Symbolic Fixpoint Algorithms for Logical LTL Games

Stanly Samuel
Computer Science and Automation
Indian Institute of Science
Bengaluru, India
stanly@iisc.ac.in

Deepak D'Souza
Computer Science and Automation
Indian Institute of Science
Bengaluru, India
deepakd@iisc.ac.in

Raghavan Komondoor
Computer Science and Automation
Indian Institute of Science
Bengaluru, India
raghavan@iisc.ac.in

***Abstract*—Two-player games are a fruitful way to represent and reason about several important synthesis tasks. These tasks include controller synthesis (where one asks for a controller for a given plant such that the controlled plant satisfies a given temporal specification), program repair (setting values of variables to avoid exceptions), and synchronization synthesis (adding lock/unlock statements in multi-threaded programs to satisfy safety assertions). In all these applications, a solution directly corresponds to a winning strategy for one of the players in the induced game. In turn, *logically-specified* games offer a powerful way to model these tasks for large or infinite-state systems. Much of the techniques proposed for solving such games typically rely on abstraction-refinement or template-based solutions. In this paper, we show how to apply classical fixpoint algorithms, that have hitherto been used in explicit, finite-state, settings, to a symbolic logical setting. We implement our techniques in a tool called GENSYS-LTL and show that they are not only effective in synthesizing valid controllers for a variety of challenging benchmarks from the literature, but often compute maximal winning regions and maximally-permissive controllers. We achieve 46.38X speed-up over the state of the art and also scale well for non-trivial LTL specifications.**

***Index Terms*—reactive synthesis, symbolic algorithms, program synthesis, program repair, two-player games**

I. INTRODUCTION

Two-player games are games played between two players called the Controller and the Environment, on a game graph or arena. The players generate an infinite sequence of states (a so-called "play") in the game by making moves alternately, from a specified set of legal moves. The Controller wins the play if the sequence of states satisfies a winning condition (e.g., a Linear-Time Temporal Logic (LTL) formula). The central question in these games is whether a player (typically the Controller) has a winning strategy from a given set of initial states (called the realizability problem), or more generally, to compute the set of states from which she wins (i.e. the winning region).

Games are a fruitful way to model and reason about several important problems in Sofware Engineering, like *controller synthesis* [1] (where a winning strategy for the Controller in the associated game directly corresponds to a valid controller for the system); *program repair* [2] (strategy corresponds to corrected program); *synchronization synthesis* [3] (strategy corresponds to appropriate placement of synchronization statements in a concurrent program); and *safety shield synthesis* [4] (winning region corresponds to region in which the neural-network based controller is allowed to operate without the shield stepping in).

Classical techniques for solving games [5]–[7], and more recent improvements [8]–[10], work on finite-state games, by iteratively computing sets of states till a fixpoint is reached. These algorithms typically allow us to compute the exact winning region and thereby answer the realizability question as well.

In recent years, *logical games* – where the moves of the players are specified by logical formulas on the state variables – have attracted much attention, due to their ability to model large or infinite-state systems. Techniques proposed for these games range from constraint solving [11], finite unrollings and generalization [12], CEGAR-based abstraction-refinement [13]–[15], counterexample-based learning [16], combination of Sygus and classical LTL synthesis [17], and solver-based enumeration [18]. Among these Beyene et al [11] address general LTL specs, while the others handle only safety or reachability specs. Furthermore, none of these techniques are able to compute precise winning regions.

In this paper we show that symbolic fixpoint techniques can be effectively applied to solve logical games with general LTL specifications. We propose a bouquet of techniques that target different classes of LTL specs, from simple specs which directly involve a safety, reachability, Büchi, or Co-Büchi condition on the states of the game, to those for which the formula automata are non-deterministic. The techniques we propose are guaranteed (whenever they terminate) to compute the *exact* winning region, and, for certain kinds of games, output a finite-memory winning strategy as well.

We show how to implement these algorithms in a logical setting, by leveraging the right tactics in available SMT solvers. We evaluate our prototype tool, called GENSYS-LTL, on a host of benchmarks from the literature. Our tool terminates on all benchmarks except one, and takes an average time of 7.1 sec to solve each benchmark. It thus outperforms the state-of-the-art tools in terms of the number of instances solved, and by an order of magnitude in terms of running time.

II. Preliminaries

We will be dealing with standard first-order logic of addition ($+$), comparison ($<$), and constants 0 and 1, interpreted over the domain of reals $\mathbb{R}$ (or a subset of $\mathbb{R}$ like the integers $\mathbb{Z}$). The atomic formulas in this logic are thus of the form $a_1x_1 + \cdots + a_nx_n \sim c$, where a_is and c are integers, x_is are variables, and "$\sim$" is a comparison symbol in $\{<, \leq, =, \geq, >\}$. We will refer to such formulas as *atomic constraints*, and to boolean combinations of such formulas (or equivalently, quantifier-free formulas) as *constraints*. We will denote the set of constraints over a set of variables V by $Constr(V)$.

For a set of variables V, a V*-valuation* (or a V*-state*) is simply a mapping $s : V \to \mathbb{R}$. Given a constraint δ over a set of variables V, and a V-state s, we say s *satisfies* δ, written $s \vDash \delta$, if the constraint δ evaluates to true in s (defined in the expected way). We denote the set of V-states by $\mathbf{V}_{\mathbb{R}}$. A *domain mapping* for V is a map $D : V \to 2^{\mathbb{R}}$, which assigns a domain $D(x) \subseteq \mathbb{R}$ for each variable x in V. We will call a V-state s whose range respects a domain mapping D, in that for each $x \in V$, $s(x) \in D(x)$), a (V, D)-state, and denote the set of such (V, D)-states by $\mathbf{V}_D$. We also denote the cardinality of a set S as $|S|$.

We will sometimes write $\varphi(X)$ to denote that the free variables in a formula φ are among the variables in the set X. For a set of variables $X = \{x_1, \ldots, x_n\}$ we will sometimes use the notation X' to refer to the set of "primed" variables $\{x'_1, \ldots, x'_n\}$. For a constraint φ over a set of variables $X = \{x_1, \ldots, x_n\}$, we will write $\varphi[X'/X]$ (or simply $\varphi(X')$ when X is clear from the context) to represent the constraint obtained by substituting x'_i for each x_i in φ.

Finally, we will make use of standard notation from formal languages. For a (possibly infinite) set S, we will view finite and infinite sequences of elements of S as finite or infinite *words* over S. We denote the empty word by ϵ. If v and w are finite words and α an infnite word over S, we denote the conatenation of v and w by $v \cdot w$, and the concatenation of v and α by $v \cdot \alpha$. We will use S^* and S^ω to denote, respectively, the set of finite and infinite words over S.

III. LTL and Automata

We will make use of a version of Linear-Time Temporal Logic (LTL) [19] where propositions are atomic constraints over a set of variables V (as in Holzmann [20], for example).

Let V be a set of variables. Then the formulas of $LTL(V)$ are given by:

$$\psi ::= \delta \mid \neg\psi \mid \psi \vee \psi \mid X\psi \mid \psi U \psi,$$

where δ is an atomic constraint over V. The formulas of $LTL(V)$ are intepreted over an infinite sequence of V-states. For an $LTL(V)$ formula ψ and an infinite sequence of V-states $\pi = s_0 s_1 \cdots$, we define when ψ is satisfied at position i in π, written $\pi, i \vDash \psi$, inductively as follows:

$$\begin{array}{lll}
\pi, i \vDash \delta & \text{iff} & s_i \vDash \delta \\
\pi, i \vDash \neg\psi & \text{iff} & \pi, i \nvDash \psi \\
\pi, i \vDash \psi \vee \psi' & \text{iff} & \pi, i \vDash \psi \text{ or } \pi, i \vDash \psi' \\
\pi, i \vDash X\psi & \text{iff} & \pi, i+1 \vDash \psi \\
\pi, i \vDash \psi U \psi' & \text{iff} & \exists k \geq i \text{ s.t. } \pi, k \vDash \psi' \text{ and} \\
 & & \forall j : i \leq j < k \to \pi, j \vDash \psi.
\end{array}$$

We say π *satisfies* ψ, written $\pi \vDash \psi$, if $\pi, 0 \vDash \psi$. We will freely make use of the derived operators F ("future") and G ("globally") defined by $F\psi \equiv \mathit{true}\, U\, \psi$ and $G\psi \equiv \neg F \neg \psi$, apart from the boolean operators $\wedge$ ("and"), $\to$ ("implies"), etc.

An ω*-automaton* [21] $\mathcal{A}$ over a set of variables V, is a tuple $(Q, I, \mathcal{T}, F)$ where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $\mathcal{T} \subseteq_{\mathit{fin}} Q \times Constr(V) \times Q$ is a "logical" transition relation, and $F \subseteq Q$ is a set of final states. The logical transition relation $\mathcal{T}$ induces a concrete transition relation $\Delta_{\mathcal{T}} \subseteq Q \times \mathbf{V}_{\mathbb{R}} \times Q$, given by $(q, s, q') \in \Delta_{\mathcal{T}}$ iff there exists $(q, \delta, q') \in \mathcal{T}$ such that $s \vDash \delta$. A *run* of $\mathcal{A}$ on an infinite sequence of V-states $\pi = s_0 s_1 \cdots$ is an infinite sequence of states $\rho = q_0 q_1 \cdots$, such that $q_0 \in I$, and for each i, $(q_i, s_i, q_{i+1}) \in \Delta_{\mathcal{T}}$.

