A General Pushdown Modular Synthesis

Ilaria De Crescenzo and Salvatore La Torre

Dipartimento di Informatica, Università degli Studi di Salerno

Abstract. The modular synthesis from a library of components (Lms) asks to construct a recursive state machine by modularly controlling a finite set of component instances taken from the library. We consider the general problem and some natural restrictions w.r.t. specifications expressed as reachability, deterministic finite automata and deterministic visibly pushdown automata. We show that the Lms problems turn out to be decidable and are EXPTIME-complete except for reachability restricted to single component instances that is NP-complete.

1 Introduction

Component-based design is a main approach for developing configurable and scalable digital systems. In this setting, the reusability of pre-existing components plays a main role. In fact, it is current practice to design specialized hardware using some base components and programming by libraries and frameworks.

A component is a piece of hardware or software that can be directly plugged into a solution or a template that needs to be customized for a specific use. In the sequential world, a general notion of component composition can be obtained by allowing to synthesize some modules from generic templates and then connect them along with other off-the-shelf modules via the call-return paradigm.

In this paper, we study a general synthesis problem for component-based pushdown systems, the *modular synthesis from a library of components* (Lms). The goal is to synthesize a recursive state machine (RSM) S [1] by composing modules instantiated from library components such that all runs of S satisfy a given specification.

We model each component as a game graph with vertices split between player 0 (pl_0) and player 1 (pl_1), and the addition of *boxes* as place-holders for *calls* to components. The library is equipped with a *box-to-component map* that is a partial function from boxes to components. An instance of a component C is essentially a copy of C along with a local strategy that resolves the nondeterminism of pl_0. An RSM S synthesized from a library is a set of instances along with a total function that maps each box in S to an instance of S and is consistent with the box-to-component map of the library.

In this paper, we give a solution to the Lms problem with winning conditions given as internal reachability objectives, or as external deterministic finite automata (FA) and deterministic visibly pushdown automata (VPA) [4]. As a further contribution, we consider two natural restrictions on the admitted solutions: 1) at most one instance of each library component is allowed in the

synthesized RSM (*single-instance* LMS), or 2) all the instances of a same library component must be controlled by a same local controller (*component-based* LMS). Note that in the *component-based* LMS there is no restriction imposed on the local strategy to be synthesized for a component and two instances of the same component can differ in the mapping of the boxes.

The single-instance LMS problem can be reduced to the modular synthesis on recursive game graphs by guessing a total box-to-component map for the library, and thus we immediately get the problem is NP-complete for reachability [3], and EXPTIME-complete for FA [2] and VPA [6] specifications. We show that the LMS and component-based LMS problems are EXPTIME-complete for any of the considered specifications. In particular, the results on reachability are obtained by adapting the results from [7]. The lower bounds for safety and VPA specifications are obtained by a standard reduction from linear-space Turing machines. The upper bound for safety specifications is based on a reduction to tree automata emptiness that is based on the notion of *library tree*: an infinite tree that encodes the library along with a choice for a total box-to-component map where both the components and the total map are unrolled. The construction is structured into several pieces and exploits the closure properties of tree automata under concatenation, intersection and union. The upper bound for VPA specification is obtained by a reduction to safety specification that exploits the synchronization between the stacks of the VPA and the synthesized RSM.

Due to the page limit some details are omitted and we do not argue explicitly the correctness of our constructions though we stress in the description the critical aspects. More details can be found in the Appendix.

Related work. The LMS problem strictly generalizes the modular synthesis on recursive game graphs [3, 2, 6] by allowing to synthesize also the the box-to-module map and synthesize multiple (possibly different) instances from each component. The LMS problem also strictly generalizes the synthesis form libraries of [10, 11] where there is no internal game within the components forming the library. In some sense, our problem generalizes their setting to library of infinitely many elements that are defined as instances of finitely many components. The synthesis from libraries of components with simple specifications has been also implemented in tools (see [9] and references therein). Other synthesis problems dealing with compositions of component libraries are [10, 5], and for modules expressed as terms of the λY-calculus [13]. Other research on pushdown games have focused on the standard notion of winning strategy. We recall that deciding standard pushdown games (i.e., where strategies may be non-modular) is known to be EXPTIME-complete for reachability specifications [16], 2EXPTIME-complete for VPA specifications and 3EXPTIME-complete for temporal logic specifications [8].

2 Modular Synthesis from Libraries

For $n \in \mathbb{N}$ and $0 \leq j < n$ with $[j, n]$ we denote the set of integers i s.t. $j \leq i \leq n$ and with $[n]$ we denote $[1, n]$. Also we let Σ be a finite alphabet.

