Modular Synthesis with Open Components[*]

Ilaria De Crescenzo and Salvatore La Torre

Dipartimento di Informatica
Università degli Studi di Salerno

Abstract. We introduce and solve a new component-based synthesis problem that combines the synthesis from libraries of recursive components introduced by Lustig and Vardi with the modular synthesis introduced by Alur et al. for recursive game graphs. We model the components of our libraries as game modules of a recursive game graph with unmapped boxes, and consider as correctness specification a target set of vertices. To solve this problem, we give an exponential-time fixed-point algorithm that computes annotations for the vertices of the library components by exploring them backwards. We also show a matching lower-bound via a direct reduction from linear-space alternating Turing machines, thus proving EXPTIME-completeness. Finally, we give a second algorithm that solves this problem by annotating in a table the result of many local reachability game queries on each game component. This algorithm is exponential only in the number of the exits of the game components, and thus shows that the problem is fixed-parameter tractable.

1 Introduction

Synthesis is the construction of a system that satisfies a given correctness specification. This problem has been studied in different settings, and in particular the controller synthesis problem has a natural formulation as a two-player game (see [13,14]). Given a description of the system, where some of the choices depend upon the input and some represent uncontrollable nondeterminism (which may depend on the interaction with the external environment), the controller synthesis problem asks to determine a controller that supplies inputs to the system such that this satisfies a given correctness specification. Synthesizing a controller corresponds to computing winning strategies in a two-player game.

For pushdown systems modeled as recursive game graphs, where the system is composed of modules that can call each other in a potentially recursive manner (the game counterpart of recursive state machines [1]), it naturally arises the notion of *modular strategy* [3]. Asking for a modular strategy in a recursive game graph equals to require that the synthesized controller is formed of a set of finite state controllers (thus adhering to the modular design of controllers), one for each of the system modules. In executing such a controller, whenever a

[*] This work was partially funded by the MIUR grants FARB 2010-2011-2012, Università degli Studi di Salerno (Italy).

module is called, the finite state controller for that module re-starts, i.e., it is oblivious of the previous history in the computation.

Component-based design plays a key role in configurable and scalable development of efficient hardware as well as software systems. For example, it is current practice to design specialized hardware using some base components that are more complex than universal gates at bit-level, and programming by using library features and frameworks.

In the modular synthesis for recursive game graphs, the call-return structure is given and cannot be modified. Therefore, the synthesis process concerns only the internal structure of each module and the modules cannot be freely composed. The work presented in this paper goes in the direction of providing new results for the automatic synthesis from components. In particular, we formulate and solve a new modular component-based synthesis problem for recursive game modules that, besides requesting the modularity of the solutions, allows also to re-configuring the call-return structure and using multiple instances of a same game module, each instance being controlled in a possibly different manner.

The game modules for our component-based synthesis are taken from a finite set (*library*) of *game components*. Game components differ from game modules in that the boxes are not mapped to any module (as an empty position in a code where we could insert a function call). Namely, a game component is a two-player finite game graphs with two kinds of vertices: standard *nodes* and *boxes*. Each box has *call* and *return* points, and each component has distinguished *entry* and *exit* nodes. The edges are *from* a node or a return *to* a node or a call within the same component. Moreover, the nodes and the returns are split among the two players (pl_0 and pl_1).

The correctness specification is given as a set of target exits $\mathcal{T}$ of a game component C_{main}. The *modular synthesis problem* asks to construct (1) a recursive game graph G by using as game modules copies of the library components and (2) a modular strategy f for pl_0 in G such that: all the maximal plays σ, starting from the entry of C_{main} and that conform to f, visit a vertex in $\mathcal{T}$. We solve this problem and address its computational complexity.

Our first contribution is a fixed-point algorithm $\mathcal{A}_1$ that decides in exponential time the above modular synthesis problem. This algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b\in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of another component C_b or undefined. Each such tuple summarizes for vertex u a reachable *local target* E (via a modular strategy of pl_0) and a set of *assumptions* $\{\mu_b\}_{b\in B}$ that are used to get across the boxes in order to reach the local target. We start from the tuples of the target exits $\mathcal{T}$ and then propagate the search backwards in the game components. Internally to each component, the search proceeds as in the standard attractor set construction [12] and it is propagated through calls to other components from the returns to the exits and then back from the entries to the calls. In this, tuples that have incompatible assumptions or refer to a different local target are treated as belonging to different searches and thus are not used together in the update rules.

Our second contribution is to show a matching lower bound by a reduction from linear-space alternating Turing machines. In the reduction, we use only four game components and $O(n)$ exits, where n is the number of cells used in the configurations of the Turing machine.