We say an ω-automaton $\mathcal{A} = (Q, I, \mathcal{T}, F)$ over V, is *deterministic* if I is singleton, and for every $q \in Q$ and V-state s, there is at most one $q' \in Q$ such that $(q, s, q') \in \Delta_{\mathcal{T}}$. Similarly we say $\mathcal{A}$ is *complete* if for every $q \in Q$ and V-state s, there exists a $q' \in Q$ such that $(q, s, q') \in \Delta_{\mathcal{T}}$.

An ω-automaton can be viewed as either a *Büchi* [22], *Co-Büchi*, *Universal Co-Büchi*, or *Safety* automaton based on how the *runs* for a given V-state sequence π are *accepted* using the final states F, described as follows. A run $\rho = q_0 q_1 \cdots$ of $\mathcal{A}$ is *accepting* by the *Büchi* acceptance condition if for infinitely many i, we have $q_i \in F$, and a V-state sequence π is accepted by $\mathcal{A}$ if there exists such a run ρ for π. A *Büchi Automaton* is an ω-automaton where F is viewed as a Büchi acceptance condition. Similarly, by the *Co-Büchi* acceptance condition ρ is *accepting* if it visits F only a finite number of times and a V-state sequence π is accepted by $\mathcal{A}$ if there exists such a run ρ for π. We call such an automaton a *Co-Büchi Automaton (CA)*. The *Universal Co-Büchi* acceptance condition states that a run ρ of $\mathcal{A}$ is *accepting* if it visits F only a finite number of times, and a V-state sequence π is accepted by $\mathcal{A}$ if *all* runs ρ for π are accepting. We call such an automaton a *Universal Co-Büchi Automaton (UCA)*. Finally, we can view an ω-automaton $\mathcal{A}$ as a *safety* automaton, by saying that $\mathcal{A}$ accepts π iff there is a run of $\mathcal{A}$ on π which never visits a state outside F. We denote by $L(\mathcal{A})$ the set of V-state sequences accepted by an ω-automaton $\mathcal{A}$.

It is well-known that any LTL formula ψ can be translated into a (possibly non-deterministic) Büchi automaton $\mathcal{A}_\psi$ that accepts precisely the models of ψ [23]. The same construction works for $LTL(V)$ formulas, by treating each atomic constraint as a propositional variable. Henceforth, for

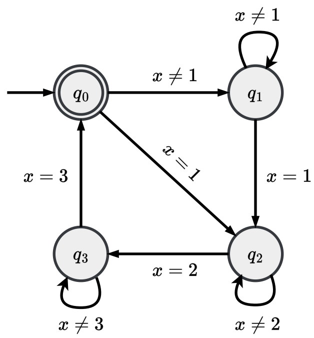


Fig. 1: Büchi automaton A_ψ for the LTL formula $\psi = G(F(x=1) \wedge F(x=2) \wedge F(x=3))$. Final states are indicated with double circles.

an $LTL(V)$ formula ψ we will denote the corresponding formula automaton by $\mathcal{A}_\psi$.

Fig. 1 shows a formula automaton $\mathcal{A}_\psi$ for the $LTL(V)$ formula $\psi = G(F(x = 1) \wedge F(x = 2) \wedge F(x = 3))$ from Example 4.1, where $V = \{x\}$. The automaton can be seen to be deterministic.

IV. LTL Games

In this section we introduce our notion of logically specified games, where moves are specified by logical constraints and winning conditions by LTL formulas. These games are similar to the formulation in Beyene et al [11].

A *2-player logical game with an LTL winning condition* (or simply an *LTL game*) is of the form

$$\mathcal{G} = (V, D, Con, Env, \psi), \quad where$$

- V is a finite set of variables.
- $D : V \to 2^{\mathbb{R}}$ is a domain mapping for V.
- Con and Env are both constraints over $V \cup V'$, representing the moves of Player C and Player E respectively.
- ψ is an $LTL(V)$ formula.

The constraint Con induces a transition relation

$$\Delta_{Con} \subseteq \mathbf{V}_D \times \mathbf{V}_D$$

given by $(s, s') \in \Delta_{Con}$ iff s and s' are (V, D)-states, and $(s, s') \vDash Con$. We use the notation $(s, s') \vDash Con$ to denote the fact that $t_{s,s'} \vDash Con$, where $t_{s,s'}$ is the valuation over $V \cup V'$ which maps each $x \in V$ to $s(x)$ and $x' \in V'$ to $s'(x)$. In a similar way, Env induces a transition relation $\Delta_{Env} \subseteq \mathbf{V}_D \times \mathbf{V}_D$. For convenience we will assume that the C-moves are "complete" in that for every (V, D)-state s, there is a (V, D)-state s' such that $(s, s') \in \Delta_{Con}$; and similarly for Player E.

A play in $\mathcal{G}$ is an sequence of (V, D)-states obtained by an alternating sequence of moves of Players C and E, with Player C making the first move. More precisely, an *(infinite) play* of $\mathcal{G}$, starting from a (V, D)-state s, is an infinite sequence of (V, D)-states $\pi = s_0 s_1 \cdots$, such that

- $s_0 = s$, and
- for all i, $(s_{2i}, s_{2i+1}) \in \Delta_{Con}$ and $(s_{2i+1}, s_{2i+2}) \in \Delta_{Env}$.

We similarly define the notion of a *finite* play w in the expected manner. We say a play π is *winning* for Player C if it satisfies ψ (i.e. $\pi \vDash \psi$); otherwise it is winning for Player E.

A strategy for Player C assigns to odd-length sequences of states, a non-empty subset of states that correspond to legal moves of C. More precisely, a *strategy* for Player C in $\mathcal{G}$ is a partial map

$$\sigma : ((\mathbf{V}_D \cdot \mathbf{V}_D)^* \cdot \mathbf{V}_D) \rightharpoonup 2^{\mathbf{V}_D}$$

satisfying the following constraints. We first define when a finite play w is *according to* σ, inductively as follows:

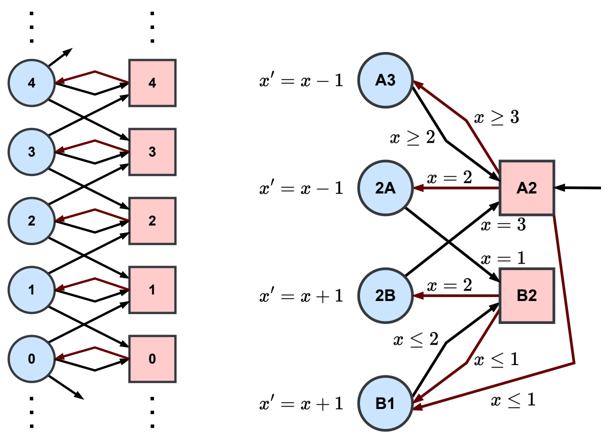
- s is according to σ iff s belongs to the domain of σ.
- if $w \cdot s$ is of odd length and according to σ, and $s' \in \sigma(w \cdot s)$, then $w \cdot s \cdot s'$ is according to σ.
- if $w \cdot s$ is of even length and according to σ, and $(s, s') \in \Delta_{Env}$, then $w \cdot s \cdot s'$ is according to σ.

For σ to be a valid strategy in $\mathcal{G}$, we require that for every finite play $w \cdot s$ of odd-length, which is according to σ, $\sigma(w \cdot s)$ must be defined and non-empty, and for each $s' \in \sigma(w \cdot s)$ we must have $(s, s') \in \Delta_{Con}$.

Finally, a strategy σ for Player C is *winning* from a (V, D)-state s in its domain, if every play that starts from s and is according to σ, is winning for Player C (i.e. the play satisfies ψ). We say σ itself is *winning* if it is winning from every state in its domain. We say Player C *wins* from a (V, D)-state s if it has a strategy which is wining from s. We call the set of (V, D)-states from which Player-C wins, the *winning region* for Player C in $\mathcal{G}$, and denote it $winreg_C(\mathcal{G})$. The analogous notions for Player E are defined similarly.

We close this section with some further notions about strategies. We say that a winning strategy σ for C is *maximal* if its domain is $winreg_C(\mathcal{G})$, and for every strategy σ' for C that is winning from a state s in $winreg_C(\mathcal{G})$, we have $\sigma'(w) \subseteq \sigma(w)$ for each odd-length play w from s according to σ'. A strategy σ for C is called a *(finitely-representable) finite memory* strategy, if it can be represented by a "Mealy-style" *strategy automaton* (see Fig. 2b). This is a finite-state automaton similar to a deterministic Büchi automaton, but with a partition of the states into controller and environment states. The initial states are environment states. The states in the domain of the strategy are those that satisfy one of the outgoing guards from the initial state. Each controller state q has a label $mov(q)$ associated with it in the form of a constraint over $V \cup V'$, which denotes a subset of moves allowed by Con. The automaton represents a strategy σ in which $\sigma(w)$ for odd-length w is given by the label $mov(q)$ of the state q reached by the automaton on reading w. Finally, a *memoryless* strategy is one that is represented by a strategy automaton with a *single* environment state.

