Games Programs Play: Analyzing Multiplayer Programs [Extended Version]

Eric Koskinen*[1], Hiroshi Unno[2], and Moshe Vardi[3]

[1] Yale University
[2] University of Tsukuba
[3] Rice University

Abstract. In this paper we present the first automatic technique for symbolically proving alternation-free μ-calculus properties of infinite-state, higher-order programs. In particular, we show how weak-recurrence solvability can be lifted from finite-state programs to higher-order recursive programs. Our strategy reduces the search for a proof of a winning strategy to nontermination of higher-order programs for safety games, termination for liveness games, and an iterated combination of the two—along with winning preconditions—for weak recurrence games. We can thus leverage existing reasoning techniques based on dependent refinement types to automatically generate abstractions. From the resulting types, one can construct the winning strategy. Our technique even allows one to solve games in which the state space consists of higher-order expressions and algebraic data-structures. We have implemented our technique in a prototype tool PGS, and discovered winning strategies for a variety of games, including safety games, liveness games, general recurrence games, and alternation-free μ-calculus verification games.

1 Introduction

Over the past 25 years the theory of games have become a fundamental tool in reasoning about programs. An obvious application of game-theoretic reasoning is in reactive-program synthesis, where the setting is naturally modeled as a game between a system and its adversarial environment [33]. Another application is in CTL model checking [9], where the truth of a formula can be modeled as a game between the existential and universal quantifiers [22]. As a result, the theory of games, and its connections with logic and automata is a major line of research in formal methods, cf. [20].

A fundamental result in this area is an efficient (linear-time) algorithm for solving infinite-duration games over *two-player weak finite game graphs*, or *weak recurrence games*, for short. A game graph is a directed graph with a notation

* Supported in part by NSF CCF Award #1421126 and, previously, by CMACS NSF Expeditions in Computing award #0926166 and the NYU Office of the Provost (UCRF).

showing which player moves at each node. The game is played by the two players, say, *Angel* and *Demon*, taking turns moving a pebble from a graph node to its neighbor along a directed edge. The winning condition for the infinite-duration game is specified as a set of recurrence nodes, which Angel is required to visit infinitely often during an infinite play, in which the players forever keep moving the pebble. Without further structure, this game is called a *recurrence* or *Büchi* game [27], which has numerous applications in formal methods [32]. Solving the game, that is, computing the set of nodes from which Angel has a winning strategy, is an algorithmic problem of continuing interest, cf. [32]. The best algorithm [7] for this problem runs in time $O(n^2)$, where n is the number of nodes in the graph. Whether a better algorithm exists, which runs in time $O(n \log n)$ or even in time $O(n)$ is a major open problem.

The *weakness* condition is an additional structure imposed on recurrence games, in which there is a decomposition of the game graph into components that are partially ordered, such that (1) each component is either *recurring*, that is, consists only of recurrence nodes or *nonrecurring*, that is, consists only of non-recurrence nodes, and (2) there is no edge from a node in a component to a node in a higher component. The intuition here is that every play necessarily descends down the component order until it stabilizes either in a recurring component, in which case Angel wins, or a nonrecurring component, in which case Demon wins. A key result in [27] is that, unlike recurrence games, weak recurrence games can be solved in linear time, i.e., $O(n)$. Furthermore, it is shown in [27] that this algorithm underlies many results in model checking, such as the linear-time solvability of CTL model checking [8] and the linear-time solvability of alternation-free μ-calculus model checking [17]. Currently, weak recurrence games constitute the largest fragment of recurrence games that is known to have linear-time solvability. Thus, the linear-time algorithm for solving weak recurrence games is one of the key result in the theory of model checking finite-state programs.

An independent line of research in formal methods is the development of semi-decision procedures for reasoning about general programs. A seminal result in this area was the development of SLAM [3], which lifted finite-state safety checking to general programs. SLAM reduces program safety to finite-state safety checking and quantifier-free theory reasoning via a process of iterative abstraction refinement. While the procedure may, in general, fail to converge, it works impressively well in practice and has been implemented in a deployed industrial tool [1, 2]. The existence of a practically effective tool for reasoning about program safety, has in turned enabled more sophisticated reasoning capabilities, including program termination [15], temporal reasoning [10, 12, 14, 13, 6, 11], and many more.

Our main aim of this paper is to achieve a similar lifting of weak-recurrence solvability from *finite* games to general *infinite-state* games. The motivation for studying general games stems from the observation that nondeterminism in programs often has two independent sources [23]. *Angelic* nondeterminism is a tool to increase expressiveness of programs by avoiding the specification

of some low-level details, enabling this nondeterminism to be resolved away at implementation time [16]. In contrast, *demonic* nondeterminism reflects uncertainty about the execution environment, for example, about the scheduling of concurrent threads. Reasoning about programs with both angelic and demonic nondeterminism requires regarding the program as a game between the program, Angel, and its environment, Demon. Thus, such reasoning must be game theoretic! (For a discussion on the need to distinguish between angelic and demonic nondeterminism in finite-state reasoning, cf. [28].)

As a motivating example, consider a Queue data-structure that is initially empty and two players: player Demon may `enqueue` one or two elements at a time, and player Angel may `dequeue` one or two elements. (Assume that dequeuing two elements from a queue containing one element gives you an empty queue with no error message). The property we are interested in here is for Angel always to be able to force the Queue to be empty. There is an obvious winning strategy for Angel: always dequeue two elements. The key observation is that Angel's nondeterminism has a different flavor than that of Demon. Angel uses nondeterministic choice to make sure that *empty* is always in reach, while Demon uses nondeterministic choice to drive the queue away from *empty*. Specifying this winning condition for Angel is outside of the realm of CTL, and requires alternation-free μ-calculus [35]. We do not know of any tools that are able to prove arbitrary alternation-free μ-calculus formulas for infinite-state programs.