Finally, we delve deeper in the computational complexity of this problem, and give a second decision algorithm $\mathcal{A}_2$ for it. The main idea here is to solve many reachability game queries "locally" to each game component and maintain a table with the obtained results to avoid recomputing. Each table entry corresponds to a game component and a set of its exits (used as targets in the query), and for the successful queries, contains a link to each table entry that has been used to reach the target (we look up into the table to propagate the search across the boxes). We observe that $\mathcal{A}_2$ takes time exponential only in the number of exits, while $\mathcal{A}_1$ takes time exponential also in the number of boxes. This is due mainly to the fact that $\mathcal{A}_1$ may compute and store exponentially many different ways of assigning the boxes to modules, in contrast, $\mathcal{A}_2$ computes and stores just one of them. Therefore, since alternating reachability in finite game graphs is already PTIME-hard, by algorithm $\mathcal{A}_2$ we get that the considered problem is PTIME-complete when the number of exits is fixed.

Related Work. The synthesis problem addressed in this paper combines the synthesis from libraries of recursive components [11] with the synthesis of modular strategies for recursive game graphs [3]. In fact, if the game components of the considered library do not contain pl_1 vertices, the problem reduces to a synthesis problem from recursive component libraries. If we instead constrain the solution to use at most one copy for each game component, we can encode our synthesis problem as a synthesis of modular strategies for recursive game graphs.

We recall that in [11] the components are modeled with transducers with call-return structures, and the correctness specification is given as a temporal logic formula over nested words. The same synthesis problem with LTL specifications and components modeled as standard finite-state transducers is addressed in [10]. In [4], this problem is formulated for synthesizing hierarchical systems bottom-up with respect to a different μ-calculus specification for each component in the hierarchy. All these synthesis problem turn out to be 2EXPTIME-complete. The synthesis from libraries of components with simple specifications has been also implemented in tools (see [8] and references therein).

Modular synthesis of recursive game graphs with several classes of ω-regular specifications is solved in [2] and is shown to be EXPTIME-complete already with finite automata specifications. The computational complexity of this problem turns out to be NP-complete for reachability specifications [3]. In [6], the modular synthesis of recursive game graphs is shown to be 2EXPTIME-complete with respect to visibly pushdown specifications. A solution to CaRet games that computes winning strategies that are modular for the recursive game graph extended with set of subformulas of the specification formula is given in [5]. The notion of modular strategy is also of independent interest and has recently found application in the automatic transformation of programs for ensuring security policies in privilege-aware operating systems [7].

2 A Modular Synthesis Problem

In this section, we define our modular synthesis problem. For this, we introduce first some preliminary notions and recall known ones.

For $n \in \mathbb{N}$, with $[n]$ we denote the set of naturals i s.t. $1 \leq i \leq n$.

Library of (game) Components. For $h, k \in \mathbb{N}$, a (h, k)-*component* is a finite graph with two kinds of vertices, the standard *nodes* and the *boxes*, and with h *entry* nodes and k *exit* nodes. Each box has h *call* points and k *return* points, and the edges take from a node/return to a node/call in the component.

Formally, for a box b, we denote with (i, b) the i-th call of b for $i \in [h]$, and with (b, i) the i-th return of b for $i \in [k]$. A (h, k)-*component* is a tuple (N, B, En, Ex, δ) where N is a finite set of nodes, B is a finite set of boxes, $En \subseteq N$ is the set of entries, $Ex \subseteq N$ is the set of exits, and $\delta : N \cup Retns \rightarrow 2^{N \cup Calls}$ where $Retns = \{(b, i) \mid b \in B, i \in [k]\}$ and $Calls = \{(i, b) \mid b \in B, i \in [h]\}$. The calls, returns and nodes of a component form its set of vertices. In the following, when we do not need to specify h and k, we simply write component.

A *game component* is a component whose nodes and returns are split into two sets P^0 and P^1, where P^0 is the set of player 0 (pl_0) positions and P^1 is the set of player 1 (pl_1) positions. We denote it as a tuple $(N, B, En, Ex, \delta, P^0, P^1)$.

For $h, k > 0$, a *library* of (game) components is a finite set $\mathcal{L}ib = \{C_i\}_{i \in [n]}$ where each C_i is a (game) (h, k)-component.

To ease the presentation we make the following standard assumptions:

- there is only one entry node for every (game) component and thus just one call for each box, i.e., we refer to (game) $(1, k)$-components;
- in each (game) component there are no transitions taking to its entry and no transitions leaving from its exits, i.e., the entries are sources and the exits are sinks in the graph representation of the component;
- there is no transition from a return to a call, i.e., two boxes are not directly connected by a single transition.

Instances from a Library. Intuitively, an instance of a (game) component C is a copy A of C where each box is mapped to an instance of a (game) component (possibly A itself). Depending on whether we consider a library of components or of game components, the instances define a *recursive state machine* [1] or a *recursive game graph* [3].