Library of components. For $k \in \mathbb{N}$, a *k-component* is a finite game graph with two kinds of vertices, the standard *nodes* and the *boxes*, and with an *entry* node and k *exit* nodes. Each box has a *call* point and k *return* points, and each edge takes from a node/return to a node/call in the component. Nodes and returns are split into player 0 (pl_0) positions and player 1 (pl_1) positions.

For a box b, we denote with $(1, b)$ the call of b and with (b, i) the i^{th} return of b for $i \in [k]$. A *k-component* C is a tuple $(N_C, B_C, e_C, Ex_C, \eta_C, \delta_C, P_C^0, P_C^1)$ where N_C is a finite set of nodes, B_C is a finite set of boxes, $e_C \in N_C$ is the entry, $Ex_C : [k] \to N_C$ is an injection that maps each i to the i^{th} exit, $\eta_C : V_C \to \Sigma$ is a labeling map, $\delta : N_C \cup Retns_C \to 2^{N_C \cup Calls_C}$ is a transition function, and P_C^0 (the pl_0 positions) and P_C^1 (the pl_1 positions) form a partition of $N_C \cup Retns_C$ where $Retns_C = \{(b, i) \mid b \in B_C, i \in [k]\}$ (*set of C returns*), $Calls_C = \{(1, b) \mid b \in B_C\}$ (*set of C calls*), and $V_C = N_C \cup Calls_C \cup Retns_C$ (*set of C vertices*).

We introduce the notion of *isomorphism* between two k-components. Intuitively, two components are isomorphic if and only if their game structures are equivalent, that is, the following three conditions must hold: it is a standard isomorphism of labeled graphs, and isomorphic vertices must be assigned to the same player and must be of the same kind.

Two k-components C and C' are *isomorphic*, denoted $C \overset{\text{iso}}{\equiv} C'$, if there exists a bijection $iso : V_C \cup B_C \to V_{C'} \cup B_{C'}$ s.t.: (1) for all $u, v \in V_C$, $v \in \delta_C(v)$ iff $iso(v) \in \delta_{C'}(iso(u))$ and (2) for $u \in V_C \cup B_C$ and $u' \in V_{C'} \cup B_{C'}$, we get $u' = iso(u)$ iff u and u'

- *have the same labeling*, i.e. $\eta_C(u) = \eta_{C'}(u')$;
- *are assigned to the same player*, i.e., $u \in P_C^j$ iff $u' \in P_{C'}^j$ for $j \in [0, 1]$;
- *are of the same kind*, i.e.:
  - u is an entry/box of C iff u' is an entry/box of C';
  - for $i \in [k]$, u is the i^{th} exit of C iff u' is the i^{th} exit of C';
  - $u = (1, b)$ iff $u' = (1, iso(b))$ and for $i \in [k]$, $u = (b, i)$ iff $u' = (iso(b), i)$ (*calls and i^{th}-returns of isomorphic boxes must be isomorphic*).

For $k > 0$, a *k-library* is a tuple $\mathcal{L}ib = \langle \{C_i\}_{i \in [0,n]}, Y_{\mathcal{L}ib} \rangle$ where:

- $\{C_i\}_{i \in [0,n]}$ is a finite set where each C_i is a k-component;
- C_0 is the *main component*;
- let $B_{\mathcal{L}ib} = \bigcup_{i \in [0,n]} B_{C_i}$ be the set of all boxes of the library components, the *box-to-component* map $Y_{\mathcal{L}ib} : B_{\mathcal{L}ib} \to \{C_i\}_{i \in [n]}$ is a partial function.

Running Example. We illustrate the definitions with an example. In Fig.1(a), we give a library of components. Each component has two exits, and $\mathcal{L}ib$ is composed of four components C_0, C_1, C_2 and C_3. 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. Entries are denoted by nodes intersecting the frame of the component on the left, and exits on the right. For example, C_0 has entry e_0 and two exits x_1 and x_2, one internal node u_1 and two boxes b_1 and b_2. With "$b_1 : C_1$" we denote that box b_1 is mapped to component C_1. The only unmapped box is b_3. To keep the figure simple, we only show the labeling of vertices with