Synthesizing winning strategies will be easier when the controller's moves are *finitely non-deterministic*, in that Con is given by a disjunction $Con_1 \vee \cdots \vee Con_k$, where each constraint Con_i is *deterministic* (in that whenever $(s, s') \vDash Con_i$ and $(s, s'') \vDash Con_i$, we have $s' = s''$). We call such a game a *finitely non-deterministic* (FND) game.



(a) Game graph (b) Strategy automaton

Fig. 2: Game graph and strategy for C in Elevator game

We illustrate some of these notions through an example below adapted from [15].

Example 4.1 (Elevator): Consider a game $\mathcal{G}_1$ which models an elevator control problem, where the system's state is represented by a single variable x of type integer, indicating the floor the elevator is currently positioned at. The controller can choose to move the elevator up or down by one floor, or stay at the same floor. The environment does nothing (simply "skips"). The specification requires us visit each of Floor 1, 2, and 3 infinitely often. The game $\mathcal{G}_1$ has the following components: the set of variables V is $\{x\}$, and the domain map D is given by $D(x) = \mathbb{Z}$. The moves of Player C and Player E are given by the constraints Con: $x' = x \vee x' = x+1 \vee x' = x-1$, and Env: $x' = x$, respectively. The LTL specification ψ is $G(F(x = 1) \wedge F(x = 2) \wedge F(x = 3))$. The game is easily seen to be finitely non-deterministic.

The "game graph" is shown in Fig. 2a. For convenience we visualize the game as having two copies of the state space, one where it is the turn of Player C to make a move (denoted by circle states on the left) and the other where it is Player E's turn to move (indicated by square states on the right). The moves of C go from left to right, while those of E go from right to left.

Player C has a winning strategy from all states; for example, by playing $x' = x - 1$ from Floor 3 and above; $x' = x + 1$ from Floor 1 and below; and $x' = x + 1$ and $x' = x - 1$ from Floor 2, depending on whether it was last in Floor 1 or 3 respectively. This finite-memory strategy is shown by the strategy automaton in Fig. 2b.

It is easy to see that a memoryless winning strategy does *not* exist for Player C, as it cannot afford to play the *same* move from state $x \mapsto 2$ (it must keep track of the direction in which the lift is coming from). □

We close this section with a description of the problems we address in this paper. The main problems we address are the following:

1) (*Winning Region*) Given an LTL game $\mathcal{G}$, compute the winning region for Player C. Wherever possible, also compute a finite-memory winning strategy for C from this region.
2) (*Realizability*) Given an LTL game $\mathcal{G}$ and an initial region in the form of a constraint $Init$ over the variables V of the game, decide whether Player C wins from every state in $Init$. If possible, compute a finite-memory winning strategy for C from $Init$.

It is easy to see that these problems are undecidable in general (for example by a reduction from the control-state reachability problem for 2-Counter Machines). Hence the procedures we give in subsequent sections may not always be terminating ones. In the sequel we focus on the problem of computing winning regions, since we can check realizability by checking if the given initial region is contained in the winning region.

V. GenSys-LTL Approach

Algorithm 1: GenSys-LTL overview

Input : LTL game $\mathcal{G} = (V, D, Con, Env, \psi)$
Output: $winreg_C(\mathcal{G})$ or an approximation of it, and a strategy σ for Player C from this region.

if $\mathcal{G}$ *is simple* **then**
  Compute $winreg_C(\mathcal{G})$ (i.e. winning reg for C in $\mathcal{G}$).
  Compute winning strategy σ.
  return $winreg_C(\mathcal{G})$, σ.
$\mathcal{A}_\psi := LTL2BA(\psi)$.
$\mathcal{A}_{\neg\psi} := LTL2BA(\neg\psi)$.
if $\mathcal{A}_\psi$ *is deterministic* **then**
  Construct simple Büchi product game $\mathcal{H} = \mathcal{G} \otimes \mathcal{A}_\psi$.
  Compute $winreg_C(\mathcal{H})$.
  Extract $winreg_C(\mathcal{G})$, winning strategy σ for C in $\mathcal{G}$.
  return $winreg_C(\mathcal{G})$, σ.
if $\mathcal{A}_{\neg\psi}$ *is deterministic* **then**
  Construct simple Co-Büchi product game $\mathcal{H} = \mathcal{G} \otimes \mathcal{A}_{\neg\psi}$. Compute $winreg_C(\mathcal{H})$.
  Extract $winreg_C(\mathcal{G})$, winning strategy σ for C.
  return $winreg_C(\mathcal{G})$, σ.
`// Both` $\mathcal{A}_\psi$ `and` $\mathcal{A}_{\neg\psi}$ `are non-det`
$k := 0$; $W_U := false$; $W_O := true$;
while $W_U \neq W_O$ **do**
  Construct on-the-fly two k-safety product automatons involving $\mathcal{G}$ with A_ψ and $A_{\neg\psi}$, respectively, and from these, extract an under-approximation W_U of $winreg_C(\mathcal{G})$ and an over-approximation W_O of $winreg_C(\mathcal{G})$, respectively. From W_U extract a winning strategy σ for Player C.
  $k = k + 1$.
return W_U, W_O, σ;

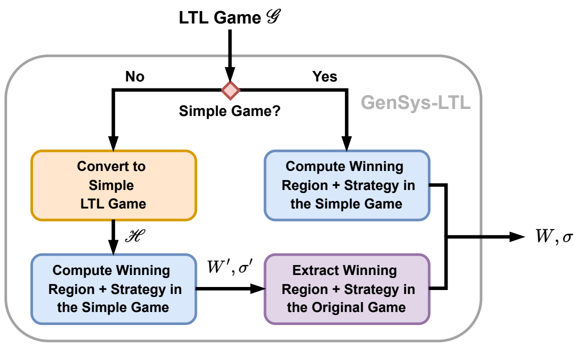


Fig. 3: Schematic overview of GenSys-LTL

Our approach consists of a bouquet of techniques. This is motivated by our objective to handle each type of LTL formula with an efficient technique suited to that type. Algorithm 1 is the "main" program or driver of our approach. Fig. 3 also summarizes the approach.

Algorithm 1 takes as input an LTL game $\mathcal{G}$. Lines 1–4 of the algorithm tackle the scenario when the given game $\mathcal{G}$ is *simple*. These are games where the formula ψ is of one of restricted forms $G(X)$ (safety), $F(X)$ (reachability), $GF(X)$ (Büchi), or $FG(X)$ (Co-Büchi), where X is a constraint over the game variables V. For these cases, we propose fixpoint procedures that directly work on the state-space of the given game $\mathcal{G}$, and that use SMT formulae to encode sets of states. Sec. VI describes these procedures in detail. Due to the infiniteness of the state-space, these fixpoint computations are not guaranteed to terminate. When they do terminate, they are guaranteed to compute the precise winning region $winreg_C(\mathcal{G})$, and, in the case of FND games, to extract a memoryless strategy automaton for these regions.

If the given formula ψ is not simple, we convert the formula as well as its negation, in Lines 5–6, into Büchi automata using a standard procedure [23]. If either of these two automata are *deterministic* (see Sec. III), we construct a product of the game $\mathcal{G}$ with the automaton, such that this product-game $\mathcal{H}$ is a simple LTL game. We then compute the winning region on this product using the fixpoint computations mentioned above. If the fixpoint computation terminates, we extract a winning region for the original game $\mathcal{G}$, and a strategy σ. These steps are outlined in Lines 7–15 of Algorithm 1, and details are presented in Sec. VII.

The hardest scenario is when both the automata are non-deterministic. For this scenario, we propose an on-the-fly determinization and winning-region extraction approach. These steps are outlined in Lines 16-19 of Algorithm 1. We present the details in Section VIII.

We state the following claim which we will substantiate in subsequent sections:

Theorem 1: Whenever Algorithm 1 terminates, it outputs the exact winning region for Player C when $\mathcal{G}$ is either simple or deterministic; in other cases it outputs a sound under- and over-approximation of the winning region for Player C in $\mathcal{G}$. Additionally, in the case when $\mathcal{G}$ is FND, upon termination Algorithm 1 outputs a strategy automaton representing a winning strategy for Player C from this region. □

VI. Simple LTL Games

Our approach reduces logical LTL games to "simple" LTL games in which the winning condition is *internal* to the game. In this section we describe this subclass of LTL games and the basic fixpoint algorithms to solve them.

A *simple* LTL game is an LTL game $\mathcal{G} = (V, D, Con, Env, \psi)$, in which ψ is an $LTL(V)$ formula of the form $G(X)$, $F(X)$, $GF(X)$, or $FG(X)$, where X is a constraint on V. We refer to games with such specifications as *safety*, *reachability*, *Büchi*, and *co-Büchi* games, respectively.

We can compute (with the possibility of non-termination) the winning regions $winreg_C(\mathcal{G})$ for each of these four types of games, and a strategy automaton representing a memoryless winning strategy for Player C, in the special case of FND games, by extending the classical algorithms for the finite-state versions of these games (see [6], [7]).

The algorithms we describe will make use of the following formulas representing sets of "controllable predecessors" in the context of different types of games. Here $Y(V)$ is a constraint representing a set of game states.

