ whenever $(s, \alpha, s') \in \rightarrow$, where $s, s' \in S$ and $\alpha \in Act$. The set of enabled actions in $s \in S$ is given by $en(s) = \{\alpha \in Act \mid \exists s \in S \ . \ s \xrightarrow{\alpha} s'\}$. A state s is called a *terminal state*, if $en(s) = \emptyset$.

A run in an LKS is a finite or infinite sequence of paths, written $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$, such that $s_i \xrightarrow{\alpha_i} s_{i+1}$ for all positions $i \geq 0$. A run can be concatenated with a path such that if $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{m-1}} s_m$ and $s_m \xrightarrow{\alpha_m} s_{m+1}$ then $\pi \ \circ \ s_m \xrightarrow{\alpha_m} s_{m+1} = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{m-1}} s_m \xrightarrow{\alpha_m} s_{m+1}$.

If a run is either infinite or the last state in the run is a terminal state then it is called maximal. We write π_i to denote the i'th state of the run and let $\Pi(s)$ denote the set of all runs starting from a state $s \in S$, such that for all $\pi \in \Pi(s)$ we have that $\pi_0 = s$. The sets $\Pi_{max}(s)$ and $\Pi_{fin}(s)$ are similarly defined as the set of all maximal and finite runs respectively starting from s. Let Π be the set of all runs, regardless of cardinality or starting state.

We define a function $last : \Pi \rightarrow S \cup \{\bot\}$ that takes a run as input and returns the last state of the run if the run is finite and $\bot$ otherwise. Let $len : \Pi_{fin} \rightarrow \mathbb{N}_0$ be a function that given a run returns the total number of states in the run.

Having defined the foundational formalism, we next define the specification language, which will express the properties of an LKS. We will use the well-known branching-time logic CTL without negation.

Definition 2 (CTL). *We define the set of CTL state formulae Ψ over the set of atomic propositions AP as follows:*

$$\psi ::= true \mid \alpha \mid a \mid \neg\psi_1$$

where $\alpha \in Act$ and $a \in AP$. The set of CTL path formulae $\varphi \in \Phi$ consists of the form:

$$\begin{aligned} \varphi &::= \psi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid E\,\rho \mid A\,\rho \\ \rho &::= X\,\varphi_1 \mid \varphi_1\,U\,\varphi_2 \end{aligned}$$

where φ_1 and φ_2 are CTL formulae. We derive the connectives $AF\ \varphi_1$ and $EF\ \varphi_1$ the usual way.

Definition 3 (Satisfaction Relation). *Let $KS = (S, s_0, Act, \rightarrow, AP, L)$ be a Labelled Kripke Structure and let $s \in S$ be a state in KS. The satisfaction relation for a formula $\varphi \in \Phi$ in s is defined as follows:*

$$\begin{array}{ll}
s \models true & \\
s \models \alpha & \textit{iff } \alpha \in en(s) \\
s \models a & \textit{iff } a \in L(a) \\
s \models \neg\psi & \textit{iff } s \not\models \varphi \\
 & \\
s \models \varphi_1 \vee \varphi_2 & \textit{iff } s \models \varphi_1 \textit{ or } s \models \varphi_2 \\
s \models \varphi_1 \wedge \varphi_2 & \textit{iff } s \models \varphi_1 \textit{ and } s \models \varphi_2 \\
s \models E\rho & \textit{iff } \exists \pi \in \Pi_{max}(s) \ . \ \pi \models \rho \\
s \models A\rho & \textit{iff } \forall \pi \in \Pi_{max}(s) \ . \ \pi \models \rho
\end{array}$$

For a run π, the satisfaction relation $\models$ for path formulae is defined by:

$$\begin{array}{ll}
\pi \models X\varphi & \textit{iff } \pi_1 \models \varphi \\
\pi \models \varphi_1 \mathcal{U} \varphi_2 & \textit{iff } \exists i \in \mathbb{N}_0 \ . \ \pi_i \models \varphi_2 \wedge \forall (0 \leq j < i) \ . \ \pi_j \models \varphi_1
\end{array}$$

Given an $KS = (S, s_0, Act, \rightarrow, AP, L)$ and a CTL formula φ, we write $KS \models \varphi$ iff $s_0 \models \varphi$. Later we will make use of a restrictive version of CTL which we call CTL$^-$. In CTL$^-$ we regard all formulas which contain both the conjunctive boolean connective ($\wedge$) and the existential path quantifier (E) as invalid. For example, a formula $(EF\, s_1) \wedge (AG\, \alpha)$ will be considered invalid.

Since LKSs do not have any notion of an environment, it becomes challenging describing open-systems using LKSs. Therefore, in the next section, we will extend LKSs to a gamed setting by introducing a notion of an environment.

3 Game-Theoretic Framework

We extend LKS into a game setting by partitioning the set of actions into controllable and uncontrollable actions, representing a system and its environment respectively.

Definition 4 (Game Labelled Kripke Structure). *A GLKS is a 7-tuple $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$, where $(S, s_0, Act_1 \uplus Act_2, \rightarrow, AP, L)$ is an LKS.*

We call the controller and the environment for players, each with their own set of actions, controllable and uncontrollable, respectively. These sets are denoted Act_1 and Act_2 for controllable and uncontrollable actions, respectively. These sets are defined by partitioning the actions into two disjoint sets $Act = Act_1 \cup Act_2$. In a given state in a GLKS, the enabled function en returns the enabled actions for all players, but since each player has their own set of actions, we require functions that produce the enabled actions for a single player. These functions are en_1 and en_2 for the controller and the environment, respectively.

Determining whether an open system adheres to a specification or not can be interpreted as an adversarial game between the controller and its environment. The environment does everything in its power to prohibit the controller from achieving its goal (satisfying the specification), and we say that the controller has a winning strategy if the controller can enforce a winning behaviour, regardless of what the environment does [16]. We define a strategy as a function that, given the complete history as a run leading up to the current state, the strategy returns the controller's action of choice. Formally we define strategies as follows:

Definition 5 (Strategy). *A controller strategy for a GLKS* $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ *is a function* $\sigma : \Pi_{fin}(s_0) \rightarrow Act_1 \cup \{\bot\}$ *that maps a run to an action or the special symbol* $\bot$ *such that*

- $\sigma(\pi) = \alpha$ *for some* $\alpha \in en_1(last(\pi))$*, or*
- $\sigma(\pi) = \bot$ *if* $en_1(last(\pi)) = \emptyset$

The game between these players is helped along by enforced progression. From Definition 5, if the controller can perform any action, the controller must suggest an action and not the $\bot$ action, forcing the controller to progress. As for the environment, it can wait for the controller to perform an action, if the controller can; otherwise, the environment must also suggest an action to perform. When determining the next action, the game first queries the environment for what action to perform, which the environment may defer to the controller. The key is that the environment cannot choose to do nothing.

Example 1. Consider the GLKS depicted in Figure 2a with $L(s) = \{s\}$ and the formula $\varphi = AF\, s_3$. A strategy for this GLKS and formula, is to perform α_1 in s_0, followed by α_4 in s_1. This is formalized by the strategy σ, where $\sigma(s_0) = \alpha_1$ and $\sigma(s_0 \xrightarrow{\alpha_1} s_1) = \alpha_4$. However this is not a winning strategy because the environment can loop forever in state s_1 by choosing to perform α_2 repeatedly, effectively prohibiting the controller from ever reaching s_3. However, a winning strategy for the GLKS and formula, would be to perform α_6 in s_0, and if the environment instead chooses to perform α_0, then the controller can simply perform α_5 in s_2.

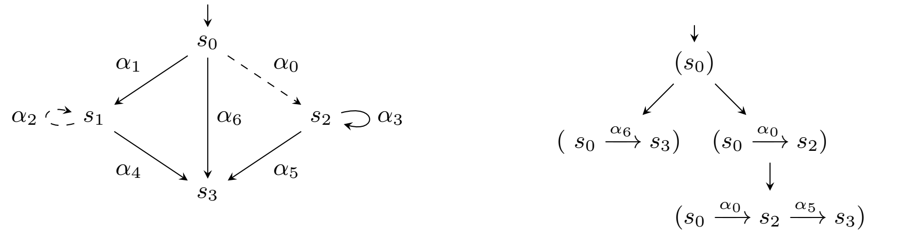
Next, we define the GLKS semantics as an unfolding, where we unfold a GLKS GS into an Labelled Kripke Structure KS, with regards to a controllable strategy σ. Formally we define the unfolding as follows:

Definition 6 (Unfolding GLKS into LKS). *Given a GLKS* $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ *and a strategy* σ *we define an LKS* $unfold(GS, \sigma) = (S', s_0, Act_1 \cup Act_2, \rightarrow', AP, L')$ *where:*

- $S' = \Pi_{fin}(s_0)$,
- $L'(\pi) = L(last(\pi))$*, and*
- $\rightarrow' = \Big\{ \big(\pi, \alpha, (\pi \circ (last(\pi) \xrightarrow{\alpha} s))\big) \;\big|\; \pi \in S' \text{ and } \alpha \in en_2\big(last(\pi)\big) \cup \big\{\sigma(\pi)\big\} \Big\}$.

Note that each state of *unfold*(GS, σ) is now symbolized by a finite run, as such that the labelling function is updated to return the set of labels given by the labelling function of the GS on the last state of the run.

Example 2. Consider the example, in Figure 2b, which illustrates the unfolding of the GLKS on Figure 2a with the winning strategy from Example 1. By observing the resulting LKS in Figure 2b, we see that the LKS tree has two branches, for which both ensure the formula $\varphi = AF\, s_3$ holds. Most interestingly we see that the right branch, demonstrates how the controller can recover when the environment forces the system to state s_2, for which the strategy instructs the controller to take action α_5, bringing the system to state s_3.



s_0
α_1
α_0
α_2
s_1
α_6
s_2
α_3
α_4
α_5
s_3
(s_0)
$(s_0 \xrightarrow{\alpha_6} s_3)$
$(s_0 \xrightarrow{\alpha_0} s_2)$
$(s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_5} s_3)$

(a) A GLKS

(b) Unfolding of the winning strategy

Fig. 2: A GLKS and a winning strategy automata for the formula $\varphi = AF\, s_3$

Given a model and a formula, determining whether or not a winning strategy exists and how such a strategy looks is the basis for the Synthesis Decision Problem and the Synthesis Problem.

Definition 7 (Synthesis Decision Problem). *Given a GLKS* $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ *and a CTL* φ*, the synthesis decision problem asks if there is a strategy* σ *such that unfold*$(GS, \sigma) \models \varphi$.

Definition 8 (Synthesis Problem). *Given a Game Labelled Kripke Structure* $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ *and a CTL* φ*, the synthesis problem is to find a strategy* σ *such that unfold*$(GS, \sigma) \models \varphi$.

3.1 The Complexity of the Synthesis Decision Problem

It is clear that solving the synthesis problem in turn also solves the synthesis decision problem. Therefore, in this section, we will demonstrate the complexity of the synthesis problem only. The method for which we proof the complexity is by reduction to QSAT.