The key technical result of the paper is showing how weak-recurrence solvability can be lifted from finite-state programs to higher-order, infinite-state programs. We work with two-player games described as procedures `angel` and `demon` that mutually invoke each other *ad infinitum*, passing the game state as arguments. We then describe a translation on these procedures to a new mutually recursive program encoding, whose non-termination entails safety and whose termination entails liveness. In this way, we can adapt prior higher-order program verification techniques based on dependent types to prove safety (via nontermination [21, 29]) and liveness (via termination [30]). Moreover, when there isn't a winning strategy for Angel from all initial states, we adapt these works to synthesize sufficient *winning preconditions*.

Next, we use these two reductions as building blocks for an the first automatic technique for solving weak recurrence games over infinite state spaces. Our algorithm traverses the weak order in a bottom-up manner, synthesizing winning preconditions of lower components and using them to find winning strategies in higher components. If the winning precondition of the initial component holds of the start state, then there is an overall winning strategy of the weak game.

There are numerous applications of this result. Well beyond safety and termination, our work is expressive enough to prove properties of programs from alternation-free fragment of the μ-calculus [19]. We have developed a prototype tool called PGS and applied our technique/tool to example safety games, liveness games, and weak recurrence games including some from AFMC verification. These games may have higher-order state spaces, algebraic data-types, etc. Our

result is the first tool for for alternation-free μ-calculus verification of programs over infinite state spaces.

2 Technical Background

2.1 Modal Mu-Calculus

The *modal μ-calculus* is an expressive logic in which one can express properties of transition systems, including all properties expressible in branching-time logics such as CTL* [24]. Formally, it is modal logic augmented with least and greatest fixpoint operators; specifically, we consider a μ-calculus where formulas are constructed from atomic formula using Boolean connectives, modalities $\Box$ and $\Diamond$ indexed over the players set (e.g., $\Box_D$ or $\Diamond_A$), and the least (μ) and greatest (ν) fixpoint operators. We assume that μ-calculus formulas are written in positive normal form (negation only applied to atomic propositions constants and variables). Most treatment of the μ-calculus focuses on its propositional version, but here we allow atomic formulas over program states. There is extensive literature about the modal μ-calculus; for a survey, see [31]. Formulas where all propositional variables are quantified with either μ or ν are called *sentences*.

Here we are interested in the *alternation-free* fragment of the μ-calculus (AFMC) in which syntactic nesting of least and greatest fixpoint operators is allowed, but semantic interaction is not allowed. For precise definitions see [18]. For example, the formula $ReachEmpty = \mu x.(empty \vee \Box_D \Diamond_A x))$ says that Angel can force its way to the empty queue. The formula $AlwaysReachEmpty = \nu y.(ReachEmpty \wedge \Box_D \Diamond_A y))$ says that Angel can always force staying in states from which forcing reachability of is possible. Intuitively, $AlwaysReachEmpty$ is the modal version of the CTL formula $\mathsf{AGEF}empty$, which takes into account the adversarial interaction of Angel and Demon. Note that $AlwaysReachEmpty$ allows μ to be nested in the scope of ν, but there is no semantic interaction, which makes the formula alternation free. In contrast, the formula $\nu x.\mu y.((p \wedge x) \vee \Diamond_A y)$ is not alternation free, since there is semantic interaction between the greatest and least fixpoint operators.

We interpret AFMC sentences over two-player transition systems, which have two transition relations, one for Angel and one for Demon, i.e., $T = (S, R_A, R_D)$, where S is a set of states and R_A and R_D are binary relations on S. The atomic formulas are interpreted over S; that is, every atomic formula can be viewed as a subset of S.

2.2 Weak Recurrence Games

A *game graph* is a directed graph $G = (V, E)$, with a partition $V = V_0 \uplus V_1$ of the node set, usually denoted as $G = (V_0, V_1, E)$. We assume that the graph has no sinks, so every node has some outgoing edges. An *infinite-duration* game is played on G *from a node* v by first placing a pebble p on v, and then moving p along edges in E. When p is placed on a node in V_0, Angel is making the move,

and when p is played on a node in V_1, Demon is making the move. The game continues forever; thus, a *play* is an infinite sequence $\pi = v_0, v_1, \ldots$ of nodes such that $\forall i \geq 0.\ (v_i, v_{i+1}) \in E$.

In a *recurrence* game, the winning condition for infinite plays is specified by means of a recurrence set $W \subseteq V$. An infinite play $\pi = v_0, v_1, \ldots$ is *winning* for Angel if the set $\{i \mid v_i \in W\}$ is infinite; otherwise it is winning for Demon. That is, Angel is required to visit W infinitely often, while Demon is trying to block such a recurrence.

A player *wins* the game if it has a winning strategy. A strategy for Angel (resp, Demon) is a function $\sigma : V^*V_0 \to V$ (resp., $\sigma : V^*V_1 \to V$) such that $\sigma(v_0 \ldots v) = v'$ only if $(v, v') \in E$. Intuitively, the strategy tells a player which edge to select, based in the full history of the game so far. Thus, a play $\pi = v_0, v_1, \ldots$ *complies* with strategies σ_0, σ_1 for Angel and Demon if $v_{i+1} = \sigma_k(v_0 \ldots v_i)$, when $v_i \in V_k$, for $k \in [0, 1]$. A strategy σ_0 is a *winning strategy* for Angel, if for every Demon strategy σ_1, every play π that complies with σ_0 and σ_1 is winning for Angel. For recurrence games it is known that it suffices to consider *memoryless* strategies, which depend only on the current state and not on the whole history.

Solving a game means deciding for a node v whether Angel has a winning strategy from v. For finite game graphs, this is an algorithmic question. As mentioned above, the best algorithm for finite recurrence games has time complexity of $O(|V|^2)$, and it is an open question whether there is an algorithm with better complexity. By restricting, however, the structure of the game, we can obtain better complexity.