- The set of controllable predecessors w.r.t. the set of states Y, for a safety specification $G(X)$ (namely states from which Player C has a *safe* move from which all environment moves result in a Y-state):

$$CP_S^X(Y) \equiv \exists V'(Con(V,V') \wedge X(V') \wedge \forall V''(Env(V',V'') \implies Y(V''))).$$

- The set of controllable predecessors w.r.t. Y, for a reachability specification $F(X)$ (namely states from which either C has a move that either gets into X, or from which all environment moves get into Y):

$$CP_R^X(Y) \equiv \exists V'(Con(V,V') \wedge (X(V') \vee \forall V''(Env(V',V'') \implies Y(V'')))).$$

- The set of predecessors w.r.t. Y for Player C (namely states from which C has a move that results in a Y-state):

$$CP_C(Y) \equiv \exists V'(Con(V,V') \wedge Y(V')).$$

- The set of predecessors w.r.t. Y for Player E (namely states from which all moves of E result in a Y-state):

$$CP_E(Y) \equiv \forall V'(Env(V,V') \implies Y(V')).$$

Algorithm 2 (ComputeWR-Safety) takes a safety game as input, and iteratively computes the safe controllable predecessors, starting with the given safe set X, until it reaches a fixpoint ($W_{old} \implies W$). Here we use a quantifier elimination procedure *QElim* which takes a logical formula with quantifiers (like $CP_S^X(W) \wedge X$) and returns an equivalent quantifier-free formula. For example, $QElim(\exists y(y \leq x \wedge x + y \leq 1 \wedge 0 \leq y))$ returns $0 \leq x \wedge x \leq 1$. Upon termination the algorithm returns the fixpoint W.

Algorithm 2: ComputeWR-Safety

Input : Safety game $\mathcal{G} = (V, D, Con, Env, G(X))$
Output: $winreg_C(\mathcal{G})$, strategy σ

$W := X$;
do
  $W_{old} := W$;
  $W := QElim(CP_S^X(W) \wedge X)$;
while $\neg(W_{old} \Rightarrow W)$;
$\sigma := ExtractStrategy_G(W)$;
return W, σ;

Algorithm 3: ComputeWR-Reachability

Input : Reachability game $\mathcal{G} = (V, D, Con, Env, F(X))$
Output: $winreg_C(\mathcal{G})$, strategy σ

$W := X$;
$C := [W]$;
do
  $W_{old} := W$;
  $W := QElim(CP_R^X(W) \vee X)$;
  $C.append(W \wedge \neg W_{old})$;
while $\neg(W \Rightarrow W_{old})$;
$\sigma := ExtractStrategy_F(C)$;
return W, σ;

Algorithm 4: ComputeWR - Büchi

Input : Büchi game $\mathcal{G} = (V, D, Con, Env, GF(X))$
Output: $winreg_C(\mathcal{G})$, strategy σ

$W := W_E := True$;
do
  $W_{E_{old}}, W_{old} := W_E, W$;
  $W := QElim(CP_C(W_E) \wedge X)$;
  $W_E := QElim(CP_E(W) \wedge X)$;
  $H := H_E := False$;
  $C := [H]$;
  do
    $H_{E_{old}}, H_{old} := H_E, H$;
    $H := QElim(CP_C(H_E) \vee W)$;
    $H_E := QElim(CP_E(H) \vee W_E)$;
    $C.append(H \wedge \neg H_{old})$;
  while $\neg(H_E \Rightarrow H_{E_{old}} \ \wedge \ H \Rightarrow H_{old})$;
  $W_E, W := H_E, H$;
while $\neg(W_{E_{old}} \Rightarrow W_E \ \wedge \ W_{old} \Rightarrow W)$;
$\sigma := ExtractStrategy_{GF}(W, C)$;
return W, σ;

Algorithm 5: ComputeWR - Co-Büchi

Input : Co-Büchi game $\mathcal{G} = (V, D, Con, Env, FG(X))$
Output: $winreg_C(\mathcal{G})$, strategy σ

$W := W_E := False$;
$C := [W]$;
do
  $W_{E_{old}}, W_{old} := W_E, W$;
  $W := QElim(CP_C(W_E) \vee X)$;
  $W_E := QElim(CP_E(W) \vee X)$;
  $H := H_E := True$;
  do
    $H_{E_{old}}, H_{old} := H_E, H$;
    $H := QElim(CP_C(H_E) \wedge W)$;
    $H_E := QElim(CP_E(H) \wedge W_E)$;
  while $\neg(H_{E_{old}} \Rightarrow H_E \ \wedge \ H_{old} \Rightarrow H)$;
  $W_E, W := H_E, H$;
  $C.append(W \wedge \neg W_{old})$;
while $\neg(W_E \Rightarrow W_{E_{old}} \ \wedge \ W \Rightarrow W_{old})$;
$\sigma := ExtractStrategy_{GF}(W, C)$;
return W, σ;

When the input game is FND (with $Con = Con_1 \vee \cdots \vee Con_k$), the call to $ExtractStrategy_G(W)$ does the following. Let

$$U_i = \quad W \wedge QElim(\exists V'(\quad Con_i(V, V') \ \wedge W(V') \ \wedge \\ \forall V''(Env(V', V'') \implies W(V'')))).$$

Then the memoryless strategy σ extracted simply offers the move Con_i whenever Player C is in region U_i. The corresponding strategy automaton essentially maintains a controller state for each constraint U_i, labelled by the move Con_i. For the strategy extraction in the rest of this section, we assume that the input game is FND.

Similarly, Algorithm 3 (ComputeWR-Reachability) takes a reachability game as input, and iteratively computes the reachable controllable predecessors, starting with the given safe set X, until it reaches a fixpoint ($W \implies W_{old}$).

To compute the memoryless strategy for reachability, we compute C that ensures that each move made by the controller from a given state ensures that it moves one step closer to X.

Let the reachability controllable predecessor for move Con_i be:

$$CP_{R_i}^X(Y) \quad \equiv \quad \exists V'(Con_i(V, V') \ \wedge (X(V') \ \vee \\ \forall V''(Env(V', V'') \implies Y(V'')))).$$

Then $ExtractStrategy_F(C)$ does the following:

$$U_i = \quad \bigvee_{j=0}^{|C|-2} QElim(CP_{R_i}^{X_j}(X_j)) \wedge C_{j+1}$$

where $X_j = C_j \vee C_{j-1} \vee C_{j-2} \vee \cdots \vee C_0$.

Thus, U_i is the set of states exclusively in W_{j+1} (which we denote by C_{j-1} which is constructed in Algorithm 3) from where Player C has a strategy to reach X by first ensuring a move to W_j, thereby ensuring moving one step closer to X. Then the memoryless strategy σ extracted offers the move Con_i whenever Player C is in the region U_i. The corresponding strategy automaton essentially maintains a controller state for each constraint U_i, labelled by the move Con_i.

Algorithm 4 (ComputeWR- Büchi) takes a Büchi game as an input, and computes a winning region from where Player C has a strategy to visit X infinitely often. In this algorithm, we require two levels of nesting to compute the winning region. Using two-step controllable predecessors (such as CP_S^X, and CP_R^X), that reason about two moves at a time causes unsoundness, if used directly. Using CP_S^X in the nested Buchi algorithm causes an underapproximation of the winning region as it is not necessary that the intermediate environment states be safe. Similarly, using CP_R^X is too weak as the intermediate states of the environment are not reasoned with correctly. It assumes that a finite play reaching an intermediate environment state in X satisfies the property, which is not true for an infinite Büchi play. Thus, we use one step controllable predecessors CP_C and CP_E (for controller and environment respectively) that reason about the game play one move at a time in the style of [6]. The strategy is also extracted similarly.

As a dual of Algorithm 4, Algorithm 5 (ComputeWR- Co-Büchi), takes a co-büchi game as an input, and computes a winning region from where Player C has a strategy to eventually visit X always.

We can now state (see our extended version [24] for proof):

Theorem 2: Whenever Algorithms 2, 3, 4, and 5 terminate, they compute the exact winning region for Player C in safety, reachability, Büchi, and co-Büchi games, respectively. For FND games, upon termination, they also output a winning strategy automaton for Player C for this region. Furthermore, for safety games this strategy is maximally permissive. □

VII. Deterministic LTL Games

In this section we discuss how to solve a game with an LTL condition ψ which is not simple, but is nevertheless *deterministic* in that either $\mathcal{A}_\psi$ or $\mathcal{A}_{\neg\psi}$ is deterministic. We begin with the case when $\mathcal{A}_\psi$ is deterministic.

Let $\mathcal{G} = (V, D, Con, Env, \psi)$ be an LTL game, and let $\mathcal{A}_\psi = (Q, \{q_0\}, \mathcal{T}, F)$ be a deterministic and complete Büchi automaton for ψ over the set of variables V. We define the *product game* corresponding to $\mathcal{G}$ and $\mathcal{A}_\psi$ to be

$$\mathcal{G} \otimes \mathcal{A}_\psi = (V \cup \{q\}, D, Con', Env', \psi') \quad \text{where}$$

- q is a new variable representing the state of the automaton such that $D(q) = \{1, 2, \cdots |Q|\}$
- $Con' = Con \wedge \bigvee_{(p,\delta,p') \in \mathcal{T}} (q = p \wedge \delta \wedge q' = p')$.
- $Env' = Env \wedge \bigvee_{(p,\delta,p') \in \mathcal{T}} (q = p \wedge \delta \wedge q' = p')$.
- $\psi' = GF(\bigvee_{p \in F} q = p)$.

