Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 354 (2006) 230–249
Theoretical Computer Science
www.elsevier.com/locate/tcs

Modular strategies for recursive game graphs☆

Rajeev Alur[a], Salvatore La Torre[b], P. Madhusudan[c,*]

[a] *University of Pennsylvania, USA*
[b] *Università degli Studi di Salerno, Italy*
[c] *University of Illinois at Urbana-Champaign, USA*

Abstract

Many problems in formal verification and program analysis can be formalized as computing winning strategies for two-player games on graphs. In this paper, we focus on solving games in recursive game graphs which 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 fixed-point computation algorithm for solving modular games such that in the worst case the 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.
© 2005 Published by Elsevier B.V.

Keywords: Model-checking; Games in verification; Pushdown systems

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 [10,26,20] for sample research and [31] 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 the μ-calculus, as well as several procedures that use tree automata emptiness for deciding various

☆ This research was supported in part by ARO URI award DAAD19-01-1-0473, and NSF awards CCR-9970925, ITR/SY 0121431, and CCR-030638. The second author was also supported by the MIUR Grant 60% 2002/2003. A preliminary version appeared in the Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and the Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 2619, 2003, pp. 363–378.

* Corresponding author.
E-mail addresses: alur@cis.upenn.edu (R. Alur), latorre@dia.unisa.it (S. La Torre), madhu@cs.uiuc.edu, madhusud@saul.cis.upenn.edu (P. Madhusudan).

0304-3975/$ - see front matter © 2005 Published by Elsevier B.V.
doi:10.1016/j.tcs.2005.11.017

logics, can be reduced to solving games [15,30]. 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" [4]; *module checking* deals with the problem of checking whether a module behaves correctly no matter in which environment it is placed [21]; 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 [14,12]; recent work in interface synthesis for Java classes [2] uses games to capture dynamic requirements a class demands of its environment.

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 games with *reachability* and *safety* winning conditions. For reachability games, the problem is to decide whether one of the players has a strategy to force the system starting from a specified node to enter one of the target nodes. In a safety game, the player must force the system to stay within a given "good" set of 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 [32,11], 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 [3]. 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 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 winning strategy while there is a global one. Recent work on the interface compatibility checking for software modules implements the global games on pushdown systems [12], but we believe that checking for existence of modular strategies matches better with the intuition for compatibility. Due to recursion, a recursive state machine behaves like a *pushdown* system and therefore its underlying global state space is infinite. Reachability in pushdown games has been studied [24,17,32,11] under the assumption that each player has access to the entire global history which includes the information of the play in all modules. The first contribution of our paper is the notion of *modular* strategies for games on RSMs. A modular strategy is a strategy that has only local memory, and thus the 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. Clearly, there are cases where there is no modular winning strategy while there is a global one.

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 [32]. Then, we proceed to formulate a fixed-point computation algorithm that generalizes the symbolic solution to reachability games. For ordinary game graphs, the fixed-point 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, we also need to keep track of the

[1] More interesting forms of interactions between the two players are possible, for instance, see alternating transition systems [4], but we will use a simple game model for this paper.

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. When a play enters a module, the strategy used for the module always drives the play such that the play exits only in a subset E of the exit nodes of the module. Also, the strategy only allows certain modules M to be called from the current module. It turns out that these two sets, E and M, are the only relevant aspects of the strategy that needs to be recorded. Consequently, in the worst case, the number of iterations of our fixed-point computation is exponential in the number of exit nodes of the modules.

We then turn to *safety* winning conditions for recursive game graphs. Since we restrict to modular strategies for one player, 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 where this memory can be persistent and can remember the history of the *past* invocations of this module. In this case, we prove reachability games to be undecidable by a reduction from the undecidability of *multi-player* games with incomplete information [25].

Related work: We have already explained the relation to the global games on pushdown systems [9,11,32]. 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 for synthesizing distributed controllers for a system (see [27,22,28] 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* [25,27]. 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 by recalling the notion of games on "flat graphs" 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. Intuitively, after a play $u_0 u_1 \dots u_j$, if $u_j \in V_p$, then player p chooses a successor vertex $u_{j+1} \in \gamma(u_j)$.

Formally, a *play* in a game graph G is a (finite or infinite) path in G. A *strategy* for player p is a function $f : V^*.V \to V$ mapping sequences (and hence plays) to vertices such that for every sequence $\pi u \in V^*.V$, if $u \in V_p$ and $\gamma(u) \neq \emptyset$, then $f(\pi u) \in \gamma(u)$. The idea is that when a play πu has been played, where $u \in V_p$, the strategy f recommends the move $f(\pi u)$. Unless u is a sink vertex, we require that f picks a successor vertex of u. A play $\pi = u_0 u_1 \dots$ is *according to* f if for every $0 \leqslant j < |\pi|$ such that $u_j \in V_p$, $f(u_0 \dots u_j) = u_{j+1}$. If 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, π is finite and there is no $v \in V$ such that πv is a play according to the strategy).

Since we are interested in *reachability* and *safety* games in this paper, we have a *winning condition* for the game given by a subset of vertices X. For reachability games, X represents the target set that must be reached, and in safety games, X represents the "good" set that the play must stay within. A flat reachability (safety) 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 is a subset of the vertices. In a reachability game, a play is winning for the protagonist if it contains a vertex in X (i.e. a play $\pi = u_0 u_1 \dots$, where $u_0 = v_0$, is winning if there is an $i < |\pi|$ such that $u_i \in X$). A play π is winning for the protagonist in a safety game if all vertices in the play are 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. We recall that flat reachability and safety games are solvable in linear time and are PTIME-complete. Moreover, if a player has a winning strategy, then it also has a memoryless winning strategy [23].

2.2. Recursive game graphs

Our main objective is to study reachability and safety games in hierarchical and recursive graph structures that are defined using several component game graphs (or game modules) that can invoke each other. We concentrate mainly on reachability games and briefly overview how the results translate to safety games in Section 5.

In this section we introduce our model. It is the recursive state machine model defined in [3,8] generalized to game graphs.

Definition 1. 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 finite nonempty set of *nodes* N_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 (respectively *player* 1) to play.
- A labeling $Y_i : B_i \to \{1, \ldots, n\}$ that assigns to every box an index of the game modules $A_1, \ldots, A_n$.
- Let $Calls_i = \{(b, e) \mid b \in B_i, e \in En_j, j = Y_i(b)\}$ denote the set of *calls* of module A_i and let $Retns_i = \{(b, x) \mid b \in B_i, x \in Ex_j, j = Y_i(b)\}$ denote the set of *returns* in A_i. Then, $\delta_i : N_i \cup Retns_i \to 2^{N_i \cup Calls_i}$ is a *transition function*.

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 $b \in B_i$, $j = Y_i(b)$ and $e \in En_j$ is an entry node 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 node $x \in Ex_j$. 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 Fig. 1. It comprises two modules A_1 and A_2, where A_1 has three boxes (b_1, b_2 and b_3) that invoke A_2. The nodes of player 0 (the protagonist) are denoted by circles and the nodes of player 1 (the adversary) are denoted by squares (all boxes belong to player 0). The only adversary node in the example is e_1, the entry node of A_1. The only place at which the protagonist has more than one enabled move is e_2, the entry node of A_2. The node e_1 is an entry node of A_1, (b_2, e_2) is a call in A_1 and (b_2, x_2) is a return in A_1.

We make some assumptions of these graphs in the sequel that enables a more readable presentation:

(R1) There is only *one* entry node in every module, i.e. $|E_i| = 1$ for every i. We refer to this unique entry node of A_i as e_i.

(R2) For every $u \in N_i \cup Retns_i$, $e_i \notin \delta_i(u)$ 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 exit nodes. Also, there are no transitions from returns to calls.

(R3) The nodes and boxes of all modules 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.

The restriction (R1) of having only a single entry is for convenience. We address how to handle multiple entries in Section 5.

The condition (R3) is without loss of generality. Hence we can extend the functions $\{Y_i\}_{i=1}^n$ to a single function $Y : B \to \{1, \ldots, n\}$. The restriction (R2) is also without loss of generality as one can always introduce new entry nodes (exit nodes) that do not have incoming (outgoing) transitions and turn the original entry nodes (exit nodes) to internal nodes. In the second case too, for every transition from a return to a call, we can introduce a new dummy node in between them.