Fix a library $\mathcal{L}ib = \{C_1, \ldots, C_n\}$ of game components.

A *recursive game graph* from $\mathcal{L}ib$ is $G = (M, m_{in}, \{S_m\}_{m \in M})$ where M is a finite set of module names, $m_{in} \in M$ is the name of the initial module and for each $m \in M$, S_m is a game module. A *game module* S_m is defined as $(N_m, B_m, Y_m, \{e_m\}, Ex_m, \delta_m, P^0_m, P^1_m)$ where:

- $Y_m : B_m \to (M \setminus \{m_{in}\})$ is a labeling function that maps every box to a game module;
- $(N_m, B_m, \{e_m\}, Ex_m, \delta_m, P^0_m, P^1_m)$ is equal to a component C of $\mathcal{L}ib$ up to a renaming of nodes and boxes such that calls and returns of a box b are 1-to-1 mapped to the entries and the exits of $M_{Y_m(b)}$, that is, denoting $Ex_{Y_m(b)} = \{x_1, \ldots, x_k\}$: the call of b is renamed to $(e_{Y_m(b)}, b)$ and each return (b, i) is renamed to (b, x_i).

The calls, returns and vertices of S_m are denoted respectively $Calls_m$, $Retns_m$ and V_m. We also assume the following notation: $V = \bigcup_m V_m$ (set of all vertices); $B = \bigcup_m B_m$ (set of all boxes); $Calls = \bigcup_m Calls_m$ (set of all calls); $Retns = \bigcup_m Retns_m$ (set of all returns); and $P^i = \bigcup_m P^i_m$ for $i = 0, 1$ (set of all positions of pl_i).

The definition of a *recursive state machine* from $\mathcal{L}ib$ can be obtained from that of recursive game graph by ignoring the splitting among pl_0 and pl_1 nodes.

A (global) *state* of G is composed of a call stack and a vertex. Formally, the states are of the form $(\gamma, u) \in B^* \times V$ where $\gamma = b_1 \ldots b_h$, $b_1 \in B_{m_{in}}$, $b_{i+1} \in B_{Y(b_i)}$ for $i \in [h-1]$ and $u \in V_{Y(b_h)}$. In the following, for a state $s = (\gamma, u)$, we denote with $V(s)$ its vertex, that is $V(s) = u$.

A *play* of G is a (possibly finite) sequence of states $s_0 s_1 s_2 \ldots$ such that $s_0 = (\epsilon, e_{m_{in}})$ and for $i \in \mathbb{N}$, denoting $s_i = (\alpha_i, u_i)$, one of the following holds:

- **Internal move:** $u_i \in (N_m \cup Retns_m) \setminus Ex_m$, $u_{i+1} \in \delta_m(u_i)$ and $\alpha_i = \alpha_{i+1}$;
- **Call to a module:** $u_i \in Calls_m$, $u_i = (b, e_{m'})$, $u_{i+1} = e_{m'}$ and $\alpha_{i+1} = \alpha_i.b$;
- **Return from a call:** $u_i \in Ex_m$, $\alpha_i = \alpha_{i+1}.b$, and $u_{i+1} = (b, u_i)$.

Modular Strategies. A *strategy* of a player pl is a function f that associates a legal move to every play ending in a node controlled by pl. A *modular strategy* [3] for G consists of a set of local strategies, that are used together as a global strategy for a player. A local strategy for a game module S can only refer to the local memory of S, i.e. the sequence of S vertices that are visited in the play in the current invocation of S.

Formally, fix $j \in \{0, 1\}$. A *modular strategy* f of pl_j is a set of functions $\{f_m\}_{m \in M}$, one for each game module, where for every m, $f_m : V^*_m.P^j_m \to V_m$ such that $f_m(\sigma.u) \in \delta_m(u)$ for every $\sigma \in V^*_m, u \in P^j_m$.

Fix a play $\sigma = s_0 s_1 \ldots s_n$ where $s_i = (\gamma_i, u_i)$ for any i. Denote with $\sigma_i = s_0 s_1 \ldots s_i$, i.e., the prefix of σ up to index i. With $ctr(\sigma_i)$ we denote $m \in M$ such that $u_i \in V_m$, that is the name of the game module where the control is after σ_i. The *local history* at σ_i, denoted $\lambda(\sigma_i)$, is the maximal sequence of S_m vertices u_j, $j \leq i$, starting with the most recent occurrence of entry e_m where $m = ctr(\sigma_i)$.

A play σ *conforms to* a modular strategy $f = \{f_m\}_{m \in M}$ of pl_j if for every $i <\mid \sigma \mid$, denoting $ctr(\sigma_i) = m$, $u_i \in P^j_m$ implies that $u_{i+1} = f_m(\lambda(\sigma_i))$.

