Modular Strategies for Infinite Games on Recursive Graphs*

Rajeev Alur[1], Salvatore La Torre[2], and P. Madhusudan[1]

[1] University of Pennsylvania
[2] Università degli Studi di Salerno

Abstract. In this paper, we focus on solving games in recursive game graphs that can model the control flow of sequential programs with recursive procedure calls. The winning condition is given as an ω-regular specification over the observable, and, unlike traditional pushdown games, the strategy is required to be *modular*: resolution of choices within a component should not depend on the context in which the component is invoked, but only on the history within the current invocation of the component. We first consider the case when the specification is given as a deterministic Büchi automaton. We show the problem to be decidable, and present a solution based on two-way alternating tree automata with time complexity that is polynomial in the number of internal nodes, exponential in the specification and exponential in the number of exits of the components. We show that the problem is Exptime-complete in general, and Np-complete for fixed-size specifications. Then, we show that the same complexity bounds apply if the specification is given as a *universal* co-Büchi automaton. Finally, for specifications given as formulas of linear temporal logic Ltl, we obtain a synthesis algorithm that is doubly-exponential in the formula and singly exponential in the number of exits of components.

1 Introduction

An interesting class of infinite-state systems that permits algorithmic verification is *pushdown systems*. Pushdown systems can model the control flow in sequential imperative programming languages with recursive procedure calls. Their analysis has been well studied theoretically [6, 20], and forms the basis of recent tools for software verification [4, 11, 12]. In this paper, we focus on solving *games* over such models. The original motivation for studying games in the context of formal analysis of systems comes from the *controller synthesis* problem [7, 16, 17]: given a model of the system where some of the choices depend upon the controllable inputs and some of the choices represent uncontrollable nondeterminism, synthesizing a controller that supplies inputs to the system so that the

* This research was supported in part by ARO URI award DAAD19-01-1-0473, NSF award CCR99-70925, and NSF award ITR/SY 0121431. The second author was also supported by the MIUR in the framework of project "Metodi Formali per la Sicurezza e il Tempo" (MEFISTO) and MIUR grant 60% 2002.

product of the controller and the system satisfies the correctness specification corresponds to computing winning strategies in two-player games. Besides the long-term dream of synthesizing correct programs from formal specifications, games are relevant for modular verification of open systems. For instance, *Alternating Temporal Logic* allows specification of requirements such as "module A can ensure delivery of the message no matter how module B behaves" [2]; *module checking* deals with the problem of checking whether a module behaves correctly no matter in which environment it is placed [14]; and the framework of interface automata allows assumptions about the usage of a component to be built into the specification of the interface of the component, and formulates compatibility of interfaces using games [10].

Among the many roughly equivalent formulations of pushdown systems, we use the model of *recursive state machines* (RSMs) [1, 5]. A recursive state machine consists of a set of component machines called *modules*. Each module has a set of *nodes* (atomic states) and *boxes* (each of which is mapped to a module), a well-defined interface consisting of *entry* and *exit* nodes, and edges connecting nodes/boxes. An edge entering a box models the invocation of the module associated with the box, and an edge leaving a box corresponds to a return from that module. To define two-player games on recursive state machines, the nodes are partitioned into two sets such that a player gets to choose the transition when the current node belongs to its own partition. While the complexity of pushdown games has already been studied [8, 20], existing algorithms for solving pushdown games assume that each player has access to the entire global history which includes the information of the play in all modules. In a recent paper, we introduced the notion of *modular* strategies for games on RSMs [3]. A modular strategy is a strategy that has only local memory, and thus, resolution of choices within a module does not depend on the context in which the module is invoked, but only on the history within the current invocation of the module. This permits a natural definition of synthesis of recursive controllers: a controller for a module can be plugged into any context where the module is invoked. Recent work on the interface compatibility checking for software modules implements the global games on pushdown systems [10], but we believe that checking for existence of modular strategies matches better with the intuition for compatibility.

In [3], we showed that solving modular reachability games is NP-complete. In this paper, we consider the general case where the winning condition is specified as an ω-regular language over the observable, using Büchi automata or linear temporal logic formulas. Compared to reachability games, there are two additional technical hurdles now. First, the winning strategy needs to produce accepting cycles. Second, the specification is external, and we require the *same* strategy to be used every time a module is invoked. Consequently, if we take the product of the game graph and the specification automaton, as is done typically, we need to find a winning strategy over the product graph that is modular as well as oblivious to the state of the specification.