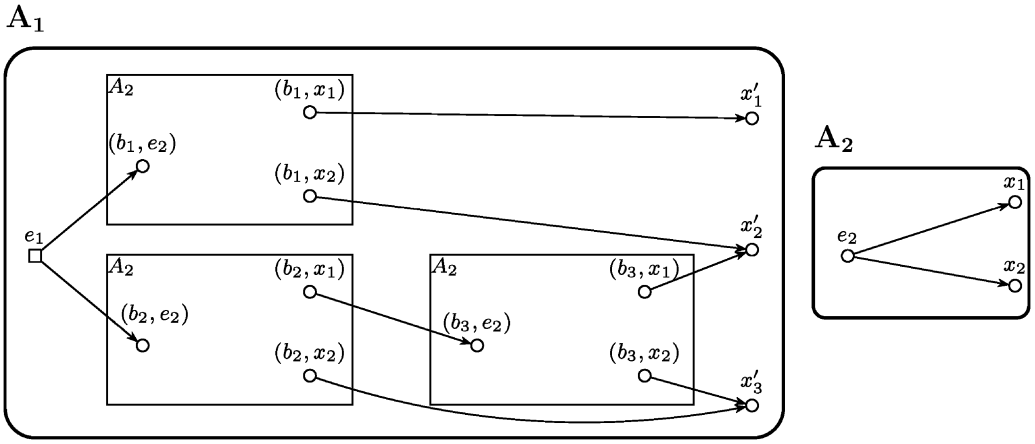

Fig. 1. A recursive game graph.

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 $\leqslant$ on the modules such that for every A_i, A_i has no calls to any A_j where $A_j \leqslant A_i$ (i.e. there is no box b in A_i such that $A_{Y(b)} \leqslant A_i$). For example, the game graph in Fig. 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 $\langle \theta, u \rangle$ where $\theta = b_1, \ldots, b_r$ is a finite (perhaps empty) sequence of boxes from B (the stack of recursive calls), and u is a node in N (the current control point). Consider a state $\langle b_1, \ldots, b_r, u \rangle$ and let j_i, $1 \leqslant i \leqslant r$, be such that $b_i \in B_{j_i}$, and let j be such that $u \in N_j$. Such a state is well-formed if $Y(b_i) = j_{i+1}$ for $1 \leqslant i < r$, and if $r \geqslant 1$, $Y(b_r) = j$. 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 node and is a p-node, or (b_r, u) is a p-return. (If $r = 0$ and u is an exit node, 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.

Definition 2. Given a recursive game graph $A = \langle A_1, \ldots, A_n \rangle$, the *global game graph* corresponding to 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 $u \in N_j$. Then for any $s' \in Q_A$, $s' \in \delta(s)$ provided one of the following holds:

Internal move: $u \in N_j \setminus Ex_j$, $u' \in \delta_j(u)$, for some $u' \in N_j$, and $s' = \langle b_1, \ldots, b_r, u' \rangle$.
Call a module: $u \in N_j$, $(b', e) \in \delta_j(u)$ (where $b' \in B_j$) and $s' = \langle b_1, \ldots, b_r, b', e \rangle$.
Return from a call: $r \geqslant 1$, $(b_r, u) \in Retns_j$, $u' \in \delta_j(b_r, u)$, for some $u' \in N_j$, and $s' = \langle b_1, \ldots, b_{r-1}, u' \rangle$.

The first case above corresponds to an internal move within a module, the second case is when a new module is called and the box gets pushed onto the stack, and the last case corresponds to returns from calls when the transition is taken from the return node determined by popping the stack. Note that the set of reachable states from a given state could be infinite if the recursive game graph is not hierarchic, while it is certainly finite if it is hierarchic.

Definition 3. A *recursive* (*reachability*) *game* is a tuple $\langle A, e_1, X\rangle$ where $A = \langle A_1, \ldots, A_n\rangle$ is a recursive 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 exit nodes of A_1.[2]

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 adopt the notion of global strategies on the global recursive games, it will lead us to a definition equivalent to that of pushdown games [32]. One can show that for every recursive game, there is a polynomial-sized pushdown automaton whose configuration game graph (as in [32]) is isomorphic to the global graph of the recursive game, and vice versa. This is an extension of how recursive state machines and pushdown automata are related (see [1] for a detailed explanation). 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. We now introduce this class of strategies and define the corresponding decision problem.

2.3. Modular strategies

In this section, we introduce the concept of a *modular strategy* for the protagonist. A modular strategy comprises a set of local strategies, one for each game module, such that when the local strategies are put together into a global strategy, the protagonist wins the game. A local strategy for a module A_i is restricted in the sense that it can only refer to the "local memory" of module A_i, 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 $u \in N_i \setminus Ex_i$ 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 is not an exit node or $r = 0$, and $\beta(\pi) = \langle b_1, \ldots, b_{r-1}\rangle$, otherwise. Now, let j be the *largest* index $0 \leqslant j \leqslant k$ 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 $j \leqslant j' \leqslant k$ are all of the form $\langle \beta(\pi), b'_1, \ldots, b'_{r'}, u'\rangle$, for some $r' \geqslant 0$. Note that there may be states $s_{j'}$, $j < j' \leqslant k$ such that $s_{j'} = \langle \beta(\pi), b'_1, \ldots, b'_{r'}, e_i\rangle$, with $r' \geqslant 1$, which denote recursive entries into A_i, but which return before s_k. We will now be interested in the suffix of π from s_j, $\alpha(\pi) = s_j \ldots s_k$.

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_θ, for a sequence $\theta = \langle b_1, \ldots, b_r\rangle$, of any state s as follows: if $s = \langle \theta, u\rangle$, where $u \in N_i$, then $\rho^i_\theta(s) = u$; if $s = \langle \theta, b, u\rangle$, where $(b, u) \in Calls_i \cup Retns_i$, then $\rho^i_\theta(s) = (b, u)$; in all other cases, $\rho^i_\theta(s) = \varepsilon$, the empty word. We extend ρ^i_θ to sequences of states: $\rho^i_\theta(s'_1 \ldots s'_l) = \rho^i_\theta(s'_1) \ldots \rho^i_\theta(s'_l)$.

We can now define a function μ_i that extracts the *local memory* for module A_i 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).

For example, consider the recursive game graph in Fig. 1. The play $\pi = \langle e_1\rangle\langle b_2, e_2\rangle\langle b_2, x_1\rangle\langle b_3, e_2\rangle\langle b_3, x_1\rangle$ is a finite play in this game and the control after π is in A_1. The current stack $\beta(\pi)$ is the empty sequence. $\alpha(\pi) = \pi$ and the local memory of π is $\mu_1(\pi) = \pi$. However, for $\pi' = \langle e_1\rangle\langle b_2, e_2\rangle\langle b_2, x_1\rangle\langle b_3, e_2\rangle$, control after π' is in A_2, $\beta(\pi') = \langle b_3\rangle$, $\alpha(\pi') = \langle b_3, e_2\rangle$ and $\mu_2(\pi') = e_2$. Note that after π', the local memory has "forgotten" the previous entry into A_2 where it went through the nodes e_2 and x_1.

[2] Note that we could define the target set to be an arbitrary subset of nodes. However, our choice will be convenient and is without loss of generality. For example, given a target set of nodes, we can modify the recursive game graph such that as soon as a target is hit, the play is forced to exit all calls along a new exit all the way up to A_1 where it exits at a particular exit node x.

We are ready now to define modular strategies. A modular strategy for player 0 is intuitively a set of functions, one for each module, that encodes how player 0 must play the game in that module. However, the choice of a move at a state can depend only on the *local memory* of the play so far in the current module. Formally:

Definition 4. Given a recursive game graph $A = \langle A_1, \ldots, A_n \rangle$, a modular strategy $\hat{f}$ for player 0 is a set of functions, $\{f_i\}_{i=1}^n$, one for each module, where for every i, $f_i : (N_i \cup Calls_i \cup Retns_i)^+ \to (N_i \cup Calls_i)$ such that whenever $u \in (N_i \cup Calls_i \cup Retns_i)$ such that $\delta_i(u) \neq \emptyset$, $f_i(wu) \in \delta_i(u)$, for each $w \in (N_i \cup Calls_i \cup Retns_i)^*$. A play π is according to $\hat{f}$ 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 $\mu_i(\pi s s') = wu$ where $w = \mu_i(\pi' s)$ and $u = f_i(w)$. [3]

Modular strategies can be seen as global strategies f such that the following property holds:

(P1) For all plays π, π' of A, if the control after π and the control after π' are both in A_i and $\mu_i(\pi) = \mu_i(\pi')$, then $f(\pi) = f(\pi')$ holds.

