Modular Strategies for Recursive Game 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 in sequential programs with recursive procedure calls. While such games can be viewed as the pushdown games studied in the literature, the natural notion of winning in our framework requires the strategies to be modular with only local memory; that is, 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. While reachability in (global) pushdown games is known to be EXPTIME-complete, we show reachability in modular games to be NP-complete. We present a fixpoint computation algorithm for solving modular games such that the worst-case number of iterations is exponential in the total number of returned values from the modules. If the strategy within a module does not depend on the global history, but can remember the history of the past invocations of this module, that is, if memory is local but persistent, we show that reachability becomes undecidable.

1 Introduction

The original motivation for studying games in the context of formal analysis of systems comes from the controller synthesis problem. Given a description of the system where some of the choices depend upon the input and some of the choices represent uncontrollable internal non-determinism, designing a *controller* that supplies inputs to the system so that the product of the controller and the system satisfies the correctness specification corresponds to computing winning strategies in two-player games. This question has been studied extensively in the literature (see [5, 15, 10] for sample research and [19] for a survey). Besides the long-term dream of synthesizing correct programs from formal specifications, games are relevant in two different contemporary contexts. First, model checking for branching-time logics such as μ-calculus, as well as several procedures that

⋆ This research was supported in part by ARO URI award DAAD19-01-1-0473, NSF CAREER award CCR97-34115, and NSF award ITR/SY 0121431. The second author was also supported by the MIUR in the framework of the project "Metodi Formali per la Sicurezza" (MEFISTO) and MIUR grant 60% 2002.

⋆⋆ For the details of the proofs of this paper we refer the reader to the technical report available at the URL: "`http://www.cis.upenn.edu/~madhusud/`".

use tree automata emptiness for deciding various logics, can be reduced to solving games [9, 18]. Second, games have been shown to be relevant for verification of open systems. For instance, the *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 [11]; 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 [8].

In traditional model checking, the model is a finite state machine whose vertices correspond to states, and whose edges correspond to transitions. To define two-player games in this model, the vertices are partitioned into two sets corresponding to the two players, where a player gets to choose the transition when the current state belongs to its own partition [1]. In this paper, we consider the richer system model of *recursive state machines* (RSMs), in which vertices can either be ordinary states or can correspond to invocations of other state machines in a potentially recursive manner. Recursive state machines can model the control flow in typical sequential imperative programming languages with recursive procedure calls.

More precisely, 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, we partition the nodes into two sets such that a player gets to choose the transition when the current node belongs to its own partition. We focus on solving the *reachability* game, that is, deciding if one of the players has a strategy to force the system starting from a specified node to enter one of the target nodes.

Due to recursion, the underlying global state-space is infinite and behaves like a *pushdown* system. While reachability in pushdown games is already studied [20, 6], we are interested in developing algorithms for games on RSMs for two reasons. First, RSMs is a more natural model of recursive systems, and studying reachability (without games) on RSMs has led to refined bounds on complexity in terms of parameters such as the number of entry and exit nodes of modules [1]. Second, 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. The first contribution of the paper is the notion of *modular* strategies for games on RSMs. 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

[1] More interesting forms of interactions between the two players are possible, for instance, see alternating transition systems [2], but we will use a simple game model for this paper.

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. Clearly, there are cases where there is no modular strategy, but there is a global one. Recent work on the interface compatibility checking for software modules implements the global games on pushdown systems [7], but we believe that checking for existence of modular strategies matches better with the intuition for compatibility.

After formulating the notion of modular strategies, we show that deciding existence of modular winning strategies for reachability games is NP-complete. In contrast, global reachability games are EXPTIME-complete [20]. Then, we proceed to formulate a fixpoint computation algorithm that generalizes the symbolic solution to reachability games. For ordinary game graphs, the fixpoint algorithm, starting with the target vertices, iteratively grows the set of vertices from which winning is ensured. In our case, when a node is found to be winning, the algorithm also needs to keep track of the strategies within different modules that were used. This labeling is needed to make sure that the same set of module strategies is used consistently everywhere to ensure modularity. It turns out that the only relevant information about a strategy used within a module is the set of exit nodes of the module that the strategy can restrict the game into. Consequently, in the worst-case, the number of iterations of our fixpoint computation is exponential in the number of exit nodes of the modules.

We also consider *safety* winning conditions for recursive game graphs. Since we restrict to modular strategies of the protagonist, on recursive game graphs safety is not dual to reachability. We prove that determining the existence of a modular strategy for a safety recursive game is also NP-complete.

Finally, we consider the case when the strategy within a module is required to have only local memory, and does not depend on the global history, but this memory can be persistent, and can remember the history of the past invocations of this module. In this case, we prove the reachability games to be undecidable by a reduction from the undecidability of multi-player games with incomplete information [14].

Related work: We have already explained the relation to the global games on pushdown systems [4, 6, 20]. The notion of modular strategies may remind the reader of games with partial information, but this is technically quite different from the standard notion of partial information, and, in fact, lowers the complexity class of the decision problem, while introducing partial information typically adds an exponential to the complexity. Another context where modular strategies have been studied is in the realm of concurrent or distributed games where the intention is to come up with distributed controllers for a system (see [16, 12, 17] and references therein). In that setting, however, looking for modular strategies quickly leads to undecidability. There are some restricted architectures that are decidable, a prominent one being the *hierarchic architectures* [14, 16]. Our problem, however, is quite different from these works since in our setting the control is always in one module, while in the concurrent setting, control can be in several modules at any given time.

2 Games on Recursive Graphs

In this section we introduce recursive games and the decision problem we wish to solve. We start recalling the notion of flat games which are the standard games on And-Or graphs.