Definition 9 (QSAT). *Let* $X = \{x_1, x_2, x_3, \ldots, x_n\}$ *be a non-empty set of boolean variables. A literal is either a variable, or a negated variable, i.e. either* *variable x, or negated variable $\neg x$. Let $L \subseteq X \cup \{\neg x \mid x \in X\}$ be the set of literals. A clause is a set of three literals. Let $C \subseteq 2^{L^3}$ be a set of clauses. Then, the QSAT problem is deciding the truth value of the equation:*

$$\exists x_1 \ . \ \forall x_2 \ . \ \exists x_3 \ . \ \ldots \forall x_n \ . \ \Big(\bigwedge_{c \in C} \bigvee_{l \in c} l \Big)$$

where n is even and $x_i \in X$.

Theorem 1 (QSAT Complexity [21]). *QSAT is PSPACE-complete.*

Theorem 2. *For a GLKS and a CTL formula the synthesis problem is PSPACE-hard.*

Proof. The proof is by reduction to QSAT. Let $X = \{x_1, x_2, \ldots, x_n\}$ be the variables of a QSAT problem and let C be the set of clauses. Notice that for every variable x_i with an odd index we have $\exists x_i$, and for every variable x_j with an even index, we have $\forall x_j$. We can think of a QSAT formula as a game with two players, one for the existential quantifiers and one for the universal quantifiers. The players then alternate by choosing values for their respective variables. We start with the existential player who assigns a value to the variable x_1 then the universal player assigns a value to x_2 then the existential player responds by assigning a value to x_3 etc. The existential player then has to ensure that for every choice then universal player makes then the existential player can respond such that the clauses will hold in the end. We can say that the existential player has a strategy, denoted f, where given the truth values of all previous choices, it can produce then a truth value for the next variable. To avoid confusion, let solutions refer to QSAT formula strategies, and let strategy refer to a GLKS strategy. With this in mind, we construct the following GLKS:

$$\begin{aligned}
S &= \{\mathit{start}\} \cup \{ \ x_i^t \mid t \in \mathbb{B} \wedge x_i \in X \ \} \\
s_0 &= \mathit{start} \\
\mathit{Act}_1 &= \{x_i \mid i \text{ is odd}\} \\
\mathit{Act}_2 &= \{x_i \mid i \text{ is even}\} \\
\rightarrow &= \{(\mathit{start}, x_1^t, x_1^t) \mid t \in \mathbb{B}\} \cup \\
&\quad\ \{(x_i^t, x_{i+1}^{t'}, x_{i+1}^{t'}) \mid x_i^t, x_{i+1}^{t'} \in S\} \\
AP &= S \\
L(x_i^t) &= \{x_i^t\}
\end{aligned}$$

The resulting GLKS can be represented graphically as shown in Figure 3. Notice that in GLKS for each action there is a state that is equal to the action. This is not depicted in Figure 3 for readability.

Let Φ_X be a function that given a literal $l_i \in c$ returns the query x_i^{true} if $l_i = x_i$ and returns the query x_i^{false} and if $l_i = \neg x_i$. We construct our formula φ as such:

$$\varphi = \bigwedge_{c \in C} AF \Big(\bigvee_{l \in c} \Phi_X(l) \Big)$$

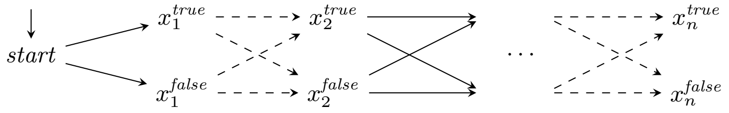


Fig. 3: Resulting GLKS by reduction of QSAT

The construction of the GLKS, ensures that for every variable $x_i \in X$ in every maximal run that the run must contain either x_i^{true} or x_i^{false} but cannot contain both. Essentially, every run is assigning truth values to every variable, i.e. if x_i^{true} is visited then $x_i = true$. Notice that the resulting GLKS has the same semantics as the QSAT game mentioned. Here the controller starts by either choosing x_1^{true} or x_1^{false}, then the environment chooses either x_2^{true} or x_2^{false}, then the controller responds, etc. The controller tries to choose actions such that φ holds.

Mapping between a history for a solution and a run for a strategy is pretty straight forward. Given a run $start \xrightarrow{x_1^{t_1}} x_1^{t_1} \xrightarrow{x_2^{t_2}} x_2^{t_2} \dots$ then we map it to the history $[t_1, t_2, \dots]$ and vice-versa. This makes it easy to map between solutions and strategies. Given a run (history), for a variable x_i if a strategy (solution) outputs x_i^{true} ($true$) then the solution (strategy) should output $true$ (x_i^{true}).

While the QSAT and CTL formula are not the same, they are still equal. The QSAT formula asks that given every history all clauses hold, and φ asks that all clauses hold on every history, which by the distributive property of the universal quantifier is the same question.

We will now proof that the QSAT formula holds iff. there exists a strategy for the resulting GLKS such that $start \models \varphi$.

- $\implies$: For any solution f, let σ be the corresponding strategy for the GLKS created from f. Let us assume that this strategy is not winning. Then there has to be at least one run π where one AF, which corresponds to a clause c, evaluates to false. If that is the case, then there are three states, $\Phi_X(l_i), \Phi_X(l_j), \Phi_X(l_k)$ where $l_i, l_j, l_k \in c$, that are not visited. Since the run π chose the corresponding truth value provided by f, and given the semantics of AF, then π is a counter-example for why f is not a solution, and therefore the original QSAT formula cannot hold, which is a contradiction.
- $\impliedby$: For any winning strategy σ, let f be the corresponding solution for the QSAT created from σ. Let us assume that f is not a solution. Then there has to be at least one set of assignments to each even variable such that there is at least one clause c which evaluates to false, which in turn means that there are three literals $l_i, l_j, l_k \in c$ which also evaluate to false. If that is the case, given the construction of the GLKS, then there would be a run where none of $\Phi_X(l_i)$, $\Phi_X(l_j)$, or $\Phi_X(l_k)$ were visited. If that is the case, then there would be one AF which could not evaluate to true, which means that σ could not be winning, which is a contradiction.

Since the reduction was done in polynomial time, we conclude that the synthesis problem for a GLKS with a CTL formula is PSPACE-hard. □

4 Solving the Synthesis Decision Problem

For model-checking problems, a natural choice of tool is to use dependency graphs as presented in [13]. However, for dependency graphs to be useful for us, some extension is needed.

4.1 Labeled Dependency Graphs

A Dependency graph [13] is a abstract mathematical graphical structure, which shows causal dependencies between its vertices. Traditionally it consists of a finite set of configurations and hyper-edges, however we have extended it by adding labels to the edges.

Definition 10 (Dependency Graph). *A Dependency Graph (DG) is a 3-tuple $D = (V, E, P)$ where V is a set of configurations, $E \subseteq V \times 2^{P \times V}$ is a set of hyper-edges, and P is a set of labels.*

Figure 4a shows a dependency graph in graphical form. Let $D = (V, E, P)$ be a DG, where for each hyper-edge $e = (v, T) \in E$, we have that $v \in V$ is the source of e and $T \subseteq 2^{P \times V}$ the target set of e. We write $v \xrightarrow{p} u$ if there exists a hyper-edge $(v, T) \in E$ such that $(p, u) \in T$. An assignment on D is an assignment function $A : V \to \{0, 1\}$ that assigns the value 0 or 1 to each configuration in the graph, interpreted as false or true respectively. Let $\mathcal{A}^D$ be the set of all assignments on D. For two assignments $A_1, A_2 \in \mathcal{A}^D$ we write $A_1 \sqsubseteq A_2$ if $A_1(v) \leq A_2(v)$ for all $v \in V$. Clearly $(\mathcal{A}^D, \sqsubseteq)$ is a complete lattice.

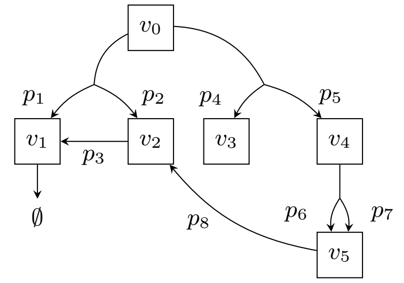
We are interested in finding the minimum fixed-point assignment. For this end we define the function $FP : \mathcal{A}^D \to \mathcal{A}^D$ as follows:

$$FP(A)(v) = \bigwedge_{(v,T) \in E} \quad \bigvee_{u \in T} A(u) \tag{1}$$

where, by convention, conjunction over the empty set is true while disjunction over the empty set is false.

An assignment A is a fixed-point assignment iff $FP(A) = A$. The function FP is monotonic wrt. $\sqsubseteq$ which, by the Knaster-Tarski theorem [22], implies that there exists a unique minimum fixed-point assignment, which we will denote by A_{min}. The assignment A_{min} on D can be computed by repeated application of the function FP on the assignment A_0 where $A_0(v) = 0$ for all $v \in V$, as shown in Figure 4. When a configuration v is assigned 1, i.e. $A(v) = 1$, we say that v propagates 1.

Theorem 3. *[13] There is a linear time (on-the-fly) algorithm to compute the minimal fixed point of a dependency graph.*



	v_0	v_1	v_2	v_3	v_4	v_5
A_0	0	0	0	0	0	0
$FP(A_0)$	0	1	0	0	0	0
$FP(FP(A_0))$	0	1	1	0	0	0
$FP(FP(FP(A_0)))$	1	1	1	0	0	1
$FP(FP(FP(FP(A_0))))$	1	1	1	0	1	1
$FP(FP(FP(FP(FP(A_0)))))$	1	1	1	0	1	1

(a) Dependency Graph $D = (V, E, P)$ (b) Fixed-point calculation of $A_{\min}$

Fig. 4: A Dependency Graph D and the fixed-point assignment $A_{\min}$

4.2 Encoding for GLKSs

As mentioned before, when synthesizing strategies, CTL formulae are not compositional. Therefore, inductively constructing a DG that can determine whether or not a strategy exists for a given GLKS and CTL formula is complicated. Instead, we will do this in two steps. First, we will construct a DG, enabling us to over-approximate whether a strategy exists. Second, we will take this previous DG and build a new DG that can determine whether a strategy exists. In this subsection, we will illustrate how we construct the first DG.

The configurations in the constructed DG are defined as $V : S \times \Phi$, where Φ is the set of all CTL formulae, and labels defined as $P : S \times \big(Act \cup \{\texttt{NIL}\}\big)$. The label is a pair where the first element denotes the state from which we are transitioning. The second element represents the action taken to get from the state in the source configuration to the state in the target configuration. That is, for a target $t = (p, v')$ where the label is $p = (s, \alpha)$ and the target configuration is $v' = (s', \varphi')$, we have that $s \xrightarrow{\alpha} s'$. When the only change between the source and target configuration is a change in the CTL formula, then the second element in the pair is NIL.