Our first result is that for a specification given as a deterministic Büchi automaton $\mathcal{B}$, the problem of deciding whether there exists a modular strategy so that the resulting plays are guaranteed to be accepted by $\mathcal{B}$ is EXPTIME-complete. The upper bound is established using an automata-theoretic solution. We first define strategy trees that encode modular strategies in the recursive game graph A in a particular way. The strategy tree contains a subtree corresponding to each module. The subtree corresponding to a module A_m specifies the choice at every existential node of A_m, and when it enters a box b corresponding to an invocation of another module $A_{m'}$, it just specifies the exit nodes of $A_{m'}$ that are guaranteed *to be avoided* by the strategy for $A_{m'}$. The next step is to define a *two-way alternating tree automaton* T that accepts the winning strategy trees. An important task of this automaton is to check the consistency of the input tree with respect to the exits to be avoided in each module. The *alternating* nature allows sending of multiple copies, and the two-way nature is exploited to move up and down the tree, rereading strategies of modules as and when needed, and thereby making sure that only one strategy for each module is used. Using the exponential translation from two-way alternating tree automata to nondeterministic tree automata [18], and then, employing the polynomial-time algorithm for checking emptiness of nondeterministic tree automata, we get a complexity bound that is polynomial in the number of internal nodes of A, exponential in the specification automaton $\mathcal{B}$, and exponential in the total number of exit nodes in A. The exponential dependence on the number of exits seems unavoidable as even reachability games are NP-hard [3]. We show that the NP upper bound applies for fixed-size specifications given by Büchi automata.

Our second result is that given a recursive game graph A whose nodes are labeled with atomic propositions, and a formula φ of linear temporal logic LTL, the problem of deciding whether there exists a modular strategy in A so that the resulting paths are guaranteed to satisfy φ, is 2EXPTIME-complete. The hardness holds even for ordinary game graphs [16]. Since translation from LTL to deterministic parity automata is doubly-exponential, using our construction for deterministic specifications would lead to a 3EXPTIME upper bound. We show that the construction for deterministic Büchi specifications can be modified to *universal* Büchi as well as co-Büchi specifications with the same complexity. Unlike a nondeterministic automaton, a universal automaton accepts a word if all runs over that word are accepting. Since the set of models of an LTL formula can be characterized by a universal co-Büchi automaton that is only exponential in the formula, we get a 2EXPTIME bound for LTL games.

We recall that two-way alternating tree automata have been used to solve synthesis problems for pushdown systems with respect to specifications given by μ-calculus formulas [13] and parity pushdown games [9]. Apart from the fact that these papers only solve for global strategies and not modular ones, the encodings of strategies are different. While in their encodings, paths to nodes of the tree represent the stack content and the two-way nature is used to push and pop elements, in our setup paths to nodes represent the local history and the two-way nature is used to re-read strategies for a module.

Due to the lack of space we omit some details in this version and refer the interested reader to the full paper.

2 Games on Recursive Graphs

Let us fix a finite alphabet Σ. Let Σ^* denote the finite words and Σ^ω denote the set of ω-words over Σ. For any $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, \ldots, n\}$.

Definition 1. *A* recursive game graph *A (for short RGG) over Σ is a tuple $(M, m_{in}, \{A_m\}_{m \in M})$, where M is a finite set of module names, $m_{in} \in M$ is the name of the initial module and for every $m \in M$, A_m is a* game module *$(N_m, B_m, Y_m, En_m, Ex_m, P^0_m, P^1_m, \delta_m, \eta_m)$ that consists of:*

- *A finite set of* nodes *N_m and a finite set of* boxes *B_m.*
- *A nonempty set of* entry *nodes $En_m \subseteq N_m$ and a nonempty set of* exit *nodes $Ex_m \subseteq N_m$.*
- *A labeling $Y_m : B_m \to M$ that assigns a module to every box.*
- *Let $Calls_m = \{(b, e) \mid b \in B_m, e \in En_{Y_m(b)}\}$ denote the set of* calls *of m and let $Retns_m = \{(b, x) \mid b \in B_m, x \in Ex_{Y_m(b)}\}$ denote the set of* returns *in m. Then, $\delta_m : N_m \cup Retns_m \to 2^{N_m \cup Calls_m}$ is a* transition function*.*
- *P^0_m and P^1_m form a partition of $N_m \cup B_m$ into the set of positions of player 0 and player 1, respectively.*
- *η_m is a labeling function $\eta_m : N_m \to \Sigma$ that associates a letter in Σ with each node.*

We assume that all the sets N_m and B_m ($m \in M$) are disjoint. Also, we let $N = \bigcup_m N_m$, $B = \bigcup_m B_m$, $Calls = \bigcup_m Calls_m$ and $Retns = \bigcup_m Retns_m$ denote the set of all nodes, boxes, calls and returns, respectively. Similarly, let $En = \bigcup_{m \in M} En_m$, $Ex = \bigcup_{m \in M} Ex_m$, $P^\ell = \bigcup_{m \in M} P^\ell_m$, for $\ell \in \{0, 1\}$. We extend the functions Y_m to a single function $Y : B \to M$ by defining $Y(b) = Y_m(b)$, where $b \in B_m$. Similarly, we extend the functions η_m to a single function $\eta : N \to \Sigma$.

An element in $Calls_m$ of the form (b, e) represents a call from m to the module m', where $Y_m(b) = m'$ and e is an entry of $A_{m'}$. An element in $Retns_m$ of the form (b, x) corresponds to the associated return of control from m' to m when the call exits from m' at exit x. The transition function hence defines moves from nodes and returns to a set of nodes and calls.