In fact, we have the following:

Proposition 1. *For any modular strategy $\hat{f}$ for player* 0, *there exists a strategy f such that* (P1) *holds and the set of plays according to $\hat{f}$ is equal to the set of plays according to f. Conversely, for any strategy f such that* (P1) *holds there exists a modular strategy $\hat{f}$ such that the set of plays according to f is equal to the set of plays according to $\hat{f}$.*

Proof. For a modular strategy $\hat{f} = \{f_i\}_{i=1}^n$, let πs be a play, let the control after πs be in A_i, and let $\theta = \beta(\pi s)$. Then $f(\pi s) = s'$ where s' is the unique successor of s such that $\rho_\theta^i(s') = f_i(\mu_i(\pi s))$. Clearly, property (P1) holds for f since $f(\pi s)$ is uniquely determined by $\mu_i(\pi s)$. Moreover, it is easy to see from the definitions that the set of plays according to f is the same as the set of plays according to $\hat{f}$. Conversely, let f be a strategy that satisfies (P1). Then, define $\hat{f} = \{f_i\}_{i=1}^n$ such that for any play π, if the control after π is in A_i, then $f_i(\mu_i(\pi)) = \rho_\theta^i(f(\pi))$, where $\theta = \beta(\pi)$ (the function f_i on other values can be defined arbitrarily). Then, from the fact that f satisfies (P1), it follows that such a modular strategy can be defined and it is easy to see that the plays according to f are precisely the plays according to $\hat{f}$. □

We sometimes refer to a component f_j of a modular strategy $\{f_i\}$ as a *local strategy*. We denote modular strategies also as f (instead of $\hat{f}$) and freely switch between whether it stands for a tuple of strategies or a global strategy that satisfies (P1).

We consider the following decision problem.

Definition 5 (Recursive game reachability problem). Given a recursive game $\langle A, e_1, X \rangle$, is there a modular winning strategy for the protagonist?

Observe that there are recursive games in which the protagonist can win only if it uses a global non-modular strategy. To see this, consider again the game graph in Fig. 1. 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 from e_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 (which chooses x_1 if b_1 is on the stack and chooses x_2 if b_2 is on the stack).

Rather than allow 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 Fig. 1, though there is no modular strategy for the protagonist for the target set $\{x_1', x_3'\}$, there is

[3] Technically, the definition of modular strategy may seem similar to that of innocent strategy in the literature on game semantics for programming languages [19]. In both innocent and modular strategies (in fact, even for distributed strategies [25]), the move of the protagonist depends on a partial history of the play and not on the complete play. However, they differ in both the associated semantics and the way the partial history is extracted.

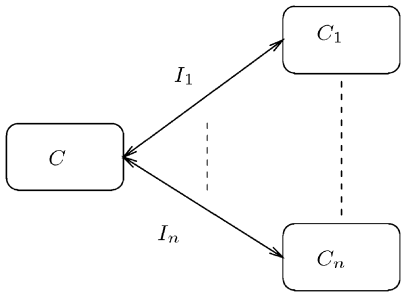

Fig. 2. Component architecture of a digital system.

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 a modular strategy.

2.4. Potential applications

In this section, we briefly illustrate settings where the use of modular and persistent strategies is quite natural and appropriate.

2.4.1. Synthesis of interfaces for components

In modular approaches to system design, a system is decomposed into components arranged according to a global architecture [29]. This approach heavily relies on interfaces—an interface abstracts from a component the information that is needed to compose it with the other parts of a system, thus allowing each component to be designed independently of others.

Consider a general architecture of a system as shown in Fig. 2, where a component C interacts with a set of components $C_1, \ldots, C_n$. For each C_j, the interface I_j of C_j to C captures the ways in which C_j will behave when fed a sequence of inputs by C. Now, suppose that we want to synthesize the interface assumptions $I_1, \ldots, I_n$, one for each component, that assure that the component C is correct with respect to a specification. Note that any interface I_j should depend only on the information explicitly passed by C to C_j.

For sequential models (that is, when at any time the control can be in at most one component), this scenario can be naturally captured using modular/persistent strategies. We can model each interface as a module with the nondeterminacy in the interface modeled as choices for the protagonist. Determining the strategy for the protagonist in these modules that will make C satisfy its specification corresponds to synthesizing interfaces. Moreover, the modularity of interfaces is captured by strategies that are "local" to each module. In particular, if along any execution, at each invocation of a component C_j, C_j does not keep memory of the previous invocations of itself (for example, this is the case when C calls each C_j as a purely functional call), then the above interface synthesis problem corresponds to that of computing modular strategies. On the other hand, if components can keep track of previous invocations, then interface synthesis corresponds to computing persistent strategies.

2.4.2. Software verification using abstraction

A recent approach to software verification consists of abstracting a computer program with a boolean program (boolean predicates are used to abstract data domains) and then checking the boolean program against the specification [7,18]. A boolean program is a program with the usual control-flow constructs of imperative languages such as C, but with the restriction that all the variables are boolean.

In the existing approaches, when the boolean program is constructed, function calls to other modules or library calls are often completely abstracted away (or treated as uninterpreted functions). Since the function calls may indeed prove important to verify the specification, a more robust approach is to ask what requirements the functions must satisfy in order for the program to meet its specification. For example, the fact that a call to a function that squares its input

always returns a positive value may be required to prove the correctness of the program, if boolean predicates keep track of whether variables are positive or not.

More precisely, given a boolean program that makes function calls $g_1, \ldots, g_n$ (the same function g_i may be called in different parts of the program), instead of modeling each call as an abstract stub, we want to synthesize a model of these functions (over the boolean variables) that ensure that the program meets its specification. We can model each g_i as a module, capture its nondeterministic output using protagonist moves, and check for a winning modular strategy. The modularity of the strategy ensures that the model of each function is independent of the context in which it is invoked. The synthesized assumptions for each function g_i can then be checked against the concrete implementation of g_i.

3. Solving reachability in recursive games

In this section, we show how to solve the recursive game reachability problem and show it is NP-complete. Let us fix a reachability game $\langle A, e_1, X\rangle$ for the rest of this section, where $A = \langle A_1, \ldots, A_n\rangle$, and each $A_i = (N_i, B_i, V_i^0, V_i^1, Y_i, En_i, Ex_i, \delta_i)$. Recall that the target set X is a subset of Ex_1, the exit nodes of A_1.

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 exit nodes a play can reach if it enters A_i and continues according to f; that is, an exit node $x \in Ex_i$ is in X_i^f if 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 modular strategy $\hat{f}$ and replace an f_i in $\hat{f}$ 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 definition.

Definition 6. 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 from $\langle e_1\rangle$ according to f that enters A_i. (In particular, $A_1 \in V$.)
- $A_i \rightarrow A_j$ iff there is some play according to f from $\langle e_1\rangle$ which has 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 $Y(b_r) = A_j$).
- $\lambda(A_i) = X_i^f$, for every $A_i \in V$.

We first make a simple observation:

Lemma 2. *Let f be a modular 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. Assume there is a cycle in C_f. Then we can find a path $A_{i_1}A_{i_2}\ldots A_{i_m}\ A_{i_{m+1}}\ldots A_{i_j}$ in C_f such that $i_1 = 1$, the elements $i_1, \ldots, i_{j-1}$ are distinct and $i_j = i_m$. By the definition of a call graph, we know that for every $1 \leqslant l < j$, there is a play according to f starting from $\langle e_1\rangle$ that reaches a point in A_{i_l} where $A_{i_{l+1}}$ is called. Since f is modular, this means that for every play π that ends in the entry node of A_{i_l}, $1 \leqslant l < j$, there is a continuation of it according to f that reaches a point in A_{i_l} where $A_{i_{l+1}}$ is called. However, before $A_{i_{l+1}}$ is called, there may be several calls to other modules (that return).

We can now construct a play according to f that starts from $\langle e_1\rangle$ that goes through e_{i_2}, then through $e_{i_3}, \ldots$ then through $e_{i_{j-1}}$ and then back to e_{i_m}, then through $e_{i_{m+1}}$, etc. forever.

Now there are two cases. If along this play, we happen to call A_1 again (note that A_1 may not be A_{i_j} for any j) then we can, instead of continuing the play in the fashion described above, continue the play as we did at the beginning starting from $\langle e_1\rangle$. In this case, we have a play that repeatedly calls A_1 and never exits these A_1 calls, and hence avoids X, which contradicts the assumption that f was winning.