2.1 Flat game graphs

A *flat game graph* is a tuple $G = \langle V, V_0, V_1, \gamma \rangle$ where V is a finite set of vertices, V_0 and V_1 define a partition of V, and $\gamma : V \to 2^V$ is a function giving for each vertex $u \in V$ the set of its *successors* in G. The game is played by two players, player 0 (the *protagonist*) and player 1 (the *adversary*). For $p = 0, 1$, the vertices in V_p are those from which only player p can move and the allowed moves are given by the function γ. A *play* of a game graph G is a (finite or infinite) path in G constructed in turns by the two players. A play $u_0 u_1 \ldots$ starting at a vertex u_0 is constructed as follows: if $u_0 \in V_p$, player p chooses a successor vertex u_1; at each step j, player p (where $u_j \in V_p$) chooses a successor vertex u_{j+1}.

A *strategy* for player p is a function $f : V^* \to V$ mapping sequences (and hence plays) to vertices. The idea is that when a play πu has been played, where $u \in V_p$, the strategy f recommends the move $f(\pi u)$. A play $\pi = u_1 u_2 \ldots$ is *according to* f if for every $j < |\pi|$ such that $u_j \in V_p$, $f(u_0 \ldots u_j) = u_{j+1}$. When a strategy for player p depends only on the current vertex of a play, i.e., if $f(\pi x) = f(\pi' x)$, for all plays π, π' and $x \in V_p$, it is called a *memoryless strategy*. A play π according to a strategy is said to be *maximal* if it cannot be continued (i.e. if π is infinite or there is no $v \in V$ such that πv is a play according to the strategy).

Since we are interested in *reachability* games in this paper, we have a *winning condition* for the game given by a subset of *target* vertices X. A flat reachability game is then a tuple $\langle G, v_0, X \rangle$ where G is a flat game graph, v_0 is the initial vertex where the plays start, and X, a subset of the vertices, is the target set. A play is winning for the protagonist if it contains a vertex in X (i.e. a play $\pi = u_0 u_1 \ldots$, where $u_0 = v_0$, is winning if there is an $i < |\pi|$ such that $u_i \in X$). A strategy for the protagonist is winning if all maximal plays according to it are winning, while a strategy for the adversary is winning if all maximal plays according to it are not winning. Hence, a winning strategy for the protagonist is one which forces the play to the target set X, no matter how the adversary chooses. Flat reachability games are PTIME-complete and if a player has a winning strategy, it also has a memoryless winning strategy [13].

2.2 Recursive game graphs

In this paper, our main objective is to study reachability games in hierarchical and recursive graph structures that are defined using several interacting component game graphs (or game modules). Our model is a generalization to game graphs of the recursive state machines model defined in [1].

$\mathbf{A_1}$

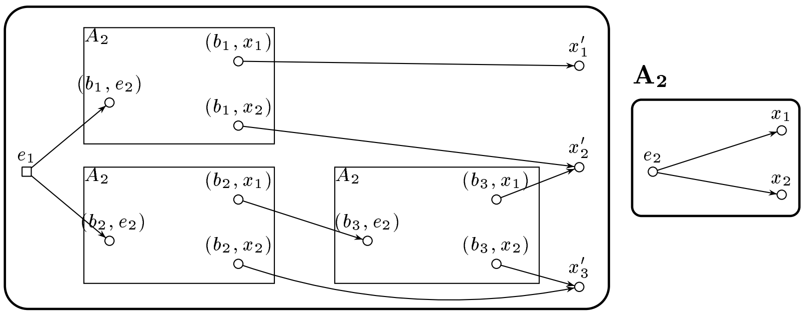

Fig. 1. A recursive game graph

A *recursive game graph* A is given by a tuple $\langle A_1, \ldots, A_n \rangle$, where each *game module* $A_i = (N_i, B_i, V_i^0, V_i^1, Y_i, En_i, Ex_i, \delta_i)$ consists of the following components:

- A set of nodes N_i which we call the *nodes* of module A_i.
- A nonempty set of *entry* nodes $En_i \subseteq N_i$ and a nonempty set of *exit* nodes $Ex_i \subseteq N_i$.
- A set of *boxes* B_i.
- Two disjoint sets V_i^0 and V_i^1 that partition the set of nodes and boxes into two sets, i.e. $V_i^0 \cup V_i^1 = N_i \cup B_i$ and $V_i^0 \cap V_i^1 = \emptyset$. The set V_i^0 (V_i^1) denotes the places where it is the turn of player 0 (resp. player 1) to play.
- A labeling $Y_i : B_i \rightarrow \{1, \ldots, n\}$ that assigns to every box an index of the game modules $A_1, \ldots, A_n$.
- Let $\mathit{Calls}_i = \{(b, e) \mid b \in B_i, e \in En_j, j = Y_i(b)\}$ denote the set of *calls* of module A_i and let $\mathit{Retns}_i = \{(b, x) \mid b \in B_i, x \in Ex_j, j = Y_i(b)\}$ denote the set of *returns* in A_i. Then, $\delta_i : N_i \cup \mathit{Retns}_i \rightarrow 2^{N_i \cup \mathit{Calls}_i}$ is a *transition function*.

We also refer to entry (exit) nodes simply as entry (exit). Nodes of N_i, for any i, which are in V_i^p are called p-nodes while returns of the form (b, u), where $b \in V_i^p$, for some i, are called *p-returns*. An element in Calls_i of the form (b, e) represents a call from A_i to the module A_j, where $j = Y_i(b)$ and e is an entry of A_j. An element in Retns_i of the form (b, x) corresponds to the associated return of control from A_j to A_i when the call exits from A_j at exit x. The transition function hence defines moves from nodes and returns to a set of nodes and calls.

To illustrate the definitions, consider the example shown in Figure 1. It comprises two modules A_1 and A_2, where A_1 has three boxes that invoke A_2. The only adversary node is e_1, the initial node of A_1.