We denote the set of *vertices* of m as $V_m = N_m \cup Calls_m \cup Retns_m$. Viewed in terms of vertices, each A_m defines a graph over the set of vertices V_m. Let $V^\ell_m = (N_m \cap P^\ell_m) \cup \{(b, x) \in Retns_m \mid b \in P^\ell_m\}$ denote the set of vertices of player ℓ. Note that returns are identified as belonging to player ℓ if the box corresponding to it belongs to player ℓ. The vertices in $Calls_m$ are not assigned to any player. (One can assign nodes to players in various ways; we have just chosen one such way, without any real loss in generality.)

Without loss of generality, we make some assumptions of these graphs in the sequel that enable a more readable presentation:

- There is only *one* entry point to every module, i.e. $|E_m| = 1$ for every m. We refer to this unique entry point of A_m as e_m. One can reduce a game-graph module with multiple entries to this setting by making copies of this module, one for each entry point, and changing the calls and returns appropriately. This causes only a cubic blow-up (see [3] where this is done for reachability specifications).

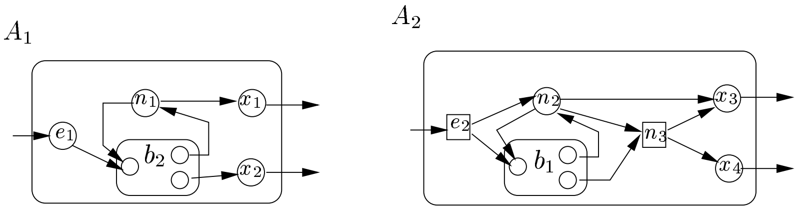

Fig. 1. An example of a recursive game graph.

– For every $u \in N_m$, $e_m \notin \delta_m(u)$ holds, and for every $x \in Ex_m$, $\delta_m(x)$ is empty. That is, within a module there are no transitions to its entry point and no transitions from its exits.
– A module is not called immediately after a return from another module. That is, for any m, $\delta_m(Retns_m) \subseteq N_m$.

As an example of a recursive game graph consider the graph in Figure 1. There are two modules A_1 and A_2, and A_1 is the initial module. We denote by squares the nodes of player 1 and by circles those of player 0. The rectangles with curved corners denote the boxes of player 0. Box b_1 is mapped by Y to module A_1 and b_2 is mapped to A_2. In this recursive game graph, all boxes belong to player 0.

A *state* of the game is an element in $(\gamma, u) \in B^* \times N$ such that either
– $\gamma = \epsilon$ and $u \in N_{m_{in}}$, or,
– $\gamma = b_1 \dots b_k$, $b_1 \in B_{m_{in}}$ and for each $i \in [1, k-1]$, if $Y(b_i) = m$, then $b_{i+1} \in B_m$, and $u \in N_{m'}$ where $Y(b_k) = m'$.

Intuitively a state of the form $(b_1\, b_2, u)$ represents the global state of the system where, if $Y(b_1) = m$ and $Y(b_2) = m'$, then the module $A_{m_{in}}$ has called module A_m by box b_1, which in turn has called module m' by box b_2 and the current node inside m' is u. Hence, in a state (γ, u), γ denotes the *stack* of calls and u denotes the current node in the last invoked module.

The following then defines the natural notion of a run of an RGG. A *run* of an RGG A is a finite or infinite non-null sequence of states, $s_0 s_1 \dots$ such that:
1. $s_0 = (\epsilon, e_0)$
2. If $s_j = (\gamma, u)$, $u \in N_m$, and $s_{j+1} = (\gamma', u')$, then one of the following holds:
   Internal move: $u \in N_m \setminus Ex_m$, $u' \in \delta_m(u)$ and $\gamma' = \gamma$.
   Call a module: $u \in N_m \setminus Ex_m$, $(b, e_{m'}) \in \delta_m(u)$, $u' = e_{m'}$ and $\gamma' = \gamma.b$.
   Return from a call: $u \in Ex_m$, $\gamma = \gamma'.b$, $u' \in \delta_{m'}((b, u))$, where $b \in B_{m'}$.

The first case above is when the control stays within module m, the second case is when a new module m' is entered via a box of m (and we "push" the box name b onto the stack) and the third is when the control exits m and returns to m' (and we "pop" the box name b from the stack). A *play* in A is a run of A. We denote by Π_f and Π_ω the set of all finite and infinite plays of A, respectively.

For a state $s = (\gamma, u)$, let $V(s)$ denote the vertex corresponding to the state: $V(s) = u$ if $\gamma = \epsilon$ or $u \in N_m \setminus Ex_m$; otherwise $V(s) = (b, u)$ where $\gamma = \gamma'.b$. Then $ctr : \Pi_f \to M$ identifies the module where the control is after any finite run and is defined as follows: for any $\pi.s \in \Pi_f$, $ctr(\pi.s) = m$, where $V(s) \in V_m$.