Otherwise, the play never enters A_1 and instead will enter A_{i_m} through $A_{i_{j-1}}$ again and again. Hence the call from A_1 never returns and the play will never reach X which again contradicts the assumption that f is 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 from $\langle e_1\rangle$ according to f reaches a state of the form $s = \langle b_1, \ldots, b_i, u\rangle$ such that $u \in N_l$ and $b_j \in B_l$ for some $l \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, i\}$). The following is immediate from the above lemma:

Corollary 3. *Recursive reachability games admit only hierarchic modular winning strategies.*

We recall that the target set X is a subset of the exit nodes 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 that can be reached according to 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:

Definition 7. A *call graph* for a game $\langle A, e_1, X\rangle$ is a tuple $C = (V, \to, \lambda)$ where $V \subseteq \{A_1, \ldots, A_n\}$, $(V, \to)$ is an acyclic graph and $\lambda(A_i) \subseteq Ex_i$, for each $A_i \in V$.

Let $C = (V, \to, \lambda)$ 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, if C is a call graph, 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 (i.e. $A_i \to A_{Y(b)}$ 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 allowed set of exit nodes $\lambda(A_{Y(b)})$ from the call graph and have edges to each of the returns corresponding to these exit nodes.

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

Lemma 4. *There exists a modular winning strategy for a recursive reachability game $\langle A, e_1, X\rangle$ if and only if there exists a call graph $C = (V, \to, \lambda)$ such that $A_1 \in V$, $\lambda(A_1) \subseteq X$ and for every $A_i \in V$, player 0 wins the reachability game in the flat game graph $\langle A_i^C, e_i, \lambda(A_i)\rangle$.*

Proof. If f is a winning strategy for $\langle A, e_1, X\rangle$, we show that its call graph, $C_f = (V, \to, \lambda)$, satisfies the requirements of the lemma. Directly from the definition we have that $A_1 \in V$ and $\lambda(A_1) \subseteq X$. Moreover, for a module $A_i \in V$, the strategy f_i in f for module A_i is basically a strategy that drives a play entering A_i to $\lambda(A_i)$ under the assumption that calls from A_i to any module A_j will result in the play exiting A_j in some return corresponding to an exit in $\lambda(A_j)$. In $\langle A_i^{C_f}, e_i, \lambda(A_i)\rangle$ all moves are as in A_i except for the moves at the calls where the adversary can move from a call to A_j to any return corresponding to an exit in $\lambda(A_j)$, if A_j is a successor of A_i in C_f. Hence the construction precisely captures the assumption f_i makes, and therefore, the strategy f_i for A_i itself serves as a winning strategy in $\langle A_i^{C_f}, e_i, \lambda(A_i)\rangle$.

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 serves as a local strategy for A_i in the recursive game. Since the call graph is acyclic, we can show that in any play according to this strategy, if the play enters a module A_i, it will eventually exit in some exit in $\lambda(A_i)$. Since $\lambda(A_1) \subseteq X$, the play will eventually reach some node in X and hence these local strategies constitute a winning modular strategy for $\langle A, e_1, X\rangle$. □

We say that a modular 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 selection made by the modular strategy depends only on the current node of the play.

As a corollary to the above lemma we have:

Corollary 5. *If there is a modular winning strategy for a recursive reachability game, then there is a modular memoryless winning strategy for it.*

Proof. By Lemma 4, we know that there is a call graph $C = (V, \rightarrow, \lambda)$ such that each of the games $\langle A_i^C, e_i, \lambda(A_i)\rangle$ is winning. Since these games are simple flat reachability games, they admit a memoryless winning strategy [23]. By the proof of Lemma 4, these memoryless winning strategies translate to a modular memoryless winning strategy for the recursive game. □

Another corollary is immediate:

Corollary 6. *Given a recursive reachability game, the problem of checking whether player* 0 *wins the game is in* NP.

Proof. The NP procedure works as follows. First, we guess a call graph $C = (V, \rightarrow, \lambda)$ such that $A_1 \in V$ and $\lambda(A_1) \subseteq X$, where X is the target set. The size of this guess is clearly polynomial. Then, for every module $A_i \in V$, we check if there is a winning strategy for player 0 in $\langle A_i^C, e_i, \lambda(A_i)\rangle$. This graph can be constructed in polynomial time. Also, such a flat reachability game can be solved in polynomial time (alternating reachability) [13]. If all the games are winning for player 0, we report that player 0 wins the recursive game. The correctness follows from Lemma 4. □

In fact, the problem is of solving recursive games is NP-complete:

Theorem 7. *The recursive game reachability problem* (*even for hierarchic game graphs*) *is* NP*-complete.*

Proof. Corollary 6 establishes membership in NP. We need to show that reachability on hierarchic game graphs is NP-hard. We do this by a reduction from the satisfiability of 3-CNF formulas, which is known to be NP-complete [16].

The intuition is that there will be modules for each variable in the CNF formula, where player 0 has to pick a valuation for the variable by picking an exit node. The initial module enables player 1 to pick any clause and call a module corresponding to that clause. The module for this clause will check whether the clause is satisfied by calling the modules corresponding to the variables in the clause. Player 0 wins if all clauses are satisfied.

Let $\varphi = c_2 \wedge \cdots \wedge c_m$ be a 3-CNF formula over the variables $y_1, \ldots, y_n$. Define a hierarchic game graph $A = \langle A_1, A_2, \ldots, A_m, A_{m+1}, \ldots, A_{m+n}\rangle$ where:

- For $j = 1, \ldots, n$, module A_{m+j} has an entry node e_{m+j}, two exit nodes x_{m+j} and $\bar{x}_{m+j}$, and no boxes. Node e_{m+j} is a 0-node.
- For $i = 2, \ldots, m$, module A_i has an entry node e_i, an exit node x_i and boxes b_{ih}, b_{ij}, b_{ik} such that $Y_i(b_{il}) = m + l$ and literals of c_i are from variables y_h, y_j, and y_k. Function δ_i is defined by: $\delta_i(e_i) = \{(b_{ih}, e_{m+h}), (b_{ij}, e_{m+j}), (b_{ik}, e_{m+k})\}$; for $l \in \{h, j, k\}$, if y_l is a literal of c_i then $x_i \in \delta_i(b_{il}, x_{m+l})$ and if $\neg y_l$ is a literal of c_i, then $x_i \in \delta_i(b_{il}, \bar{x}_{m+l})$. The node e_i is a 0-node.
- A_1 has an entry node e_1, an exit node x_1, and boxes $b_2, \ldots, b_m$. For every $i = 2, \ldots, m$, $Y_1(b_i) = i$, $(b_i, e_i) \in \delta(e_1)$ and $x_1 \in \delta_1(b_i, x_i)$. The node e_1 is a 1-node.

The construction of A is illustrated in Fig. 3. It is easy to verify that φ is satisfiable if and only if there exists a winning strategy of the protagonist in the reachability game $\langle A, e_1, \{x_1\}\rangle$.

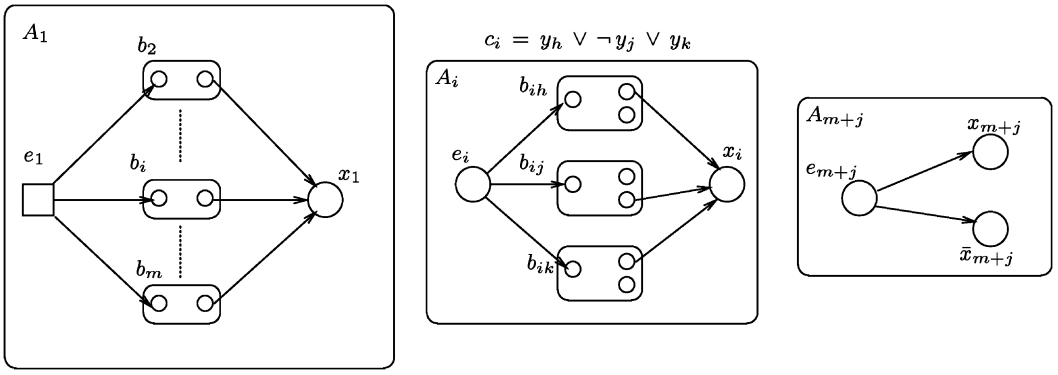

Fig. 3. Reduction from 3-CNF: graphical representation of module A_1, module A_i with $i \in \{2, \ldots, m\}$, and module A_{m+j} with $j \in \{1, \ldots, n\}$.

Note that the requirement that the strategy be modular is important as we want the valuation for the variables to be chosen independent of the context in which it is called (i.e. independent of the clause that is being checked). Also, note that the above game is hierarchic. □

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 [23] adapted to our setting, and it computes the vertices where player 0 can win in an incremental 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 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 (Fig. 4) 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. It turns out that to ensure this for each module A_i, it is enough to keep track of the set of exit nodes of A_i that the strategy will force any play entering A_i into.