Similarly, for the case when $\mathcal{A}_{\neg\psi}$ is a deterministic and complete Büchi automaton for $\neg\psi$, we define the *product game* corresponding to $\mathcal{G}$ and $\mathcal{A}_{\neg\psi}$ to be

$$\mathcal{G} \otimes \mathcal{A}_{\neg\psi} = (V \cup \{q\}, D, Con', Env', \psi') \quad \text{where}$$

- $\psi' = FG(\bigvee_{p \notin F} q = p)$.

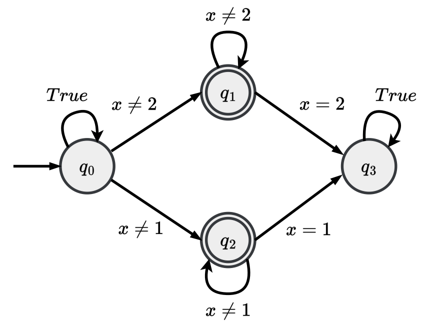


Fig. 4: Universal Co-Büchi automaton $A_{\neg\psi}$ for the specification $\psi := G(F(x = 1) \wedge F(x = 2))$

The definitions of Con' and Env' remain the same as that of the product game $\mathcal{G} \otimes \mathcal{A}_\psi$. For the product game $\mathcal{G} \otimes \mathcal{A}_{\neg\psi}$, in order to satisfy the specification ψ, we need to visit the final states of $\mathcal{A}_{\neg\psi}$ finitely often. This is equivalent to visiting the non-final states eventually always, as the definition of ψ' states. We note that if $\mathcal{G}$ is finitely non-deterministic, so is $\mathcal{G} \otimes \mathcal{A}_\psi$ and $\mathcal{G} \otimes \mathcal{A}_{\neg\psi}$.

Theorem 3: Let $\mathcal{G}$, with $\mathcal{A}_\psi$ (resp. $\mathcal{A}_{\neg\psi}$) deterministic, be as above. Let W' be the winning region for Player C in $\mathcal{G} \otimes \mathcal{A}_\psi$ (resp. $\mathcal{G} \otimes \mathcal{A}_{\neg\psi}$). Then the winning region for Player C in $\mathcal{G}$ is $W = \{s \mid (s, q_0) \in W'\}$. Furthermore, when $\mathcal{G}$ is finitely non-deterministic, given a finitely-representable memoryless strategy for C in $\mathcal{G} \otimes \mathcal{A}_\psi$ (resp. $\mathcal{G} \otimes \mathcal{A}_{\neg\psi}$), we can construct a finitely-representable finite-memory strategy for C in $\mathcal{G}$.

Proof: See our extended version [24]. ■

VIII. On The Fly Determinization Approach

When the automata A_ψ and $A_{\neg\psi}$ are non-deterministic, the product game $\mathcal{H}_\psi$ of the given game $\mathcal{G}$ with A_ψ and the product game $\mathcal{H}_{\neg\psi}$ of $\mathcal{G}$ with $A_{\neg\psi}$ both will be non-deterministic. It has been recognized in the literature [9] that non-deterministic automata need to be determinized to enable a precise winning region to be inferred.

A. Overview of determinization

We adopt the basic idea of k-safety determinization from the *Acacia* approach [10] for finite games and extend it to the setting of infinite games. We introduce our determinized product game construction intuitively below, and later formally in Sec. VIII-B. The underlying game graph $\mathcal{G}$ we use here for illustration is based on the Elevator game in Example 4.1. We simplify the example to admit just two controller moves, namely, $x' = 1$ and $x' = 2$, while the environment does not change the floor x in its moves. The given LTL property ψ is $G(F(x = 1) \wedge F(x = 2))$. Fig. 4 depicts the Universal Co-Büchi automaton $A_{\neg\psi}$ for this property, which happens to be non-deterministic.

The approach takes as parameter an integer $k \geq 0$, and generates a determinized version of the product of the game $\mathcal{G}$ and the automaton $A_{\neg\psi}$. A portion of the (infinite) determinized product for our example under consideration is depicted in

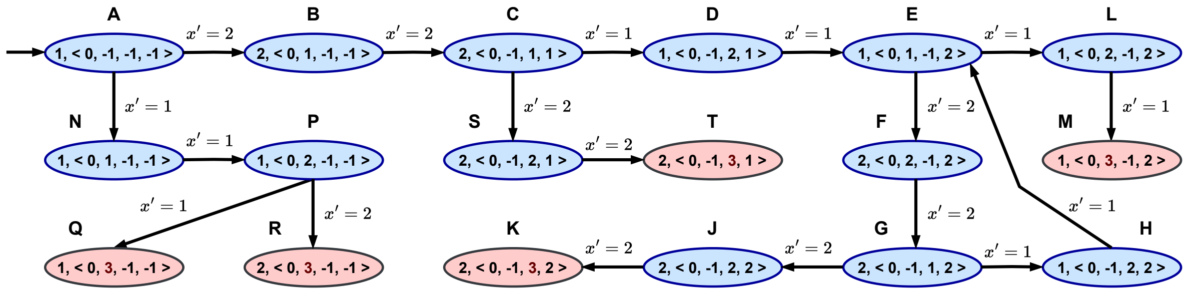


Fig. 5: Determinized 2-safety game for $A_{\neg\psi}$ where $\psi := G(F(x = 1) \wedge F(x = 2))$

Fig. 5, for $k = 2$. Each state of the determinized product is a pair of the form (s, v), where s a state of the underlying game $\mathcal{G}$ (i.e., a value of x in the example), and v is a vector of counts (the vectors are depicted within angle brackets). Each vector intuitively represents the subset of automaton states that the game could be in currently, with $v(i) > -1$ indicating that the automaton state $q_i \in Q$ belongs to the subset. If $v(i) > -1$, the value $v(i)$ further indicates the count of the maximum number of final states that can be visited along plays in the underlying game graph that reach automaton state q_i and that correspond to plays of the product graph that reach the current state (s, v). The moves of the two players in the product graph are alternating. For conciseness, we avoid showing the environment states in square., which do not make any updates to the game state. The initial states of the product graph are the ones whose vector component is $c_0 = \langle 0, -1, -1, -1 \rangle$, which represents the *initial* subset $\{q_0\}$. One of the initial states of the product graph is depicted in Fig. 5 (there are an infinite number of them, corresponding to all possible values of the game variable x).

We pick state E in Fig. 5 to illustrate the subset construction. q_2 is not present in E because from none of the automaton states that are present in the subset in product state D (i.e., q_0, q_2 or q_3), transitions to q_2 are possible as per the automaton in Fig. 4, when the value of x is 1 (as x has value 1 in product state D). And q_3 has a count of 2 in E because q_2 had count of 2 in state D and a q_2 to q_3 transition is possible when x has value 1 as per Fig. 4.

The product game shown in Fig. 5 can be seen to deterministic. This means that if a product state (s, v) has two successors (s_1, v_1) and (s_2, v_2), then $s_1 \neq s_2$. *Safe* states of the product graph are ones where no element of the vector exceeds k. Successor states of unsafe states will themselves be unsafe. In the figure unsafe states are indicated in red color (and have entries greater than 2 in the vectors).

A unique product game graph exists as per the construction mentioned above for a given value of k. This product game graph is said to be *winning* for Player C if satisfies the following conditions: (I) At least one of the safe product states has $c_0 = \langle 0, -1, \ldots, -1 \rangle$ as its vector, (II) For every safe product state from which the controller moves, at least one of the successors is a safe state, and (III) for every safe product state from which the environment moves, none of the successors are unsafe states. Otherwise, higher values of k will need to be tried, as indicated in the loop in Lines 16-19 in Algorithm 1. The product game graph in Fig. 5 happens to be winning.

If a product game graph is winning, then the game state components s of the product states of the form (s, c_0) in the product graph, where $c_0 = \langle 0, -1, \ldots, -1 \rangle$, constitute, in general, an under-approximation of the winning region $winreg_C(\mathcal{G})$. The under-approximation in general increases in size as the value of k increases. Note, in the loop we also compute a strategy for Player E by constructing a determinized product using A_ψ. Using this it can be detected when the current value of k yields the precise region $winreg_C(\mathcal{G})$. (If the underlying game is finite, such a k is guaranteed to exist.)

B. Formal presentation of deterministic product construction

We present here our SMT-based fixpoint computation for computing the product game graph of the kind introduced above, for a given bound k. We use formulas to represent (finite or infinite) portions of product graphs symbolically. The free variables in any formula are underlying game variables V and a vector-typed variable c. The solution to a formula is a (finite or infinite) set of states of a product graph.

$Aut(P, V, Q)$ is a given formula that encodes the logical transition relation $\mathcal{T}$ of the Büchi automaton $A_{\neg\psi}$. A triple (q, s, q') is a solution to $Aut(P, V, Q)$ iff $(q, s, q') \in \Delta_{\mathcal{T}}$. For instance, for the automaton in Fig. 4, $Aut(P, V, Q)$ would be $(P = q_0 \wedge Q = q_0) \vee (P = q_0 \wedge Q = q_1 \wedge x \neq 2) \vee \cdots$. $\mathit{final}(P)$ is a given formula that evaluates to 1 if P is a final state in the automaton $A_{\neg\psi}$ and otherwise evaluates to 0.