The construction of the dependency graph requires the use of two functions *next* and Δ. Given a configuration $v \in V$ the function $next_i : V \to 2^{(S\times Act_i)\times V}$, where $i \in \{1, 2\}$, is defined as $next_i\big((s, \varphi)\big) = \big\{\big((s, \alpha), (s', \varphi)\big) \bigm| \alpha \in en_i(s) \wedge s \xrightarrow{\alpha} s'\big\}$. We define $next\big((s, \varphi)\big) = \bigcup_{i\in\{1,2\}} next_i\big((s, \varphi)\big)$. The intuition is that the *next* function takes in a tuple (s, φ) and returns a set of targets, one for each state that the GLKS can reach from state s with one transition. All target configurations returned from *next* have the formula φ. Given a configuration $v \in V$, the function $\Delta : V \to 2^{2^{S\times Act_i \times V}}$ is defined as:

$$\Delta\big((s, \varphi)\big) = \begin{cases} \big\{next_2\big((s, \varphi)\big)\big\} & \text{if } en_1(s) = \emptyset \\ \big\{next_2\big((s, \varphi)\big) \;\cup\; \{c\} \mid c \in next_1\big((s, \varphi)\big)\big\} & \text{otherwise} \end{cases}$$

When a CTL formula uses the universal temporal operator (A), then we need to be able to respond to every action that the uncontrollable can make.

Therefore all outgoing hyper-edges from the source configuration must include a target set, containing all states generated by performing player 2 actions, which is the idea of the Δ function. It outputs a set of sets where each set combines a single configuration which contains a successor state generated by a player 1 action, with all the configurations which have successors states generated by player 2 actions. In the case where player 1 has no action, then only player 2 actions are included.

The construction of a DG from a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ and a CTL formula φ is inductively defined by a set of compositional encoding rules, one for each rule in the abstract syntax for the CTL path formula, starting with the root configuration (s_0, φ). Whatever the form of the formula, the encoding rules specify which hyper-edges should be included in the set of hyper-edges E.

State formulas When $\varphi = \psi$ we add:

$$\begin{cases} \big\{\big((s,\varphi),\ \emptyset\big)\big\} & \text{if } s \models \psi \\ \emptyset & \text{otherwise} \end{cases} \tag{2}$$

Recall that $\psi ::= \mathit{true} \mid \alpha \mid a \mid \neg\psi_1$. Either we add an edge with no targets, if $s \models \psi$, or we add no edge, if not. In this way, this configuration only propagates one iff. $s \models \psi$.

Disjunction When $\varphi = \varphi_1 \vee \varphi_2$, we add:

$$\left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_1)\big)\right\}\right),\ \left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_2)\big)\right\}\right)\right\} \tag{3}$$

In Equation 3, we model disjunction by adding two hyper-edges: one for checking if φ_1 holds and one for checking if φ_2 holds. Observe that even though the formula changes between the source and target configuration, no GLKS action has been taken. In cases like these, we use `NIL`.

Conjunction When $\varphi = \varphi_1 \wedge \varphi_2$, we add:

$$\left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_1)\big),\ \big((s,\texttt{NIL}),(s,\varphi_2)\big)\right\}\right)\right\} \tag{4}$$

Contrary to Disjunction, we model conjunction in Equation 4 by adding only one hyper-edge but with two targets, one for checking if φ_1 holds and one for checking if φ_2 holds.

Existential Next When $\varphi = EX\varphi_1$, we add:

$$\begin{cases} \emptyset & \text{if } en(s) = \emptyset \\ \big\{\big((s,\varphi),\ \{c\}\big) \mid c \in next\big((s,\varphi_1)\big)\big\} & \text{otherwise} \end{cases} \tag{5}$$

We use the *next* function in Equation 5 to find each of the next targets and create one hyper-edge per target. These targets all have the formula φ_1. This ensures that if only one of the targets propagates one, then the source configuration will as well.

Universal Next When $\varphi = AX\varphi_1$, we add:

$$\big\{(s,\varphi)\big\} \times \Delta\big((s,\varphi_1)\big) \tag{6}$$

We use the Δ function in Equation 6 to create one edge per successor. Each edge has a target set which includes one target produced by a controller action and all of the targets produced by environment actions. These targets all have the formula φ_1. Thus it requires at least one of the configurations generated by the controller and all of the configurations generated by the environment to propagate one.

Existential Until When $\varphi = E\,\varphi_1 \mathcal{U}\, \varphi_2$, we add:

$$\begin{gathered} \left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_2)\big)\right\}\right)\right\} \cup \\ \left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_1)\big),\ c\right\}\right) \,\middle|\, c \in next\big((s,\varphi)\big)\right\} \end{gathered} \tag{7}$$

Since the formula holds if φ_2 holds the rule shown in Equation 6, then we add an edge with the only target being $((s,\texttt{NIL}),(s,\varphi_2))$, which checks whether φ_2 holds on the current state.

Similar to Existential Next, the second clause of Equation 7 enumerates each successor configuration, but here we retain the original formula. We add each of these successor configurations to a set, along with the target $((s,\texttt{NIL}),(s,\varphi_1))$, which checks that φ_1 holds in the current state. Each of these sets then forms an edge which we add to the dependency graph.

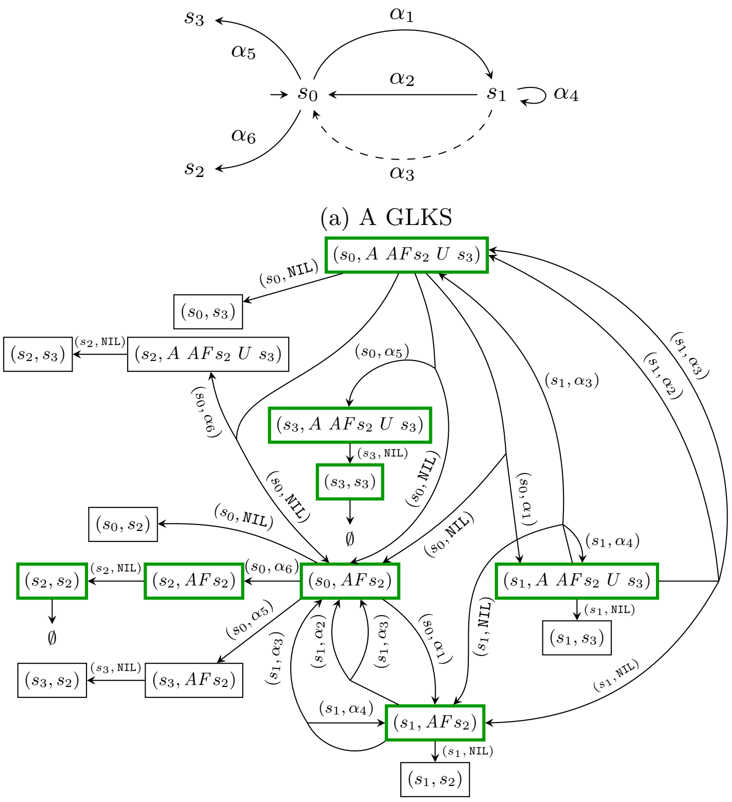
Universal Until When $\varphi = A\,\varphi_1 \mathcal{U}\, \varphi_2$, we add:

$$\begin{gathered} \left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_2)\big)\right\}\right)\right\} \cup \\ \left\{\left((s,\varphi),\ \left\{\big((s,\texttt{NIL}),(s,\varphi_1)\big)\right\} \cup n\right) \,\middle|\, n \in \Delta\big((s,\varphi)\big)\right\} \end{gathered} \tag{8}$$

The first clause of Equation 8 is the same as in Existential Until, which adds an edge that checks whether φ_2 holds on the current state.

Similar to Universal Next, we use the Δ function in Equation 6 to create one edge per successor set, but here we retain the original formula. To each of these successor sets we add the target $((s, \texttt{NIL}), (s, \varphi_1))$, which checks that φ_1 holds in the current state. Finally, each the successor sets are then formed into edges and added to the dependency graph.

Example 3. When encoding a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ and a CTL formula φ into a dependency graph, we start with (s_0, φ) as the root configuration and then add hyper-edges according to the encoding rules. An example is given in Figure 5. Highlighted with a thick green border are the configurations which propagate one. In order to keep the labels on the dependency graph in Figure 5b small, we have replaced the actions in Figure 5a with indexed α symbols.



(a) A GLKS

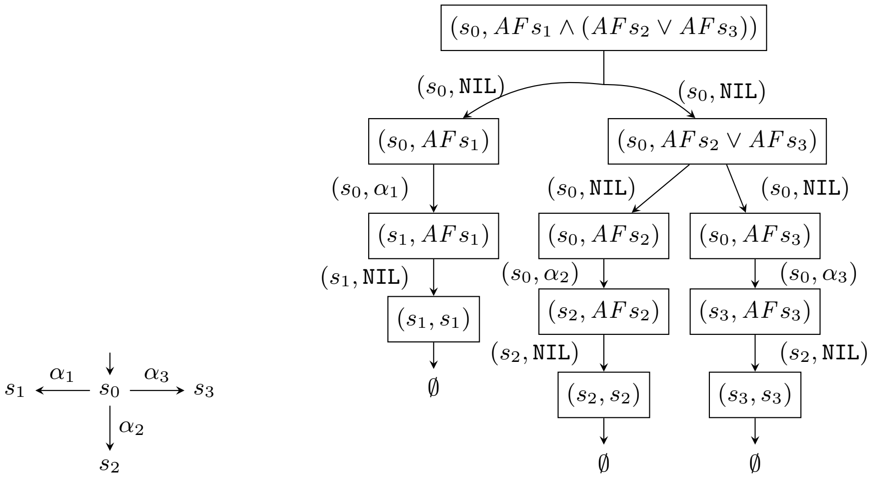
(b) The dependency graph constructed from the GLKS in Figure 5a and the CTL formula $A\ AF s_2\ U\ s_3$

Fig. 5: Encoding example

4.3 Over-approximating the Synthesis Decision Problem

After constructing the dependency graph, we compute the minimum fixed-point assignment. In model checking, the minimum fixed-point assignment assigns 1 to the root of the dependency graph iff the model conforms to the specification. Dalsgaard et al. use this property in their work [6]. If model checking was our goal, we could stop here, but this is not enough to solve the synthesis problem.

However, the approach of reducing the synthesis decision problem into computing minimum fixed-point assignments on dependency graphs is an overapproximation of the synthesis decision problem. Observe that if the dependency graph propagates 0, there has to exist some property that does not hold regardless of strategy. But there are cases where the model conforms to certain specifications, which results in the root of the constructed dependency graph propagating one where there cannot exist a winning strategy. Consider for instance the dependency graph in Figure 6b constructed from the GLKS shown in Figure 6a and the CTL formula $\varphi = AF s_1 \wedge (AF s_2 \vee AF s_3)$.