For a set of maximal finite plays Π, let $Final(\Pi) = \{s \mid \text{there is a play of the form } \pi s \in \Pi\}$. 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 modular strategy $\hat{f} = \{f_i\}$ such that the following hold:

A1. If Π_v is the set of all maximal plays starting at $\langle v \rangle$ and consistent with $\hat{f}$, then Π_v contains only finite plays and $Final(\Pi_v) = \{\langle x_i \rangle \mid x_i \in E_i\}$. Further, the plays in Π_v do not enter any module 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 $\hat{f}$. 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 means 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.

Algorithm REACH

Initially, each exit node $x \in X$ is labeled by the tuple $\langle E_1, \ldots, E_n \rangle$, 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 the 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 the 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 $\langle E_1', \ldots, E_n' \rangle$ 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 of box b. 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 every $l \neq i$, 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 $\langle E_1, \ldots, E_n \rangle$ such that $E_1 \subseteq X$.

Fig. 4. Algorithm REACH.

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 $E' = E''$.

The intuition behind the rules are as follows:

Rule 1: This rule makes a 0-node or 0-return u inherit a label of a successor. Clearly, if there is a strategy $\hat{f}$ from the successor which meets conditions A1 and A2, we have a similar strategy from u which first picks this successor and then follows $\hat{f}$.

Rule 2: 1-nodes and 1-returns u get labeled using this rule. First, we check if the labels on all successors are consistent with respect to strategies for other modules. Note that a play starting at u can be taken to any of its successors by player 1. If the protagonist then plays according to the strategies prescribed at that node (by the inductive hypothesis), then we could reach any of the nodes in $\bigcup_{h=1}^{k} E_i^h$; this is why we set E_i' to this set. Note that for any other module A_j, E_j' is set to the common set of exit nodes which the successor labels agree upon. But there is a subtle point here—we must make sure that the strategies we pick for other modules be non-recursive. This will be explained in the proofs below.

Rule 3: This rule activates an exit node of a module A_j when there is a box b in a module A_i which is a call to A_j and which has a return (b, x) that has a label. Note that the activation of this exit node is similar to the initialization step for module A_1—we label the exit node with $\top$ for all components except the jth one, which is labeled $\{x\}$.

Rule 4: Calls are labeled using this rule. A call in A_i to A_j can get labeled if the entry node of A_j ensures that there is a strategy which will take plays entering A_j to some exit nodes X' and the returns in A_i corresponding to these exit nodes are labeled such that they agree on the assumptions on all modules. Note that we require $E_i^0 = \top$, which by (A1) demands that the strategy for A_j must not call A_i, and hence we avoid recursive calls. The rest of the intuition is same as for Rule 2.

We prove now that REACH solves the reachability problem for recursive games.

Lemma 8 (*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 modular strategy $\hat{f} = \{f_i\}$ such that* (A1) *and* (A2) *hold.*

Proof. We prove the lemma by induction on the number of applications of the update rules. For the initial labeling the lemma is trivially true. Suppose by induction that after the hth updating, the lemma holds. We distinguish the following cases for the $(h+1)$th updating, depending on which rule was used.

Rule 1: Suppose that $v \in \delta(u)$ where u is a 0-node (or a 0-return), $u, v \in A_i$ and v has already been labeled with $\langle E_1, \ldots, E_n \rangle$. By the inductive hypothesis, there exists a modular strategy $\hat{f} = \{f_j\}$ satisfying A1 and A2 with respect to the labeling $\langle E_1, \ldots, E_n \rangle$ at v.

Using Rule 1, if u is labeled $\langle E_1, \ldots, E_n \rangle$, then let a new modular strategy $\hat{f}' = \{f'_j\}$ be such that $f'_j = f_j$ for $j \neq i$ and $f'_i(u) = v$, and $f'_i(uv\sigma) = f_i(v\sigma)$ for any sequence $uv\sigma$. Using the fact that $\hat{f}$ satisfies properties A1 and A2, clearly $\hat{f}'$ also satisfies A1 and A2.

Rule 2: Suppose that $\{v_1, \ldots, v_k\} = \delta(u)$ where u is a 1-node (or a 1-return) of A_i and for $l = 1, \ldots, k$, v_l has been labeled already with $\langle E^l_1, \ldots, E^l_n \rangle$. Let the condition of Rule 2 hold: i.e. for $l \neq i$, $E^1_l, \ldots, E^k_l$ are pairwise consistent. Assume Rule 2 is used and u gets labeled with $\langle E_1, \ldots, E_n \rangle$ by REACH using the labels $\langle E^l_1, \ldots, E^l_n \rangle$ for $l = 1, \ldots, k$.

By the inductive hypothesis, for each $l \in \{1, \ldots, k\}$ there exists a strategy $\hat{f}^l = \{f^l_j\}$ satisfying the properties A1 and A2 with respect to the labeling $\langle E^l_1, \ldots, E^l_n \rangle$ at v_l. We define a strategy $\hat{f} = \{f_j\}$ for meeting the conditions A1 and A2 for the new label for the node u as follows. For $j \neq i$, set $f_j = f^l_j$ where l is the least index h such that $E^h_j \neq \top$. Also, let $f_i(uv_l\sigma) = f^l_i(v_l\sigma)$, for every $l \in \{1, \ldots, k\}$ and $\sigma \in (N_i \cup Calls_i \cup Retns_i)^*$.

The only subtle point here is to make sure that the call graph for the strategy $\hat{f}$ is indeed acyclic. If this is so, it is easy to see that $\hat{f}$ satisfies the properties A1 and A2, using the fact for any $l \neq i$, the sets $E^1_l, \ldots, E^k_l$ are pairwise consistent.

Let $J = \{j \mid j \neq i, E_j \neq \top\}$. The problem is that for any module A_j ($j \in J$), we pick strategies from the various strategies recommended at the successor vertices v_l ($l = 1, \ldots, k$). We must make sure that while doing this, we do not introduce recursive calls.

First note that the call graph of every $\hat{f}^j$ must be acyclic (Lemma 2). For any $j \in J$, let us say that f_j gets *defined at h* if h is the least index for which $E^h_j \neq \top$. Now note that if f_j is defined at h, then for any play entering A_j, if the play when within A_j calls $A_{j'}$, then $f_{j'}$ is defined at h' where $h' \leqslant h$. Using this fact it is easy to argue that the call graph of $\hat{f}$ is acyclic.

Rule 3: In case we update the label of $x \in Ex_i$ by Rule 3, the lemma is trivially true since the ith component of the new label of x is $\{x\}$ and the other components are $\top$.

Rule 4: Consider the call (b, e), with $b \in B_i$ and $e \in N_j$. Suppose that $(b, v_1), \ldots, (b, v_k)$ are returns and let $\langle E^l_1, \ldots, E^l_n \rangle$ be a label of (b, v_l), for every $l = 1, \ldots, k$, already computed by REACH. Also, let $\langle E^0_1, \ldots, E^0_n \rangle$ be a label of e already computed by REACH. Denote by $\langle E_1, \ldots, E_n \rangle$ the labeling of v computed by REACH from $\langle E^l_1, \ldots, E^l_n \rangle$, for $l = 0, \ldots, k$, applying Rule 4. We construct a strategy f similar to the case for Rule 2, giving priorities starting from f^0 up to f^k. One can argue, in a similar fashion as we did for Rule 2, that f satisfies the properties A1 and A2 of the lemma with respect to $\langle E_1, \ldots, E_n \rangle$. However, there is one extra property we have to show in this case: that the strategy starting at (b, e) does not call A_i again. This is, however, true because of the condition of the rule which checks that $E^0_i = \top$. Since $E^0_i = \top$, the play when in A_j cannot call A_i again. When the return node (b, v_l) is reached, we know that the play can no longer call A_i by induction hypothesis. □

Lemma 9 (*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 reachability game $\langle A, e_1, X \rangle$, then e_1 is labeled by algorithm* REACH *with some label $\langle E_1, \ldots, E_n \rangle$, where $E_1 \subseteq X$.*

Proof. Let f be a winning strategy of the protagonist in the reachability game $\langle A, e_1, X \rangle$.

Before proving the lemma, we define a tree corresponding to the winning strategy f. Let Π be the set of all plays starting from $\langle e_1 \rangle$ that are consistent with f. Since f is winning, each play in Π reaches a state $\langle b_1, \ldots, b_r, x \rangle$ where

$x \in X$. However, since we know that f must be hierarchic, it is easy to see that all plays in Π in fact end in $\langle x \rangle$ with $x \in X$. Note that Π has no infinite plays.