In a *weak* game graph there is a partition of the node set into a finite partially ordered set of disjoint components. Thus, $V = P_1 \uplus \ldots \uplus P_m$, with a partial order $\leq$ defined on $\{P_1, \ldots, P_m\}$, such that if $(u, v) \in E$ with $u \in P_i$ and $v \in P_j$, then $P_j \leq P_i$. Intuitively, a move along an edge cannot ascend the component order. Thus, if $\pi = v_0, v_1, \ldots$ is an infinite play, then there is a component P_k and an index $l \geq 0$ such that $v_r \in P_k$ whenever $r \geq l$. Intuitively, every play weakly descends the component order until it stabilizes in some component. Because the partition is finite, this feature of weak games holds even when the game graph is infinite. Note that the partition $V = P_1 \uplus \ldots \uplus P_m$ is orthogonal to the partition $V = V_0 \uplus V_1$; that is, one can have that both $V_0 \cap P_j \neq \emptyset$ and $V_1 \cap P_j \neq \emptyset$ for $1 \leq j \leq m$.

In a *weak recurrence game* the recurrence set W has to be *compatible* with partition $V = P_1 \uplus \ldots \uplus P_m$. That is, either $P_j \subseteq W$ or $P_j \cap W = \emptyset$ for $1 \leq j \leq m$. Recall that every play in a weak game stabilizes in one of the components. Thus, from the players point of view, every component is winning or losing.

A key result of [27] is that finite weak recurrence games can be solved in linear time. The intuition is that the game can be solved one component at time, going up the component order. For a bottom component $P \subseteq W$, Angel's goal is to avoid sink states. Thus, for that component the game reduces to a safety game, which can be solved in a linear time using a careful fixpoint computation (e.g., as in [4]). For a bottom component $P \cap W = \emptyset$, Demon's game becomes a safety

game. Once the bottom components are solved, higher components can be solved using the same ideas, avoiding moves into losing states, and taking moves when possible into winning states.

Theorem 1. [27] *Finite Weak recurrence games can be solved in linear time.*

The rest of the paper deals with weak recurrence games over *infinite* game graphs.

2.3 Alternating Automata

Alternating automata play a key role in algorithmic program verification [34]. We are interested here in *symmetric alternating automata*, which can be viewed as a normal form for the μ-calculus [26]. A symmetric alternating automaton $A = (Q, \Sigma, q_0, \delta, \alpha)$, where Σ is a finite set of atomic *literals* (atomic formulas or their negation), Q is a finite state set, $q_0 \in Q$ is an initial state, δ is the transition function, and α is the acceptance condition.

Let $B^+(X)$ be the set of positive Boolean formulas over a set X. A set $Y \subseteq X$ satisfies a formula θ in $B^+(X)$ if θ is true when variables in Y are assigned **true**, and variables in $X \setminus Y$ are assigned **false**. We call Y a *satisfying subset*.

To define the transition function, we first define the *modal closure* of a set X denoted $modal(X)$, which consists of X itself, and, in addition, for each element $x \in X$ and modality M (e.g, $\Diamond_A$), also the expression Mx. The transition function is now defined as $\delta : Q \to B^+(modal(Q \cup \Sigma))$; that is, it associates a positive Boolean function over $modal(Q \cup \Sigma)$ with each automaton state q. We call $\delta(q)$ the *transition formula* for a state $q \in Q$.

Symmetric alternating automata run on a two-player transition system $T = (S, R_A, R_D)$. A *run* of A over T starting from t_0 is a tree τ labeled by elements of $S \times Q$, with the root labeled by (t_0, q_0). The rule for constructing the tree is as follows. Let x be a node of τ labeled by (s, q). Let $\theta = \delta(q)$ be the transition formula. Then there is a satisfying subset $P \subseteq modal(Q \cup \Sigma)$ such that for each element $p \in P$ we have:

- if p is a state q', then x has a child y labeled with (s, q')
- if p is a literal ℓ, then ℓ must be **true** in s, and x is a leaf.
- if p is the expression $\Diamond_A o$ (resp., $\Diamond_D o$), then there is a state $s' \in S$ such that $(s, s') \in R_A$ (resp., $(s, a') \in R_D$) and a child y of x labeled with (s', o). If o is a literal, then it must be satisfied by s'.
- If p is the expression $\Box_A o$ (resp., $\Box_D o$), then for each state $s' \in S$ such that $(s, s') \in R_A$ (resp., $(s, s') \in R_D$) there is a child y of x labeled with (s', o). If o is a literal, then it must be satisfied by s'.

It remains to define the acceptance condition α, which is defined as a condition on the infinite branches of the run. Such an infinite branch is labeled by an infinite word $(s_0, q_0), (s_1, q_1), \ldots$ over $S \times Q$. We define here acceptance by means of a recurrence set $W \subseteq Q$ (this is called a Büchi condition [27]), and we

say that the branch is accepting if W is visited infinitely often, and the run is accepting if all branches are accepting.

We focus attention here on *weak* automata, where there is a partially-ordered partition $Q_1 \uplus \ldots \uplus Q_m$ of Q that is compatible with W (so each Q_i is either contained in or disjoint from W), and such that if $q \in P_i$ and q' occurs in $\delta(q)$ with $q' \in P_j$, then $P_j \leq P_i$. We call the automata that meet this condition *symmetric weak alternating automata* [27, 25].

The key result is a translation from AFMC to symmetric weak alternating automata.

Theorem 2. [27, 25] *For each AFMC sentence φ, we can construct a symmetric weak alternating automaton A_φ such that φ is true at a state t of a transition system T iff A_φ has an accepting run on T from t. Furthermore, the size of A_φ is linear in the size of φ.*

2.4 Reduction to Games

In this section we remind the reader that the question of whether a state in two-player transition system satisfies an AFMC sentence can be reduced to solving weak recurrence games. We are given a transition system $T = (S, R_A, R_D)$ and an AFMC sentence A_φ. By Theorem 2, we can assume that we have the symmetric weak alternating automaton $A_\varphi = (Q, \Sigma, q_0, \delta, W)$.