(a) A GLKS GS

(b) A minimized DG D constructed from Figure 6a

Fig. 6: An example where no winning strategy exists but $A_{\min}((s_0, \varphi)) = 1$

Note that the dependency graph in Figure 6b only shows configurations that propagate 1. The reason why the root configuration propagates 1 is that the left operand $AF s_1$ and the right $(AF s_2 \vee AF s_3)$ are not necessarily evaluated on the same runs. There exists a run $\pi_1 = s_0 \xrightarrow{\alpha_1} s_1$ in which $AF s_1$ holds. There also exists a run $\pi_2 = s_0 \xrightarrow{\alpha_2} s_2$ in which $AF s_2$ holds and thereby $(AF s_2 \vee AF s_3)$ as well. However it is easy to see that $\pi_1 \neq \pi_2$. Said in other words, in order for a controller to make $(AF s_2 \vee AF s_3)$ hold for the GLKS in Figure 6a the controller would need to choose both α_1 and either α_2 or α_3 at the same time, which obviously can not be done.

When the dependency graph requires a strategy to perform different actions simultaneously, we call it a strategy conflict. It is, therefore, interesting to determine whether a dependency graph has any conflicts. A naive method for finding conflicts is to look at hyper-edges with multiple targets to determine if they contain different actions. However, there are two problems with this method. First, not all conflicts happen immediately; they may occur much later in the dependency graph. Second, in some cases, the targets may be labeled with `NIL`, which is not an action. For instance, in Figure 6b the hyper-edge contains two targets, each labeled with `NIL`. The left configuration has an outgoing edge labeled with the action α_1. The right configuration, however, only has outgoing edges labeled with `NIL`. Would this be considered a conflict?

These problems seem only to occur when we have hyper-edges with more than one target or when edges are labeled with `NIL`. We can eliminate hyper-edges with more than one target by transforming the dependency graph into disjunctive normal form. When in this form, it should be easy to remove targets with `NIL`. In that regard, the following section introduces Meta Dependency Graph that do precisely this.

4.4 Meta Dependency Graphs

In this section we demonstrate how to convert dependency graphs into Meta Dependency Graph (MDG). An MDG $MG = (V, E, P)$ is defined just like the dependency graphs are defined in Subsection 4.1, however the configurations are defined as $V = 2^{S \times \Phi}$. To avoid confusion, we will refer to configurations of a meta dependency graph as meta-configurations.

The goal is to structure the dependency graph so that it allows us to identify strategy conflicts and reject those configurations that require them. To do this, we transform the original dependency graph into disjunctive normal form. Each meta-configuration then represents a conjunctive clause of the disjunctive normal form. The meta-configurations allow us to quickly identify when conflicts happen, as they will always occur in the immediate successor edges. Along with some minor corrections, which we will come back to later, removing meta-configurations that require a strategy conflict will make the meta-dependency graph propagate one iff there is a winning strategy.

The conversion consists of two algorithms: the `dgToMdg` algorithm doing the actual DG to MDG conversion and one auxiliary algorithm `Close`.

The `Close` Algorithm The algorithm's name is inspired by Epsilon Closure of Non-deterministic Finite Automata [17], where epsilon is removed by combining some states. The goal of the `Close` algorithm is the same as Epsilon Closure, but instead of ϵ our culprit is `NIL`. The `Close` Algorithm is shown in Algorithm 1. The output `Close` contains sets of triplets $V \times Act \times V$, where the first element represents the source configuration, the second element represents the action taken, and the third element represents the destination configuration.

Algorithm 1: Algorithm for removing `NIL` from a DG

```
Input  : A DG $D = (V, E, P)$ and a set $W$ of configurations
Output: A set of sets of triples $V \times Act \times V$
Function Close($D$, $W$):
    $successors = \emptyset$ ;
    for $v = (s, \varphi) \in W$ do
        $configSuccessors = \emptyset$ ;
        for $(v, T) \in E$ do
            $edgeSuccessors = \{\emptyset\}$ ;
            for $\big((s, \alpha), v'\big) \in T$ do
                $targetSuccessors = \emptyset$ ;
                if $\alpha = NIL$ then $targetSuccessors :=$ Close($D$, $\{v'\}$) ;
                else $targetSuccessors := \Big\{\{(v, \alpha, v')\}\Big\}$ ;
                $edgeSuccessors :=$ Combine($edgeSuccessors$, $targetSuccessors$) ;
            end
            $configSuccessors := configSuccessors \cup edgeSuccessors$ ;
        end
        $successors :=$ Combine($successors$, $configSuccessors$) ;
    end
    return $successors$;
```

```
Input  : Two sets of sets
Output: A set of sets containing all elements of the input sets
Function Combine($S_1$, $S_2$):
    if $S_1 = \emptyset$ then return $S_2$;
    if $S_2 = \emptyset$ then return $S_1$ ;
    return $\{t \cup k \mid t \in S_1 \land k \in S_2\}$ ;
```

Observing the encoding rules in Subsection 4.2, we see that for every rule, except for the Universal Next encoding rule, every edge that has more than one target, it is the case that at least one of these targets contains `NIL`. To remove these, we use the disjunctive normal form. For instance, given the DG D from Figure 6b, and the set $\{v\}$ where $v = (s_0, AF s_1 \land (AF s_2 \lor AF s_3))$ is the root, then the `Close` would return the following:

$$\texttt{Close}(D, \{v\}) = \begin{Bmatrix} \{(v, \alpha_1, (s_1, AF s_1)), (v, \alpha_2, (s_2, AF s_2))\}, \\ \{(v, \alpha_1, (s_1, AF s_1)), (v, \alpha_3, (s_3, AF s_3))\} \end{Bmatrix} \tag{9}$$

Note that since the DG D from Figure 6b ignores configurations that do not propagate one, the above equation does so as well. The full output contains 18 sets. Observing the output, we see that each of the sets have triplets with different actions, the first has α_1 and α_2, the second has α_1 and α_3.

The `Close` Algorithm is a bit intricate, yet not overly complex. It is a trebly nested for loop, each managing different sets of sets. When talking about termination guarantees, then the for-loops are not of concern since all data structures in the algorithm are finite. However, there is a recursive call on line 9, but this is not a problem, as we prove in Lemma 1.

Lemma 1. *Given a dependency graph $D = (V, E, P)$ constructed by the encoding rules in Subsection 4.2 and a set of configurations $W \subseteq V$, then the* `Close` *function terminates.*

Proof. By the dependency graph encoding rules in Subsection 4.2, it is clear that for $v, v' \in V$ where $v = (s, \varphi)$, $v' = (s', \varphi')$ that $\varphi \geq \varphi'$ if there exists $(v, T) \in E$ where $(\alpha, v') \in T$. Following the same rules, observe that if $v = v'$ then we have $\alpha \neq \texttt{NIL}$. Finally, if $\alpha = \texttt{NIL}$ then we can see that $\varphi > \varphi'$. With this in mind, for any configuration $v = (s, \varphi) \in W$, for any edge $(v, T) \in T$, and for any action and target configuration $(\alpha, v') \in T$ where $v' = (s', \varphi')$ we see that the recursion on line 9 can only happen if $\varphi > \varphi'$ due to the condition on line 9. Since the formula is always getting smaller for every time an recursion happens, then the recursion must eventually stop. Since all sets and formulas involved are finite, then the function must terminate at some point. □

By Lemma 1 we know that Algorithm 1 terminates which must mean that there is no infinite path starting from a configuration in a DG, where every target configuration along the path has the action $(s, \texttt{NIL})$ for some s. That is, there is no two $v, v' \in V$ such that $v \xrightarrow{(s,\texttt{NIL})} v' \xrightarrow{(s',\texttt{NIL})}{}^{*} v$. This means that for any $v \in V$ there is a finite number of consecutive `NIL` actions before an action $\alpha \in Act$ must be taken along any path starting with v. We can therefore define the `NIL` distance function $dist : V \to \mathbb{N}_0$ that returns the maximum number of consecutive `NIL` actions along any path starting from v and is inductively defined as $dist(v) = \max(\{dist(v') + 1 \mid v' \in V \text{ and } v \xrightarrow{(s,\texttt{NIL})} v'\})$ where $\max(\emptyset) = 0$. Finally, let $Dist(V) = \max(\{dist(v) \mid v \in V\})$. Using the *dist* function, we can show that `Close` preserves the minimum fixed-point assignment.

Lemma 2. *For a dependency graph $D = (V, E, P)$ constructed by the encoding rules in Subsection 4.2 and a set $W \subseteq V$ we have that $A_{min}(w) = 1$ for all $w \in W$ iff $\exists S \in$ `Close`(D, W) where $A_{min}(v') = 1$ for all $(v, \alpha, v') \in S$.*

Proof. The proof is by mathematical induction on $Dist(W) \leq n$. u

For **$n = 0$.** We are assured that $\alpha \neq \texttt{NIL}$ for every target and thus only line 10 can be reached. This simplifies the cases a bit. Following the execution of the algorithm we see that each $S \in \texttt{Close}(D, W)$ is a union between target triplets from edges, one set for each configuration $v \in W$. Because of this, if $A_{min}(w) = 1$ for all $w \in W$ then there must exist an edge from each w such that all of its targets propagate one, and because of that then there must exist $S \in \texttt{Close}(D, W)$ such that $A_{min}(v') = 1$ for all $(v, \alpha, v') \in S$. Thereafter, if there exist $S \in \texttt{Close}(D, W)$ such that $A_{min}(v') = 1$ for all $(v, \alpha, v') \in S$ then every $w \in W$ must have at least one edge where every target configuration propagates one which in turn means $A_{min}(w) = 1$ for all $w \in W$.

Inductive step. When the if-statement on line 9 is reached and $\alpha = \texttt{NIL}$, then by the induction hypothesis $A_{min}(v') = 1$ iff there exists $S \in \texttt{Close}(D, \{v'\})$ such that $A_{min}(w') = 1$ for all $(w, \alpha, w') \in S$. Therefore, by the time line 13 is executed, since *edgeSuccessors* is added to *configSuccessors* we are sure that $A_{min}(v') = 1$ iff. there exists $S \in$ *configSuccessors* such that $A_{min}(w') = 1$ for all $(w, \alpha, w') \in S$. Finally, since *configSuccessors* is combined with *successors* for every $w \in W$, then $A_{min}(w) = 1$ for all $w \in W$ iff. $S \in \texttt{Close}(D, W)$ such that $A_{min}(v') = 1$ for all $(v, \alpha, v') \in S$.

Thereby proving that for $W \subseteq V$ we have that $A_{min}(w) = 1$ for all $w \in W$ iff. $\exists S \in \texttt{Close}(D, W)$ where $A_{min}(v') = 1$ for all $(v, \alpha, v') \in S$. □

We have now proven termination and the preservation of the minimum fixed-point assignment. The only property left to prove that we are interested in is the removal of NIL in the labels.