We can now define local histories that describe, for every finite play π such that $ctr(\pi) = m$, the history of the play within m since the current invocation of the module m. For a state $s = (b_1 \dots b_r, u)$, an *s-history* is $\langle \beta_1, \beta_2, \dots, \beta_r \rangle$, where for each $i \in [r]$, $\beta_i \in V_m^*$ where $b_i \in B_m$. In other words, each β_i is a sequence of vertices of the module b_i belongs to. We define a *history* function Hst that associates with every finite play $\pi.s$ an s-history $Hst(\pi.s)$ as follows.

- If $\pi = s_0 = (\epsilon, e_{m_{in}})$, then $Hst(\pi) = \langle e_{m_{in}} \rangle$.
- Let $\pi = \pi'.(\gamma', u')$ where $\pi' = \pi''.(\gamma, u)$, and let $Hst(\pi') = \langle \beta_1, \dots, \beta_r \rangle$. Then:
  Internal move: If $\gamma' = \gamma$, then $Hst(\pi) = \langle \beta_1, \dots, \beta_{r-1}, \beta_r.u' \rangle$.
  Call: If $\gamma' = \gamma.b$ with $Y(b) = m'$, then $Hst(\pi) = \langle \beta_1, \dots, \beta_r, e_{m'} \rangle$.
  Return: If $\gamma = \gamma'.b$, then $Hst(\pi) = \langle \beta_1, \dots, \beta_{r-1} \rangle$.

Intuitively, in the beginning of the run, when the control is at m_{in}, the history is a single string e_{in}. On an internal move, the last element in the history tuple gets updated, while the other tuples remain unchanged. On a call, the last element in the history tuple also freezes, and a new entry is created and initialized to the entry corresponding to the called module. When the call returns, the last tuple in the history gets erased. The history function hence keeps a *stack* of histories: for each module in the stack of calls, there is a record of the moves that have happened in that module during the play thus far.

We now define the *local history*, μ, after a finite play π. For any play π, $\mu(\pi) = \beta_k$ where $Hst(\pi) = \langle \beta_1, \dots, \beta_k \rangle$. In other words, the local history is the fragment of the play inside the current module $m = ctr(\pi)$ since the corresponding entry into m. Note that the local history is a sequence of vertices that correspond to internal nodes, calls or returns of the module m.

A *modular strategy* is intuitively a set of functions, one for each module, that encodes how player 0 must play in the game. However, which move to make at a state can depend only on the *local history* of the play so far in the current module. Formally, a modular strategy is a set of functions, $f = \{f_m\}_{m \in M}$, one for each module, where for every m, $f_m : V_m^*.V_m^0 \to V_m$ such that $f_m(\pi v) \in \delta_m(v)$, for every $\pi \in V_m^*$, $v \in V_m^0$.

A *play π in A according to a modular strategy f* is a run $s_0 s_1 \dots$ such that for every $i < |\pi|$, if $ctr(s_0 \dots s_i) = m$ and $V(s_i) \in V_m^0$, then $s_{i+1} = (\gamma', u')$, where $f_m(\mu(s_0 \dots s_i))$ is either u' or (b, u'), for some $b \in B$. In other words, if a prefix of the play ends in a player 0 vertex, the move recommended by f for the local memory in the current module must be taken.

Winning conditions. A *winning set over Σ* is a *regular* ω-language over Σ, i.e. a regular language $\mathcal{L} \subseteq \Sigma^\omega$. A *recursive game* is a pair $(A, \mathcal{L})$ where A is a recursive game graph and $\mathcal{L}$ is a winning set, both over Σ. Let us extend η to states by defining $\eta((\gamma, u)) = \eta(u)$. This then extends to plays: $\eta(s_0 s_1 \dots) = \eta(s_0)\eta(s_1)\dots$. A play π is said to be *winning* if $\eta(\pi) \in \mathcal{L}$, the winning set. A modular strategy is *winning* if every play according to it is winning. We consider the following decision problem:

"Given a recursive game $(A, \mathcal{L})$ is there a modular winning strategy for the protagonist?"

In this paper we solve this problem for $\mathcal{L}$ given by *safety* and *deterministic/universal Büchi/co-Büchi* automata, and by LTL formulas.