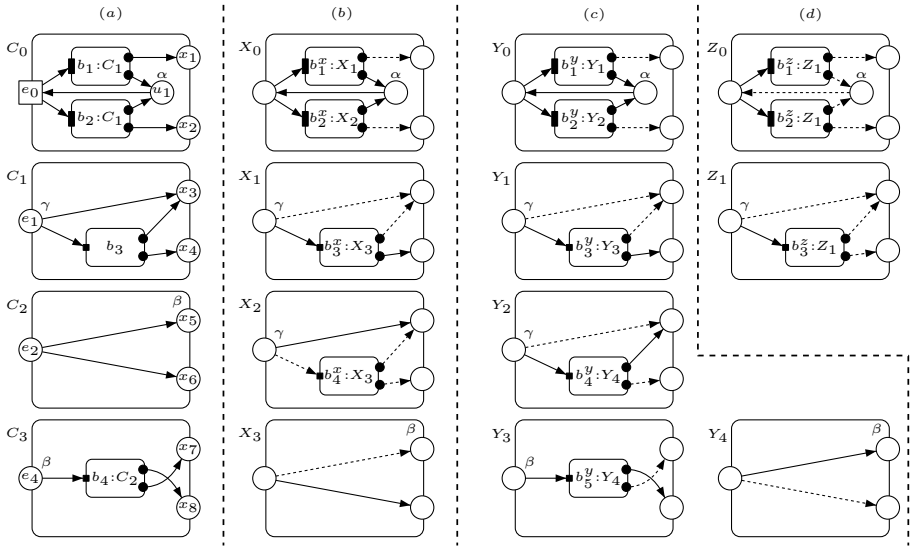

Fig. 1. A library (a) and RSMs from it: unrestricted (b), same local strategy for instances of the same component (c), and at most one instance for each component (d).

labels α, β and γ, and hide the labeling for all the remaining vertices (meaning that they are labeled with any other symbol).

Notes. For the ease of presentation, we have imposed a few restrictions. First, in the definition of library, $Y_{\mathcal{L}ib}$ can map a box to each component but the main component C_0. We observe that this is in accordance with the choice of many programming languages where the main function cannot be called by other functions and is without loss of generality of our results. Second, multiple entries can be handled by making for each component as many copies as the number of its entries, and accommodating calls and returns accordingly. Third, all the components of a library have the same number of exits that also matches the number of returns for each box. This can be relaxed at the cost of introducing a notion of *compatibility* between a box and a component, and map boxes to components only when they are compatible. We make a further assumption that is standard: in the components there are no transitions leaving from exits (assigning them to pl_0 or pl_1 is thus irrelevant).

Instances and recursive state machines. We are interested in synthesizing a *recursive state machine* (RSM) from a library of components. Such a machine is formed by a finite number of *instances* of library components, where each instance is isomorphic to a library component and resolves the nondeterminism of pl_0 by a finite-state local strategy. The boxes of each instance are mapped to instances in the machine with the meaning that when a call of a box b is reached

then the execution continues on the entry of the mapped instance and when the i^{th} exit of such instance is reached then it continues at the i^{th} return of b (as in the recursive call-return paradigm). The box-to-instance map of an RSM must agree with the box-to-component map of the library when this is defined.

We observe that our definition of RSM differs from the standard one (given in [1]) in that (i) each finite-state machine is implicitly given by a component and a finite-state local strategy, and (ii) the nodes are split between pl_0 and pl_1. (However the last is immaterial since the nondeterminism of pl_0 is completely resolved by the local strategies.)

For a component C, a *local strategy* is $f : V_C^*.P_C^0 \to Calls_C \cup N_C$ such that $f(w.u) \in \delta_C(u)$. The strategy is *finite-state* if it is computable by a finite automaton (we omit a formal definition here, see [15]).

An *instance of* C is $I = (G, f)$ where G is s.t. $G \overset{\text{iso}}{\equiv} C$ holds and f is a finite-state local strategy of G. For example, in Fig.1, X_1 and X_2 are two instances of C_1 that differ on the local strategy (we have denoted with dashed edges the transitions that cannot be taken because of the local strategies). Also, Y_1 is an instance of C_1 and has the same local strategy as X_1. Note that, though the local strategies used in this example are memoryless, this is not mandatory and thus the number of instances of each component with different local strategies is in general unbounded.

Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$. A *recursive state machine (RSM) from* $\mathcal{L}ib$ is $S = \langle \{I_i\}_{i\in[0,m]}, Y_S\rangle$ where:

- for $i \in [0, m]$, $I_i = (G_i, f_i)$ is an instance of a component C_{j_i} from $\mathcal{L}ib$;
- I_0 is an instance of the main component C_0;
- the *box-to-instance* map $Y_S : \bigcup_{i\in[0,m]} B_{G_i} \to \{I_i\}_{i\in[m]}$ is a total function that is consistent with $Y_{\mathcal{L}ib}$, i.e., for each $i \in [0, m]$ and $b \in B_{G_i}$, denoting with b' the box of C_{j_i} that is isomorphic to b, it holds that if $Y_{\mathcal{L}ib}(b') = C_{j_h}$ then $Y_S(b) = G_h$.