We make some assumptions of these graphs in the sequel that enable a more readable presentation (these will turn out to be without loss of generality for the problems we consider):

– There is only *one* entry point to every module, i.e. $|E_i| = 1$ for every i. We refer to this unique entry point of A_i as e_i.
– For every $u \in N_i$, $e_i \notin \delta_i(u)$ holds, and for every $x \in Ex_i$, $\delta_i(x)$ is empty. That is, there are no transitions from a module to its own entry nodes and no transitions from its exits.
– The nodes and boxes of all agents are disjoint. Let $B = \bigcup_i B_i$ denote the set of all boxes and $N = \bigcup_i N_i$ denote the set of all nodes. We extend the functions $\{Y_i\}_{i=1}^n$ to a single function $Y : B \to \{1, \ldots, n\}$.

Note that the above definition allows recursive calls—a module can call itself directly or indirectly. We say that a recursive game graph is *hierarchic* if this cannot happen. Formally, a recursive game graph $\langle A_1, \ldots A_n \rangle$ is hierarchic if there is an ordering $\leq$ on the modules such that for every A_i, A_i has no calls to any A_j where $A_j \leq A_i$ (i.e. there is no box b in A_i such that $A_{Y(b)} \leq A_i$). For example, the game graph in Figure 1 is hierarchic.

To define the notions of play and strategy for a recursive game graph, we first give the semantics of our model by defining a flat game graph associated with it. This is similar to the way one associates a flat model that describes the behavior of a recursive state machine. A (global) *state* of a recursive game graph $A = \langle A_1, \ldots, A_n \rangle$ is a tuple $\langle \bar{b}, u \rangle$ where $\bar{b} = b_1, \ldots, b_r$ is a finite (perhaps empty) sequence of boxes from B, and u is a node in N. Consider a state $\langle b_1, \ldots, b_r, u \rangle$ such that $b_i \in B_{j_i}$ for $1 \leq i \leq r$ and $u \in N_j$. Such a state is well-formed if $Y(b_i) = j_{i+1}$ for $1 \leq i < r$, and if $r \geq 1$, $Y(b_r) = j$. If τ denotes the empty sequence, note that any state of the form $\langle \tau, u \rangle$ is well-formed; we henceforth use $\langle u \rangle$ to denote $\langle \tau, u \rangle$. Intuitively, a well-formed state $\langle b_1, b_2, u \rangle$ denotes the configuration where b_1 is a call to a module which in turn called another module using b_2 in which the current node is u. When the last module exits, the control goes back to the corresponding return of b_2, and so on. Henceforth, we assume states to be well-formed and denote by Q_A the set of global states of A.

According to the partition of nodes and boxes in the recursive game graph, the states are also classified as protagonist and adversary states. For $p = 0, 1$, a state $\langle b_1, \ldots, b_r, u \rangle$ is a p-state if either u is not an exit and is a p-node, or, (b_r, u) is a p-return. (If $r = 0$ and u is an exit, the definition will not matter since there are no transitions from it; we hence choose these states to be, say, 0-states.) We denote by Q_p the set of p-states.

The *global game graph* corresponding to a recursive game graph A is $G_A = (Q_A, Q_0, Q_1, \delta)$ where the global transition function δ is given as follows. Let $s = \langle b_1, \ldots, b_r, u \rangle$ be a state with $u \in N_j$ and $b_r \in B_m$. Then, $s' \in \delta(s)$ provided one of the following holds:

1. $u' \in \delta_j(u)$ for a node u' of A_j, and $s' = \langle b_1, \ldots, b_r, u' \rangle$.
2. $(b', e) \in \delta_j(u)$ for a box b' of A_j, and $s' = \langle b_1, \ldots, b_r, b', e \rangle$.
3. u is an exit node of A_j, $u' \in \delta_m(b_r, u)$ for a node u' of A_m, and $s' = \langle b_1, \ldots, b_{r-1}, u' \rangle$.
4. u is an exit node of A_j, $(b', e) \in \delta_m(b_r, u)$ for a box b' of A_m, and $s' = \langle b_1, \ldots, b_{r-1}, b', e \rangle$.

The first case above is when the control stays within the module A_j, the second case is when a new module is entered via a box of A_j, the third is when the control exits A_j and returns to A_m, and the last case is when the control exits A_j and immediately enters a new module via a box of A_m. Note that the set of reachable states from, say, a state $\langle e_i \rangle$ ($e_i \in En_i$) could be infinite if the recursive game graph is not hierarchic (if it is hierarchic, the global reachable graph is finite).

A *recursive game* is a tuple $\langle A, e_1, X \rangle$ where $A = \langle A_1, \ldots A_n \rangle$ is a game graph, e_1 is the entry node of A_1, and $X \subseteq Ex_1$ is the target set, which is a subset of the exits of A_1. The *global recursive game* corresponding to $\langle A, e_1, X \rangle$ is the flat game $\langle G_A, \langle e_1 \rangle, X' \rangle$ where X' is the set of all global states $s = \langle b_1, \ldots, b_r, u \rangle$ where $u \in X$. A (winning) *global strategy* in a recursive game is a (winning) strategy in the global recursive game.

If we use the notion of global strategies, it will lead us to the definition of games equivalent to pushdown games [20]. One can show that for every recursive game, there is a polynomial-sized pushdown automaton whose configuration game graph (as in [20]) is isomorphic to the global graph of the recursive game, and vice-versa. We depart from pushdown games at this point in that we require a particular kind of strategy (namely a modular strategy) that wins these games. In the next section, we introduce this class of strategies and define the corresponding decision problem.

2.3 Modular strategies

We are interested in *modular strategies* for the protagonist in each game module such that when the strategies are put together, the protagonist wins the game. Such strategies are restricted in that they can only refer to the "local history" of each module, that is, the portion of the play corresponding to the "current" invocation of the module.