We now define the product $G_{T,\varphi} = (V, E, W_G)$, which is a weak recurrence game, by taking the product of T and A_φ. Let $Form(A_\varphi)$ be the set of all subformulas of transition formulas in A_φ. Take $V = (S \times (Q \cup Form(A_\varphi)) \cup \{win, lose\}$. That is, the nodes of $G_{T,\varphi}$ are states of T labeled by either states of Q or subformulas of transition formulas. There are also two special states *win* and *lose*. We now define the edges E of $G_{T,\varphi}$, as well as the partition $V = V_0 \uplus V_1$.

- *win* and *lose* each have a single edge to itself. $win \in W_G$ and $lose \notin W_G$. Thus, if the play hits *win* then Angels wins, and if it hits *lose* Angel loses.
- If an edge from a node in V_0 is required (below) to go to (s, ℓ), where $\ell \in \Sigma$, then that edge goes instead to *win* if s satisfies ℓ and to *lose*, otherwise.
- V_0 consists of all nodes of the form (s, q), with $s \in S$ and $q \in Q$, as well of nodes of the form (s, θ), where θ is a disjunction or a $\Diamond$ expression.
- V_1 consists of all nodes of the form (s, q), with $s \in S$ and $q \in Q$, as well as nodes of the form (s, θ), where θ is a conjunction or a $\Box$ expression.
- Consider a node $(s, q) \in V_0$: there is a single edge from from that node to (s, θ), where $\theta = \delta(q)$.
- Consider a node $(s, \theta_1 \vee \theta_2) \in V_0$: there are edges from that node to the nodes (s, θ_1) and (s, θ_2).
- Consider a node $(s, \Diamond_A o) \in V_0$ (resp., $(s, \Diamond_D o) \in V_0$): there are edges to all nodes of the form (s', o) with $(s, s') \in R_A$ (resp., $(s, s') \in R_D$).
- Consider a node $(s, \theta_1 \wedge \theta_2) \in V_1$: there are edges from that node to the nodes (s, θ_1) and (s, θ_2).
- Consider a node $(s, \Box_A o) \in V_1$ (resp., $(s, \Box_D o) \in V_1$): there are edges to all nodes of the form $(s', 0)$ with $(s, s') \in R_A$ (resp., $(s, s') \in R_D$).

– $W_G = (S \times W) \cup \{win\}$.

It remains to define the weak structure of $G_{T,\varphi}$. We first define a partial order on $Q \cup Form(A_\varphi)$ We start with the partial order defined on Q by its weak structure, add to it the natural subformula order defined on $Form(A_\varphi)$, and add to it the rule that $\delta(q) \leq q$. We now define a partial order on $V = (S \times (Q \cup Form(A_\varphi)) \cup \{win, lose\}$. We put *win* and *lose* at the bottom of the order. Then we say that $(s, o) \leq (s', o')$ iff $o \leq o'$. Since $Q \cup Form(A_\varphi)$ is finite, this defines a weak structure on $G_{T,\varphi}$.

Theorem 3. [27, 25] *Consider a transition system T and an AFMC formula φ. Then a state s of T satisfies φ iff Angel has a winning strategy in $G_{T,\varphi}$ from the node (s, q_0).*

3 Symbolic infinite-state games

For the rest of this paper we will work with symbolic games. In particular, we describe a representation of an infinite two-player game as two higher-order procedures `angel` and `demon` that recursively invoke each other forever. When combined with a predicate $\mathcal{W}$ that indicates whether a given state is in the recurrence set, we will can then define the induced recurrence game.

The procedures `angel` and `demon` are given in continuation-passing style (CPS) so they do not return values but, rather, invoke each other, passing the state on the stack. We denote these procedures:

$$P = \{\texttt{angel}\ x_1\ ...\ x_n = e_{\texttt{angel}}\ ,\quad \texttt{demon}\ x_1\ ...\ x_n = e_{\texttt{demon}}\}$$

Expression $e_{\texttt{angel}}$ is the body of `angel` (resp. `demon`) and is given by

$$\begin{array}{rl} e ::= & x \mid c \mid \texttt{angel} \mid \texttt{demon} \mid \texttt{let}\ x\ = e'\ \texttt{in}\ e'' \mid x\, x' \\ \mid & x\ op\ x' \mid \texttt{if}\ x\ \texttt{then}\ e'\ \texttt{el}\ e'' \mid \lambda t.e' \mid \texttt{*} \end{array}$$

Expressions e are variables x, constants c, function names `angel` and `demon`, let expressions $\texttt{let}\ x = e'\ \texttt{in}\ e''$, function applications $x\, y$, constant operations $x\ op\ y$, conditional branches $\texttt{if}\ x\ \texttt{then}\ e'\ \texttt{el}\ e''$, lambda abstractions $\lambda t.e'$, and nondeterministic choice `*`. The constants include the boolean constants `true` and `false`, the unit constant (), and integer constants. The operator *op* include boolean and integer operators such as $+$, $-$, $\leq$. The expression `*` represents nondeterministic choice from a bounded domain. If `*` is contained within $e_{\texttt{angel}}$ then it is interpreted as *angelic nondeterminism* and if it is contained within $e_{\texttt{demon}}$ then it is interpreted as *demonic nondeterminism*. The body $e_{\texttt{angel}}$ may not call `angel` (resp. for $e_{\texttt{demon}}$ and `demon`).

We write $fv(e)$ for the free variables of e. For simplicity, we restrict branch conditions, operator arguments and the expressions in function applications to variables. But note that a non-variable form can be encoded by using `let`. For convenience, we use the non-variable forms when the encoding is clear from the context. As usual, applications associate to the left so that $e_0\, e_1\, e_2 = (e_0\, e_1)\, e_2$.

$$\frac{e_1 \rightsquigarrow e_2}{E[e_1] \rightsquigarrow E[e_2]}\ \textbf{E1} \qquad \frac{[\![op]\!](c_1, c_2) = c}{c_1\ op\ c_2 \rightsquigarrow c}\ \textbf{Op} \qquad \frac{}{(\lambda x.\ e)\ v \rightsquigarrow e[v/x]}\ \textbf{App}$$

$$\frac{\texttt{F}\ \bar{x} = e \in P}{\texttt{F}\ v \rightsquigarrow e[v/x]}\ \textbf{Fun} \qquad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \rightsquigarrow e[v/x]}\ \textbf{Let}$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{el}\ e_2 \rightsquigarrow e_1}\ \textbf{If1} \qquad \frac{}{\texttt{if false then}\ e_1\ \texttt{el}\ e_2 \rightsquigarrow e_2}\ \textbf{If2}$$

Fig. 1: Reduction semantics of mutually recursive `angel` and `demon`.