Lemma 3. *Given a dependency graph D constructed by the encoding rules in Subsection 4.2 and a set $W \subseteq V$, then $\alpha \neq$ NIL for all $(s, \alpha, v') \in S \in$ Close(D, W).*

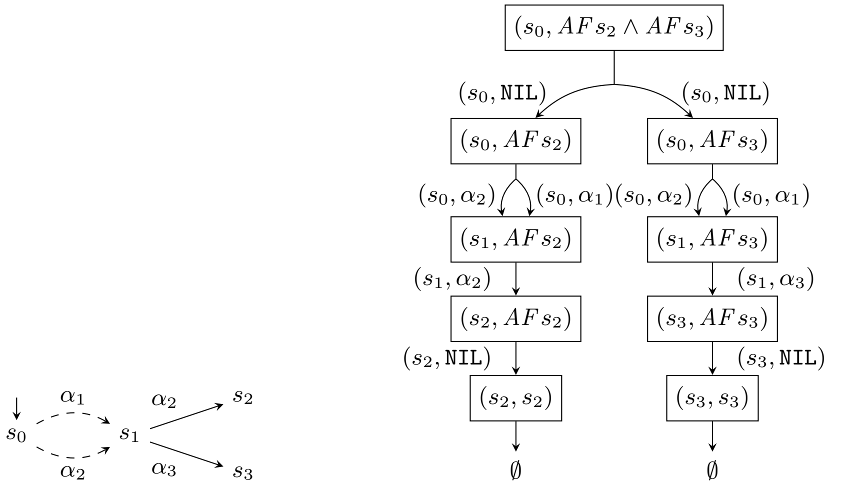
Proof. If for some $((s, \alpha), v') \in T$ we see that $\alpha = \texttt{NIL}$ then the algorithm recurses on line 9. However if $\alpha \neq \texttt{NIL}$ then a triplet (v, α, v') is added on line 10. This does not change however often the algorithm recurses. Because of this no triplet (v, α, v') can be added where $\alpha = \texttt{NIL}$. □

The dgToMdg Algorithm The Close Algorithm allows us to remove NIL edges and quickly identify strategy conflicts. One would think that we should recursively call Close to transform the whole dependency graph to achieve our goal. However, this is unfortunately not quite enough. The issue this time around is the uncontrollable actions. To illustrate the issue, consider the dependency graph in Figure 7b constructed from Figure 7a. Calling Close with this DG and the root produces the following set:

$$\left\{ \left\{ \begin{matrix} ((s_0, AF s_2), \alpha_1, (s_1, AF s_2)), ((s_0, AF s_2), \alpha_2, (s_1, AF s_2)), \\ ((s_0, AF s_3), \alpha_1, (s_1, AF s_3)), ((s_0, AF s_3), \alpha_2, (s_1, AF s_3)) \end{matrix} \right\} \right\} \tag{10}$$

This set has only one element, which has four triplets. However, these triplets do not agree on which action is taken; the first triplet has α_1, the second has α_2, the third triplet has α_1, and finally, the fourth has α_2. This disagreement allows the controller to react to the same action twice, which is not what we want as this enables the controller to reach a conflict. Furthermore, the controller should not respond to configuration changes but rather actions that are taken. So, if we feed this element to Close, the result will be a set of four sets, where each set has two elements:

$$\left\{ \begin{matrix} \{((s_1, AF s_2), \alpha_3, (s_3, AF s_2)), ((s_1, AF s_3), \alpha_3, (s_3, AF s_3))\}, \\ \{((s_1, AF s_2), \alpha_2, (s_2, AF s_2)), ((s_1, AF s_3), \alpha_2, (s_2, AF s_3))\}, \\ \{((s_1, AF s_3), \underline{\alpha_3}, (s_3, AF s_3)), ((s_1, AF s_2), \underline{\alpha_2}, (s_2, AF s_2))\}, \\ \{((s_1, AF s_2), \underline{\alpha_3}, (s_3, AF s_2)), ((s_1, AF s_3), \underline{\alpha_2}, (s_2, AF s_3))\} \end{matrix} \right\} \tag{11}$$



(a) A GLKS with uncontrollable actions

(b) A minimized DG D constructed from Figure 7a

Fig. 7: An example GLKS with a resulting DG

Notice that the last two sets have both α_3 and α_2 which are fired in Equation 11. This makes the meta-dependency graph propagate 1 when there is a conflict. To mitigate this, we group each element of the result of `Close` by the source state, action taken, and controllability of said action.

The `dgToMdg` Algorithm uses the queue *queue* to manage the configuration it has yet to process, while V' functions both as the set of meta-configurations for the MDG and as the set of already processed meta-configurations. The algorithm repeatedly calls `Close` on line 8, groups each of its sets by controllability on line 14 and line 15, and by state and action on line 17 and line 18. Finally, `dgToMdg` adds edges to each of these groups on line 25.

This brings our algorithm to a close, as it now has tackled every issue we can throw at it. First, the following proof will show that the algorithm terminates, subsequently, we show that the algorithm does not include hyper-edges with more than one target. Unfortunately, the final theorem remains to be proven.

Lemma 4. *Given a finite dependency graph D constructed by the encoding rules in Subsection 4.2, the root configuration r of the dependency graph D, and the minimum fix-point assignment A_{min} of D, the algorithm `dgToMdg`, described in Algorithm 2, terminates.*

Proof. From Lemma 1 we know that any call to `Close` terminates. The line where the queue grows is line 22. This line requires *target* to not be an element V'. This element is then added to V' on line 24. Notice that *target* $\subseteq V$. Since V' never decreases in size, and the maximum size it can be is 2^V, then the if-criteria on line 22 will eventually become, and remain, false, when V' cannot grow any

Algorithm 2: Meta Dependency Graph without `NIL`

```
Input  : A DG $D = (V, E, P)$, the root configuration $r$, and the minimal
         assignment $A_{min}$
Output: A new dependency graph $(V', E', P)$ which only propagates one if
         there exists a strategy
Function dgToMdg($D$, $r$, $A_{min}$):
    $queue := \emptyset$ ;
    $V' := \{\{r\}\}$ ;
    $E' = \emptyset$ ;
    enqueue($\{r\}$, $queue$) ;
    while $queue \neq \emptyset$ do
        $W :=$ dequeue($queue$) ;
        for $successor \in$ Close($D$, $W$) do
            if $\exists (v, \alpha, v') \in successor \,.\, A_{min}(v') \neq 1$ then continue;
            if $successor = \emptyset$ then
                $E' := E' \cup \{\, (W, \emptyset) \,\}$ ;
                continue;
            end
            $edges_1 := \big\{(v, \alpha, v') \in successor \mid \alpha \in Act_1\big\}$ ;
            $edges_2 := successor \setminus edges_1$ ;
            if $\big|\big\{(s, \alpha) \mid ((s, \varphi), \alpha, v') \in edges_1\big\}\big| > 1$ then continue;
            $grouped_1 = \Big\{ \, \Big\{ \big((s, \varphi), \alpha, v'\big) \in edges_1 \Big\} \mid s \in S \wedge \alpha \in Act_1 \, \Big\}$ ;
            $grouped_2 = \Big\{ \, \Big\{ \big((s, \varphi), \alpha, v'\big) \in edges_2 \Big\} \mid s \in S \wedge \alpha \in Act_2 \, \Big\}$ ;
            for $edges \in grouped_1 \cup grouped_2$ do
                if $edges = \emptyset$ then continue;
                $target := \{v' \mid (v, \alpha, v') \in edges\}$;
                if $target \notin V'$ then enqueue($target$, $queue$) ;
                $(s, \alpha) \in \big\{(s', \alpha') \mid \big((s', \varphi), \alpha', v'\big) \in edges\big\}$ ;
                $V' := V' \cup \{target\}$ ;
                $E' := E' \cup \big\{ \, \big( \, W, \big\{ \, \big((s, \alpha), target\big) \, \big\} \, \big) \, \big\}$ ;
            end
        end
    end
    return $(V', E')$ ;
```

more. Because of the polling on line 7, the queue must eventually become empty, which in-turn makes the algorithm terminate. □

Lemma 5. *Given a dependency graph D constructed by the encoding rules in Subsection 4.2, the root configuration r of the dependency graph D, and the minimum fix-point assignment A_{min}, the meta dependency graph generated from $D' = \texttt{dgToMdg}(D, r, A_{min})$ has no hyper-edge with more than one target.*

Proof. The only place where E' is updated is on line 25. On this line, the edge added only has a single target. Since this is the only place where E' is updated, it must be the case that all added edges only have a single target. □

Theorem 4. *Given a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$, a CTL formula φ, a dependency graph D constructed from GS and φ by the encoding rules in Subsection 4.2, the root configuration $r = (s_0, \varphi)$ of the dependency graph D, and the minimum fix-point assignment A_{min}, the meta dependency graph generated from $D' = \texttt{dgToMdg}(D, r, A_{min})$ propagates one iff. there exist a strategy σ such that $unfold(GS, \sigma) \models \varphi$.*

From here on out, in order to avoid confusion, we will use the logic CTL^-, as the next sections will focus on the *synthesis problem.*

5 Strategy Automata

Albeit strategies take complete runs in, representing these strategies as simple maps can require an infinite amount of memory. This is because the runs can be arbitrary long. Another way of describing strategies is by using automata, which allows us to express arbitrary runs with a finite amount of memory. A natural choice of an automaton to represent strategies are Mealy machines [14]. We formally define Strategy automates for GLKS as Mealy machines as follows:

Definition 11 (Strategy Automaton for GLKS). *A strategy automaton for a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ is a 5-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ such that:*

- *Q is a finite, non-empty set of states,*
- *$\Sigma = S$ is the input alphabet,*
- *$\Gamma = Act$ is the output alphabet,*
- *$\delta : Q \times \Sigma \rightarrow Q \times \Gamma$ is a transition function, such that $\delta(q, s) = (q', \alpha) \wedge \alpha \in en_1(s)$ if $en_1(s) \neq \emptyset$ and $\delta(q, s) = (q', \bot)$ if $en_1(s) = \emptyset$, and*
- *$q_0 \in Q$ is the initial state.*

To determine the next action to take, at any given state in a strategy automaton M, we introduce an output function out_M, which when given a automaton state $q \in Q$ and a input string $w \in \Sigma^+$, returns an action $\alpha \in Act_1$. We say that a strategy produced by the strategy automaton M is called σ_M.

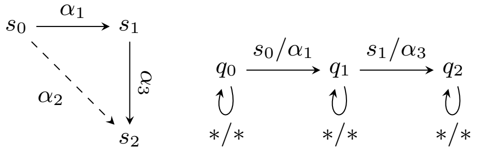
Definition 12 (Strategy Automaton Output). *The output of a strategy automaton $M = (Q, \Sigma, \Gamma, \delta, q_0)$ is defined by a function out_M mapping a state and an input string to an output $out_M(q, w) \in \Gamma$. The function out_M is defined inductively on the length of the input w as follows:*

$$\begin{aligned} out_M(q, a) &= \alpha && \textit{where } (q', \alpha) = \delta(q, a) \\ out_M(q, a \cdot w) &= out_M(q', w) && \textit{where } (q', \alpha) = \delta(q, a) \end{aligned}$$

where $a \in \Sigma$ and $w \in \Sigma^*$.