Modular Synthesis from Libraries of Game Components. A *modular game over a library* is $(\mathcal{L}ib, C_{main}, \mathcal{T})$ where $\mathcal{L}ib$ is a library of game components, $C_{main} \in \mathcal{L}ib$ and $\mathcal{T}$ is a set of exits of C_{main}.

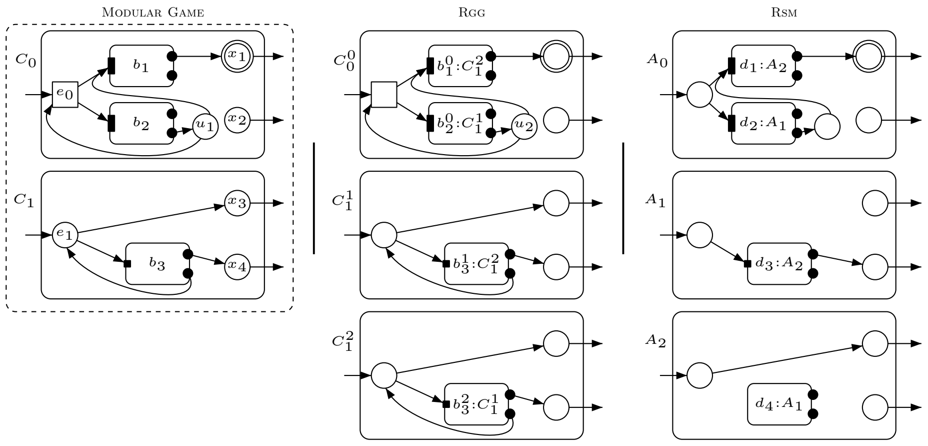

Fig. 1. An example of modular synthesis

Given an instance $(\mathcal{L}ib, C_{main}, \mathcal{T})$ of a modular game over a library, the *modular synthesis problem* is the problem of determining whether: for some recursive game graph G from $\mathcal{L}ib$ whose initial module is an instance of C_{main}, there exists a modular strategy f for pl_0 in G such that all the maximal plays that conform to f reach an exit of the initial module of G that corresponds to an exit in $\mathcal{T}$.

Such a strategy f for pl_0 is called a *winning modular strategy.*

Example. We illustrate the definitions with an example. In the first column of Fig. 1, we give $(\mathcal{L}ib, C_0, \{x_1\})$, an instance of a modular game over a library of game components. Each game component has two exits, and $\mathcal{L}ib$ is composed of two game components C_0 and C_1. In the figure, we denote the nodes of pl_0 with circles and the nodes of pl_1 with squares. Rounded squares are used to denote the boxes. The target is marked with a double circle. C_0 has one entry e_0, two exits x_1 and x_2, and two boxes b_1 and b_2. C_1 has one entry e_1, two exits x_3 and x_4, and one box b_3.

In the second column of the figure, we show one of the possible recursive game graphs that can be obtained from $\mathcal{L}ib$ and whose initial module C_0^0 is an instance of C_0. Note that we have marked as target the vertex of C_0^0 that corresponds to (i.e., is a copy of) x_1. The other modules C_1^1 and C_1^2 are instances of C_1. Note that each box now is mapped to a game module, for example b_1^0 is mapped to C_1^2. Also, the box b_3^1 of C_1^1 is mapped to C_1^2 and the box b_3^2 of C_1^2 is mapped to C_1^1 thus forming a cycle in the chain of recursive calls.

Consider a modular strategy for pl_0, where the local strategy of C_0^0 selects the call from u_2, the local strategy of C_1^1 selects the call from its entry and the local strategy for C_1^2 selects the upper exit from its entry. This strategy is winning and modular. In the third column of the figure, we show a recursive state machine, obtained from the considered recursive game graph by resolving the moves of pl_0 according to this modular strategy. To simplify the figure, we have deleted all the unreachable transitions. Clearly, each run of this machine reaches the target. Also, note that in the considered game it is not possible to win if we do not instantiate at least two instances of C_1.

3 Solving Our Modular Synthesis Problem

In this section, we describe an exponential-time fixed-point algorithm to solve the modular synthesis problem.

We fix a library of game components $\mathcal{L}ib = \{C_{main}, C_1, \ldots, C_n\}$ and a target set $\mathcal{T}$ of C_{main} exits.