We write $e_1; e_2$ for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ such that $x \notin fv(e_2)$. Without loss of generality, we assume that bound variables are distinct. We say that an expression is closed if it contains no free variables.

The small-step reduction semantics are given in Figure 1. The semantics is fairly standard. Evaluation contexts and values are:

$$E ::= [] \mid v\ E \mid E\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{el}\ e_2 \mid \texttt{let}\ x = E\ \texttt{in}\ e$$
$$v ::= \texttt{F} \mid c \mid \lambda x.\ e$$

We assume that there are no terminating traces: `angel` will eventually call `demon` (resp. for `demon`). The code `angel` and `demon` induce a recurrence game as follows.

Definition 1 (Induced recurrence game). *For a program P, initial condition $\mathcal{I}$, and recurrence predicate $\mathcal{W}$, the induced game has state space given by evaluation contexts. Moreover, there is an edge from $(F_1\ v_1)$ to $(F_2\ v_2)$ if and only if, for some v such that $\mathcal{I}(v)$, `angel` $v \rightsquigarrow^* E_1[F_1\ v_1]$ and $E_1[F_1\ v_1] \rightsquigarrow^+ E_2[F_2\ v_2]$ without reaching an expression of the form $E[F\ v]$. The induced recurrence set is $\{F\ v \mid \mathcal{W}(v)\}$.*

For symbolic weak *games, we assume that each component is specified by a corresponding predicate $c_1, ..., c_n$. Each induced components C_i is $\{F\ v \mid c_i(v)\}$.*

4 Solving symbolic games

In this section we first describe how to solve safety and liveness games. We use these as building blocks to then give an algorithm for solving weak recurrence games. Finally, we implement the translation from Section **??** in our symbolic setting, reducing the question of whether a state in two-player transition system satisfies an AFMC sentence can be reduced to solving weak recurrence games

4.1 Solving Safety & Liveness Games

We describe a simple transformation $\mathcal{E}$ for a given recurrence program P and recurrence set $\mathcal{W}$ to reduce the search for a winning strategy to a program analysis problem. The key idea is to create a new recurrence between two methods

`enc_angel` and `enc_demon` that each first check whether recurrence set $\mathcal{W}$ has been reached and, if not, return `unit`. In this way, since `angel`/`demon` never return, the return of `unit` indicates an event of interest: reachability of $\mathcal{W}$. We first show how this can be used for safety games or liveness games.

Definition 2 (Encoding). *For $P = \{\texttt{angel}\ x_1\ ...\ x_n = e_{\texttt{angel}}, \texttt{demon}\ x_1\ ...\ x_n = e_{\texttt{demon}}\}$, initial condition $\mathcal{I}$, and formula φ, let*

$$\mathcal{E}(P,\mathcal{I},\varphi) = \begin{cases} \texttt{let rec enc\_angel } x_1\ ...\ x_n \texttt{ =} \\ \quad \texttt{if } \varphi(x_1,...,x_n) \texttt{ then () else } \hat{e}_{\texttt{angel}} \\ \texttt{and enc\_demon } x_1\ ...\ x_n \texttt{ =} \\ \quad \texttt{if } \varphi(x_1,...,x_n) \texttt{ then () else } \hat{e}_{\texttt{demon}} \\ \texttt{and main() =} \\ \quad \texttt{let } (x_1,...,x_n) \texttt{ = (*,...,*) in} \\ \quad \texttt{assume } \mathcal{I}(x_1,...,x_n)\texttt{; enc\_angel } x_1\ ...\ x_n \end{cases}$$

where $\hat{e}$ is a transformation on e that replaces calls to `angel` with calls to `enc_angel` (resp. for `demon`).

Theorem 4 (Safety Games). *For all $P, \mathcal{I}, \mathcal{W}$, if $\mathcal{E}(P,\mathcal{I},\mathcal{W})$* does not terminate*, then for all $s \in \mathcal{I}$ there exists a winning strategy for Angel to avoid $\mathcal{W}$.*

Proof. Assume not. Then $\mathcal{E}(P,\mathcal{I},\mathcal{W})$ always diverges and there is a $s_0 \in \mathcal{I}$ from which there is no winning strategy to avoid $\mathcal{W}$. If $\mathcal{E}(P,\mathcal{I},\mathcal{W})$ always diverges then $\mathcal{E}(P,\{s_0\},\mathcal{W})$ always diverges. $\mathcal{E}(P,\{s_0\},\mathcal{W})$ divergence must by via an infinite mutual recurrence between `enc_angel` and `enc_demon`. Since $\hat{e}_{\texttt{angel}}$ and $\hat{e}_{\texttt{demon}}$ are in CPS, the only way for $\mathcal{E}(P,\{s_0\},\mathcal{W})$ to terminate is when $\mathcal{W}$ holds and () is returned. Therefore all choices for angelic nondeterminism avoid $\mathcal{W}$. Contradiction.

Implementation. Our implementation begins by transforming `angel` and `demon` into `enc_angel` and `enc_demon`. The resulting encoding $\mathcal{E}(P,\mathcal{I},\mathcal{W})$ reduces safety games to proving non-termination. We therefore can prove non-termination of $\mathcal{E}(P,\mathcal{I},\mathcal{W})$ by leveraging a number of different existing techniques [21, 29, 30]. In our implementation, we use [21] which is able to infer refinement types automatically to prove that `main : unit` $\rightarrow$ **false**, that is, that `main` does not terminate.