Examples of RSM for the library from Fig.1(a) are given in Fig.1(b)–(d).

We assume the following notation: $V_S = \bigcup_{i\in[0,m]} V_{G_i}$ (set of all vertices); $B_S = \bigcup_{i\in[0,m]} B_{G_i}$ (set of all boxes); $En_S = \bigcup_{i\in[0,m]} \{e_{G_i}\}$ (set of all entries); $Ex_S = \bigcup_{i\in[0,m]} Ex_{G_i}$ (set of all exits); $Calls_S = \bigcup_{i\in[0,m]} Calls_{G_i}$ (set of all calls); $Retns_S = \bigcup_{i\in[0,m]} Retns_{G_i}$ (set of all returns); and $P_S^j = \bigcup_{i\in[0,m]} P_{G_i}^j$ for $j = 0, 1$ (set of all positions of pl_j).

A *state* of S is (γ, u) where $u \in V_{Y_S(b_h)}$ is a vertex and $\gamma = \gamma_1 \dots \gamma_h$ is a finite sequence of pairs $\gamma_i = (b_i, \mu_i)$ with $b_i \in B_S$ and $\mu_i \in V^*_{Y_S(b_i)}$ for $i \in [h]$ (respectively, *calling box* and *local memory* of the called instance).

In the following, for a state $s = (\gamma, u)$, we denote with $V(s)$ its vertex u. Moreover, we define the *labeling map of* S, denoted η_S, from the labeling η_{G_i} of each instance I_i in the obvious way, i.e., $\eta_S(s) = \eta_{G_i}(V(s))$ for each $V(s) \in V_{G_i}$ and $i \in [0, m]$. η_S naturally extends to sequences.

A *run* of S is an infinite sequence of states $\sigma = s_0 s_1 s_2 \dots$ such that $s_0 = ((\epsilon, e_{G_0}), e_{G_0})$ and for $i \in \mathbb{N}$, denoting $s_i = (\gamma_i, u_i)$ and $\gamma_i = (b_1, \mu_1) \dots (b_h, \mu_h)$, one of the following holds:

– **Internal pl_1 move:** $u_i \in (N_S \cup Retns_S) \setminus Ex_S$, and $u_i \in P_S^1$, then $u_{i+1} \in \delta_S(u_i)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_h, \mu_h.u_{i+1})$;
– **Internal pl_0 move:** $u_i \in (N_S \cup Retns_S) \setminus Ex_S$, $u_i \in P_S^0$ and $u_i \in V_{G_j}$ with $j \in [0, m]$, then $u_{i+1} = f_j(\mu_h)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_h, \mu_h.u_{i+1})$.
– **Call to an instance:** $u_i = (1, b) \in Calls_S$, $u_{i+1} = e_{Y_S(b)}$ and $\gamma_{i+1} = \gamma_i.(b, e_{Y_S(b)})$;
– **Return from a call:** $u_i \in Ex_S$ and u_i corresponds to the j^{th} exit of an instance I_h, then $u_{i+1} = (b, j)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_{h-1}, \mu_{h-1}.u_{i+1})$.

An *infinite* RSM is defined as an RSM where we just relax the request that the set of instances is finite. We omit a formal definition and retain the notation. Note that the definitions of state and run given above still hold in this case.

Synthesis problem. Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i \in [0,n]}, Y_{\mathcal{L}ib} \rangle$ with alphabet Σ.

A *library game* is $(\mathcal{L}ib, \mathcal{W})$ where $\mathcal{L}ib$ is a library of components and $\mathcal{W}$ is a *a winning set*, i.e., a language $\mathcal{W} \subseteq \Sigma^\omega$.

The *modular synthesis from libraries* (Lms, for short) is the problem of determining if for a given library game $(\mathcal{L}ib, \mathcal{W})$ there is an RSM $S = \langle \{I_i\}_{i \in [0,m]}, Y_S \rangle$ from $\mathcal{L}ib$ that *satisfies* $\mathcal{W}$, i.e., $\eta_S(\sigma) \in \mathcal{W}$ for each run σ of S.