Intuitively, our algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b \in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of a game component or undefined (we use $\bot$ to denote this). The intended meaning of such tuples is that: there is a local strategy f of pl_0 in C such that starting from u, each maximal play conforming to f reaches an exit within E, under the assumption that: for each box $b \in B$, if μ_b is defined, then from the call of b the play continues from one of the returns of b corresponding to a $x \in \mu_b$ (if μ_b is undefined means that no play conforming to f visits b starting from u). Thus, each tuple $(u, E, \{\mu_b\}_{b \in B})$ summarizes for vertex u a reachable *local target* E and a set of *assumptions* $\{\mu_b\}_{b \in B}$ that are used to get across the boxes.

For computing Φ, we use the concept of compatibility of the assumptions. Namely, we say that two assumptions μ and μ' are *compatible* if either $\mu = \mu'$, or $\mu' = \bot$, or $\mu = \bot$ (i.e., there is at most one assumption that has been done). Moreover, we say that the assumptions $\mu_1, \ldots, \mu_m$ are *passed to* μ if $\mu = \bigcup_{i \in [m]} \mu_i$ (we assume that $\bot \cup X = X \cup \bot = X$ holds for each set X).

The set Φ is initialized with all the tuples of the form $(u, \mathcal{T}, \{\bot\}_{b \in B_{main}})$ where $u \in \mathcal{T}$ and B_{main} is the set of boxes of C_{main}. Then, Φ is updated by exploring the components backwards according to the game semantics, and in particular: within the components, tuples are propagated backwards as in an attractor set construction, by preserving the local target and passing to a node the assumptions of its successors (provided that multiple assumptions on the same box are are passed they are pairwise compatible); the exploration of a component is started from the exits with no assumptions on the boxes, whenever the corresponding returns of a box b have been discovered with no assumptions on b; the visit of a component is resumed at the call of a box b, whenever (1) there is an entry of a component that has been discovered with local target X and (2) there is a set of b returns corresponding to the exits X with no assumptions on b (thus, that can be responsible for discovering the exits in X as in the previous case) and with compatible assumptions on the remaining boxes; if this is the case, then the call is discovered with the assumption X on box b and passing the local target and the assumptions on the other boxes as for the above returns.

Below, we denote with b_x the return of a box b corresponding to an exit x (recall that all game components of a library have the same number of exits, and so do the boxes). The update rules are formally stated as follows:

Update 1: For a pl_0 vertex v, we add $(v, E, \{\mu_b\}_{b\in B})$ provided that there is a transition from v to u and $(u, E, \{\mu_b\}_{b\in B}) \in \Phi$ (the local target and the assumptions of a v successor are passed on to a pl_0 vertex v).

Update 2. For a pl_1 vertex v, denote $u_1, \ldots, u_m$ all the vertices s.t. there is a transition from v to u_i, $i \in [m]$, then we add $(v, E, \{\mu_b\}_{b\in B})$ to Φ provided that for each $i, j \in [m]$ and $b \in B$: (1) there is a $(u_i, E_i, \{\mu_b^i\}_{b\in B}) \in \Phi$, (2) $E_i = E_j$, (3) μ_b^i and μ_b^j are compatible, and (4) $\mu_b = \bigcup_{i\in[m]} \mu_b^i$ (all the v successors must be discovered under the same target and with compatible assumptions; target and assumptions are passed on to a pl_1 vertex v).

Update 3. For an exit u, we add a tuple $(u, E, \{\perp\}_{b\in B'})$ to Φ provided that $u \in E$ and for a box b' it holds that there are tuples $(b'_x, E_x, \{\mu_b^x\}_{b\in B}) \in \Phi$, one for each $x \in E$, such that for all $x, y \in E$ and $b \in B$, (1) $\mu_{b'}^x = \perp$, (2) $E_x = E_y$, and (3) μ_b^x and μ_b^y are compatible (the discovery of the exits follows the discovery of the corresponding returns under compatible assumptions and the same local target).

Update 4. For a call u of a box b', we add a tuple $(u, E_u, \{\mu_b^u\}_{b\in B})$ to Φ provided that (i) there is an entry e s.t. $(e, E_e, \{\mu_b^e\}_{b\in B'}) \in \Phi$, (ii) for each return b'_x, $x \in E_e$, there is a tuple $(b_x, E, \{\mu_b^x\}_{b\in B}) \in \Phi$ s.t. all these tuples satisfy (1), (2) and (3) of Update 3, and moreover, (iii) $E_u = E$, $\mu_b^u = \bigcup_{x\in E_e} \mu_b^x$ for $b \neq b'$, and $\mu_{b'}^u = E_e$ (the discovery of a call u of box b' follows the discovery of an entry e from exits E_e that in turn have been discovered by matching returns b'_x, $x \in E$; thus on u we propagate the local target and the assumptions on the boxes $b \neq b'$ of the returns b'_x and make an assumption E_e on box b').

We compute Φ as the fixed-point of the recursive definition given by the above rules and outputs "YES" iff $(e, \mathcal{T}, \{\mu_b\}_{b\in B_{main}}) \in \Phi$ for the entry e of C_{main}.