3 Automata for winning strategies

3.1 Background: Words, trees and automata

An automaton on ω-words over Σ is $\mathcal{A} = (Q, q_1, \delta, W)$ where Q is a finite set of states, $q_1 \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is a transition function and W is a *winning condition*. Depending on the winning condition we have a Büchi, co-Büchi, or a parity automaton. A *Büchi* (resp. *co-Büchi*) winning condition is $W \subseteq Q$ and requires that a state from W repeats infinitely often (resp. only finitely often). A *parity* condition is a function $W : Q \to [c]$, for some $c \in \mathbb{N}$, that associates a colour to every element in Q. It requires that the minimal colour seen infinitely often is even. A run of $\mathcal{A}$ over a word $\sigma_0\sigma_1 \ldots \in \Sigma^\omega$ is an ω-sequence of states $p_0p_1 \ldots$ such that $p_0 = q_1$ and $\forall i \in \mathbb{N}$, $p_{i+1} \in \delta(p_i, \sigma_i)$. A run is accepting if it satisfies the winning condition W. An automaton is *deterministic* if $|\delta(q, \sigma)| \leq 1$ for every $q \in Q$, $\sigma \in \Sigma$. For either deterministic or nondeterministic automata a word $\alpha \in \Sigma^\omega$ is accepted if *there exists* an accepting run on it. For *universal* automata α is accepted if *all* runs on it are accepting. A *safety* automaton is a deterministic Büchi automaton $\mathcal{A} = (Q, q_1, \delta, W)$ where $W = Q$. In other words, it accepts a word as long as it has a run over the word, thus we omit mentioning the winning condition.

Let $k \in \mathbb{N}$ and Δ be a finite alphabet. A Δ-labelled k-tree is (T_k, ν) where $T_k = (Z, E)$ is a tree with $Z = [k]^*$ and $E = \{(y, y.d) \mid y \in [k]^*, d \in [k]\}$, and $\nu : Z \to \Delta$ is a labelling function that labels every vertex of the tree with a letter in Δ. To distinguish vertices of trees and vertices of recursive game graphs, we refer to the former as *tree-vertices*. The tree-vertex ϵ is the *root* of the tree T_k, and for every $y \in Z$, tree-vertex $y.d$ is the d^{th} child of y. For any set X, let $\mathcal{B}^+(X)$ denote the set of boolean formulae over X using conjunctions and disjunctions only (negation is not allowed). For any subset F of X, we say that F *satisfies* a formula $\varphi \in \mathcal{B}^+(X)$ if φ evaluates to true assigning the elements in F to *true* and the other elements in X to *false*.

A *two-way alternating parity tree automaton* (see [18]) over Δ-labelled k-trees is $\mathcal{A} = (Q, q_1, \delta, W)$, where Q is a finite set of states, $q_1 \in Q$ is the initial state, W is a parity condition on Q and $\delta : Q \times \Delta \to \mathcal{B}^+((\{-1, 0, 1, \ldots, k) \times Q)$. Intuitively, $\{-1, 0, \ldots, k\}$ code the directions from a tree-vertex, where $\{1, \ldots, k\}$ stand for the k children of the tree vertex, -1 stands for the parent of the tree vertex, and 0 stands for the current tree-vertex itself. Let us extend the definition of concatenation of words over $[k]^*$ as follows: $(xi.(-1)) = x$ and $x.0 = x$, for any $x \in [k]^*$, $i \in [k]$, i.e. when a word is concatenated with -1, it removes the last letter and concatenating with 0 is the identity function. A *one-way nondeterministic tree automaton* can be seen as a two-way alternating tree automaton where the transition function is always a disjunction of formulas of the kind $\bigwedge_{j=1}^{k}(j, q_j)$, i.e. the automaton guesses nondeterministically to send exactly *one* copy of itself in each *forward* direction.

A run of $\mathcal{A}$ over a Δ-labelled k-tree (T_k, ν), where $T_k = (Z, E)$, is a labelled tree $T_\rho = (R_\rho, E_\rho)$ where each tree-vertex in R_ρ is labelled with a pair (x, q) where $x \in Z$ is a tree-vertex of the input tree and $q \in Q$ is a state of the automaton $\mathcal{A}$, such that: (a) the root of T_ρ is labelled (ϵ, q_1), and (b) if a tree-vertex y of T_ρ is labelled (x, q), then we require that there is a set $F \subseteq \{-1, 0, 1, \ldots, k\} \times Q$ such that F satisfies $\delta(q, \nu(x))$ and for each $(i, q') \in F$, y has a child labelled $(x.i, q')$. A run is *accepting*, if for every *infinite* path in the run tree, if one projects the second component of the labels along the path, then it is a sequence of states in Q that satisfies the winning condition of $\mathcal{A}$. Note that there is no condition for *finite* paths of the run tree. An automaton $\mathcal{A}$ accepts a Δ-labelled k-tree T iff there is an accepting run of $\mathcal{A}$ on T; the language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all Δ-labelled k-trees that $\mathcal{A}$ accepts.

Proposition 1.
- *Let $\mathcal{A}$ be a two-way alternating parity tree automaton. Then there is a one-way nondeterministic parity tree automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, the number of states in $\mathcal{A}'$ is exponential in the number of states in $\mathcal{A}$, and the number of colours in the parity condition of $\mathcal{A}'$ is linear in the number of colours in the parity condition of $\mathcal{A}$ [18].*
- *The emptiness of one-way parity tree automata can be checked in time that is polynomial in the number of states and exponential in the number of colours in the parity condition (see [17]).*

3.2 Strategy trees

Let us fix a set of h modules $M = \{m_1, \ldots, m_h\}$, which is ordered, and a finite alphabet Σ for this section. Let us fix a recursive game graph over Σ, $A = (M, m_1, \{A_m\}_{m \in M})$, where each $A_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m)$. By Δ_A we denote the set of labels $\{root, dummy\} \cup (V \times \{\top, \bot\})$.