As an example, consider the Lms queries $\mathcal{Q}_i = (\mathcal{L}ib, \mathcal{W}_i)$, $i \in [3]$, where $\mathcal{L}ib$ is from Figure 1(a) and denoting $\Sigma = \{\alpha, \beta, \gamma\}$: $\mathcal{W}_1$ is the set of all ω-words whose projection into Σ gives the word $(\gamma\alpha)^\omega$, $\mathcal{W}_2$ is the set of all words whose projection into Σ gives a word in $(\gamma\beta\alpha + \gamma\beta^2\alpha)^\omega$, and $\mathcal{W}_3$ is the set of all ω-words with no occurrences of β. The RMSs from Fig.1(b)–(d) are solutions of the Lms queries $\mathcal{Q}_1, \mathcal{Q}_2$ and $\mathcal{Q}_3$ respectively. In the figure, we use circles to denote all the nodes, this is to stress that the splitting between the two players is not meaningful any more.

3 Safety Lms

A *safety automaton* A is a deterministic finite automaton with no final states, and the language accepted by A, denoted $\mathcal{W}_A$, is the set of all ω-words on which A has a run (see [15] for the automata on ω-words). We denote a safety automaton by $(\Sigma, Q, q_0, \delta_A)$ where Σ is a finite set of input symbols, Q is a finite set of states,

$q_0 \in Q$ is the initial state, and $\delta_A : Q \times \Sigma \to Q$ is a partial function (the transition function).

In the *safety* Lms the winning set is given by the set of words accepted by a safety automaton. In this section we show that deciding this problem is EXPTIME-complete. Our decision procedure consists of reducing the problem to checking the emptiness of tree automata. We assume familiarity with tree automata and refer the interested reader to [14] (see also Appendix A for the main definitions).

Overview of the construction. Fix a safety Lms query $(\mathcal{L}ib, \mathcal{W}_A)$ where $\mathcal{L}ib = \langle \{C_i\}_{i \in [0,n]}, Y_{\mathcal{L}ib} \rangle$ is a library and $A = (\Sigma, Q, q_o, \delta_A)$ is a safety automaton.

We aim to construct an automaton $\mathcal{A}$ that accepts a tree that *encodes* an RSM S synthesized from $\mathcal{L}ib$ iff S satisfies $\mathcal{W}_A$.

For the RSM encoding we introduce the notions of component tree and library tree. For a component $C \in \mathcal{L}ib$, the *component tree* of C is a tree where the subtree rooted at the first child of the root is essentially the unrolling of C from its entry node and the other children of the root are leaves s.t. each box of C is mapped to exactly one of them. A *library tree* is a tree obtained by starting with the component tree of the main component and then iteratively gluing at each leaf corresponding to a box b: any component tree, if $Y_{\mathcal{L}ib}(b)$ is not defined, and the component tree of $Y_{\mathcal{L}ib}(b)$, otherwise.

For a library tree t, denote with $\mathit{Roots}(t)$ the set of all nodes of t that correspond to a root of a component tree. A set $\mathcal{I} = \{I_x\}_{x \in \mathit{Roots}(t)}$ is *compatible* with t if I_x is an instance of the component corresponding to the component tree rooted at x. Such a set $\mathcal{I}$ and the total box-to-component mapping defined by the concatenation of component trees in t define a possibly infinite RSM (it is infinite iff $\mathit{Roots}(t)$ is infinite). Denote $S_{\mathcal{I},t}$ such RSM.

Intuitively, the automaton $\mathcal{A}$ checks that the input tree t is a library tree of $\mathcal{L}ib$ and that there is a set of instances $\mathcal{I}$ that is compatible with t s.t. $S_{\mathcal{I},t}$ satisfies $\mathcal{W}_A$. For this, $\mathcal{A}$ simulates the safety automaton A on the unrolling of each component and on pl_0 nodes also guesses a move of the local strategy (this way we also guess an instance of the component). To move across the boxes, $\mathcal{A}$ uses a *box summary* that is guessed at the root of the component tree. For $x \in \mathit{Roots}(t)$, denoting with C_x the corresponding component and with x_b the child of x corresponding to a box b of C_x, the box summary guessed at x essentially tells for each such b: (1) the associated component C_{x_b} in t, and (2) a non empty set $Q' \subseteq Q$ and sets $Q^b_{q,i} \subseteq Q$ for $i \in [k]$ and $q \in Q'$ s.t. for any run π of $S_{\mathcal{I},t}$ that starts at the entry of the instance I_{x_b} and ends at its i^{th} exit, if the safety automaton A starts from q and reads the sequence of input symbols along π then it reaches a state of $Q^b_{q,i}$. The above assumption (2) is called a *pre-post condition* for C_b. The correctness of the pre-post condition for each such C_b is checked in the simulation of $\mathcal{A}$ on the unrolling of C_{x_b} (at the first child of x_b).