Observe that, the total number of tuples of the form $(u, E, \{\mu_b\}_{b\in B})$ is bounded by $|\mathcal{L}ib|\, 2^{O(k\beta)}$ where k is the number of exits of each game component in $\mathcal{L}ib$ and β is the maximum over the number of boxes of each game component. Therefore, the algorithm always terminates and takes at most time exponential in k and β, and linear in the size of $\mathcal{L}ib$.

Soundness of the algorithm is a consequence of the fact that each visit of a game component is done according to the standard attractor set construction, and repeated explorations of each component are kept separate by allowing to progress backwards in the graph only with the same local target and compatible assumptions on the boxes. By not allowing to change the box assumptions (when defined), we ensure that we cannot cheat by using different assumptions in repeated visits of a box within the same exploration. The computed strategy is clearly modular since we compute it locally to each graph component. Note that we can end up computing more than a local strategy for each graph component, but this does not break the modularity of the solution since this happens when in the computed solution we use different instances of the component. Also, observe that for each game component we construct at most a local strategy for each possible subset of its exits, thus we bound the search of a solution to modular strategies of this kind.

To prove completeness, we first observe that using standard arguments one can show that:

Lemma 1. *If there is a modular winning strategy for an instance of the modular synthesis problem over a library $\mathcal{L}ib$, then there is a winning modular strategy f for a recursive game graph G from $\mathcal{L}ib$ such that: for each two instances S and S' of a same game component in $\mathcal{L}ib$, the sets of exits visited along any play conforming to f in S and S' differ.*

Observe that by the above lemma, we can restrict the search for a solution within the modular strategies of the instances of a $\mathcal{L}ib$ that have at most 2^k copies of each game component, where k is the number of exits for the components. Therefore, combining this with the results from [3] we get a simple argument to show membership to NEXPTIME of the considered problem.

The next step in the completeness argument is to show that if there is a winning modular strategy f as for Lemma 1, then our algorithm outputs YES. Denoting with G the recursive game graph from $\mathcal{L}ib$ for which f is winning, this can be shown by proving by induction on the structure of G that: if on a game module S of G that is an instance of $C \in \mathcal{L}ib$, f forces to visit a set of exits corresponding to the exits X of C, then the algorithm adds to Φ the tuples $(x, X, \{\bot\}_{b \in B})$ for each $x \in X$ and eventually discovers the entry of C with local target X. We omit the proof of this here.

Therefore, we get that the algorithm is a solution of the modular synthesis problem from game component libraries, and the following theorem holds.

Theorem 1. *The modular synthesis problem from libraries of game components with k exits and at most β boxes can be solved in time linear in the size of $\mathcal{L}ib$ and exponential in k and β.*

4 Computational Complexity Analysis

Lower-bound. We reduce the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components, thus showing EXPTIME-hardness for this problem.

Consider a linear-space alternating Turing machine A and an input word $w = a_1 \dots a_n$. Without loss of generality, we assume that the transition function δ of A is the union of two functions δ_1 and δ_2 where $\delta_i : Q \times \Sigma \to \{L, R\} \times Q$ for $i \in [2]$, and Q is the set of control locations, Σ is the tape alphabet, and L/R cause to move the tape head to left/right. A configuration of A is represented as $b_1 \dots (q, b_i) \dots b_n$ where b_j is the symbol at cell j of the input tape for $j \in [n]$, q is the control state and the tape head is on cell i. The control states are partitioned into states where the $\exists$-player can move, and states where the $\forall$-player can move. A computation of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation ρ that reaches a configuration with a final state on all the plays conforming to ρ.

Denoting $h = n\,|\Sigma|\,(|Q|+1)$, fix two sets $X = \{x_1, \dots, x_h\}$ and $Y = \{y_1, \dots, y_h\}$ such that each x_i and y_i correspond exactly to a symbol and a position in a configuration of A (i.e., for each symbol in $\Sigma \cup Q \times \Sigma$ we have exactly n variables from X

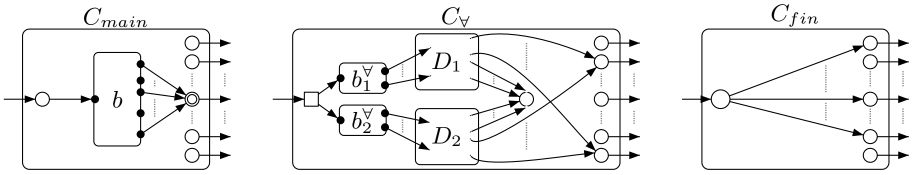

Fig. 2. Graphical representation of the game components C_{main}, $C_\forall$ and C_{fin}

and n from Y, one for each position on the tape). We can encode each configuration $\sigma_1 \dots \sigma_n$ of A by setting to true a variable x_j (resp. y_j) iff it corresponds to a σ_i for $i \in [n]$ (that is, to a configuration symbol and its position in the configuration). It is well-known that for each δ_i, we can construct a Boolean circuit (using only the logical gates AND and OR) with inputs $\bar{x} = x_1 \dots x_h$ and outputs $\bar{y} = y_1, \dots, y_h$, such that if $\bar{x}$ is an encoding of a configuration, then $\bar{y}$ is the next configuration after the application of the only possible transition of δ_i.