We define a formula $Succ_k(c, V, c')$ as follows:

$$\begin{aligned} \forall q.\, c'(q) = max\{&min(c(p) + \mathit{final}(q), k+1) \mid \\ &p \in Q, Aut(p, V, q), c(p) \geq 0\}, \\ &\text{if } \exists p \text{ such that } Aut(p, V, q) \wedge c(p) \geq 0 \\ = -1&, \text{ otherwise.} \end{aligned}$$

Intuitively, a triple (v, s, v') is a solution to $Succ_k(c, V, c')$ iff the product state (s', v') is a successor of the product state (s, v) for some s'.

Our approach is to use an iterative shrinking fixpoint computation to compute the *greatest fixpoint* (GFP) W of the function CP defined below.

$$\begin{aligned} CP_k(X) \;\equiv\; & G(V,c) \wedge \\ & \exists V', c'.\ (Con(V,V') \wedge Succ_k(c,V,c') \wedge G(V',c') \\ & \wedge \forall V'', c''.\ ((Env(V',V'') \wedge Succ_k(c',V,c'')) \\ & \implies X(V'',c''))). \end{aligned}$$

The argument of and the return value from the function above are both formulas in V, c, representing sets of product graph states. $G(V,c)$ represents safe product states, and checks that all elements of c are $\geq$ -1 and $\leq k$. The fixpoint computation is not guaranteed to terminate due to the infiniteness in the underlying game graph $\mathcal{G}$. If it does terminate, the formula W, after replacing the free variable c with the initial vector $c_0 = \langle 0, -1, \ldots, -1 \rangle$, is returned. This formula will have solutions iff the value of k considered was sufficient to identify a non-empty under-approximation of $winreg_C(\mathcal{G})$. The formula's solution is guaranteed to represent the *maximal* winning product graph that exists (and hence the maximal subset of $winreg_C(\mathcal{G})$) for the given value of k.

If W has solutions, we infer a strategy σ for Player C as follows. The following utility function σ_{prod} returns a formula in free variables V, whose solutions are the next game states to transition to when at a product state (s_1, c_1) in order to ensure a winning play.

$$\sigma_{prod}(s_1, c_1) \;=\; \exists c_2.\, Con(s_1, V) \wedge Succ_k(c_1, s_1, c_2) \wedge W(V, c_2)$$

We introduce a utility function $DestPair$, whose argument is a play in the underlying game $\mathcal{G}$, and that returns the product state in the determinized product graph reached by the play.

$$\begin{aligned} DestPair(s) &= (s, c_0) \\ DestPair(w \cdot s) &= (s, c),\ \text{such that} \\ & (DestPair(w) = (s', c') \wedge Succ_k(c', s', c) \end{aligned}$$

Finally, the strategy in terms of the underlying game $\mathcal{G}$ is defined as follows (where w is a play in the underlying game):

$$\sigma(w) \;=\; \sigma_{prod}(DestPair(w)).$$

IX. Implementation

We implement all fixpoint approaches in our protoype tool GenSys-LTL which extends our earlier tool GenSys [25] to support general LTL specifications. GenSys-LTL is implemented using Python and uses the Z3 theorem prover [26] from Microsoft Research as the constraint solver under the hood. GenSys-LTL uses Z3 to eliminate quantifiers from formulas resulting from the fixpoint iterations and check satisfiability. In all fixpoint approaches mentioned in this paper, large formulas are generated in every iteration, containing nested quantifiers. This formula blowup can quickly cause a bottleneck affecting scalability. The reason is that Z3 chokes over large formulas involving nested quantifiers. Thus, it is necessary to eliminate quantifiers at every step. We use quantifier elimination tactics inherent in Z3 to solve this issue. We use variations of [27] and simplification tactics in parallel, to achieve efficient quantifier elimination.

To convert a given LTL formula into an equivalent Buchi automaton, we use the Spot library [28] which efficiently returns a complete and state based accepting automaton for a given LTL specification. We also constrain Spot to return a deterministic Buchi automaton whenever possible, and then choose our approach appropriately. However, in this prototype version of GenSys-LTL , this encoding is done manually. GenSys-LTL is available as an open source tool on GitHub[1].

X. Evaluation

To evaluate GenSys-LTL we collect from the literature a corpus of benchmarks (and corresponding temporal specifications) that deal with the synthesis of strategies for two-player infinite-state logical LTL games. The first set of benchmarks were used in the evaluation of the *ConSynth* [11] approach. These target program repair applications, program synchronization and synthesis scenarios for single and multi-threaded programs, and variations of the Cinderella-Stepmother game [29], [30], which is considered to be a challenging program for automated synthesis tasks. The second set of benchmarks were used to evaluate the *Raboniel* [15] approach, which contains elevator, sorting, and cyber-physical examples, and specification complexity ranging from simple LTL games to ones that need products with Büchi automata. The third set of benchmarks are from *DTSynth* approach evaluation [16], and involve safety properties on robot motion planning over an infinite grid.

We compare our tool GenSys-LTL against two comparable tools from the literature: ConSynth [11] and Raboniel [15]. We do not compare against tools such as DTSynth [16] that only handle safety (not general LTL) specifications. We executed GenSys-LTL and Raboniel on our benchmarks on a desktop computer having six Intel i7-8700 cores at 3.20GHz each and 64 GB RAM. We were able to obtain a binary for ConSynth from other authors [16], but were unable to run it due to incompatibilities with numerous versions of Z3 that we tried with it. Hence, for the benchmarks in our suite that previous papers [11], [16] had evaluated ConSynth on, we directly picked up results from those papers. There is another comparable synthesis tool we are aware of, *Temos* [17]. We were unable to install this tool successfully from their code available on their artifact and from their GitHub repository, due to numerous dependencies that we could not successfully resolve despite much effort.

Table I shows the experimental results of all our approaches in comparison with ConSynth and Raboniel, with a timeout of 15 minutes per benchmark. The first column depicts the name of the benchmark: each benchmark includes a logical game specification and a temporal property winning condition. Column **Type** indicates whether the game variables in the underlying game $\mathcal{G}$ are reals or integers. Column **P** indicates the player (C or E) for which we are synthesizing a winning

[1] https://github.com/stanlysamuel/gensys/tree/gensys-ltl

region. Column **S** indicates whether the given benchmark falls in the Simple LTL category (G, F, FG, GF), or whether it needs an automaton to be constructed from the LTL property (Gen). |V| is the number of game variables. Letting ψ denote the given temporal property, Column **DB?** indicates whether the automaton A_ψ is deterministic, while Column **DCB?** indicates whether the automaton $A_{\neg\psi}$ is deterministic. In both these columns, the numbers within brackets indicate the number of automaton states.

The remainder of this section summarizes our results for the two main problems we address in this paper, namely, winning region computation, and realizability (see Section IV).

A. Winning region computation

Columns **G-S** to **OTF** in Table I indicate the running times of different variants of our approach, in seconds, for winning region computation (i.e., when an initial set of states is not given). The variant **G-S** is applicable when the given game is a simple game, and it involves no automaton construction or product-game formation (see Section VI). Variant **GF-P** (resp. **FG-P**), involves product constructions with property automata, and is applicable either when the given game is simple or when A_ψ (resp. $A_{\neg\psi}$) is deterministic (see Section VII). The **OTF** variant (see Section VIII) is applicable in all cases, as it is the most general. Any entry **T/O** in the table denotes a timeout, of 15 minutes while "N/A" indicates not-applicable.

We observe that when the game is simple, computing the winning region is fastest using simple game fixpoint approaches (Variant G-S). If both automatons are deterministic, then the FG-P computation is faster than the GF-P computation. This is because the former does not require a nested loop, as compared to the latter. The OTF approach is slower than the other approaches in most of the cases due to the cost of determinization, but is the only approach that was applicable in one of the benchmarks (Cinderella $C = 1.4$ with a non-simple temporal property). OTF took 7.7 seconds in this case, and returned a non-empty under-approximation of the winning region. The k parameter value given to OTF is indicated in Column **K**.

Our approach is very efficient as per our evaluations. Only on one of the benchmarks did none of the variants terminate within the timeout (Repair-Critical, with non-simple temporal property). On each of the remaining benchmarks, at least one variant of our approach terminated within 43 seconds at most.

The other approaches ConSynth and Raboniel are only applicable when an initial set of states is given, and not for general winning region computation.

B. Realizability

Recall that in this problem, a set of *initial states* is given in addition to the temporal property, with the aim being to check if the chosen player wins from every state in this set. The last three columns in Table I pertain to this discussion. Column **G** indicates the running time of the *most suitable* variant of our approach for the corresponding benchmark; what we mean by this, Variant **G-S** whenever it is applicable, else **FG-P** if it is applicable, else **GF-P** if it is applicable, otherwise **OTF**.

Column **C** indicates ConSynth's running times, for the benchmarks for which results were available in other papers. The rows where we show ConSynth's results in red color are ones where we are unsure of its soundness; this is because ConSynth does not determinize non-deterministic automatons, whereas in the literature it has been recognized that in general determinization is required for synthesis [9].