In Figure 8b we graphically illustrate a strategy automaton M for the GLKS GS shown in Figure 8a. The transition $q_0 \xrightarrow{s_0/\alpha_1} q_1$ denotes that when M is in automaton state q_0 and gets the state s_0 from GS as input, then M should propose action α_1 . Given that the transition function is total, then transitions for all automaton state and GLKS state pairs are define. This means that, e.g., when q_0 reads s_1 as input, then M must propose α_3. In order to show correct strategy automata, these transitions must technically be included when graphically showing the strategy automaton. However a valid run in GS would never start with the state s_1 and therefore it would never be the case that the transition $q_0 \xrightarrow{s_1/\alpha_3} q_1$ would be taken. Including these transitions when showing the strategy automaton graphically (transitions such as $q_0 \xrightarrow{s_1/\alpha_3} q_0$ and $q_0 \xrightarrow{s_2/\perp} q_0$) would distract the reader from seeing the transitions that are of importance. In Figure 8b, we denote these transitions by a $\xrightarrow{*/*}$ transition and in the sequel, we omit these loops and assume their existence.

Figure 8c shows the input/output relation specified in M from q_0. The intuition is the same for all other states in the strategy machine. Note that when we have input string $x = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_3} s_3$, we should perform $\perp$ (the do nothing action), due to s_2 being a terminal state.



(a) GLKS GS (b) Strategy Automaton M

Run	Output
x	$out_M(q_0, x)$
s_0	α_1
$s_0 \xrightarrow{\alpha_1} s_1$	α_3
$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_3} s_2$	$\perp$

(c) Strategy output for GS Figure 8a

Fig. 8: A GLKS GS with a Strategy Automaton M and output function

Some strategies require infinite memory, while others are finite. The amount of memory that a strategy may use is called its bound. For strategy automata, memory is given in states. We say that a strategy that uses k states is k-bounded. Formally k-bounded strategies are defined as follows:

Definition 13 (k-bounded Strategy). *Given a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$, a strategy σ is k-bounded if there exists a strategy automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \gamma)$ where $|Q| = k$ such that $\sigma(\pi) = out_M(q_0, \pi)$ for every $\pi \in \Pi_{fin}(s_0)$.*

5.1 Memory Hierarchy

In the following lemmas, we prove the differing expressivity of k-bounded strategies for varying values of k. We let the notation EX^k denote that the temporal operator EX is nested k times. These proofs serve to show that strategies are of different expressive power depending on how much memory they have.

Lemma 6. *Every k-bounded strategy is also $(k+1)$-bounded.*

Proof. It is easy to see that any $(k+1)$-bounded strategy automaton can simulate a k-bounded strategy automaton, by having the k-bounded strategy automaton being exactly encoded in the $(k+1)$-bounded strategy automaton and then subsequently let one state be disconnected from the rest of the $(k+1)$-bounded strategy automaton. This state will neither hinder nor help the $(k+1)$-bounded strategy automaton. □

Lemma 7. *For every k there exists an GLKS and CTL formula that has a k-bounded winning strategy but no $(k-1)$-bounded winning strategy.*

Proof. Consider the GLKS GS in Figure 9 and assume that $L(s_0) = \{s_0\}$ and $L(s_1) = \{s_1\}$. For any $k \geq 0$, we can construct a winning strategy automaton M_k for a formula $EX^k s_1$ on the GS as follows: let there be k states in M_k ordered from q_k to q_1, with q_k as the initial state. Let $q_n \xrightarrow{s_0/\alpha_0} q_{n-1}$ be the transition between q_n and q_{n-1} for all $k \leq n \leq 2$ and let q_1 contain the self loop $q_1 \xrightarrow{s_0/\alpha_1} q_1$.

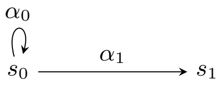


Fig. 9: A GLKS GS

The construction is shown in Figure 10. Clearly M_k is a winning strategy automaton for the formula $EX^k s_1$ on GS any $k \geq 1$. Proving that M_k is the smallest winning strategy automaton will be proof by contradiction. Assume that there exists a $(k-1)$-bounded strategy automaton M_{k-1} for $EX^k s_1$. In order for M_{k-1} to be a winning strategy automaton for the formula $EX^k s_1$ in GS the strategy needs to first output the α_0 action $k-1$ times and then output the α_1 action. Notice that this is also the only winning strategy. In order for M_{k-1}

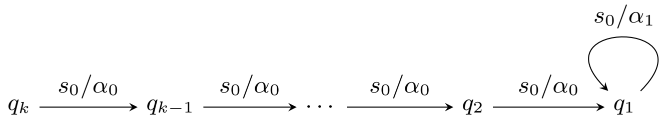


Fig. 10: The Strategy Automaton M_k

to output α_0 and still have the possibility of later outputting α_1 the strategy automaton needs to do so with the transition $\xrightarrow{s_0/\alpha_0}$, while transitioning to a state which has not been visited before. This transition must happen $k-1$ times, each time going to a state previously not visited. However with only $k-2$ such transitions possible in M_{k-1}, then we can conclude by the pigeonhole principle that one of two things must happen. Either there exist a already visited state in M_{k-1} that is reached by the $\xrightarrow{s_0/\alpha_0}$ transition, which will make M_{k-1} output the α_0 action infinitely often, or M_{k-1} outputs the α_0 action fewer than $k-1$ times and then outputs the α_1 action and thereby moving GS to s_1 too soon. Neither of these situations is winning for the formula $EX^k s_1$ in GS, which implies that M_{k-1} cannot exist. Therefore M_k is the smallest winning strategy automaton for the formula $EX^k s_1$ on GS. □

5.2 Most Permissive Strategy

In this section, we introduce the notion of nondeterministic strategy automaton. As mentioned earlier we can use strategy automata to represent strategies, and we will show how we can use nondeterministic strategy automaton to represent multiple strategy automata in a single automaton. We shall now formally define nondeterministic strategy automaton, for which will be the basis for Most Permissive Strategies (MPS). They are formally defined as follows:

Definition 14 (NSA). *A nondeterministic strategy automaton for a GLKS* $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ *is a 6-tuple* $N = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *where* Q, Σ, Γ, q_0 *are defined as in Definition 11 and:*

- $\delta : Q \times \Sigma \rightarrow 2^{Q \times \Gamma}$ *is a transition function and* $o = \{\alpha \mid (q', \alpha) = \delta(q, s)\}$, *where* $o \subseteq en_1(s)$ *if* $en_1(s) \neq \emptyset$ *and* $o \subseteq \{\bot\}$ *if* $en_1(s) = \emptyset$,
- $F \subseteq Q$ *is a set of final states.*

There are two differences between strategy machines and NSAs. The first is the nondeterminism. When given a state and input the nondeterminism allows an NSA to nondeterministically choose between a set of outputs and states. The second is the acceptance criteria. Alike Nondeterministic Strategy Automata [17], an NSA $N = (Q, \Sigma, \Gamma, \delta, q_0, F)$ has a set F of accepting state, which allows N to accept or reject strings dependending on whether or not N visits a state in F or not.

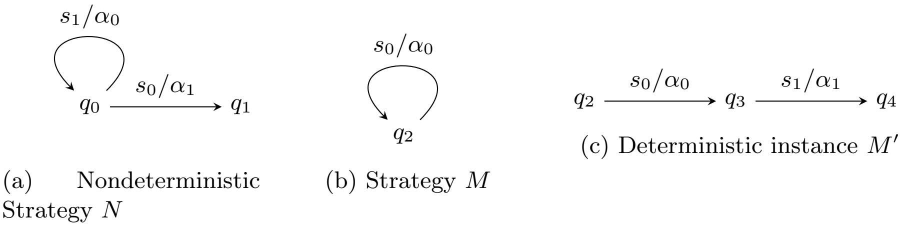
To determine whether or not a strategy machine is part of an NSA, we first need to formally define a relation between both machines. If a machine is said to part of an NSA, we say that that machine is an instance of the NSA. The following definition introduces the notion of simulating a non-deterministic machine N with a deterministic machine M, ensuring compatibility between M and N. We define the simulation relation as follows:

Definition 15 (Simulation Relation). *A binary relation $\mathcal{R} \subseteq Q_M \times Q_N$, where Q_M and Q_N are the set of states for a deterministic and nondeterministic strategy automaton respectively, is a simulation iff $q_1 \mathcal{R} q_2$ and if $q_1 \xrightarrow{s/\alpha} q_1'$ then there exists a transition $q_2 \xrightarrow{s/\alpha} q_2'$ such that $q_1' \mathcal{R} q_2'$.*

We say that q_2 simulates q_1, written $q_1 \leq q_2$, if there is a simulation relation that relates them. Given a deterministic and a nondeterministic strategy automaton, $M = (Q_M, \Sigma, \Gamma, \delta', q_0)$ and $N = (Q_N, \Sigma, \Gamma, \delta, q_0', F)$ respectively, then N simulates M, written $M \leq N$, if $q_0 \leq q_0'$.

Given a strategy automaton, we define a trace w as $w = q_0 q_1 q_2 \ldots$, such that for all positions $i \leq 0$, there exists a transition $\xrightarrow{s/\alpha}$ such that $q_i \xrightarrow{s/\alpha} q_{i+1}$. We let w_i denote the i'th state of the trace.

The simulation relation defined in Definition 15 ensures that a deterministic strategy machine M is compatible with an nondeterministic strategy machine N. However this is not enough to ensure that a machine M is an instance of N, because we have no notion of a acceptance criteria for the simulation. Consider the example in Figure 11 showing a nondeterministic strategy N in Figure 11a and a deterministic strategy in Figure 11b. Note that in q_1 is a final state of N depicted. We can quickly see that N can simulate M, however strategy M is not an instance of N, because strategy M cannot force N to go the the final state q_1.



(a) Nondeterministic Strategy N

(b) Strategy M

(c) Deterministic instance M'

Fig. 11: An NSA, a simulation, and an instance

To define acceptance simulation criteria, we must ensure that given a deterministic strategy machine M and deterministic strategy machine that $M \leq N$. After that, we must ensure that for all behaviors in M, at least one simulation forces N to a final state. Finally, we formally define the acceptance as a deterministic instance:

Definition 16 (Deterministic Instance). *A deterministic strategy automaton, $M = (Q_M, \Sigma, \Gamma, \delta', q_0')$, is a deterministic instance of a nondeterministic* *strategy automaton, $N = (Q_N, \Sigma, \Gamma, \delta, q_0, F)$, written $M \leq^F N$, if $M \leq N$ and for every infinite trace w in M, there exists a trace w' in N such that $w_i \leq w'_i$ for all i and $w'_j \in F$ for some j.*

Recall the NSA N from Figure 11a, an instance M' of this NSA is shown in Figure 11c. This alternate M' has that, $M' < N$ and contrary to the automaton M in Figure 11b, M' forces N to go to the final state q_1. Because of this have that M' adheres to both Definition 15 and Definition 16, and therefore such that $M' \leq^F N$.

We say that a NSA N is winning for a formula φ on a GLKS GS if every deterministic instance M of N is winning. That is, given an N, we say that it is winning iff $M \leq^F N \implies \mathit{unfold}(GS, \sigma_M) \models \varphi$.