From each such circuit we can construct a game graph by replacing each AND gate with a node of pl_1 and each OR gate with a node of pl_0. We denote with D_1 and D_2 the game graphs corresponding to the above circuits for δ_1 and δ_2, respectively. The encoding of the bits is done by reachability, that is, true at an input x_i corresponds to connecting it to a vertex that can lead to the target, and false otherwise. Since the circuits compute a next configuration, from each output wire y_i that evaluates to true we will be able to get to the target by a strategy that resolves the choices on the pl_0 nodes (and thus the OR gates), and this will not be possible for those y_i that evaluates to false.

We construct a library $\mathcal{L}ib$ containing exactly the game components C_{main}, $C_\forall$, $C_\exists$, and C_{fin} (see Fig. 2). Each component has exactly h exits, each one corresponding to a variable x_i for $i \in [h]$. In C_{main}, we arbitrarily select an exit as the only vertex in the target $\mathcal{T}$, and link to it all the returns of the box that encode the initial configuration (we can assume that A has only one initial state). In $C_\forall$, all the exits are wired as inputs to both D_1 and D_2 except for those that correspond to states of the $\exists$-player. We add a pl_0 node that has no out-going edges and is wired as input to D_1 and D_2 for the remaining inputs. The outputs of D_1 and D_2 are wired respectively to the boxes $b_1^\forall$ and $b_2^\forall$, and the calls of these boxes are connected to the entry, that is a pl_1 node. $C_\exists$ is as $C_\forall$ except that the entry is a pl_0 node and the exits that are not connected correspond to $\forall$-player states. The component C_{fin} has just the entry and the exits. The entry is a pl_0 node and is connected to all the exits that correspond to a final state.

It is simple to verify that if, starting from an instance of C_{main}, we map the boxes such that to reproduce an accepting computation of A, then we get a recursive game graph that admits a modular winning strategy of pl_0. Vice-versa, suppose that there is a modular winning strategy of pl_0 in the synthesis problem $(\mathcal{L}ib, C_{main}, \mathcal{T})$. First, observe that since the returns from which we reach the target encode a legal initial configuration, each game module to which we map the box b will have the corresponding exits with the same property. Moreover, in order to reach backwards the entries of all the used instances of C_{main}, $C_\forall$, and $C_\exists$, at some point we need to use a copy of C_{fin}. Now, if the initial state is a $\forall$-player state and we map b to an instance of $C_\exists$, since the exit encoding the head position and the state will not be wired to D_1 and D_2, in all the modules below in the hierarchy of calls, none of such exits will be connected to the target. Thus, also the entry of each copy of C_{fin} in this hierarchy would not be connected to the target, and so all the entries up to the entry of the copy of C_{main}, thus contradicting the hypothesis. A contradiction can be shown also in the dual case. Thus, at any point we must have mapped each box to an instance of either $C_\exists$ or $C_\forall$ depending on whether the next move is of the $\exists$-player or the $\forall$-player. Since, the graphs D_1 and D_2 ensure the correct propagations of the reachability according to the computed configurations, we can correctly reconstruct a computation ρ of A from the modular strategy. Moreover, since a winning modular strategy ensures that each maximal sequence of module calls ends with a call to an instance of C_{fin}, then each play of ρ ends in a final configuration and thus ρ is accepting, that concludes the proof.

Lemma 2. *There is a polynomial-time reduction from the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components. Moreover, the resulting library has four game components each one with at most two boxes and a number of exits which is linear in the size of the input word.*

Complexity and fixed-parameter tractability. The algorithm from the previous section, say $\mathcal{A}_1$, shows membership to EXPTIME for the modular synthesis problem. Therefore, by Lemma 2, we get:

Theorem 2. *The modular reachability problem is* EXPTIME*-complete.*

Note that $\mathcal{A}_1$ takes time exponential in both the number of boxes β and the number of exits k. We sketch a different algorithm that shows that this problem is indeed in PTIME when the number of exits for each game component is fixed.