To define formally a modular strategy for a recursive game graph A, we introduce some notation. For a play π, the control after π is in A_i if the current node is in A_i; however, if the current node is an exit node, then the control is in the module that made the last call. Formally, for a play πs, we say that the *control after πs is in A_i* if $s = \langle b_1, \ldots, b_r, u \rangle$, with $u \in N_i \backslash Ex_i$ or $(b_r, u) \in Retns_i$. Note that the control after a play can be in at most one module.

Consider a play $\pi = s_0 s_1 \ldots s_k$ and let the control after π be in A_i. Let $s_k = \langle b_1, \ldots, b_r, u \rangle$. Then we define the current stack of π to be $\beta(\pi) = \langle b_1, \ldots, b_r \rangle$, if $u \notin Ex$ and $\beta(\pi) = \langle b_1, \ldots, b_{r-1} \rangle$, otherwise. Now, let j be the *largest* index $0 \leq j \leq k$ such that $s_j = \langle \beta(\pi), e_i \rangle$. Intuitively, s_j corresponds to the activation of A_i that led to s_k. The states $s_{j'}$, where $j \leq j' \leq k$ are all of the form $\langle \beta(\pi), b'_1, \ldots, b'_{r'}, u' \rangle$, for some $r' \geq 0$. Note that there may be states $s_{j'}$, $j < j' \leq k$ such that $s_{j'} = \langle \beta(\pi), b'_1, \ldots, b'_{r'}, e_i \rangle$, with $r' \geq 1$, which denote recursive entries into A_i, but which have returned before s_k. We will now be interested in $\alpha(\pi) = s_j \ldots s_k$, the suffix of π from s_j.

What we want to do now is to project $\alpha(\pi)$ to the nodes, calls and returns in A_i, discarding fragments of runs in modules called from A_i. To do this, let us define a projection function ρ^i_β, for a sequence $\beta = \langle b_1, \ldots, b_r \rangle$, of any state s as follows: if $s = \langle \beta, u \rangle$, where $u \in N_i$, then $\rho^i_\beta(s) = u$; if $s = \langle \beta, b, u \rangle$, where $(b, u) \in \mathit{Calls}_i \cup \mathit{Retns}_i$, then $\rho^i_\beta(s) = (b, u)$; in all other cases, $\rho^i_\beta(s) = \varepsilon$, the empty word. We extend ρ^i_β to sequences of states: $\rho^i_\beta(s'_1 \ldots s'_l) = \rho^i_\beta(s'_1) \ldots \rho^i_\beta(s'_l)$.

We can now define a function μ_i that extracts the *local memory* from the play π: $\mu_i(\pi) = \rho^i_{\beta(\pi)}(\alpha(\pi))$. Thus the local memory of a play π ending in a state s_k stands for the fragment of the play in the current module that gives the sequence of nodes, calls and returns in A_i that led to the state s_k, ignoring the sub-plays in called modules (including recursive calls to itself).

A *modular* strategy f for player 0 is a strategy for player 0 such that for all plays π, π' of A, if the control after π and π' are both in A_i and $\mu_i(\pi) = \mu_i(\pi')$ then $f(\pi) = f(\pi')$ holds. In other words, the advice of the strategy f for any play π which is currently in A_i depends only on the local memory $\mu_i(\pi)$.

Since a modular strategy depends only on the local memories, we can alternatively view the modular strategy as a set of strategies, one for each game module. A strategy for a game module A_i is a function $f_i : (N_i \cup \mathit{Calls}_i \cup \mathit{Retns}_i)^* \to (N_i \cup \mathit{Calls}_i)$. A *local strategy* $\widehat{f}$ is $\{f_i\}_{i=0}^n$ where f_i is a strategy for A_i. A play π is according to $\widehat{f}$ if for every prefix $\pi' s s'$ of π, if the control after $\pi' s$ is in A_i and s is in Q_0, then the following holds: let $\mu_i(\pi' s) = w$, then $\mu_i(\pi s s') = wu$ where $u = f_i(w)$.

For a modular strategy f, we can associate with it a local strategy. Let π be a play consistent with f. Set $\beta = \beta(\pi)$ and let the control after π be in A_i. Then, $f_i(\mu_i(\pi)) = \rho^i_\beta(f(\pi))$ (the function f_i on other values can be defined arbitrarily). Then it is easy to see that the plays according to f are precisely the plays according to the local strategies. Conversely, given a local strategy $\widehat{f} = \{f_1, \ldots, f_n\}$, we can associate a modular strategy f with it. Let πs be a play consistent with $\widehat{f}$, let the control after πs be in A_i, and let $\beta(\pi s) = \beta$. Then $f(\pi s) = s'$ where s' is the unique successor of s such that $\rho^i_\beta(s') = f_i(\mu_i(\pi s))$. Again, it is easy to see that f is modular and that the sets of plays according to f and $\widehat{f}$ are the same. In the sequel, we freely switch between modular strategies and the corresponding local strategies.

We consider the following decision problem.

> Given a recursive game $\langle A, e_1, X \rangle$, is there a modular winning strategy (or equivalently, a local strategy that is winning) for the protagonist?

Consider the game graph in Figure 1. Note that the only place where the protagonist has a choice is in picking the move from e_2. If the target set is $\{x'_2, x'_3\}$ then the protagonist has a winning modular strategy where it chooses to move to the exit node x_2 in A_2. For the target set $\{x'_1, x'_3\}$, there is no modular strategy for the protagonist that is winning. However, it is easy to see that there is a global winning strategy.

The idea behind modular strategies is that it is more appropriate to look for strategies for the modules rather than a monolithic strategy. However, rather than allowing a strategy for a module A_i to remember only the play from the *last* call to A_i, we could allow the strategy to remember *all* parts of the play when it was inside A_i. That is, we could allow strategies to have a *persistent* memory where it is allowed to remember how the play evolved in all the previous calls to the module. For example, in the recursive game in Figure 1, though there is no modular strategy for the protagonist for the target set $\{x_1', x_3'\}$, there is a persistent strategy (the strategy for A_2 picks x_1 when it is first called and picks x_2 on the second call). Checking for persistent strategies, however, turns out to be undecidable (see Section 5 for details).