The use of most permissive strategies is beneficial when looking for a particular strategy [4] because we can check whether that strategy is an instance (contained) of the corresponding MPS; hence the MPS acts as a verification for the given strategy instance. we formally define a Most Permissive Strategy Automaton, and is defined as followed:

Definition 17 (Most Permissive Strategy Automaton). *An NSA N is the most permissive strategy automaton for a formula φ and a GLKS GS, if every deterministic instance M of N is a winning strategy for φ on GS and if every M that is a winning strategy for φ on GS is a deterministic instance of N.*

Now that we have formally defined most permissive strategies and deterministic instances, we will in the next section introduce an synthesis algorithm, that is able to construct a MPS.

5.3 Algorithm for finding the most permissive strategy

In this section we introduce the algorithm for finding most permissive strategies. This algorithm is shown in Algorithm 3. We start by defining the notion for update, which will be used in Algorithm 3 when creating transitions and states. We define the update δ as follows:

Definition 18 (Update δ). *Given an NSA $N = (Q, \Sigma, \Gamma, \delta, q_0, F)$ we define $N[q' \xrightarrow{s'/\alpha} q''] = (Q \cup q'', \Sigma, \Gamma, \delta', q_0, F)$ where $q' \in Q$, $s \in \Sigma$, and $\alpha \in \Gamma$ such that:*

$$\delta'(q, s) = \begin{cases} \{(\alpha, q'')\} \cup \delta(q', s') & \textit{if } q = q' \land s = s' \\ \delta(q, s) & \textit{otherwise} \end{cases}$$

What is essentially in happening in Algorithm 3 is that initially we start with a strategy that has no behaviour $N^{\perp}$ in line 2, and from here we gradually build the MPS. The key points are in line 10 where we identify final states an update N subsequently in line 11, and in 16 where we create a new state W' and update the corresponding transitioning from W to W'. The algorithm returns the most permissive strategy N in line 21, when the queue is empty in line 5.

Algorithm 3: MPS Synthesis

```
Input  : An MDG $D = (V, E, P)$, the root meta-configuration $r$, and the
         minimum fix-point assignment $A_{min}$ of $D$
Output: A Most Permissive Non-deterministic Strategy Automaton
Function Synth($D$, $r$, $A_{min}$):
    $N := N^{\perp} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ;
    $visited := queue := \emptyset$ ;
    Push($r$, $queue$) ;
    while $queue \neq \emptyset$ do
        $W :=$ Poll($queue$) ;
        Push($W$, $visited$) ;
        for $(W, T) \in E$ do
            if $T = \emptyset$ then
                $N := N[W \xrightarrow{*/*} W]$ ;
                $N := (Q, \Sigma, \Gamma, \delta, q_0, F \cup \{W\})$ ;
                continue;
            end
            for $(s, \alpha, W') \in T$ do
                if $A_{min}(W') \neq 1$ then continue;
                $N := N[W \xrightarrow{s/\alpha} W']$ ;
                if $W' \notin visited$ then Push($W'$, $queue$) ;
            end
        end
    end
    return $N$ ;
```

Theorem 5. *Given as input a dependency graph $D = (V, E, P)$, constructed from a GLKS $GS = (S, s_0, Act_1, Act_2, \rightarrow, AP, L)$ and a formula φ using the rules in Subsection 4.2, then Algorithm 3 outputs a Most Permissive Strategy for GS and φ.*

In order to find an instance of an NSA $N = (Q, \Sigma, \Gamma, \delta, q_0, F)$ we use a modified version of the alternating reachability as defined in [5]. The way it works is by defining a *safe* set of states. Initially, this is $safe = F$. We then select any $q \in Q$ such that $q \xrightarrow{s/\alpha} q'$ where $q \notin F$ but $q' \in F$. If for every outgoing edge from $q \xrightarrow{\alpha'} q''$ that $q'' \in safe$ and $\alpha' \in Act_2$ then we add q' to *safe*. If there are other edges from q with controllable actions, we remove them. When there are no more edges to add *safe* we can be sure that the NSA is deterministic.

Finally, this allows us to find an MPS for the running example from Figure 1 with the CTL^- formula $\varphi = (AF\ s_2) \wedge (AF\ s_1)$, and then to find an instance of this MPS, allowing us to solve the problem. This MPS is given in Figure 12 and finally the instance is given in Figure 13.

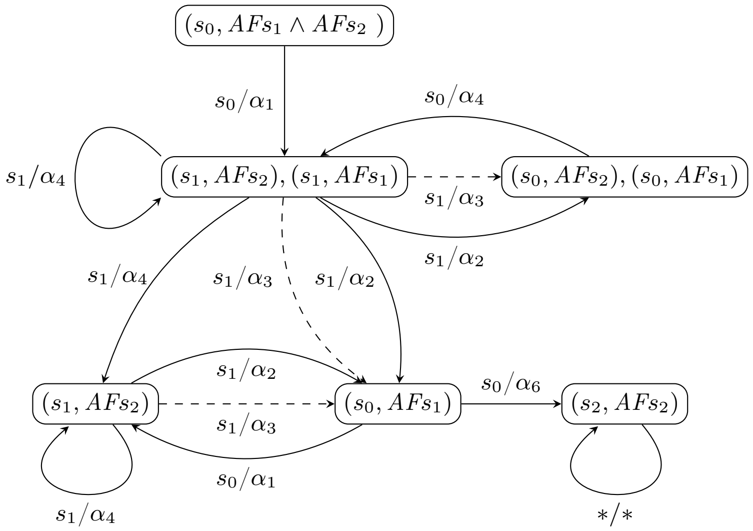


Fig. 12: Most Permissive Strategy corresponding to the running example 1

6 Software Implementation

This section presents PetriGAAL (**Petri G**ame **Aal**borg) [23], our open-source, cross-platform tool Game Labelled Kripke Structure strategy synthesis. It is written in Java 16 and is built to be extensible.

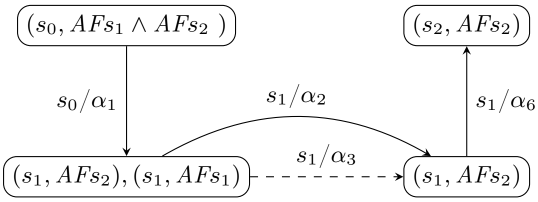


Fig. 13: Strategy Instance corresponding to the running example 1

It uses Petri nets [15] as the model for the GLKS. Petri nets are a graphical and mathematical modeling tool that has been a topic of interest in the scientific community for over 50 years. For that reason there exists multiple software tools for the modeling and automatic verification of Petri nets. One of those software tools is the TAPAAL [7] tool suite, where we make use of its modeling capabilities. The reason why we chose TAPAAL is that it is readily available, the authors of this paper are familiar with it, and it has support for Petri games, i.e., Petri nets with transitions partitioned into player 1 and 2.

PetriGAAL is capable of taking a GLKS model as input by parsing *tapn* files, which is the fileformat used by TAPAAL, and encodes it as a GLKS. This is safe to do since every Petri game can be encoded as a GLKS, as long as the Petri game is bounded. PetriGAAL also includes a *pnml* parser, however in order to include a player 2, an extra attribute is required on each transition in the *pnml* file to denote where the transition belongs.

After taking a GLKS model and a CTL$^-$ formula as input, PetriGAAL will output the most permissive strategy (if it exists), along with a deterministic instance of that most permissive strategy and all the intermediatary stages, i.e., the dependency graph and meta dependency graph. The procedure of how PetriGAAL computes the output is shown in Figure 14.

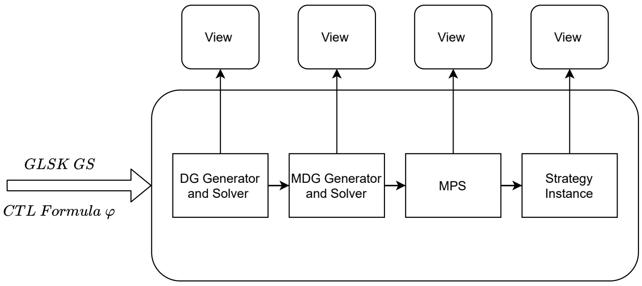


Fig. 14: Flow Chart Petrigaal

Visualizing dependency graphs and meta dependency graphs can be helpful to determine why no strategy exists. PetriGAAL visualizes both dependency graphs and meta dependency graphs along with most permissive strategies and deterministic instances. We use Cytoscape.js [8] and DOT [9] for rendering and node-layout, respectively.

When the tool is booted up, the first thing that the user sees is shown in Figure 15. At the top are 4 different tabs, one for each stage in the procedure. It is possible to select a model on the left hand side by pressing the *Browse* button on the upper left hand corner and then a file browser appears with which the user can use to find the model. PetriGAAL assumes that the CTL/CTL^- formula is contained in a `formula` file that resides in the same directory as the GLKS model. Once the model and the query has been loaded in, then the user can press the *Synthesize* button.

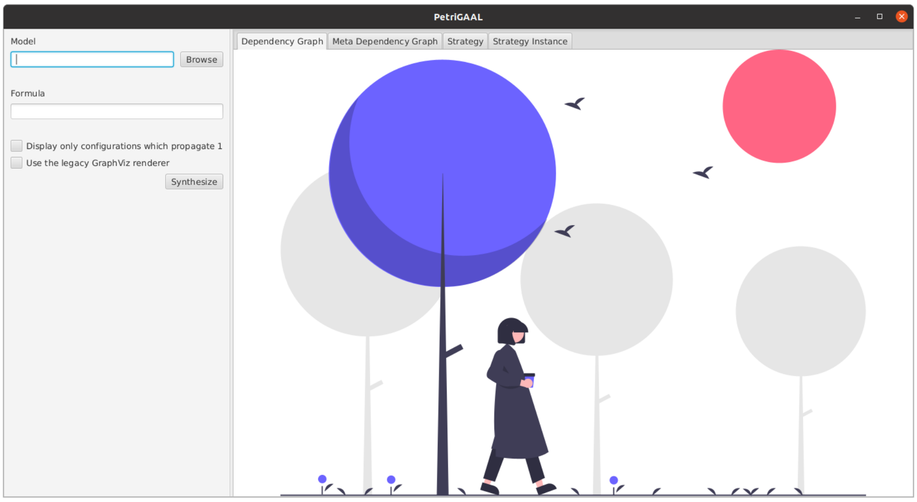
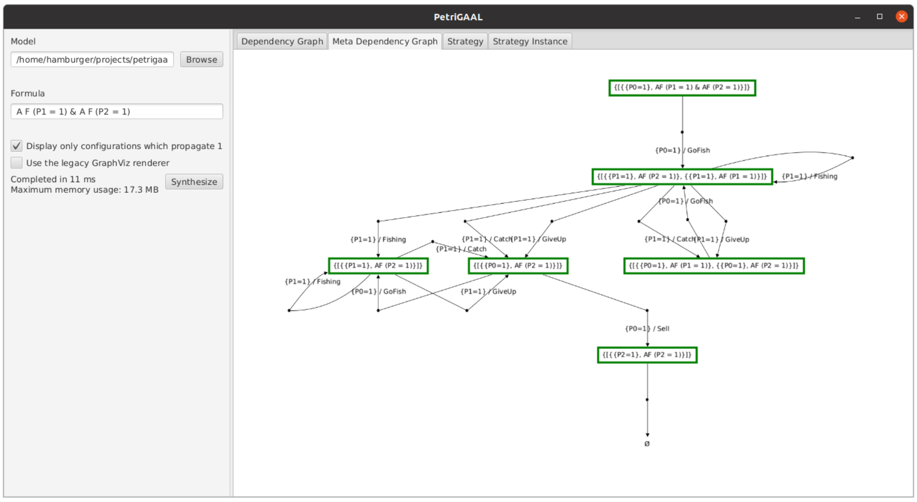


Fig. 15: The PetriGAAL homescreen

An example of the meta dependency graph that the tool produces for the fish running example given in Figure 1 and the CTL$^-$ query $AFs_1 \wedge AFs_2$ is shown Figure 16.