A *strategy tree* is a Δ_A-labelled k-tree and is intended to encode a modular strategy in the following way. First, we have the special symbol *root* labelling the root of the tree. The subtree rooted at the i^{th} child of the root will encode the modular strategy for m_i. The root of the subtree for m is labelled by the entry point of m and tagged with $\top$. (The other children of the root, if any, are marked to be dummy tree-vertices and will not encode any information). No other tree-vertices are labelled from the set $En \times \{\top, \bot\}$.

The subtree for a module m encodes the strategy for the module m by unravelling the graph A_m and annotating each tree-vertex with either $\top$ or $\bot$, with $\top$ intuitively encoding that a move to the corresponding node is possible while a $\bot$-tag signifies that the strategy will not allow a play to visit this node.

If a tree-vertex v of the subtree for m is labelled $(p, \top)$, then:
- When $p \in (N_m \setminus Ex_m) \cup Retns_m$, the children of v are labelled by the successors of p along with a $\top/\bot$ annotation. Further, if p is a player 0 vertex, then the strategy must choose exactly one successor of p, and thus exactly one of the annotations of the children is $\top$. If p is a player 1 vertex, then *all* moves of player 1 need to be accounted for, hence all children are tagged with $\top$.
- When $p \in Calls_m$, we *do not* encode calling the other module; we instead simply have children corresponding to the returns from the called module, with any $\top/\bot$ annotation. This hence encodes an assumption that the call to

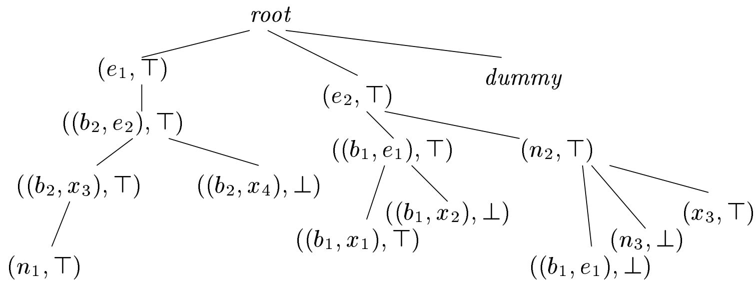

Fig. 2. A fragment of a strategy tree.

the other module will definitely not end in any return which is tagged with $\bot$, no matter how player 1 plays. Note that we do not put any constraints at this point that this assumption is indeed true—we take care of this later when we check these trees using automata.

If a tree-vertex of the subtree for m is labelled $(p, \bot)$, this denotes a move disabled by the strategy and we do not continue to define the strategy from it; we hence label the entire subtree under it as *dummy*. Similarly, if a tree-vertex is labelled (p, ζ) where $p \in Ex_m$, then this signifies reaching an exit node of the module and again we do not encode the strategy after this point.

For example, consider the recursive game graph in Figure 1, and a modular strategy such that the choices for player 0 are resolved selecting (b_2, e_2) from n_1 and x_3 from n_2. A fragment of the corresponding strategy tree is shown in Figure 2. Note that the subtree encoding the strategy for module A_1 assumes that when A_2 is called, it will not exit at x_4, and the subtree for A_2 assumes that when A_1 is called, it will not exit at x_2. In this example, the strategy does indeed guarantee these assumptions. It is easy to see that:

Proposition 2. *There exists an effectively constructible one-way nondeterministic tree automaton $\mathcal{A}_{str}$ of size $O(|\mathcal{A}|)$ that accepts a Δ-labelled k-tree if and only if it is a strategy tree.*

3.3 Automata accepting winning strategy trees

In this section we sketch the construction of an automaton $\mathcal{A}_{win}$ that accepts a strategy tree iff it represents a winning strategy with respect to a specification given as a safety automaton $\mathcal{B} = (Q, q_1, \delta_{\mathcal{B}})$. In this construction it is crucial the use of an *avoid component* in states of $\mathcal{A}_{win}$. An avoid component $(x, q) \in Ex \times Q$ corresponds to the assumption that a play must not end at the exit x with the specification state q. There is also another type of avoid component denoted $\$_m$ that is used instead for the assumption that the play must not exit at all the current invocation of the module m. We will return to the avoid component later.

Automaton $\mathcal{A}_{win}$ is a two-way alternating tree automaton that performs three main tasks. The first task is to simulate $\mathcal{B}$ so that we ensure the satisfaction of the specification on an accepted strategy tree. This is achieved basically simulating the transitions from $\delta_{\mathcal{B}}$, whenever a node is read. On reading an exit node the automaton also checks that the assumption of the avoid component is fulfilled.