From now on, when the context is clear, we use the term *strategy* to mean modular strategies.

3 Solving recursive games

Let us fix a reachability game $\langle A, e_1, X\rangle$ for the rest of this section, where $A = \langle A_1, \ldots, A_k\rangle$, and each $A_i = (N_i, B_i, V_i^0, V_i^1, Y_i, En_i, Ex_i, \delta_i)$.

Consider f, a modular strategy for $\langle A, e_1, X\rangle$. The key to deciding recursive games is the observation that whether f is winning or not is primarily determined by finding, for each A_i, the set of exit nodes of A_i which the local strategy f_i will lead a play entering A_i to. Let X_i^f denote the set of exits a play can reach if it enters A_i and continues according to f; that is, an exit point $x \in Ex_i$ is in X_i^f if there is a play according to f of the form $\pi\langle \bar{b}, e_i\rangle\pi'\langle \bar{b}, x\rangle$. Since the strategy f is modular, such exits x are the ones for which there is a play according to f of the form $\langle e_i\rangle\pi''\langle x\rangle$. In fact, if we take a winning strategy $\widehat{f}$ and replace an f_i in $\widehat{f}$ with a different strategy f_i' which calls the same modules that f_i calls and leads to the same set of exit nodes of A_i, then this will also be a winning strategy. This motivates the following definitions.

For a modular strategy f for $\langle A, e_1, X\rangle$, the *call graph of f* is $C_f = (V, \rightarrow, \lambda)$ such that $(V, \rightarrow)$ is a graph where:

- $V \subseteq \{A_1, \ldots, A_n\}$ is the set of all A_i such that there is a play according to f that enters A_i.
- $A_i \rightarrow A_j$ iff there is some play according to f from $\langle e_1\rangle$ where there is a call from A_i to A_j (i.e. there is a play of the form $\langle e_1\rangle\pi\langle b_1, \ldots, b_r, e\rangle$ with $b_r \in B_i$ and $Y(b_r) = A_j$).
- For every $A_i \in V$, let $\lambda(A_i) = X_i^f$.

We first make a simple observation:

Lemma 1. *Let f be a winning strategy for $\langle A, e_1, X\rangle$ and let $C_f = (V, \rightarrow, \lambda)$ be the call graph of f. Then, $(V, \rightarrow)$ is acyclic.*

Proof. If f is a strategy whose call graph has a cycle, then one can show that there is a play according to f that makes calls to the modules recursively according to the cycle in the call graph, and hence never reaches the target set. Such an f will not be winning. □

For a strategy f on $\langle A, e_1, X\rangle$, we say that f is hierarchic if the plays according to f make no recursive calls (i.e., no play according to f reaches a state of the form $s = \langle b_1, \ldots, b_i, u\rangle$, where $\exists l \in \{1, \ldots, n\}$ such that $u \in N_l$ and $b_j \in N_l$, for some $j \in \{1, \ldots i\}$). The following is immediate from the above lemma:

Corollary 1. *Reachability games admit only hierarchic winning strategies.*

We recall that the target set X is a subset of the exits of module A_1. An interesting consequence of the above result is that when we consider modular strategies, the only global states of the target set which can be reached on a winning modular strategy are $\langle x \rangle$, $x \in X$.

Motivated by the above lemma, we give a general definition of a call graph: a *call graph* is a tuple $C = (V, \rightarrow, \lambda)$ where $V \subseteq \{A_1, \ldots, A_n\}$, $(V, \rightarrow)$ is an acyclic graph and $\lambda(A_i) \subseteq Ex_i$, for each $A_i \in V$.

Let $C = (V, \rightarrow, \lambda)$ be a call graph for the game $\langle A, e_1, X \rangle$. Let A_i be a module of the game. We now define a game graph A_i^C which is a flat game graph associated with A_i and C, where, intuitively, we replace each call (b, e_j) to a module A_j by a vertex where player 1 can take the game to any return (b, x_j) where x_j is in $\lambda(A_j)$. In other words, we are defining a game graph under the assumption that a call to a module A_j could result in returns corresponding to $\lambda(A_j)$ and we want to solve the game for A_i under these assumptions. The game graph A_i^C will also prohibit any calls to modules that it is not supposed to call, in accordance with the call graph C.

Formally, A_i^C is defined as follows: $A_i^C = (S_i, S_i^0, S_i^1, \gamma_i)$ where

- $S_i = N_i \cup Calls_i \cup Retns_i$.
- $S_i^0 = (V_i^0 \cap N_i) \cup \{(b, x) \in Retns_i \mid b \in V_i^0\}$.
- $S_i^1 = (V_i^1 \cap N_i) \cup \{(b, x) \in Retns_i \mid b \in V_i^1\} \cup Calls_i$.
- The transition function γ_i is defined as follows:
  1. If $u \in N_i \cup Retns_i$, $\gamma_i(u) = \delta_i(u)$.
  2. If $(b, e) \in Calls_i$ and $A_{Y(b)}$ is a successor of A_i in C, then $\gamma_i((b, e)) = \{(b, x) \mid x \in \lambda(A_{Y(b)})\}$.
  3. If $(b, e) \in Calls_i$ and $A_{Y(b)}$ is not a successor of A_i in C, then $\gamma_i((b, e)) = \emptyset$.

The graph A_i^C is thus obtained by taking the vertices as the nodes, calls and returns of A_i. The nodes and returns are partitioned into 0-nodes and 1-nodes as in A_i. Also, the calls are all deemed to be 1-nodes. The transition function follows the transition function of A_i for nodes and returns. For a call (b, e), the transition function maps it to an empty set if A_i is not permitted to call $A_{Y(b)}$ according to the call graph C. Note that if a play reaches such a call, then it is maximal and hence losing for player 0. If A_i is permitted to call $A_{Y(b)}$, then we take the expected set of exit nodes $\lambda\left(A_{Y(b)}\right)$ from the call graph and have edges to each of the returns corresponding to these exits.