Let V_f be the set of all elements $(\pi, v) \in (\Pi \times \bigcup_{i=1}^{n}(N_i \cup Calls_i \cup Retns_i))$ such that π is of the form $\pi' s$ where $s = \langle b_1, \ldots, b_r, u \rangle$ and where either $v = u$, or u is an entry/exit node and $v = (b_r, u)$. Hence, for every $\pi \in \Pi$, where $\pi = \pi' s$ with $s = \langle b_1, \ldots, b_r, u \rangle$, we add an element of the form (π, v) into V_f if u is a node in N, and two elements of the form (π, v) into V_f if u is an entry or an exit node.

Let $T_f = (V_f, \rightarrow_f)$ be a graph such that $(\pi, v) \rightarrow_f (\pi', v')$ iff the following holds:

- If $v = (b, e)$ is a call, then $\pi' = \pi$ and $v' = e$.
- If $v = x$ is an exit node, and $\pi = \pi'' s$ where $s = \langle b_1, \ldots, b_r, x \rangle$, with $r \geqslant 1$, then $\pi' = \pi$ and $v' = (b_r, x)$.
- If v is not a call nor an exit node, then $\pi' = \pi s$ for some s and v' is a non-entry node or a call.

It is easy to see that T_f is a tree rooted at $(\langle e_1 \rangle, e_1)$ and has leaves of the form $(\pi\langle x \rangle, x)$ where $x \in X$. It is very similar to the usual strategy-tree of a strategy, except that we explicitly label each play with the current node where the play is and add edges from calls to entry nodes, and exit nodes to returns which are implicit moves in the game.

Denote by E_i, for $i = 1, \ldots, n$, the set of all exit nodes in Ex_i visited along any maximal play of f starting at $\langle e_1 \rangle$ (we assume that if a module A_i is not visited, then $E_i = \top$).

We claim that for every $v \in N_i \cup Calls_i \cup Retns_i$ such that there is a node of the form (π, v) in T_f, v is labeled by REACH with some label $\langle E_1^v, \ldots, E_n^v \rangle$ where:

(1) Let Z_i be the smallest subset of the exit nodes of A_i such that the plays starting at $\langle v \rangle$ and consistent with f end in $\langle x \rangle$, for some $x \in Z_i$. Then $E_i^v = Z_i$.
(2) For $j \neq i$, $E_j^v = \top$ or $E_j^v = E_j$.
(3) For $j \neq i$, $E_j^v = \top$ if $A_i \not\rightarrow^* A_j$, where $\rightarrow^*$ is the transitive closure of the relation defined by the edges of the call graph of f (see Definition 6).

If this claim is true, then since $(\langle e_1 \rangle, e_1) \in T_f$, e_1 must get a label $\langle E_1^{e_1}, \ldots, E_n^{e_1} \rangle$ such that $E_1^{e_1} = E_1 \subseteq X$, which proves the lemma.

We prove the above claim by structural induction on T_f, from the leaves up to the root. The base case is trivially true since all leaves of T_f correspond to a target vertex and the claim is true by the initialization step of REACH. Consider now a node (π, v) such that the claim is true for all the nodes in the subtree under it. If $v \in (N_i \setminus Ex_i) \cup Retns_i$ is either a 0-node or a 0-return, then (π, v) has a single successor (π', v') and by update Rule 1 REACH labels v with the label of v', and the claim is true by the inductive hypothesis. If $v \in (N_i \setminus Ex_i) \cup Retns_i$ is either a 1-node or a 1-return, let $(\pi_1, v_1), \ldots, (\pi_k, v_k)$ be all the successors of (π, v) and $\langle E_1^{v_1}, \ldots, E_n^{v_1} \rangle, \ldots, \langle E_1^{v_k}, \ldots, E_n^{v_k} \rangle$ be, respectively, the labels associated with them by REACH which must exist by the induction hypothesis. By property (2) of the inductive hypothesis, for $j \neq i$, the sets $E_j^{v_1}, \ldots, E_j^{v_k}$ are pairwise consistent, the update Rule 2 of REACH can be applied and the label $\langle E'_1, \ldots, E'_n \rangle$ is computed for v, where $E'_j = \bigcup_{l=1}^{k} E_j^{v_l}$ for $j = 1, \ldots, n$. It is easy to see that properties (1) and (2) hold with respect to this label for v. Also, property (3) holds, since by property (3) of the inductive hypothesis if $A_i \not\rightarrow^* A_j$, $j \neq i$, then $E_j^{v_l} = \top$ for every $l = 1, \ldots, k$. Thus, by update Rule 2, $E'_j = \top$. By update Rule 3 and the inductive hypothesis the claim clearly holds when $v \in Ex_i$.

If $v = (b, e_j) \in Calls_i$, then since f is modular we know that for every $x_j \in E_j$ there is a play starting at $\langle e_j \rangle$ that ends up in $\langle x_j \rangle$. Thus, for every $x_j \in E_j$ there is a node in T_f of the form $\langle \pi_j, x_j \rangle$ below $\langle \pi, v \rangle$. By the induction hypothesis, these have labels that satisfy properties (1)–(3). Also, $\langle \pi, e_j \rangle$, which is the only successor of $\langle \pi, v \rangle$ in T_f, must be labeled by some $\langle E'_1, \ldots, E'_n \rangle$ such that: (1) $E'_j = E_j$, since f is modular and thus plays starting at $\langle e_j \rangle$ end up in $\{\langle x_j \rangle \mid x_j \in E_j\}$; (2) $E'_i = \top$, directly by property (3) of the induction hypothesis on $\langle \pi, e_j \rangle$ since $A_i \rightarrow A_j$. We can now show, using arguments similar to those for the case when v was a 1-node, that Rule 4 can be applied to get a label for v that satisfies the claim. □

The soundness of REACH follows from Lemma 8, by condition A1 (with $v = e_1$), while Lemma 9 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 and linear 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 fixed-point. We have the following theorem.

Theorem 10. *Algorithm* REACH *solves the recursive reachability game problem in time that is linear in the size of the game graph and exponential in the number of exit nodes.*

Proof. By Lemmas 8 and 9, REACH solves the reachability game on recursive game graphs. Let $A = \langle A_1, \ldots, A_n \rangle$ be a recursive game graph. Let $ex = \sum_{i=1}^{n} |Ex_i|$ denote the total number of exits in A, let $K_v = |N \cup Calls \cup Retns|$ denote the number of vertices and let K_e denote the number of edges in A. Since the ith component of each label is either a subset of Ex_i or $\top$, the number of labels used by our algorithm is bounded by 2^{ex}. A simple implementation, which maintains for each label a list of vertices that have that label, works in time at most cubic in $K_v + K_e$ and exponential in ex. However, a more efficient implementation can be obtained if we look upon the rules of the algorithm as solving a reachability game on a flat finite graph. This graph will have vertices of the form $(u, \langle E_1, \ldots, E_n \rangle)$, for every node u and label $\langle E_1, \ldots, E_n \rangle$. The set we want to reach is the set of vertices that corresponds to the initial labeling of the algorithm, that is, $\{(x, \langle E_1, \ldots, E_n \rangle) \mid E_1 = \{x\} \text{ and } E_j = \top, \forall j > 1\}$. The initial set of vertices are those of the form $(e_1, \langle E_1, \ldots, E_n \rangle)$ where $E_1 \subseteq X$.

To implement the rules as a game that is not too large, we need to carefully choose auxiliary vertices that witness the various conditions that the rules demand. For example, to encode Rule 2 for $(u, \langle E'_1, \ldots, E'_n \rangle)$, u is a vertex of A_i, we can build auxiliary nodes that step through the successors $v_1, \ldots, v_k$ of u, one at a time. For this, we need to make sure that the successors have labels that are consistent with $\langle E'_1, \ldots, E'_n \rangle$ and also that for every E'_j such that $E'_j \neq \top$, there is some successor v_l which has a label $\langle E^l_1, \ldots, E^l_n \rangle$ such that $E^l_j = E'_j$. Technically, this can be achieved for example by guessing at each v_i the sets $E'_j \neq \top$ that still need to be matched and the subset of exits within E'_i that still need to be covered by the remaining successor vertices $v_{i+1}, \ldots, v_k$. The encoding for Rule 4 is similar, except that one first has to guess a subset of exits of the called module.