The second task is to ensure that plays are generated from the entry point of the initial module m_{in} starting with the initial state of the specification, and that this invocation of m_{in} does not exit. If this call exits then this would mean that there are finite maximal plays according to the strategy, which are by definition losing for player 0. Hence, the first transition sets the avoid component to every possible pair $(x, q) \in Ex_{m_{in}} \times Q$.

The last task is to ensure that when a return of the input tree is tagged with $\perp$, no matter how player 1 plays, the call to the other module will definitely not end in such a return (see Section 3.2). Assuming that this return is from a module m at exit x, the above requirement is checked sending a copy of the automaton to the root of the subtree corresponding to m for *every* avoid component of the form (x, q).

The winning condition we choose for $\mathcal{A}_{win}$ is trivial: we just map every state to the colour 0; hence the winning condition simply requires that there be a run over the tree. There is a trick that is needed in the construction to keep the size of $\mathcal{A}_{win}$ small. When reading a $\top$-tagged return (b, x) $(x \in Ex_{m'})$, the strategy encodes the assumption that when m' is called, it may return at exit x, but it does not contain information on what the specification state can be when it returns. The automaton hence will guess what the set of possible specification states can be when the call returns. Since we would like to keep the size of the transition function polynomial, we let the automaton step through the specification states q one by one, but staying at the same tree vertex, and guess whether the current specification state q is a possibility as the state on a return. We omit further details here. Thus, we have the following.

Lemma 1. *$\mathcal{A}_{win}$ is a two-way alternating tree automaton that accepts a strategy tree iff it corresponds to a winning strategy. Further, the size of $\mathcal{A}_{win}$ is $O(|Q|^2 \cdot |Ex|)$ where Q is the set of states in the specification automaton and Ex is the set of all exits in the recursive game graph.*

We now convert $\mathcal{A}_{win}$ to a one-way nondeterministic tree automaton and take its intersection with the automaton $\mathcal{A}_{str}$ that accepts strategy trees, to get a one-way nondeterministic automaton $\mathcal{A}'$ which accepts a tree iff it corresponds to a winning strategy tree. Since we can prove EXPTIME-hardness, via a direct reduction from linear-space Turing machines, and using Proposition 1, we have:

Theorem 1. *For a recursive game graph A and a specification safety automaton $\mathcal{B}$, the problem whether there is a winning modular strategy for player 0 is* EXPTIME-*complete, and can be decided in time $O(|A| \cdot \exp(|Ex| \cdot |\mathcal{B}|))$.* [1]

[1] $\exp(x)$ stands for 2^x.

4 Complexity of modular games

4.1 Handling other ω-regular specifications

The construction given in Section 3.3 can be extended to games with specifications given as deterministic Büchi or co-Büchi automata on words. We consider here only the case of Büchi automata; the case for co-Büchi automata is dual.

Let $\mathcal{B} = (Q, q_1, \delta_{\mathcal{B}}, W)$ be a deterministic Büchi automaton on words, and A be a recursive game graph. We can extend the automaton A_{win} of Section 3.3 to a two-way alternating tree automaton $\mathcal{A}'_{win}$ that accepts strategy trees corresponding to winning strategies in this game.

The main modifications are as follows. When the automaton reads a call in the subtree for a module m, recall that we guessed for every return labelled $\top$, the set of specification states the play could be at when it exits from the called module. In the case of Büchi specifications, we need to guess more about what happened during the call. More precisely, we need to know whether *all* possible sub-plays that return at this exit and specification state would have definitely met a state in W or not. If the automaton guesses that this is so, it must send a copy to the called module to check this, and also signal a Büchi final state (for the tree automaton) before continuing the play in the current module.

Again, by Proposition 1, and the lower bound given in Theorem 1, we have.

Theorem 2. *Deciding recursive game graphs with deterministic Büchi (or co-Büchi) automaton specifications is* EXPTIME*-complete.*

We can also extend the tree automaton from Section 3.3 to handle specifications given as *universal* Büchi (or co-Büchi) specifications. In the construction of the automaton, whenever we were updating the specification state, we now need to create a copy for *each possible update* of the specification state.

Theorem 3. *Deciding recursive game graphs with universal Büchi (or co-Büchi) automaton specifications is* EXPTIME*-complete.*

The above theorem in fact shows that we can solve games for any ω-regular specification. If $\mathcal{L}$ is any ω-regular language, then its complement, $\bar{\mathcal{L}}$ is also ω-regular and can be accepted by some nondeterministic Büchi automaton $\mathcal{B}'$. It is easy to see that if $\mathcal{B}'$ is viewed as a *universal co-Büchi automaton* $\mathcal{B}$ (we keep the same states and the same transitions but interpret the automaton as universal and co-Büchi), then $\mathcal{B}$ accepts a word α iff $\mathcal{B}'$ rejects α. Hence $\mathcal{B}$ accepts $\mathcal{L}$ and we can solve games against this automaton.