The main idea is to solve many reachability game queries on standard finite game graphs, where each query asks to determine for a game component C and a subset of its exits E: if there exists a modular strategy f of pl_0 such that all the maximal plays, which conform to f and start from the entry of C, reach one of the exits from E. To avoid recomputing, the results of such queries are stored in a table T, and the algorithm halts when no more queries can be answered positively.

To solve the query for a component C and a set of its exits E, we extend the standard attractor set construction. Namely, we accumulate the winning set for pl_0 as usual for nodes and returns. To add the call of a box b, we look in the table for a positively answered query whose target set correspond to returns of b that are already in the winning set. If the entry of C is added to the winning set, then we update the T entry for E and C to YES, and store the links to the table entries that have been used to add the calls (observe that we just need to store exactly a link for each box that is traversed to win in the game query in order to synthesize the recursive game graph and the winning modular strategy).

With similar arguments as those used in Section 3, we can show that pl_0 has a winning modular strategy in the input modular synthesis problem if and only if the T entry for the target set $\mathcal{T}$ is set to YES. Since the size of the table is exponential in k and linear in β, and that solving the "local" reachability games is linear in the size of the game component and in the size of the table, we get that the whole algorithm takes time exponential in k and linear in β (and the size of the library). Since already alternating reachability is PTIME-hard, we get:

Theorem 3. *The modular reachability problem for a fixed number of exits is* PTIME-*complete.*

We observe that $\mathcal{A}_1$ computes all the solutions of the kind as from Lemma 1, by trying all the possible ways of assigning each box with all the game components. This causes the exponential in the number of boxes, but also gives a quite simple and direct way to show completeness. Moreover, the fixed-point updates of $\mathcal{A}_1$ can be implemented quite efficiently and only the sets of exits from which we can reach the target (in a series of calls) are used in the computation.

Algorithm $\mathcal{A}_2$ arbitrarily computes, for each game component and each set of exits, only one assignment of each box with a game module. Moreover, it computes (several times) all the game queries, even those with exits that cannot reach the global target $\mathcal{T}$.

Both algorithms can be used to synthesize the winning modular strategy as a recursive state machine. Also, we can modify them to compute optimal winning modular strategies with respect to some criteria, such as minimizing the number of modules, the depth of the call stack or the number of used exits.

5 Conclusion

In this paper, we have introduced a formulation of the synthesis problem that generalizes both the modular synthesis of recursive game graphs and the synthesis from component libraries. We have solved this problem for reachability specifications, and in particular, we have shown that it is EXPTIME-complete and is fixed-parameter tractable when the number of exits is fixed.

Besides the optimization problems mentioned at the end of previous section, we see several other future directions that could be investigated.

In our formulation, the number of instances of each component that are allowed in a solution is unbounded. It is realistic to consider some limitations, in particular, we plan to investigate variations of the considered synthesis problem where in each solution there is at most one game module that instantiates each component, or where for all the game modules that instantiate a same component we require the same local function.

We have considered only reachability specifications. It is natural to investigate more complex specifications such as regular or pushdown specifications expressed as temporal logic formulas or automata models. Moreover, to synthesize more succinct solutions, it could be interesting to investigate the effect of a hierarchical labeling such as in [9].

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. 27(4), 786–818 (2005)
2. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for infinite games on recursive graphs. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 67–79. Springer, Heidelberg (2003)
3. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for recursive game graphs. Theor. Comput. Sci. 354(2), 230–249 (2006)
4. Aminof, B., Mogavero, F., Murano, A.: Synthesis of hierarchical systems. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 42–60. Springer, Heidelberg (2012)
5. De Crescenzo, I., La Torre, S.: Winning caret games with modular strategies. In: Fioravanti, F. (ed.) CILC. CEUR Workshop Proceedings, vol. 810, pp. 327–331. CEUR-WS.org (2011)
6. De Crescenzo, I., La Torre, S.: Visibly pushdown modular games. Technical Report, University of Salerno, pp. 1–19 (2013)
7. Harris, W.R., Jha, S., Reps, T.: Secure programming via visibly pushdown safety games. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 581–598. Springer, Heidelberg (2012)
8. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 215–224 (May 2010)
9. La Torre, S., Napoli, M., Parente, M., Parlato, G.: Verification of scope-dependent hierarchical state machines. Inf. Comput. 206(9-10), 1161–1177 (2008)
10. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 395–409. Springer, Heidelberg (2009)
11. Lustig, Y., Vardi, M.Y.: Synthesis from recursive-components libraries. In: D'Agostino, G., La Torre, S. (eds.) GandALF. EPTCS, vol. 54, pp. 1–16 (2011)
12. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic 65(2), 149–184 (1993)
13. Thomas, W.: Infinite games and verification (Extended abstract of a tutorial). In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)
14. Thomas, W.: Facets of synthesis: Revisiting church's problem. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 1–14. Springer, Heidelberg (2009)