One can define such a graph which has $(K_v + K_e)2^{O(ex)}$ vertices and edges. Since reachability games on flat graphs can be solved in time linear in the number of vertices and edges of the game graph [33], we obtain an implementation of the algorithm that runs in time $(K_v + K_e)2^{O(ex)}$. □

5. Extensions

5.1. 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 $\langle A, e_1, X_{\text{good}} \rangle$, where A is a recursive game graph, e_1 is the entry node of the game module A_1, and X_{good} is a subset of nodes of A. A play of such a game is winning if all visited states are of the form $\langle b_1, \ldots, b_r, u \rangle$ where $u \in X_{\text{good}}$. 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 2 and Corollary 3 do not hold). Despite this, deciding safety recursive games is also NP-complete:

Theorem 11. *Solving safety games on recursive* (*as well as hierarchic*) *game graphs is* NP*-complete.*

Proof. The hardness result is directly obtained from the reduction given in the proof of Theorem 7. For membership in NP, we can design an algorithm that first guesses a set of modules V (with $A_1 \in V$) and for each $A_i \in V$, a set $E_i \subseteq Ex_i \cap X_{\text{good}}$. The algorithm then builds a flat game graph A'_i (which is similar to the graph A^C_i defined in Section 3) where calls to $A_j \notin V$ are declared unsafe, each call to $A_j \in V$ is replaced with a player 1 node that has edges to returns corresponding to exits in E_j, and for all the other nodes u, u is unsafe if either $u \notin X_{\text{good}}$ or $u \in Ex_i \setminus E_i$. Each flat safety game A'_i is then solved and the algorithm declares that the protagonist wins if all these safety games are winning. Intuitively, each flat game graph A'_i checks whether the protagonist can keep the play safe in A_i without making any call to modules not in V, either forever or till the play reaches an exit node in E_i. While doing so, we assume that calls to any $A_j \in V$ will return (if at all) at returns corresponding to exits in E_j. It is easy to see that the protagonist wins iff there is such a witness. □

5.2. 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 node where the play enters, it could follow 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 node, but where each new module has only one entry node. We suitably change the calls from other modules so that they call the corresponding modules with the appropriate entry node. 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. Note that in a game with multiple entry nodes, there could be a winning strategy such that plays according to it call modules recursively—however, by the above reduction, there cannot be recursive calls to modules with *the same entry node*. Consider a recursive game G with n modules and let m_e be the maximum number of entry nodes of any module. Then, it is easy to see that the above reduction produces a game graph with at most $n \cdot m_e$ single-entry modules and the overall size of the game graph is at most $|G|^3$, where $|G|$ is the size of G.

5.3. 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 [6]). 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 k internal variables, then we can model this by having $2^{r_{\mathrm{i}}} \cdot |En_i|$ entry nodes, $2^{r_{\mathrm{o}}} \cdot |Ex_i|$ exit nodes, and $2^{r_{\mathrm{i}}+r_{\mathrm{o}}+k} \cdot |N_i|$ internal nodes (we assume all variables to 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 any module, but cannot know the exact evolution of the called module. This makes our setting a natural setup where we can deal with the construction of skeletons of program modules which achieve a particular specification.

5.4. Persistent memory strategies

Modular strategies, where a strategy is allowed to remember only the part of the play since it last entered a module, are one way of realizing the idea that each module must have its own strategy and is not allowed to know what happens in other modules. However, we can relax this condition 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 was in this module. We can realize such a 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. Formally, we define a projection operator that projects a play onto a module A_i. For any $i \in \{1, \ldots, n\}$, let $\zeta_i(s)$, for any state s, be defined as follows: if $s = \langle \theta, u\rangle$, where $u \in N_i$, then $\zeta_i(s) = u$; if $s = \langle \theta, b, u\rangle$, where $(b, u) \in Calls_i \cup Retns_i$, then $\zeta_i(s) = (b, u)$; in all other cases, $\zeta_i(s) = \varepsilon$, the empty word. We extend ζ_i to sequences of states: $\zeta_i(s'_1 \ldots s'_l) = \zeta_i(s'_1) \ldots \zeta_i(s'_l)$.

A *persistent 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 $\zeta_i(\pi) = \zeta_i(\pi')$ then $f(\pi) = f(\pi')$ holds. In other words, the advice of the strategy f for any play π that is currently in A_i depends only on the projection $\zeta_i(\pi)$ of the play onto the nodes, calls, and returns of the module.

This subtle difference with modular strategies, however, changes the problem of solving games dramatically and 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 [25] which we formalize below.

5.4.1. Multi-player games

Peterson and Reif in [25] show that multi-player games with incomplete information, where there are a set of cooperating players who play against an adversary but have incomplete information about each other, are undecidable to solve. We present here a similar simplified problem, 2Pl-1Ad, which is a game where two players play against one

adversary and the two players have incomplete information about each other. Note that in this game, there is no game graph; instead the players choose letters from some alphabets in turns.

Let α and β be two players playing against an adversary. Players α and β form team 0 and play against team 1 comprising a lone player, the adversary. Let us fix four disjoint alphabets Σ_i^γ, where $i \in \{0, 1\}$ and $\gamma \in \{\alpha, \beta\}$. The game is played as follows. The adversary picks $\gamma \in \{\alpha, \beta\}$, and plays a letter in Σ_1^γ. Player γ responds to this by playing a letter in Σ_0^γ. Now it is again the adversary's turn, and it picks a γ and continues as before.

In addition, we have a special symbol $\$_\gamma \in \Sigma_0^\gamma$ which the player γ uses to signal that it has finished playing the game. In case $\$_\gamma$ is played, the adversary can no longer pick γ (though it may continue playing with the other player). Thus plays are restricted: a play is a sequence in $((\Sigma_1^\alpha \cdot \Sigma_0^\alpha) + (\Sigma_1^\beta \cdot \Sigma_0^\beta))^*$ such that if $\pi = \pi' \$_\gamma \pi''$, for any γ, π' and π'', then $\pi'' \restriction (\Sigma_1^\gamma \cup \Sigma_0^\gamma) = \varepsilon$ (where $\restriction$ stands for the projection operation).

Let $L \subseteq ((\Sigma_1^\alpha \cdot \Sigma_0^\alpha) + (\Sigma_1^\beta \cdot \Sigma_0^\beta))^*$ be a regular language which identifies the winning plays for team 0, i.e. a play is winning for team 0 iff it belongs to L.

A strategy for team 0 is a pair of functions $\hat{g} = \{g_\alpha, g_\beta\}$ where $g_\gamma : \Sigma_1^\gamma.(\Sigma_0^\gamma \cdot \Sigma_1^\gamma)^* \to \Sigma_0^\gamma$, $\gamma \in \{\alpha, \beta\}$. A play π is said to be according to a strategy $\hat{g}$ for team 0 if for every prefix $\pi' a a'$ of π, if the length of π' is even and $a \in \Sigma_1^\gamma$, then it must be that $a' = g_\gamma(\pi \restriction (\Sigma_1^\gamma \cup \Sigma_0^\gamma))$. In other words, a play π is according to $\hat{g}$ if at every point in the play, if it is the turn of γ to play, then γ plays according to the strategy g_γ which selects a symbol depending only on the projection of the play to the moves it can see, namely $\Sigma_1^\gamma \cup \Sigma_0^\gamma$.

A strategy for team 0 is said to be winning if all plays according to it are finite and are winning. The problem 2Pl-1Ad is now this: given a set of alphabets Σ_i^γ, $i \in \{0, 1\}$, $\gamma \in \{\alpha, \beta\}$, and given a regular language $L \subseteq ((\Sigma_1^\alpha \cdot \Sigma_0^\alpha) + (\Sigma_1^\beta \cdot \Sigma_0^\beta))^*$, is there a winning strategy for team 0?

The following is easy to see from the proof of the undecidability of multi-player games in [25]:

Proposition 12 (*Peterson and Reif*). *The problem* 2Pl-1Ad *is undecidable*.

5.4.2. Reduction to recursive games with persistent strategies

Let us reduce 2Pl-1Ad to the problem of checking whether there is a persistent strategy for player 0 in a recursive game.

Let us fix an instance of 2Pl-1Ad by fixing the alphabets Σ_i^γ, $i \in \{0, 1\}$, $\gamma \in \{\alpha, \beta\}$, and a regular language L given as a deterministic finite automaton $\mathcal{A} = (Q, q_{in}, \delta, F)$ with the usual interpretation.

There are three modules A_1, A_2 and A_3. A_1, the initial module, will make calls to A_2 and A_3; A_2 and A_3 do not call any other module. A_2 has an entry node for each letter in Σ_1^α and an exit node for each letter in Σ_0^α. Similarly, A_3 has entries and exits for Σ_1^β and Σ_0^β, respectively. A_1 can call A_2 with entry nodes corresponding to any letter in Σ_1^α and A_2 will pick an exit corresponding to a letter in Σ_0^α. Similarly, A_1 will call A_3 with letters in Σ_1^β and get back letters in Σ_0^β. The moves to exit nodes in A_2 and A_3 will be controlled by player 0 and hence will correspond to some strategies g_α and g_β for the 2Pl-1Ad-game. The moves in A_1 are controlled by player 1, which nondeterministically picks γ and calls the module corresponding to γ with a nondeterministically chosen letter in Σ_1^γ. A_1 also keeps track of whether a player returns $\$_\gamma$, in which case the game graph is such that it cannot call the module corresponding to γ again.