We can now state the main result for which we have developed the definitions above:

Lemma 2. *There exists a winning strategy for $\langle A, e_1, X \rangle$ if and only if there exists a call graph $C = (V, \rightarrow, \lambda)$ such that $\lambda(A_1) \subseteq X$ and for every $A_i \in V$, player 0 wins the reachability game $\langle A_i^C, e_i, \lambda(A_i) \rangle$.*

Proof. If f is a winning strategy for $\langle A, e_1, X\rangle$, we show that its call graph C_f satisfies the requirements of the lemma. In fact, the strategy f_i for A_i itself serves as a winning strategy in $\langle A_i^{C_f}, e_i, \lambda(A_i)\rangle$, for each A_i in the call graph C_f.

Conversely, if there is a call graph $C = (V, \rightarrow, \lambda)$ that satisfies the properties of the lemma, then the strategy for any $A_i^C \in V$ serves as a strategy for A_i in the recursive game and these strategies in fact constitute a winning strategy. $\square$

We say that a local strategy $\{f_i\}_{i=1}^n$ is *memoryless* if for every pair of sequences $\sigma u, \sigma' u \in (N_i \cup Calls_i \cup Retns_i)^*$, $f_i(\sigma u) = f_i(\sigma' u)$. That is, if the local strategies' selection depends only on the current node of the play.

As a corollary to the above lemma we have:

Corollary 2. *If there is winning strategy for a recursive game, then there is a local memoryless winning strategy for it.*

We can now show that solving reachability games is NP-complete:

Theorem 1. *Reachability on recursive (as well as hierarchic) game graphs is* NP*-complete.*

Proof. The NP procedure works as follows. First, we guess a calling graph $C = (V, \rightarrow, \lambda)$ such that $A_1 \in V$ and $\lambda(A_1) \subseteq X$, where X is the target set. Then, for every module A_i, we check if there is a winning strategy for player 1 in $\langle A_i^C, e_i, \lambda(A_i)\rangle$. If all the games are winning for player 0, we report that player 0 wins the recursive game. It is easy to see that this works in polynomial time. Correctness follows from Lemma 2.

For the lower bound, we reduce the satisfiability problem of 3-CNF formulas to solving a reachability game on hierarchic game graphs. The intuition is that there is a module for each variable, where player 0 has to pick a valuation for the variable by picking an exit point. The initial module enables player 1 to pick any clause and call a module corresponding to that clause. In the module for each clause, player 0 points to a literal that witnesses the clause satisfaction by calling the module of the corresponding variable. Player 0 wins if all clauses are satisfied. Since a modular strategy picks essentially a valuation of the variables independent of the context in which it is called, player 0 wins iff the formula is satisfiable. $\square$

4 A labeling algorithm

In this section, we describe an exponential-time realization of the nondeterministic procedure sketched in the previous section. The algorithm we present is an extension of the usual *attractor set* construction on flat graphs [13] adapted to our setting, and, it computes the vertices where player 0 can win in an incremental on-the-fly fashion.

Our algorithm iteratively labels vertices of a recursive game graph with tuples of sets of exit nodes according to some initialization and update rules. The

Algorithm Reach

Initially, each exit $u \in X$ is labeled by tuples $\langle E_1, \ldots, E_n \rangle$, where $E_1 = \{x\}$ and $E_i = \top$, for every $i > 1$. All the other nodes, calls, and returns are unlabeled.

Labels are updated according to the following rules:

Rule 1: For a 0-node (or a 0-return) v of A_i, if $\langle E_1, \ldots, E_n \rangle$ labels $v' \in \delta_i(v)$ then add $\langle E_1, \ldots, E_n \rangle$ to labels of v.

Rule 2: For a 1-node (or a 1-return) v of A_i, $\delta_i(v) = \{v_1, \ldots, v_k\}$, if (a) $\langle E_1^h, \ldots, E_n^h \rangle$ labels v_h for $h = 1, \ldots, k$, and (b) for every $l \neq i$, $E_l^1, \ldots, E_l^k$ are pairwise consistent, then add $\langle E'_1, \ldots, E'_n \rangle$ to labels of v, where $E'_j = \bigcup_{h=1}^{k} E_j^h$ for $j = 1, \ldots, n$.

Rule 3: For a return (b, x) of A_i labeled by $\langle E_1, \ldots, E_n \rangle$ where $Y_i(b) = j$, add $\langle E'_1, \ldots, E'_k \rangle$ to labels of x where $E'_j = \{x\}$ and $E'_l = \top$ for $l \neq j$.

Rule 4: For a call (b, e) of A_i such that $Y_i(b) = j$, let $(b, x_1), \ldots, (b, x_k)$ be any k distinct returns from (b, e). Suppose that for $h = 1, \ldots, k$, $\langle E_1^h, \ldots, E_n^h \rangle$ labels (b, x_h) and $\langle E_1^0, \ldots, E_n^0 \rangle$ labels e in A_j. If $E_i^0 = \top$, $E_j^0 = \{x_1, \ldots, x_k\}$, and $E_l^0, \ldots, E_l^k$ are pairwise consistent for $l \neq i$, then add $\langle E_1, \ldots, E_n \rangle$ to labels of (b, e), where $E_l = \bigcup_{h=0}^{k} E_l^h$ for $l = 1, \ldots, n$.

The algorithm halts when there are no more labels that can be added. Then, it gives an affirmative answer if and only if e_1 is labeled with a tuple $\langle E_1, \ldots, E_k \rangle$ such that $E_1 \subseteq X$.

Fig. 2. Algorithm Reach.

algorithm halts when the computed labeling reaches its fixed-point, i.e., there are no new labels that can be added.