Example 1. Consider the following recurrence (given as `safe_int3` in Figure 3) that involves angelic non-determinism. Here $\mathcal{W} \equiv \texttt{x} < 0$ and $\mathcal{I} \equiv \texttt{x} = 0$.

```
let rec angel x =
  let n = rand_int() in
  if n >= x then demon (x + 1) else demon x
and demon x = angel (x - 1)
and main() =
  let x = * in assume I(x); angel x
```

Our implementation infers invariants for `enc_angel` and `enc_demon` and treats `rand_int` as *angelic nondeterminism* as in [29, 21]. The refinement types inferred

for `rand_int` provide the conditions on the angelic nondeterminism (i.e. which choices) that are necessary to prove that $\mathcal{E}(P, \mathcal{I}, \mathcal{W})$ does not terminate. In this example, we infer the condition that $\texttt{rand\_int} : \texttt{unit} \to \{i : \texttt{int} \mid i \geq \texttt{x}\}$.

Theorem 5 (Liveness Games). *For all $P, \mathcal{I}, \mathcal{W}$, if $\mathcal{E}(P, \mathcal{I}, \mathcal{W})$ always* terminates*, then for all $s \in \mathcal{I}$, there exists a winning strategy for Angel to reach $\mathcal{W}$.*

Proof. Assume not. Then $\mathcal{E}$ always terminates and there exists some $s_0 \in \mathcal{I}$ from which there no winning strategy for Angel. However, $\mathcal{E}(P, \mathcal{I}, \mathcal{W})$ always terminates, so $\mathcal{E}(P, \{s_0\}, \mathcal{W})$ also always terminates. Pick a trace of $\mathcal{E}(P, \{s_0\}, \mathcal{W})$. Since Since `angel/demon` are given in CPS, it must be via the return of () in `enc_angel` or `enc_demon`. This case can only happen if $\mathcal{W}$ holds. Thus there is a strategy for Angel to reach $\mathcal{W}$. Contradiction.

Implementation. Our encoding $\mathcal{E}(P, \mathcal{I}, \mathcal{W})$ reduces liveness games to proving termination (total correctness). Proving non-termination of $\mathcal{E}(P, \mathcal{I}, \mathcal{W})$ can be done with a number of different existing techniques [21, 29, 30]. In our implementation, we use [30] which is able to the infer rank functions that witness the termination of `main`.

We denote by $\mathsf{PreC}_{\mathsf{safe}}[\mathcal{E}(P, \mathcal{I}, \mathcal{W})]$ the application of tools to solve safety games and $\mathsf{PreC}_{\mathsf{live}}[\mathcal{E}(P, \mathcal{I}, \mathcal{W})]$ the application of tools to solve liveness games. When Angel cannot win the game from all initial states, it still may be possible for Angel to win from *some* states.

Definition 3 (Winning Precondition). *For a given program P and formula φ, a* winning safety precondition *is a ψ such that $\mathsf{PreC}_{\mathsf{safe}}[\mathcal{E}(P, \varphi, \mathcal{W})]$ is* **true***, and similar for winning liveness precondition.*

Our implementation adapts prior work [21] to obtain winning preconditions. $\mathsf{PreC}_{\mathsf{safe}}$ will return the winning precondition for the safety game and $\mathsf{PreC}_{\mathsf{live}}$ will return the winning precondition for the liveness game. The technique in [21] was sufficient for the examples in the evaluation, but we believe a form of CEGAR may improve efficiency. We leave this to future work.

4.2 Solving Weak Games

Figure 2 gives our algorithm for solving weak recurrence games. We will use the techniques in the previous section as building blocks: $\mathsf{PreC}_{\mathsf{safe}}$ and $\mathsf{PreC}_{\mathsf{live}}$ and the winning safety/liveness preconditions that they generate. Recall that each component in a weak game is either entirely *fair* or entirely *unfair* and that there is a partial order $\preceq$ over these components.

The algorithm proceeds to compute the winning safety/liveness recondition for each component of the game in a bottom-up fashion. This mapping from ith component to its precondition ρ_i is represented as a function that is initially defined by initial_precond (Line 3) and then refined (Lines 13 and 16) each time a new component is considered. Initially, the function initial_precond maps base

```
 1 type pcmap = int → option formula
 2
 3 let initial_precond : pcmap = λ (i:int) .
 4    if (base(C_i) && fair(C_i)) then Some(true);
 5    elseif (base(C_i) && unfair(C_i)) then Some(false);
 6    else None
 7
 8 let rec solve (precond : pcmap) : pcmap =
 9    let i = choose({ i | ∀C_j. C_j ⪯ C_i ⇒ (precond j) ≠ None }) in
10    if i = [] then pcmap else
11    let J = successors of C_i in
12    if C_i is fair then
13      solve (λ k. if k = i
14              then Some(PreC_safe[E(P, true, ⋁_{j∈J}(¬(precond j) ∧ c_j)]) else precond k)
15    else
16      solve (λ k. if k = i
17              then Some(PreC_live[E(P, true, ⋁_{j∈J}((precond j) ∧ c_j)]) else precond k)
18
19 let main() =
20    match ((solve initial_precond) 1) with
21    | Some(φ) ⇒ if s_0 ⊨ φ then return "winning strategy for Angel"
22    | _ ⇒ return "unknown"
```

where $\mathsf{base}(C_i) \equiv \nexists C_j.C_j \preceq C_i$.

Fig. 2: Bottom-up algorithm for solving weak recurrence games. The algorithm iteratively computes the winning precondition (precond i) for each ith component. If the precondition of the first component C_1 holds of the initial state s_0, then there is a winning strategy for Angel.

components—those components that don't have any successors—to the formula **true** when the component is fair and **false** when it is unfair. The outer procedure main begins by passing this initial_precond to solve.