A_1 also keeps track of the current state of the automaton $\mathcal{A}$ on the play that is being generated. Since A_1 has complete information of the play (it generates the letters in Σ_1^γ and observes the response through the return from the module that it calls), it can keep track of this information. At the nodes corresponding to final states, however, we force the module to exit at an exit node x_1 of A_1.

It is now easy to see that there is a winning strategy for team 0 in the 2Pl-1Ad-game iff there is a persistent winning strategy for player 0 in the above recursive game with the target set $\{x_1\}$. This is because persistent strategies for A_2 and A_3 directly translate back and forth to strategies for player α and β, respectively, and preserves the property of being winning in the respective games. We hence have:

Theorem 13. *The problem of checking whether there is a winning persistent strategy in a given hierarchical game is undecidable*.

6. Conclusions

The main contribution of this paper is the study of a new notion of a strategy for games on recursive/hierarchic state machines, namely the notion of a modular strategy. This is a very natural notion and in fact corresponds to a game between a team of players (one player for each module) against a global adversary. While such team games have been studied before, they usually turn out to be undecidable; however, solving for modular strategies in our framework is decidable.

In terms of applications, we believe that modular strategies are natural and appropriate in the context of synthesizing assumptions a set of modules makes about its environment in order to satisfy its specification. For example, in [12], the authors design interfaces for software modules by solving global games on pushdown systems; these interfaces are however interfaces to a global environment. If the environment itself is made of several components, then our framework of modular strategies allows the synthesis of separate interfaces to each of these environment components in a natural way. In this paper, we have concentrated solely on reachability and safety specifications. Further, these specifications were stated simply in terms of a set of states of the game graph, and not as a separate specification independent of the game. Recently, in [5], we have removed both these restrictions and studied recursive games against specifications given by automata (over infinite words) or temporal logics such as LTL. The results of the current paper scale well to these specifications and result in algorithms that are still polynomial in the game graph and exponential in the number of exits.

References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, M. Yannakakis, Analysis of recursive state machines, ACM Trans. Programming Languages and Systems, 2004, to appear.
[2] R. Alur, P. Černý, P. Madhusudan, W. Nam, Synthesis of interface specifications for Java classes, in: 32nd ACM Symp. Principles of Programming Languages, 2005, pp. 98–109.
[3] R. Alur, K. Etessami, M. Yannakakis, Analysis of recursive state machines, in: Proc. 13th Internat. Conf. Computer Aided Verification, CAV'01, Lecture Notes in Computer Science, Vol. 2102, Springer, Berlin, 2001, pp. 207–220.
[4] R. Alur, T.A. Henzinger, O. Kupferman, Alternating-time temporal logic, J. ACM 49 (5) (2002) 1–42.
[5] R. Alur, S. La Torre, P. Madhusudan, Modular strategies for infinite games on recursive graphs, in: Proc. CAV 2003, 15th Internat. Conf. Computer Aided Verification, Lecture Notes in Computer Science, Vol. 2725, Springer, Berlin, 2003, pp. 67–79.
[6] T. Ball, S.K. Rajamani, Bebop: a symbolic model checker for boolean programs, in: Proc. SPIN 2000 Workshop on Model Checking of Software, Lecture Notes in Computer Science, Vol. 1885, Springer, Berlin, 2000, pp. 113–130.
[7] T. Ball, S. Rajamani, The SLAM project: debugging system software via static analysis, in: Proc. 29th Annu. ACM Symp. Principles of Programming Languages, 2002, pp. 1–3.
[8] M. Benedikt, P. Godefroid, T. Reps, Model checking of unrestricted hierarchical state machines, in: Proc. ICALP 2001, 28th Internat. Colloq. Automata, Languages, and Programming, Lecture Notes in Computer Science, Vol. 2076, Springer, Berlin, 2001, pp. 652–666.
[9] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: applications to model-checking, in: Proc. Eighth Conf. Concurrency Theory, Warsaw, July 1997, Lecture Notes in Computer Science, Vol. 1243, Springer, Berlin, 1997, pp. 135–150.
[10] J.R. Büchi, L.H.G. Landweber, Solving sequential conditions by finite-state strategies, Trans. AMS 138 (1969) 295–311.
[11] T. Cachat, Symbolic strategy synthesis for games on pushdown graphs, in: Proc. 29th Internat. Colloq. Automata, Languages and Programming, ICALP'02, Lecture Notes in Computer Science, Vol. 2380, 2002, pp. 704–715.
[12] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, F.Y.C. Mang, Interface compatibility checking for software modules, in: Proc. 14th Internat. Conf. Computer-Aided Verification (CAV '02), Lecture Notes in Computer Science, Vol. 2404, Springer, Berlin, 2002, pp. 428–441.
[13] A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, Alternation, J. Assoc. Comput. Mach. 28 (1) (1981) 114–133.
[14] L. de Alfaro, T.A. Henzinger, Interface automata, in: Proc. Ninth Annu. Symp. Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109–120.
[15] E.A. Emerson, Model checking and the mu-calculus, in: N. Immerman, Ph. Kolaitis (Eds.), Proc. DIMACS Symp. Descriptive Complexity and Finite Models, American Mathematical Society Press, 1997, pp. 185–214.
[16] M. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, W. Freeman and Co., San Francisco, 1979.
[17] D. Harel, D. Raz, Deciding emptiness for stack automata on infinite trees, Inform. and Comput. 113 (2) (1994) 278–299.
[18] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, W. Weimer, Temporal-safety proofs for systems code, in: CAV 02: Proc. 14th Conf. Computer Aided Verification, Lecture Notes in Computer Science, Vol. 2404, Springer, Berlin, 2002, pp. 526–538.
[19] J.M.E. Hyland, C.-H.L. Ong, On full abstraction for PCF: I, II, and III, Inform. and Comput. 163 (2) (2000) 285–408.
[20] O. Kupferman, M.Y. Vardi, Church's problem revisited, Bull. Symbolic Logic 5 (2) (1999) 245–263.
[21] O. Kupferman, M. Vardi, P. Wolper, Module checking, Inform. and Comput. 164 (2) (2001) 322–344.
[22] P. Madhusudan, P.S. Thiagarajan, A decidable class of asynchronous distributed controllers, in: Proc. 13th Internat. Conf. Concurrency Theory (CONCUR '02), Lecture Notes in Computer Science, Vol. 2421, Springer, Berlin, 2002, pp. 145–160.

[23] R. McNaughton, Infinite games played on finite graphs, Ann. Pure Appl. Logic 65 (1993) 149–184.
[24] D.E. Muller, P.E. Schupp, The theory of ends, pushdown automata, and second-order logic, Theoret. Comput. Sci. 37 (1985) 51–75.
[25] G.L. Peterson, J.H. Reif, Multiple-person alternation, in: Proc. 20th IEEE Symp. Foundations of Computer Science, 1979, pp. 348–363.
[26] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proc. 16th ACM Symp. Principles of Programming Languages, Austin, January 1989.
[27] A. Pnueli, R. Rosner, Distributed reactive systems are hard to synthesize, in: Proc. 31st IEEE Symp. Foundation of Computer Science, 1990, pp. 746–757.
[28] K. Rudie, W.M. Wonham, Think globally, act locally: decentralized supervisory control, IEEE Trans. Automat. Control 37 (11) (1992) 1692–1708.
[29] C. Szyperski, D. Gruntz, S. Mure, Component Software—Beyond Object-Oriented Programming, Addison-Wesley, ACM Press, 2002.
[30] W. Thomas, Languages, automata, and logic, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Language Theory, Vol. III, Springer, Berlin, 1997, pp. 389–455.
[31] W. Thomas, Infinite games and verification, in: Proc. Internat. Conf. Computer Aided Verification CAV'02, Lecture Notes in Computer Science, Vol. 2404, Springer, Berlin, 2002, pp. 58–64.
[32] I. Walukiewicz, Pushdown processes: games and model-checking, Inform. and Comput. 164 (2) (2001) 234–263.
[33] M. Yannakakis, Graph-theoretic methods in database theory, in: Proc. Ninth ACM Symp. Principles of Database Systems, 1990, pp. 230–242.