Column **R** indicates Raboniel's running times, obtained from our own runs of their tool. We were not able to encode three benchmarks into Raboniel's system due to the higher complexity of manually encoding these benchmarks; in these cases we have indicated dashes in the corresponding rows.

It is observable that our approach is much more efficient than the two baseline approaches. We terminate within the given timeout on one all but one benchmark, whereas ConSynth times out on three benchmarks and Raboniel on eight. Considering the benchmarks where both our approach and Raboniel terminate, our approach is **46x** faster on average (arithmetic mean of speedups). Considering the benchmarks where both our approach and ConSynth terminate, our approach is **244x** faster on average.

A case-by-case analysis reveals that we scale in the challenging Cinderella case where the bucket size C is $1.9(20)$ (i.e., 9 repeated 20 times). We also scale gracefully in the simple elevator examples (Simple-3 to Simple-10), as the number of floors increases from 3 to 10, as compared to Raboniel. We solve the Cinderella benchmark for $C = 1.4$ with the general LTL specification in 301 seconds (using OTF, with $k = 1$), which is another challenging case. Raboniel times out for this case.

A detailed list of the specifications used is given in our extended version [24].

C. Discussion on non-termination

There do exist specifications where GENSYS-LTL will not terminate. We share this issue in common with Raboniel. Consider the game specification: $V = \{x\}, Con(x, x') := x' == x - 1 \vee x' = x + 1, Env(x, x') := x' == x, Init(x) := x \geq 0, \psi(x) := F(x < 0)$. This example is realizable. However, GENSYS-LTL will not terminate as it will keep generating predicates $x \leq 1$, $x \leq 2$, $x \leq 3$, and so on, which can never cover the initial region $x \geq 0$.

XI. Related Work

Explicit-state techniques for finite-state games. This line of work goes back to Büchi and Landweber [5], who essentially studied finite-state games with a Büchi winning condition, and showed that a player always has a finite-memory strategy (if she has one at all). Games with LTL winning conditions, where the players play symbols from an input/output alphabet respectively, were first studied by Pnueli and Rosner [31] who showed the realizabilty question was decidable in double exponential time in the size of the LTL specification. A recent line of work [8], [9] proposes a practically efficient solution to

TABLE I: Comparison of all approaches of GenSys-LTL with ConSynth and Raboniel. Times are in seconds. **T/O** denotes a timeout after 15 minutes. Abbreviations: **P** for Player, **S** for Specification, **DB** for Deterministic Büchi, **DCB** for Deterministic Co-Büchi, **G-S** for GenSys-Simple Game, **GF-P** for Product Büchi Game, **FG-P** for Product Co-Büchi Game, **OTF** for On-The-Fly approach, **K** for OTF bound for which solution was found, **C** for ConSynth, **R** for Raboniel, **G** for GenSys-LTL

Game	**Type**	**P**	**S**	**|V|**	**DB?(|Q|)**	**DCB?(|Q|)**	**G-S**	**GF-P**	**FG-P**	**OTF**	**K**	**C**	**R**	**G**
Cinderella ($C = 2$)	Real	C	G	5	Y (2)	Y (2)	0.4	2.4	0.8	0.7	0	**T/O**	**T/O**	0.4
Cinderella ($C = 3$)	Real	C	G	5	Y (2)	Y (2)	0.3	2.8	0.7	0.7	0	765.3	**T/O**	0.3
Repair-Lock	Int	C	G	3	Y (2)	Y (2)	0.3	1.0	0.4	0.4	0	2.5	3.1	0.3
Repair-Critical	Int	C	G	8	Y (2)	Y (2)	29.0	666.0	29.5	123.0	0	19.5	-	29.0
Synth-Synchronization	Int	C	G	7	Y (2)	Y (2)	0.3	0.6	0.3	0.4	0	10.0	-	0.3
Cinderella ($C = 1.4$)	Real	E	F	5	Y (2)	Y (2)	0.3	1.0	0.3	2.7	3	18.0	**T/O**	0.3
Cinderella ($C = 1.4$)	Real	C	GF	5	Y (2)	**N (3)**	43.0	130.0	N/A	101.0	1	436.0	**T/O**	43.0
Cinderella ($C = 1.4$)	Real	C	Gen	5	**N (7)**	**N (5)**	N/A	N/A	N/A	7.7	0	4.7	**T/O**	301.0
Cinderella ($C = 1.9(20)$)	Real	C	G	5	Y (2)	Y (2)	42.0	**T/O**	**T/O**	**T/O**	**T/O**	-	**T/O**	42.0
Repair-Critical	Int	C	Gen	8	Y(40)	**N (6)**	N/A	**T/O**	N/A	**T/O**	**T/O**	53.3	-	**T/O**
Simple-3	Int	C	Gen	1	Y (5)	**N (6)**	N/A	3.3	N/A	309.0	6	-	1.8	3.3
Simple-4	Int	C	Gen	1	Y (6)	**N (7)**	N/A	4.1	N/A	**T/O**	**T/O**	-	2.2	4.1
Simple-5	Int	C	Gen	1	Y (7)	**N (8)**	N/A	5.8	N/A	**T/O**	**T/O**	-	5.1	5.8
Simple-8	Int	C	Gen	1	Y(10)	**N(11)**	N/A	15.6	N/A	**T/O**	**T/O**	-	27.4	15.6
Simple-10	Int	C	Gen	1	Y(12)	**N(13)**	N/A	30.3	N/A	**T/O**	**T/O**	-	108.0	30.3
Watertank-safety	Real	C	G	2	Y (2)	Y (2)	0.3	0.6	0.3	0.3	0	-	19.4	0.3
Watertank-liveness	Real	C	Gen	1	Y (3)	**N (4)**	N/A	2.5	N/A	0.7	0	-	51.0	2.5
Sort-3	Int	C	FG	3	Y (2)	N/A	1.2	1.1	N/A	0.3	0	-	51.0	1.1
Sort-4	Int	C	FG	4	Y (2)	N/A	2.2	1.2	N/A	0.4	0	-	650.1	1.2
Sort-5	Int	C	FG	5	Y (2)	N/A	5.2	1.2	N/A	0.4	0	-	**T/O**	1.2
Box	Int	C	G	2	Y (2)	Y (2)	0.3	0.6	0.3	0.4	0	3.7	1.2	0.3
Box Limited	Int	C	G	2	Y (2)	Y (2)	0.2	0.6	0.3	0.3	0	0.4	0.3	0.2
Diagonal	Int	C	G	2	Y (2)	Y (2)	0.2	0.6	0.2	0.4	0	1.9	6.4	0.2
Evasion	Int	C	G	4	Y (2)	Y (2)	0.7	1.8	0.8	0.6	0	1.5	3.4	0.7
Follow	Int	C	G	4	Y (2)	Y (2)	0.7	1.8	0.8	0.6	0	**T/O**	94.0	0.7
Solitary Box	Int	C	G	2	Y (2)	Y (2)	0.3	0.5	0.2	0.4	0	0.4	0.3	0.3
Square 5 * 5	Int	C	G	2	Y (2)	Y (2)	0.3	0.6	0.3	0.4	0	**T/O**	**T/O**	0.3

these games, which avoids the expensive determinization step, based on Universal Co-Büchi Tree Automata. [10] extend this direction by reducing the problem to solving a series of safety games, based on a k-safety automaton (k-UCW) obtained from a Universal Co-Büchi Word (UCW) automaton.

Symbolic fixpoint techniques. One of the first works to propose a symbolic representation of the state space in fixpoint approaches was [32] in the setting of discrete-event systems. More recently Spectra [33] uses BDDs to represent states symbolically in finite-state safety games. For infinite-state systems, [1] proposes a logical representation of fixpoint algorithms for boolean and timed-automata based systems, while [34] characterizes classes of arenas for which fixpoint algorithms for safety/reachability/Buchi games terminate. An earlier version of our tool called GenSys [25] uses a symbolic fixpoint approach, but is restricted to safety games only. In contrast to all these works, we target the general class of LTL games. A recent preprint [35] uses acceleration-based techniques to alleviate divergence issues in the fixpoint-based game solving approach. Their approach can terminate in certain cases where our approach does not terminate, such as the one explained in Sec. X-C. However their technique does not attempt to compute the exact winning region.

Symbolic CEGAR approaches. [36] considers infinite-state LTL games and proposes a CEGAR-based approach for realizability and synthesis. Several recent works consider games specified in Temporal Stream Logic (TSL). [13] considers uninterpreted functions and predicates and convert the game to a bounded LTL synthesis problem, refining by adding new constraints to rule out spurious environment strategies. [14], [15], [17] consider TSL modulo theories specifications and give techniques based on converting to an LTL synthesis problem, using EUF and Presburger logic, and Sygus based techniques, respectively. In contrast to our techniques, these techniques are not guaranteed to compute precise winning regions or to synthesize maximally permissive controllers.

Symbolic deductive approaches. In [11] Beyene et al propose a constraint-based approach for solving logical LTL games, which encodes a strategy as a solution to a system of extended Horn Constraints. The work relies on user-given templates for the unknown relations. [12] considers reachability games, and tries to find a strategy by first finding one on a finite unrolling of the program and then generalizing it. [16], [18], [37] consider safety games, and try to find strategies using forall-exists solvers, a decision-tree based learning technique, and enumerative search using a solver, respectively. In contrast, our work aims for precise winning regions for general LTL games.