We give $\mathcal{A}$ as the composition of several tree automata: $\mathcal{A}_{\mathcal{L}ib}$ checks that the input tree is a library tree, and each $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ checks on the unrolling of C that the pre-post condition $\mathcal{P}$ holds by assuming that the box-summary $\mathcal{B}$ holds.

Component and library trees. Library trees can be formally defined as the ω-concatenation over component trees. The construction of $\mathcal{A}_{\mathcal{L}ib}$ can be obtained from the automata accepting the component trees for $\mathcal{L}ib$ accordingly. The construction of these automata is fairly standard. The formal definitions, more details and examples can be found in Appendix A. Here we just give:

Proposition 1. *There exists an effectively constructible Büchi tree automaton $\mathcal{A}_{\mathcal{L}ib}$ of size linear in the size of $\mathcal{L}ib$, that accepts a tree if and only if it is a library tree of $\mathcal{L}ib$.*

Box summary. A *pre-post condition* $\mathcal{P}$ is a set of tuples $(q, [Q_1, \ldots, Q_k])$ where $q \in Q$ and $Q_i \subseteq Q$ for each $i \in [k]$, and s.t. for any pair of tuples $(q, [Q_1, \ldots, Q_k])$,

$(q', [Q'_1, \ldots, Q'_k]) \in \mathcal{P}$: $q \neq q'$, and $Q_i = \emptyset$ implies $Q'_i = \emptyset$ for each $i \in [k]$. For such a pre-post condition $\mathcal{P}$, each q is a *precondition* and each tuple $[Q_1, \ldots, Q_k]$ is a *postcondition*.

A *box summary* of an instance of C is a tuple $\mathcal{B}_C = \langle \hat{Y}_C, \{\mathcal{P}_b\}_{b \in B_C} \rangle$, where $\hat{Y}_C : B_C \to \{C_i\}_{i \in [n]}$ is a total map that is consistent with the library box-to-component map $Y_{\mathcal{L}ib}$ and for each box $b \in B_C$, $\mathcal{P}_b$ is a pre-post condition.

Sketch of the $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ construction. Fix a component C, a pre-post condition $\mathcal{P} = \{(q_i, [Q_{i_1}, ..., Q_{i_k}])\}_{i \in [h]}$ and a box summary $\mathcal{B} = \langle \hat{Y}_C, \{\mathcal{P}_b\}_{b \in B_C} \rangle$.

Denote T_C the component tree of C and T^1_C the subtree rooted at the first child of T_C. Recall that T^1_C corresponds to the unrolling of C from the entry node. For a local strategy f for C, a path $x_1 \ldots x_j$ of T^1_C *conforms to* f if the corresponding sequence of C vertices $v_1 \ldots v_j$ is s.t. for $i \in [j-1]$ if v_i is a node of pl_0 then $v_{i+1} = f(v_1 \ldots v_i)$.

For each path π of T^1_C, a run of the safety automaton A on π is a run where a state q is updated (1) according to a transition of A, from a tree-node corresponding to a node or a return of C, and (2) by nondeterministically selecting a state from Q_i with $(q, [Q_1, \ldots, Q_k]) \in \mathcal{P}_b$ (i.e., a state from the postcondition for box b in $\mathcal{B}$), from a tree-node corresponding to a call $(1, b)$ to one corresponding to a return (b, i). Note that, in the above definition we do not consider the case of an empty postcondition for a return. This is fine for our purposes since we will use it only for paths where this case will not happen.

We construct $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ s.t. it rejects any tree other than T_C and accepts T_C iff:

(P1) There is a local strategy f for C s.t. for $i \in [h], j \in [k]$, and for each path π of T^1_C from the root to the j^{th} exit that conforms to f, each run of A on π starting from q_i ends at a state in Q_{i_j}.

$\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ is a tree automaton that accepts a tree if it halts on an accepting state on the finite paths. (No acceptance condition is required on infinite paths.)

The states of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ are: an initial state q_s, an accepting sink state q_a, a rejecting sink state q_r, a state q_e, a state q_b for each box b of C, and states of the form $(R_1, \ldots, R_h)$ where $R_i \subseteq Q$ for $i \in [h]$.

At the root of T_C, from q_s the automaton enters q_e on the first child and q_b on the child corresponding to b, for each box b of C. From q_b, it accepts if the current node corresponds to b. From q_e, it behaves as from $(\{q_1\}, \ldots, \{q_h\})$ if the current node corresponds to the entry of C (recall that $q_1, \ldots, q_h$ are the preconditions of $\mathcal{P}$).

In each run of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$, for a state of the form $(R_1, \ldots, R_h)$ at a tree-node x, we keep the following invariant: for $i \in [h]$, R_i is the minimal set of states ending a run of A starting from q_i on the path from the root of T^1_C up to x.