Each iteration of solve begins by choosing a component C_i such that all of C_i's successors have already been solved (Line 9). It then sets up a new game for this component, which may be a safety game $\mathsf{PreC}_{\mathsf{safe}}$ if C_i is fair, or else a liveness game $\mathsf{PreC}_{\mathsf{live}}$ if C_i is unfair. In either case, this game is comprised of a disjunction over all subcomponents J. Each disjunct has the jth subcomponent predicate c_j as well as its winning precondition precond j. This winning precondition is negated if C_i is fair in order to find a winning strategy that avoids the subcomponent. Notice that losing successor components (those components for which there does not exist a winning strategy for Angel) will have preconditions that are **false**. The result of PreC solving this game is a winning precondition for component i (refinement of **true**) that is used in the partial definition of a new function passed in the recursive call to solve (Lines 13 and 16).

When there are no more components for which a precondition has not yet been computed, solve terminates (Line 10) and main checks that $s_0 \models (\texttt{precond}\ 0)$.

Theorem 6 (Soundness). *If the algorithm in Figure 2 returns "winning strategy," then there is a winning strategy for Angel.*

Proof. *It suffices to show that* solve *computes winning preconditions for all components. We do this by induction on* solve precond *and the partial order $\preceq$ of the components. The base case is trivial. In the inductive case for component i, we assume that* precond *maps each immediate successor component j to its precondition ρ_j and show that* solve *will recur with a second argument that maps i to its precondition ρ_i. By the inductive hypothesis we know that (*precond j*) is the winning precondition for component j. There are now two cases:*

1. *C_i is fair.* $\mathsf{PreC}_{\mathsf{safe}}$ *will return the condition for Angel to avoid reaching a losing subcomponent: $\bigvee_{j \in J}(\neg(\textsf{precond}\ j) \wedge c_j)$. That is,* $\mathsf{PreC}_{\mathsf{safe}}$ *returns the condition under which there is a winning strategy for Angel to avoid satisfying any jth losing precondition $\neg(\textsf{precond}\ j)$.*
2. *C_i is unfair.* $\mathsf{PreC}_{\mathsf{live}}$ *will return the condition for Angel reach a winning subcomponent: $\bigvee_{j \in J}((\textsf{precond}\ j) \wedge c_j)$. That is,* $\mathsf{PreC}_{\mathsf{live}}$ *returns the condition under which there is a winning strategy for Angel to satisfy some jth winning precondition (*precond j*).*

4.3 Solving Alternation-Free Mu-Calculus Games

In this symbolic setting, we have implemented the translation described in Section 2.4 that reduces the question of whether a state in two-player transition system satisfies an AFMC sentence can be reduced to solving weak recurrence games. Our technique first takes a μ-calculus formula φ and transition system P and creates a weak recurrence game described by component predicates $c_1, ..., c_n$. In this game the state space is a pair consisting of (fst) the weak game state and (snd) the state from the original two-player transition system. We use integers to represent the weak game state, assigning a distinct integer to each of the (finitely many) weak game states. The two-player transition system state is described as a single variable, but this is expressive enough because our implementation supports tuples, higher order functions and algebraic data types (in addition to integers, booleans, unit).

5 Evaluation

Implementation. We have developed a prototype tool called PGS (Program-Games Solver), capable of automatically discovering winning strategies for games over infinite-state spaces. The tool takes safety, liveness, weak recurrence, and AFMC verification games described as mutually recursive functions in the syntax of the OCaml functional language. The tool supports integers, booleans, tuples, higher order functions, and algebraic data types. As a backend termination and non-termination checker, PGS uses a combination of previous techniques [30, 29, 21].

Name	Type	LOC	Result	Time	Precond.
safe_int1	safety	4	✓	0.775	No
safe_int2	safety	5	✓	1.418	No
safe_int3	safety	9	✓	9.050	No
safe_list	safety	11	✓	2.384	No
safe_fun	safety	15	✓	7.862	No
2bits	safety	14	✓	9.570	No
3bits	safety	18	✓	33.827	No
5game	safety	20	✓	82.526	No
live_int1	liveness	4	✓	0.794	No
live_int2	liveness	8	✓	0.915	No
live_int3	liveness	16	✓	5.231	No
live_list	liveness	14	✓	8.839	No
live_fun	liveness	15	✓	4.984	No
weak_rec	weak recurrence	18	✓	6.411	No
afmc_nu1	$p \Rightarrow \nu z.(q \wedge \Diamond_A \Box_D z)$	4	✓	29.519	Yes
afmc_nu2	$p \Rightarrow \nu z.(q \wedge \Diamond_A \Box_D z)$	4	✓	36.298	Yes
afmc_mu	$\mu z.(p \vee \Diamond_A \Box_D z)$	4	???	28.223	Yes

Fig. 3: The results of applying our prototype tool PGS to a variety of games.

Evaluation. We have evaluated our tool with a variety of games over infinite state spaces. We have conducted the experiments on a machine with Intel(R) Xeon(R) CPU E5620 (2.40 GHz, 4 GB of memory). The experiment results are summarized in Table 3. The first and the second columns show the name and the type of each game. AFMC verification games are represented as the shape of the AFMC formula where p and q are literals specific to the game. We also give the lines of code "**LOC**". We report the result "**Result**" and the total time "**Time**" in seconds for each game. "✓" indicates that our tool automatically discovered a winning strategy and "???" indicates that our tool reported unknown whether a winning strategy exists. The column "**Precond.**" shows whether winning preconditions were used to solve the game.

The games with names *_int*, *_list, and *_fun are respectively over integers, inductively defined lists, and functions. The OCaml codes of the games and their verification log messages output by PGS are available as a supplementary material. These games are small but tricky: game solving required PGS to automatically synthesize non-trivial invariants, ranking functions, and strategies for random number generation on not only integers but also algebraic data structures and higher-order functions (see the PGS log messages for details). PGS successfully solved most of them within a minute. We here found that winning precondition inference is essential for solving weak recurrence games reduced from AFMC verification games. PGS failed to solve the game afmc_mu because the backend termination checker was not able to automatically find necessary invariants for solving 1 out of 4 components. The other 3 components were successfully solved. We here expect PGS can benefit from future improvements of termination and non-termination verification techniques.