XII. Conclusion

In this paper we have shown that symbolic fixpoint techniques are effective in solving logical games with general LTL specifications. Going forward, one of the extensions we would like to look at is strategy extraction for general (non-FND) games. Here one could use tools like AE-Val [38] that synthesize valid Skolem functions for forall-exists formulas.

A theoretical question that appears to be open is whether the class of games we consider (with real domains in general) are *determined* (in that one of the players always has a winning strategy from a given starting state).

Acknowledgment: The authors would like to thank Rayna Dimitrova and Philippe Heim for their comments on a preprint of our paper, which helped us improve key parts of our paper.

References

[1] E. Asarin, O. Maler, and A. Pnueli, “Symbolic controller synthesis for discrete and timed systems,” in *Proc. Third Intl. Workshop on Hybrid Systems, Ithaca, USA, 1994*, ser. LNCS, vol. 999. Springer, 1994, pp. 1–20.

[2] B. Jobstmann, A. Griesmayer, and R. Bloem, “Program repair as a game,” in *Proc. Intl. Conf. Computer Aided Verification (CAV 2005) Edinburgh, 2005*, ser. LNCS, vol. 3576. Springer, 2005, pp. 226–238.

[3] M. T. Vechev, E. Yahav, and G. Yorsh, “Abstraction-guided synthesis of synchronization,” in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2010), Madrid, 2010*. ACM, 2010, pp. 327–338.

[4] H. Zhu, Z. Xiong, S. Magill, and S. Jagannathan, “An inductive synthesis framework for verifiable reinforcement learning,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2019), Phoenix, USA, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 686–701.

[5] J. R. Büchi and L. H. Landweber, “Solving sequential conditions by finite-state strategies,” *Trans. American Mathematical Society*, vol. 138, pp. 295–311, 1969.

[6] W. Thomas, “On the synthesis of strategies in infinite games,” in *Proc. Symp. Theoretical Aspects of Computer Science (STACS 1995), Munich, Germany, 1995*, ser. LNCS, vol. 900. Springer, 1995, pp. 1–13.

[7] O. Maler, A. Pnueli, and J. Sifakis, “On the synthesis of discrete controllers for timed systems (an extended abstract),” in *Proc. Symp. Theoretical Aspects of Computer Science (STACS 1995), Munich, Germany, 1995*, ser. LNCS, vol. 900. Springer, 1995, pp. 229–242.

[8] O. Kupferman and M. Y. Vardi, “Safraless decision procedures,” in *Proc. IEEE Symp. Foundations of Computer Science (FOCS 2005), 2005, Pittsburgh, USA*. IEEE Computer Society, 2005, pp. 531–542.

[9] R. Bloem, K. Chatterjee, and B. Jobstmann, “Graph games and reactive synthesis,” *Handbook of Model Checking*, pp. 921–962, 2018.

[10] E. Filiot, N. Jin, and J.-F. Raskin, “Antichains and compositional algorithms for LTL synthesis,” *Formal Methods in System Design*, vol. 39, pp. 261–296, 2011.

[11] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko, “A constraint-based approach to solving games on infinite graphs,” in *Symp. on Principles of Programming Languages, POPL 2014, San Diego, USA, 2014*. ACM, 2014, pp. 221–234.

[12] A. Farzan and Z. Kincaid, “Strategy synthesis for linear arithmetic games,” *Proc. ACM Symp. Principles of Programming Languages (POPL 2016)*, vol. 2, pp. 61:1–61:30, 2018.

[13] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, “Temporal stream logic: Synthesis beyond the bools,” in *Proc. Intl. Conf. Computer Aided Verification (CAV 2019) , New York, USA, 2019*, ser. LNCS, vol. 11561. Springer, 2019, pp. 609–629.

[14] B. Finkbeiner, P. Heim, and N. Passing, “Temporal stream logic modulo theories,” in *Proc. Intl. Conf. Foundations of Software Science and Computation Structures (FOSSACS 2022), Munich, Germany, 2022*, ser. LNCS, vol. 13242. Springer, 2022, pp. 325–346.

[15] B. Maderbacher and R. Bloem, “Reactive synthesis modulo theories using abstraction refinement,” in *Proc. Formal Methods in Computer-Aided Design (FMCAD 2022)*, 2022, pp. 315–324.

[16] D. Neider and O. Markgraf, “Learning-based synthesis of safety controllers,” in *Proc. Formal Methods in Computer Aided Design (FMCAD 2019)*, 2019, pp. 120–128.

[17] W. Choi, B. Finkbeiner, R. Piskac, and M. Santolucito, “Can reactive synthesis and syntax-guided synthesis be friends?” in *Proc. ACM SIGPLAN Intl Conf. Programming Language Design and Implementation (PLDI 2022)*. New York, USA: ACM, 2022, p. 229–243.

[18] K. Wu, Y. Qiao, K. Chen, F. Rong, L. Fang, Z. Lai, Q. Dong, and L. Xiong, “Automatic synthesis of generalized winning strategy of impartial combinatorial games,” in *Proc. Intl. Conf. Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, 2020*. IFAAMS, 2020, pp. 2041–2043.

[19] A. Pnueli, “The temporal logic of programs,” in *Proc. Symp. Foundations of Computer Science (FOCS 1977), Providence, USA, 1977*. IEEE Computer Society, 1977, pp. 46–57.

[20] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[21] W. Thomas, “Automata on infinite objects,” in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier and MIT Press, 1990, pp. 133–191.

[22] J. R. Büchi, *On a Decision Method in Restricted Second Order Arithmetic*. New York, NY: Springer New York, 1990, pp. 425–435.

[23] P. Wolper, M. Y. Vardi, and A. P. Sistla, “Reasoning about infinite computation paths (extended abstract),” in *Proc. Symp. Foundations of Computer Science (FOCS 1983), Tucson, USA, 1983*. IEEE Computer Society, 1983, pp. 185–194.

[24] S. Samuel, D. D’Souza, and R. Komondoor, “Towards Efficient Controller Synthesis Techniques for Logical LTL Games,” 2023.

[25] ——, “GenSys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces,” in *Proc. ACM Joint European Software Engineering Conference and Symp. Foundations of Software Engineering (ESEC/FSE 2021) (Demo Track), Athens, Greece, 2021*. ACM, 2021, pp. 1585–1589.

[26] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Intl. conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Springer, 2008, pp. 337–340.

[27] N. Bjorner and M. Janota, “Playing with quantified satisfaction,” in *Proc. Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-20)*, ser. EPiC, vol. 35. EasyChair, 2015, pp. 15–27.

[28] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, “Spot 2.0—a framework for LTL and-automata manipulation,” in *Proc. Intl. Symp. Automated Technology for Verification and Analysis (ATVA 2016), Chiba, Japan, 2016*. Springer, 2016, pp. 122–129.

[29] M. H. L. Bodlaender, C. A. J. Hurkens, V. J. J. Kusters, F. Staals, G. J. Woeginger, and H. Zantema, “Cinderella versus the wicked stepmother,” in *Proc. IFIP TC 1/WG 202 Intl. Conf. Theoretical Computer Science (TCS 2012)*. Berlin, Heidelberg: Springer, 2012, p. 57–71.

[30] A. Hurkens, C. Hurkens, and G. Woeginger, “How cinderella won the bucket game (and lived happily ever after),” *Mathematics Magazine*, vol. 84, pp. 278–283, 10 2011.

[31] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proc. ACM Symp, Principles of Programming Languages (POPL 1989), Austin, USA, 1989*. ACM Press, 1989, pp. 179–190.

[32] G. Hoffmann and H. Wong-Toi, “Symbolic synthesis of supervisory controllers,” in *Proc. American Control Conference (ACC 1992)*, 1992, pp. 2789–2793.

[33] S. Maoz and J. O. Ringert, “Spectra: a specification language for reactive systems,” *Software and Systems Modeling*, vol. 20, no. 5, pp. 1553–1586, 2021.

[34] L. de Alfaro, T. A. Henzinger, and R. Majumdar, “Symbolic algorithms for infinite-state games,” in *Proc. 12th Intl. Conf. Concurrency Theory (CONCUR 2001), Aalborg, Denmark, 2001*, ser. LNCS, vol. 2154. Springer, 2001, pp. 536–550.

[35] P. Heim and R. Dimitrova, “Solving Infinite-State Games via Acceleration,” 2023.

[36] T. A. Henzinger, R. Jhala, and R. Majumdar, “Counterexample-guided control,” in *Proc. Intl. Coll. Automata, Languages and Programming, (ICALP 2003), Eindhoven, The Netherlands, 2003*, ser. LNCS, vol. 2719. Springer, 2003, pp. 886–902.

[37] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen, “Validity-guided synthesis of reactive systems from assume-guarantee contracts,” in *Proc. Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018), Thessaloniki, Greece, 2018*, ser. LNCS, vol. 10806. Springer, 2018, pp. 176–193.

[38] G. Fedyukovich, A. Gurfinkel, and A. Gupta, “Lazy but effective functional synthesis,” in *Proc. Intl. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2019), Cascais, Portugal, 2019*, ser. LNCS, vol. 11388. Springer, 2019, pp. 92–113.