From a tree-node corresponding to a node or a return of C, the transitions of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ update each R_i as in a standard subset construction provided that there is a transition of A from all the states in $\bigcup_{j \in [h]} R_j$ (a run is unsafe if A halts). The updated state is entered on all the children from pl_1 vertices, and on only one nondeterministically selected child from pl_0 vertices (guess of the local strategy).

The update on tree-nodes corresponding to a call $(1, b)$ of C is done according to the pre-post condition $\mathcal{P}_b$. In particular, denoting $\mathcal{P}_b = \{(q'_i, [Q'_{i_1}, \ldots, Q'_{i_k}])\}_{i \in [h']}$, from $(R_1, \ldots, R_h)$ we enter q_a on the tree-node corresponding to any return (b, j) that is not reachable according to $\mathcal{P}_b$, i.e., each $Q'_{i_j} \neq \emptyset$ (we accept since the guessed local strategy excludes such paths and thus the condition $\mathcal{P}$ does not need to be checked). On the reachable returns (b, j), we enter the state $(R'_1, \ldots, R'_h)$ where $R'_i = \bigcup_{q'_d \in R_i} Q'_{d_j}$ for $i \in [h]$, i.e., we collect the postconditions of the j^{th} exit for each precondition of $\mathcal{P}_b$ that applies.

At a tree-node corresponding to the i^{th} exit of C, $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts iff $\mathcal{P}$ is fulfilled, i.e., A enters a state $(R_1, \ldots, R_h)$ s.t. $R_i \subseteq Q_i$.

Lemma 1. *$\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts T_C iff property P1 holds. Moreover, the size of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ is linear in the number of C boxes and exponential in the number of A states.*

The construction of $\mathcal{A}$. We first construct an automaton $\mathcal{A}'$. For this, we extend the alphabets such that $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts the trees that are obtained from the component tree T_C of C by labeling the leaf corresponding to b, for each box b of C, with any tuple of the form $(\hat{Y}(b), \mathcal{P}_b, \mathcal{B}_b)$ where $\mathcal{B} = \langle \hat{Y}, \{\mathcal{P}_b\}_{b \in B_C} \rangle$ and $\mathcal{B}_b$ is any box summary for $\hat{Y}(b)$. Let $\mathcal{L}^C_{\mathcal{P},\mathcal{B}}$ be the set of all trees accepted by any such automaton.

Let $\mathcal{P}_0 = \{(q_0, [\emptyset, \ldots, \emptyset])\}$ where q_0 is the initial state of A and Lab be the set of all labels $(C, \mathcal{P}, \mathcal{B})$ s.t. C is a component, $\mathcal{P}$ is a pre-post condition of C, and $\mathcal{B}$ is a box summary of C. For each box summary $\mathcal{B}_0$ for C_0 denote $\mathcal{T}_{\mathcal{B}_0}$ the language $\mathcal{L}^{C_0}_{\mathcal{P}_0,\mathcal{B}_0} \cdot_{\bar{c}} (\langle \mathcal{L}^C_{\mathcal{P},\mathcal{B}} \rangle_{(C,\mathcal{P},\mathcal{B}) \in Lab})^{\omega_{\bar{c}}}$ where $\bar{c} = \langle (C, \mathcal{P}, \mathcal{B}) \rangle_{(C,\mathcal{P},\mathcal{B}) \in Lab}$, i.e., the infinite trees obtained starting from a tree in $\mathcal{L}^{C_0}_{\mathcal{P}_0,\mathcal{B}_0}$ and then for all $(C, \mathcal{P}, \mathcal{B}) \in Lab$ iteratively concatenating at each leaf labeled with $(C, \mathcal{P}, \mathcal{B})$ a tree from $\mathcal{L}^C_{\mathcal{P},\mathcal{B}}$ until all such leaves are replaced.

By standard constructions (see [14]), we construct the automaton $\mathcal{A}'$ that accepts the union of the languages $\mathcal{T}_{\mathcal{B}}$ for each box summary $\mathcal{B}$ of the main component.

The automaton $\mathcal{A}$ is then taken as the intersection of $\mathcal{A}_{\mathcal{L}ib}$ and $\mathcal{A}'$. Thus, from Proposition 1, Lemma 1 and [14], we get that the size of $\mathcal{A}$ is exponential in the size of $\mathcal{L}ib$ and A. Since the emptiness of (Büchi) nondeterministic tree automata can be checked in linear time and if the language is not empty then it is possible to determine a finite witness of it (*regular tree*) [14], and since such witness ensures both the finiteness of the local strategies and of the number of instances of the corresponding RSM, we get (the EXPTIME lower bound can be shown with a direct reduction from alternating linear-space Turing machines, see Appendix B):

Theorem 1. *The safety* Lms *problem is* EXPTIME*-complete.*

4 On modular synthesis problems

Other formulations of the modular synthesis. We start introducing two variations of the Lms problem based on two natural restrictions on the admissible RSMs.