Let $A = \langle A_1, \ldots, A_n \rangle$ be a recursive game graph. Consider the reachability game $\langle A, e_1, X \rangle$. We use a special symbol $\top$ and we overload the set union operator with the rule $\top \cup E = E$, for any set E. We use v to denote a node, a call, or a return of A_j, for any j. We describe now the algorithm Reach (Figure 2) that solves the modular reachability problem. Algorithm Reach consists of labeling iteratively each v, with tuples of the form $\langle E_1, \ldots, E_n \rangle$, where each E_i is either a subset of Ex_i or is the symbol $\top$. Each v could be labeled at any time with a *set* of such labels.

The reason we need to keep these labels, as opposed to just a set in the attractor-set computation, is to ensure that the strategy we construct is modular. Since the exit nodes which a strategy f_i takes us to in A_i is the important information we need to know, these labels will ensure that we can consistently pick a single strategy f_i for each A_i in the end.

For a set of maximal finite plays Π, let $Final(\Pi) = \{s \mid$ there is a play of the form $\pi s \in \Pi\}$. Intuitively, $Final(\Pi)$ is the exact set of states the plays in Π end in.

When $v \in N_i \cup Calls_i \cup Retns_i$ gets a label $\langle E_1, \ldots, E_n \rangle$, it is supposed to mean that there exists a local strategy $\widehat{f} = \{f_i\}$ such that the following hold:

A1. If Π_v is the set of all maximal plays starting at $\langle v \rangle$ and consistent with $\widehat{f}$, then Π_v contains only finite plays and $Final(\Pi) = \{\langle x_i \rangle \mid x_i \in E_i\}$. Further, the plays in Π_v do not enter any A_j, where $E_j = \top$, nor does it re-enter A_i.

A2. Let l be such that $E_l \neq \top$ and $l \neq i$. Let Π_l be the set of maximal plays starting at $\langle e_l \rangle$ and consistent with $\widehat{f}$. Then Π_l consists of only finite plays and $Final(\Pi) = \{\langle x_l \rangle \mid x_l \in E_l\}$. Further, the plays in Π_l never enter any module A_j, where $E_j = \top$, nor does it enter A_i.

Intuitively, when a node in A_i is labeled with $\langle E_1, \ldots, E_n \rangle$, it signifies that there is a set of strategies for each module A_j, where $E_j \neq \top$, such that the strategies drive the play from the current node to some node in E_i. Moreover, these strategies never make the play enter a module $A_{j'}$ for which strategies are not assumed (i.e. where $E_{j'} = \top$). Also, the strategy for A_j drives any play entering it to the set E_j. Finally, the strategies are guaranteed not to call A_i

For any $E', E'' \in 2^{Ex_i} \cup \{\top\}$, we say that E' and E'' are *consistent* if $E' = \top$, or $E'' = \top$, or $E' = E''$.

Rule 1 makes a 0-node or 0-return u inherit a label of a successor. The idea is that there is a strategy from u, where the protagonist picks this successor and plays then according to the strategy assured at the successor, inductively. At a 1-node or a 1-return u, the protagonist has no choice—it can be taken to any of the successor nodes. The rule hence labels u only if all the successors agree upon the exit-nodes that they assume for all components. Rule 3 activates an exit of a module when a return corresponding to that exit is activated in some module; the activation is similar to the way we initialized our algorithm by activating the exits corresponding to the target. Calls are labeled using Rule 4: we check whether the assumptions on all modules by both the called module's entry point as well as the returns corresponding to the call are consistent, check whether the called module does not call A_i and label the call by an appropriate label. We prove now that REACH solves the reachability problem for recursive games.

Lemma 3 (Soundness). *Let $A = \langle A_1, \ldots, A_n \rangle$ be a recursive game graph. If a node, a call, or a return v of A_i is labeled by* REACH *with $\langle E_1, \ldots, E_n \rangle$, then there exists a local strategy $\widehat{f} = \{f_i\}$ such that (A1) and (A2) hold.*

Proof. The lemma is proved by induction on the number of applications of the update rules. For the initial labeling the lemma is trivially true. The induction step is fairly straightforward from the inductive hypothesis and the intuition behind the labeling. For example, if Rule 2 is applied to label a node u in A_i, then we can pick strategies for the modules from the set of strategies of its successors whose existence is guaranteed by the induction hypothesis. The only subtle point is that the selection of local strategies needs to be done carefully to ensure that the constructed strategy is not recursive (i.e. their call graph is not recursive). For Rule 2, we pick a strategy for a module A_j ($j \neq i$) as the strategy for A_j at a successor v'; however, we ensure that for every $A_{j'}$ which is called by this strategy, we pick the strategy for $A_{j'}$ using the strategy at v'. □

Lemma 4 (Completeness). *Let $A = \langle A_1, \ldots, A_n \rangle$ be a recursive game graph and $X \subseteq Ex_1$. If there exists a winning strategy for the protagonist in the reach-*

ability game $\langle A, e_1, X\rangle$, then e_1 is labeled by algorithm Reach *with $\langle E_1, \ldots, E_n\rangle$ where $E_1 \subseteq X$.*

Proof. The gist of the proof is as follows. Let f be a winning strategy of the protagonist in the reachability game $\langle A, e_1, X\rangle$, and let $C_f = (V, \rightarrow, \lambda)$ be its call graph. We can then define a finite strategy tree with f that encodes all the plays according to f as labels of paths in the tree. One can also associate each vertex of the tree with a node in the recursive game—namely, the node reached on the play corresponding to the vertex. We then show by induction on this tree, from the leaves to the root, that for every vertex v in the tree, if the node corresponding to it is u in A_i, then u gets a label $\langle E_1^u, \ldots, E_m^u\rangle$ such that: $E_i \subseteq \lambda(A_i)$ and for every $j \neq i$, E_j is consistent with $\lambda(A_j)$, if $A_j \in V$, and $E_j = \top$, for every $A_j \notin V$. The lemma then holds from the fact that the root gets such a label. □

The soundness of Reach follows from Lemma 3, by condition A1 (with $v = e_1$), while Lemma 4 proves its completeness. Reach can require exponential time since it can generate exponentially many labels. A careful analysis shows that it works in fact in time exponential in the total number of exit nodes, exponential in the maximum out-degree of the underlying graphs, but polynomial in the size of the recursive game graph. Also, note that Reach can stop once the initial node gets the appropriate label, even before reaching the fixpoint.