We can also deal with LTL-specifications ([15]) over a set of propositions $\mathcal{P}$, assuming the nodes in the game graph are labelled over $\Sigma = 2^{\mathcal{P}}$. Given an LTL formula ϕ, we can construct a nondeterministic Büchi automaton $B_{\neg\phi}$ over $2^{\mathcal{P}}$ that accepts a word iff it satisfies $\neg\phi$ [19]. Moreover, the size of this automaton is exponential in $|\phi|$. By the previous observation, this automaton when viewed as a universal co-Büchi automaton accepts the models of ϕ. In other words, for any LTL formula, we can construct an exponential sized universal co-Büchi automaton accepting its models. Invoking Theorem 3 and using the fact that LTL-games are 2EXPTIME-hard [16] even for normal graphs, we have:

Theorem 4. *Deciding* LTL *recursive game graphs is* 2EXPTIME*-complete.*

4.2 Structure Complexity

Our algorithm for deterministic automata specifications works in time exponential in the number of exits in the recursive game graph. We could ask whether this exponential is necessary by asking for the *structural complexity* of the problem, i.e. what is the complexity for fixed ω-regular specifications. Since a reachability game on a recursive game graph can be formulated by a simple fixed automaton specification, and since reachability games in recursive game graphs are NP-hard [3], we cannot hope to do polynomial in the game graph. However, we can show that for fixed specifications, the problem is in NP.

The NP upper bound can be shown by direct means, not involving automata. Let us sketch the proof for deterministic safety automata. When a module m invokes a module m' at a specification state q, the only relevant information that it needs is the set of exits m' could return at, and at each of these exits, the possible states the specification automaton can be in. This is a polynomial-sized information that we can guess. For each module m and specification state q, we build a graph $G_{m,q}$ which is a product of the game module for m and the specification automaton starting at state q. The problem then boils down to finding whether, for every module m, there is a strategy for the game module for m such that, when this strategy is played on $G_{m,q}$, for every $q \in Q$ (in the obvious way), the strategy meets the assumption pertaining to how module m should behave when invoked at q. In doing this, whenever we call another module in $G_{m,q}$, we can "plug-in" a graph that captures the assumption on how the other modules will behave. For any m, solving the game graphs $G_{m,q}$ simultaneously is akin to solving partial information games, which causes an exponential blow-up only in the size of the specification state space Q, which is a constant. Hence the problem can be solved in NP. The procedure extends to universal Büchi and co-Büchi automata with some effort.

Theorem 5. *Deciding recursive game graphs for fixed ω-regular specifications is* NP*-complete.*

5 Conclusions

In this paper, we have solved the problem of deciding the existence of modular strategies in infinite games over recursive structures for winning conditions specified using ω-automata or linear temporal logic. We have argued that the notion of modular strategies, compared to the traditional definition of global strategies, is more appropriate for designing modules that can be plugged in any context. Our solution is automata theoretic, and can be generalized to allow other types of specifications such as branching time logics. In terms of future work, we are exploring the application of games to generate interface abstractions, and efficient implementations using a combination of BDD-based symbolic techniques and SAT solvers.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. 13th Intern. Conf. on Computer Aided Verification, CAV'01*, LNCS 2102, p. 207–220. Springer, 2001.
2. R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.
3. R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *Proc. 9th Intern. Conf. on Tools and Algorithms for the Construction and the Analysis of Systems, TACAS'03*, LNCS 2619, p. 363–378. Springer, 2003.
4. T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th Intern. Conf. on Computer Aided Verification, CAV'01*, p. 260–264, 2001.
5. M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. 28th Intern. Coll. on Automata, Languages and Programming, ICALP'01*, LNCS 2076, p. 652–666. Springer, 2001.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory, CONCUR'97*, LNCS 1243, p. 135–150, 1997. Springer, 1997.
7. J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295 – 311, 1969.
8. T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proc. 29th Intern. Coll. on Automata, Languages and Programming, ICALP'02*, LNCS 2380, p. 704–715. Springer, 2002.
9. T. Cachat. *Two-Way Tree Automata Solving Pushdown Games*, p. 303–317. LNCS 2500. Springer, 2002.
10. A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *Proc. 14th Intern. Conf. on Computer Aided Verification, CAV '02*, LNCS 2404, p. 428–441. Springer, 2002.
11. H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. 9th ACM Conf. on Comp. and Comm. Security*, 2002.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th Intern. Conf. on Computer Aided Verification, CAV '00*, LNCS 1855, p. 232–247. Springer, 2000.
13. O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th Intern. Conf. on Computer Aided Verification, CAV '00*, LNCS 1855, p. 36–52. Springer, 2000.
14. O. Kupferman, M. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
15. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, p. 46 – 77, 1977.
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.
17. W. Thomas. Infinite games and verification. In *Proc. 14th Intern. Conf. on Computer Aided Verification, CAV'02*, LNCS 2404, p. 58–64. Springer, 2002.
18. M. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Intern. Coll. on Automata, Languages, and Programming, ICALP'98*, LNCS 1443, p. 628–641. Springer, 1998.
19. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1 – 37, 1994.
20. I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, January 2001.