The configurations with green borders are the configurations which propagate 1 and therefore it is also possible to see why the root of the (meta) dependency graph does or does not propagate 1. All the configurations and hyper-edges on screen can be moved by the cursor, so it is possible to manually change the layout of the (meta) dependency graph. PetriGAAL can also filter away all configurations that do not propagate 1. This can be done by checking the *Display only configurations which propagate 1* checkbox. PetriGAAL can also use the *GraphViz* rendering engine (the default is Cytoscape.js). Notice also that the computation time and memory usage is listed to the left of the *Synthesize* button.



Model

/home/hamburger/projects/petrigaa Browse

Formula

A F (P1 = 1) & A F (P2 = 1)

☑ Display only configurations which propagate 1

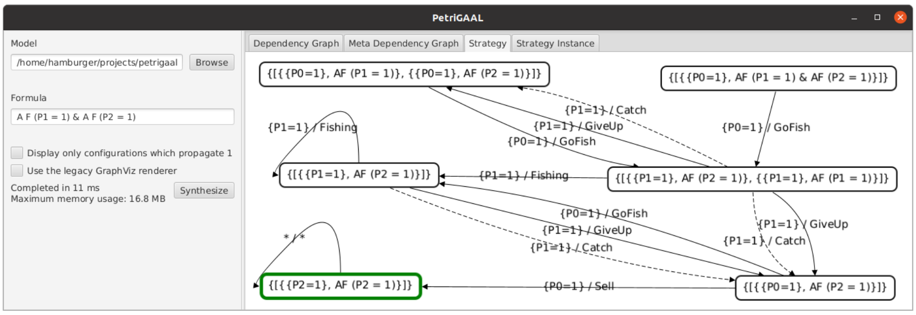
☐ Use the legacy GraphViz renderer

Completed in 11 ms
Maximum memory usage: 17.3 MB

Synthesize

Fig. 16: MDG constructed from the running example

The most permissive strategy produced by PetriGaal is shown in Figure 17. The solid transitions are what actions the controller proposes at a given state and the dashed transitions are the relevant uncontrollable actions that we observe. The state with the green border is the final state.



Model

/home/hamburger/projects/petrigaal Browse

Formula

A F (P1 = 1) & A F (P2 = 1)

☐ Display only configurations which propagate 1

☐ Use the legacy GraphViz renderer

Completed in 11 ms
Maximum memory usage: 16.8 MB

Synthesize

Fig. 17: Most Permissive Strategy for the running example

To recap, PetriGAAL is a almost complete work environment for GLKS strategy synthesis that graphically renders every stage in the procedure, from a GLKS and a CTL^{-} formula all the way to a deterministic instance of the produces most permissive strategy.

7 Conclusion

We presented a framework that solves the *Synthesis Decision Problem* and the *Synthesis Problem* in a time-branching setting. The framework is able to solve the *Synthesis Decision Problem* with the use of Negation-free CTL, however, the framework requires a restricted version CTL, namely CTL$^-$, to solve the *Synthesis Problem.* We demonstrated the difficulties when working with synthesis in a time-branching setting, as it is not compositional in the structure of the formula. We solve this complication by introducing Meta dependency graphs which are in disjunctive normal form. These Meta dependency graphs alone were able to solve the *Synthesis Decision Problem.* However, to solve the *Synthesis Problem*, we presented an algorithm able to produce the most permissive strategy for any CTL$^-$ formula φ together with a GLKS GS. The synthesis algorithm takes in a most permissive strategy (allows all winning behavior) as input and returns a strategy instance for a given CTL$^-$ formula φ together with a GLKS GS.

Moreover, we presented an implementation called PetriGAAL, which implements all core concepts in the framework. PetriGAAL is set up with a nice GUI that visually demonstrates all steps in the verification phase. The tool allows the user to inspect the resulting Dependency graph, Meta dependency graph, most permissive strategy, and strategy instance for a given CTL$^-$ formula and a system model. On a final note, we did not prove our main claim, which states given a Meta dependency graph, the graph only propagates one if and only if there exists a strategy such that $GS \models \varphi$. However, we strongly feel that this claim is true. The feeling is based on the experience we gained in testing the implementation. However, this remains future work.

7.1 Future Work

We plan to extend our framework to be able to solve the *Synthesis Problem* for the full CTL. This work would entail introducing epsilon edges to the construction of most permissive strategies. Furthermore, work on the PetriGAAL implementation is required to make the tool more useful. For instance, currently, both the dependency graph generation and evaluation are being done using the global algorithm [13]. Instead, an on-the-fly approach would probably result in a significant speed-up. In addition, in [6], Dalsgaard et al. show how to use multiple threads for dependency graph evaluation. Applying this approach to PetriGAAL would only require only minor modifications to the algorithm. Currently, the PetriGAAL implementation only accepts Petri Nets. It would be a nice feature to add other modeling languages such as Game Labelled Kripke Structure.

Acknowledgment. We want to thank our supervisors Jiří Srba and Peter Gjøl Jensen, for helping us along the way. Their engaging supervisor meetings and dedication of time and effort are very much appreciated.

Bibliographical Remarks The following sections are either taken from or based on our prior work [18]:

- Definition 1,
- all definitions between Definition 1 and Definition 2,
- Section 3,
- Definition 4,
- Definition 5,
- Definition 6,
- Definition 8,
- Subsection 4.1, and
- the *next* and *delta* functions in Subsection 4.2

References

[1] Thomas Ågotnes, V. Goranko, and W. Jamroga. "Alternating-time temporal logics with irrevocable strategies". In: *TARK '07*. 2007.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. "Alternating-time temporal logic". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, pp. 100–109. DOI: `10.1109/SFCS.1997.646098`.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Vol. 26202649. Jan. 2008. ISBN: 978-0-262-02649-9.

[4] Julien Bernet, David Janin, and Igor Walukiewicz. "Permissive strategies: From parity games to safety games". In: *RAIRO - Theoretical Informatics and Applications* 36 (July 2002). DOI: `10.1051/ita:2002013`.

[5] Krishnendu Chatterjee and Monika Henzinger. "Efficient and Dynamic Algorithms for Alternating BüChi Games and Maximal End-Component Decomposition". In: *J. ACM* 61.3 (June 2014). ISSN: 0004-5411. DOI: `10.1145/2597631`. URL: `https://doi.org/10.1145/2597631`.

[6] Andreas Engelbredt Dalsgaard et al. "A Distributed Fixed-Point Algorithm for Extended Dependency Graphs". In: *Fundam. Informaticae* 161.4 (2018), pp. 351–381. DOI: `10.3233/FI-2018-1707`. URL: `https://doi.org/10.3233/FI-2018-1707`.

[7] Alexandre David et al. "TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 492–497. DOI: `10.1007/978-3-642-28756-5\_36`. URL: `https://doi.org/10.1007/978-3-642-28756-5%5C_36`.

[8] Max Franz. *Cytoscape.js*. Online. Can also be found on GitHub under the same name. May 2013. URL: `https://js.cytoscape.org/`.

[9] Emden R. Gansner et al. "A Technique for Drawing Directed Graphs". In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 19.3 (1993), pp. 214–230.

[10] Isabella Kaufmann, Kim Guldstrand Larsen, and Jiří Srba. “Synthesis for Multi-weighted Games with Branching-Time Winning Conditions”. In: *Application and Theory of Petri Nets and Concurrency.* Ed. by Ryszard Janicki, Natalia Sidorova, and Thomas Chatain. Cham: Springer International Publishing, 2020, pp. 46–66. ISBN: 978-3-030-51831-8.

[11] Dexter Kozen. “Results on the propositional μ-calculus”. In: *Automata, Languages and Programming.* Ed. by Mogens Nielsen and Erik Meineche Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 348–359. ISBN: 978-3-540-39308-5.

[12] Orna Kupferman et al. “Open Systems in Reactive Environments: Control and Synthesis”. In: *CONCUR 2000 — Concurrency Theory.* Ed. by Catuscia Palamidessi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 92–107. ISBN: 978-3-540-44618-7.

[13] Xinxin Liu and Scott A. Smolka. “Simple Linear-Time Algorithms for Minimal Fixed Points (Extended Abstract)”. In: *Automata, Languages and Programming, 25th International Colloquium, ICALP’98, Aalborg, Denmark, July 13-17, 1998, Proceedings.* Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998, pp. 53–66. DOI: 10.1007/BFb0055040. URL: https://doi.org/10.1007/BFb0055040.

[14] G. H. Mealy. “A method for synthesizing sequential circuits”. In: *The Bell System Technical Journal* 34.5 (Sept. 1955), pp. 1045–1079. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[15] Carl Adam Petri. “Communication with automata”. eng. PhD thesis. Universität Hamburg, 1966.

[16] A. Pnueli and R. Rosner. “On the Synthesis of a Reactive Module”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190. ISBN: 0897912942. DOI: 10.1145/75277.75293. URL: https://doi.org/10.1145/75277.75293.

[17] M. O. Rabin and D. Scott. “Finite Automata and Their Decision Problems”. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114.

[18] Beinir Ragnuson, Sigmundur Vang, and Bogi Napoleon Wennerström. “Efficient Synthesis for Games with Branching-Time Winning Conditions”. In: (2021).

[19] Grzegorz Rozenberg and Joost Engelfriet. “Elementary net systems”. In: Apr. 2006, pp. 12–121. ISBN: 978-3-540-65306-6. DOI: 10.1007/3-540-65306-6_14.

[20] Klaus Schneider. *Verification of Reactive Systems. Formal Methods and Algorithms.* 1st ed. Springer-Verlag Berlin Heidelberg, 2004. URL: https://www.springer.com/gp/book/9783540002963.

[21] Michael Sipser. “Introduction to the Theory of Computation”. In: 1st. International Thomson Publishing, 1996, pp. 284–287. ISBN: 053494728X.

[22] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math.* 5.2 (1955), pp. 285–309. URL: `https://projecteuclid.org:443/euclid.pjm/1103044538`.

[23] Bogi Napoleon Wennerström, Beinir Ragnuson, and Sigmundur Vang. *PetriGAAL*. Online. June 2021. URL: `https://github.com/boginw/petrigaal`.