6 Conclusion

We have shown the first technique to automatically solve weak-recurrence games over infinite state spaces. This encompasses the verification question for alternation-free μ-calculus sentences over two-player transition systems. Our work reduces the problem to safety and liveness over a finite set of components, using the *winning precondition* of lower components in the search for winning conditions of higher components. We have implemented our tool, applied it to a variety of examples, and made it available at:

`http://www-kb.is.s.u-tokyo.ac.jp/~uhiro/weakgame/`

Other related work. In Section 1, we surveyed the related work on model checking, alternation-free μ-calculus, and games. In recent years there have been numerous techniques for proving safety, liveness, and temporal properties of infinite-state programs. This includes LTL [12], $\forall$CTL [14], CTL [13, 6], and CTL* [11]. Beyene et al. [5] proposed a constraint-based approach for solving games on infinite graphs. To the best of our knowledge, none of these techniques are able to verify general AFMC sentences.

References

1. Ball, T., Cook, B., Levin, V., and Rajamani, S. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods* (2004), pp. 1–20.
2. Ball, T., Levin, V., and Rajamani, S. K. A decade of software model checking with SLAM. *Commun. ACM 54*, 7 (2011), 68–76.
3. Ball, T., and Rajamani, S. The SLAM toolkit. In *Proc. 13th Int. Conf. on Computer Aided Verification* (2001), vol. 2102 of *Lecture Notes in Computer Science*, Springer, pp. 260–264.
4. Beeri, C., and Bernstein, P. Computational problems related to the design of normal form relational schemas. *ACM Trans. on Database Systems 4* (1979), 30–59.
5. Beyene, T. A., Chaudhuri, S., Popeea, C., and Rybalchenko, A. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014* (2014), pp. 221–234.
6. Beyene, T. A., Popeea, C., and Rybalchenko, A. Solving existentially quantified horn clauses. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings* (2013), pp. 869–882.
7. Chatterjee, K., and Henzinger, M. Efficient and dynamic algorithms for alternating büchi games and maximal end-component decomposition. *J. ACM 61*, 3 (2014), 15:1–15:40.
8. Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languagues and Systems 8*, 2 (1986), 244–263.
9. Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 1999.
10. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Proving that programs eventually do something good. In *Proc. 34th ACM Symp. on Principles of Programming Languages* (2007), pp. 265–276.
11. Cook, B., Khlaaf, H., and Piterman, N. On automation of ctl* verification for infinite-state systems. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I* (2015), pp. 13–29.
12. Cook, B., and Koskinen, E. Making Prophecies with Decision Predicates. In *POPL'11* (2011).
13. Cook, B., and Koskinen, E. Reasoning about nondeterminism in programs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (2013), ACM.
14. Cook, B., Koskinen, E., and Vardi, M. Y. Temporal property verification as a program analysis task. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (2011), pp. 333–348.
15. Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *Proc. ACM Conf. on Programming Language Design and Implementation* (2006), pp. 415–426.
16. Dijkstra, E. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (1975), 453–457.
17. Emerson, E., Jutla, C., and Sistla, A. On model-checking for fragments of μ-calculus. In *Proc. 5th Int. Conf. on Computer Aided Verification* (1993), vol. 697 of *Lecture Notes in Computer Science*, Springer, pp. 385–396.
18. Emerson, E., and Lei, C.-L. Modalities for model checking: Branching time logic strikes back. In *Proc. 12th ACM Symp. on Principles of Programming Languages* (1985), pp. 84–96.

19. Emerson, E., and Lei, C.-L. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE Symp. on Logic in Computer Science* (1986), pp. 267–278.
20. Grädel, E., Thomas, W., and Wilke, T. *Automata, Logics, and Infinite Games: A Guide to Current Research*, vol. 2500 of *Lecture Notes in Computer Science*. Springer, 2002.
21. Hashimoto, K., and Unno, H. Refinement type inference via horn constraint optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings* (2015), pp. 199–216.
22. Hintikka, J., and Sandu, G. Game-theoretical semantics. In *Handbook of Logic and Language*, J. van Benthem and A. ter Meulen, Eds. Elsevier, 1997.
23. Jifeng, C. H. H. *Unifying theories of programming*, vol. 14. Prentice Hall, 1998.
24. Kozen, D. Results on the propositional μ-calculus. *Theoretical Computer Science 27* (1983), 333–354.
25. Kupferman, O., and Vardi, M. An automata-theoretic approach to modular model checking. 87–128.
26. Kupferman, O., and Vardi, M. μ-calculus synthesis. In *25th Int. Symp. on Mathematical Foundations of Computer Science* (2000), vol. 1893 of *Lecture Notes in Computer Science*, Springer, pp. 497–507.
27. Kupferman, O., Vardi, M., and Wolper, P. An automata-theoretic approach to branching-time model checking. *Journal of the ACM 47*, 2 (2000), 312–360.
28. Kupferman, O., Vardi, M., and Wolper, P. Module checking. *Information and Computation 164* (2001), 322–344.
29. Kuwahara, T., Sato, R., Unno, H., and Kobayashi, N. Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II* (2015), pp. 287–303.
30. Kuwahara, T., Terauchi, T., Unno, H., and Kobayashi, N. Automatic termination verification for higher-order functional programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (2014), pp. 392–411.
31. Lenzi, G. The modal μ-calculus: a survey. *TASK Quarterly 9*, 3 (2005), 293–316.
32. Mazala, R. Infinite games. In *Automata logics, and infinite games*, Lecture Notes in Computer Science 2500. Springer, 2002, pp. 23–38.
33. Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages* (1989), pp. 179–190.
34. Vardi, M. Alternating automata and program verification. In *Computer Science Today –Recent Trends and Developments* (1995), vol. 1000 of *Lecture Notes in Computer Science*, Springer, pp. 471–485.
35. Vardi, M. Branching vs. linear time: Final showdown. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2001), vol. 2031 of *Lecture Notes in Computer Science*, Springer, pp. 1–22.