5 Extensions

Safety recursive games: Consider a recursive game graph A. A safety condition requires that the plays stay within a set of *good* vertices, or equivalently that *bad* vertices are avoided. A *safety recursive game* is $\langle A, e_1, X_{good}\rangle$, where A is a recursive game graph, e_1 is the entry point of the game module A_1, and X_{good} is a subset of A nodes (let us assume that all exits are good). A play of such a game is winning if all the visited states are of the form $\langle b_1, \ldots, b_r, x\rangle$ where $x \in X_{good}$. If we restrict to modular strategies, safety recursive games are not dual to reachability recursive games. This is because in both definitions, while the protagonist is forced to use a modular strategy, the adversary is allowed to use an arbitrary strategy. In contrast with reachability, winning modular strategies in safety recursive games may not be hierarchic (in particular Lemma 1 and thus Corollary 1 do not hold). Despite this, deciding safety recursive games is also NP-complete. The hardness result is directly obtained from the reduction given in the proof of Theorem 1. For membership to NP, we can prove that the existence of a winning strategy f can be witnessed by a (possibly cyclic) call graph and a tuple $\langle E_1, \ldots, E_n\rangle$, where E_i is a subset of the exits of A_i, with the meaning that: 1) in each module A_i, we can visit only calls to modules that are successors in the call graph, and 2) the winning local strategy in A_i stays within the safe set X_{good} and it either never exits A_i or it exits through an exit in E_i. Once this is guessed, checking that indeed a strategy is winning reduces to solving safety games on flat graphs whose total size is polynomial in the size of the recursive game graph.

Handling multiple entries: Consider a recursive game $\langle A, e_1, X\rangle$ where each module is allowed to have multiple entry nodes. The semantics of the game and modular strategies are the natural extensions of those for the single-entry setting. When a play enters a module, since the strategy for the module knows the entry point where the module enters, it could follow completely different strategies for the different entry nodes and still remain modular. Hence, we can replace every module which has, say, m entry nodes with a set of m modules, one for each entry point, but where each new module has only one entry. We suitably change the calls from other modules so that they call the corresponding modules with the appropriate entry point. It is easy to see that one can check for a modular strategy on the original game by checking for a modular strategy in this game. Consider a recursive game G with n modules and let m_e be the maximum number of entries to any module. Then, it is easy to see that the above reduction produces a game graph with at most $n \cdot m_e$ modules and the overall size of the game graph is at most $|G|^3$, where $|G|$ is the size of G.

Recursive games with variables: In modeling programs, it is natural to have variables over a finite domain that are abstractions of actual variables and which can be passed from module to module (see, for instance, the SLAM model checker [3]). We can extend our setup to handle this by augmenting nodes with the values of variables. These variables can be local, global or passed as parameters when calls are made to other modules. If a module A_i has r_i input variables, r_o output variables and s internal variables, then we can model this by having $2^{r_i} \cdot |En_i|$ entry nodes, $2^{r_o} \cdot |Ex_i|$ exits, and $2^{r_i+r_o+k} \cdot |N_i|$ internal nodes (we assume all variables be boolean). Note that a modular strategy will be such that the strategy for a module can take into account the parameters that are sent and returned when calling the module, but cannot know the exact evolution of the of the called program. This makes our setting a very natural setup where we can deal with the construction of skeletons of program modules which achieve a particular specification.

Persistent memory strategies: As mentioned in Section 2.3, we can relax the condition of modular strategies and instead ask for a *persistent strategy*, where a strategy for a module can remember not only the play from the last call to the module, but *all* parts of the play when the play entered into this module. The idea is that we can realize the strategy for a module as a program which has a static memory to store all that happens within the module, and use this information to drive the play. However, it turns out that checking whether there is a persistent strategy in a given recursive game is *undecidable*. The reduction is from the undecidability of solving multi-player games with incomplete information [14]. We have:

Theorem 2. *The problem of checking whether there is a winning persistent strategy in a given hierarchical game is undecidable.*

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of the 13th International Conference on Computer Aided Verification, CAV'01*, LNCS 2102, pages 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. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *LNCS*, pages 135–150, Warsaw, July 1997. Springer.
5. J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
6. T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Automata, Languages and Programming, 29th Int'l Coll., ICALP, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *LNCS*, pages 704–715. Springer.
7. A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th Int'l Conf. on Computer-Aided Verification*, LNCS 2404, pages 428–441. Springer, 2002.
8. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.
9. E. A. Emerson. Model checking and the mu-calculus. In N. Immerman and P. Kolaitis, editors, *Proceedings of the DIMACS Symposium on Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society Press, 1997.
10. O. Kupferman and M. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, June 1999.
11. O. Kupferman, M. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
12. P. Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)*, LNCS 2421, pages 145–160. Springer, 2002.
13. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149–184, 1993.
14. G. Peterson and J. Reif. Multiple-person alternation. In *Proc. 20st IEEE Symposium on Foundation of Computer Science*, pages 348–363, 1979.
15. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
16. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symposium on Foundation of Computer Sc.*, pages 746–757, 1990.
17. K. Rudie and W. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.
18. W. Thomas. Languages, automata, and logic. *Handbook of Formal Language Theory*, III:389–455, 1997.
19. W. Thomas. Infinite games and verification. In *Proceedings of the International Conference on Computer Aided Verification CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, pages 58–64. Springer, 2002.
20. I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, January 2001.