Fix a library $\mathcal{L}ib$. An RSM S from $\mathcal{L}ib$ is *component-based* if for any two S instances $I = (G, f)$ and $I' = (G', f')$ of a component C from $\mathcal{L}ib$, the local strategies f and f' *must coincide* (up to a renaming). Moreover, S is *single-instance* if it has *at most* one instance of each library component.

The *component-based* (resp. single-instance) Lms problem is the Lms problem restricted to component-based (resp. single-instance) RSMs.

Let $\mathcal{L}ib$ be the library from Fig.1(a). The RSM in Fig.1(b) is not component-based (and thus single-instance). In fact, X_1 and X_2 are instances of C_1 and use two different local strategies. The RSM in Fig.1(c) instead is component-based but not single-instance since Y_1 and Y_2 are two instances of C_1 (note that even if they have the same local strategy, they differ on the reachable vertices because the box is mapped differently). The RSM from Fig.1(d) is clearly single-instance.

Let $\mathcal{W}_1, \mathcal{W}_2$ and $\mathcal{W}_3$ be the winning conditions given at the end of Section 2. Observe that they are all expressible by safety automata. Moreover, there is no component-based RSM from $\mathcal{L}ib$ that satisfies $\mathcal{W}_1$ and no single-instance RSM from $\mathcal{L}ib$ that satisfies $\mathcal{W}_2$.

Solving the safety single-instance and component-based Lms *problems.* The construction given in Section 3 is based on the notion of library tree that essentially encodes the components and the box-to-instance map of an RSM. The local strategies are guessed on-the-fly by the tree automaton. To constrain the RSM to be component-based we should guess a strategy for each component C and then use it when visiting each component tree of C in the input library tree. This requires to prove first boundedness of the local strategies if there is a component-based RSM that satisfies the winning condition.

A simpler solution can be obtained by adapting the solution given in [2] for the synthesis of *modular strategies*. This problem is a particular case of the single-instance Lms problem where the box-to-component map is total, i.e., each box is pre-assigned. The solution given in [2] is based on the notion of *strategy tree* that unrolls each component as a subtree of the root and encodes in the labels of this encoding a local strategy. To adapt their automaton construction for the component-based Lms we just need to guess the mapping for the boxes that are not mapped by the box-to-component map of the library, every time a component subtree is visited.

Also, we can solve the single-instance Lms problem, by guessing a total assignment of the boxes in the library and then solving the modular synthesis problem on the resulting RGG. Thus, we get:

Theorem 2. *The safety component-based (resp. single-instance)* Lms *problem is* EXPTIME*-complete.*

Other winning conditions. A standard winning condition for synthesis problems is *reachability*, i.e., each run of the solution should reach a vertex of a target set.

The Lms problem with reachability winning condition is solved in [7].

For the single-instance Lms problem we reason as for the safety single-instance Lms and since the synthesis of modular strategies with reachability

winning conditions is NP-complete [3], we also get that the reachability single-instance LMS problem is NP-complete.

For the component-based LMS problem we can modify the algorithm proposed in [7] to fix the local strategy for each component. It is simple to verify that, if there is a component-based RSM that satisfies the reachability condition then there exists a component-based RSM S that satisfies it s.t. all the local strategies are memoryless (going through cycles does not help with reachability winning conditions). Therefore, we can solve the component-based LMS problem by guessing a local strategy for each component and then applying the algorithm as in [7]. We refer the reader to Appendix C for the details.

Theorem 3. *The reachability component-based (resp. single-instance)* LMS *problem is* EXPTIME*-complete (resp.* NP*-complete).*

A *visibly pushdown automaton* (VPA) is a pushdown automaton where the stack operations are determined by the input symbols: a call symbol causes a push, a return causes a pop and an internal causes just a change of the finite control [4]. We give a reduction from the LMS problem with deterministic VPA specifications (VPA LMS) to the safety LMS problem. The labeling of the library components is synchronized with the VPA, in the sense that calls are labeled with call symbols, returns with return symbols and all the nodes with internals. This synchronization is exploited in our construction. We add a new component C_{stack_i} for each library component C_i and construct a safety automaton that preserves the finite control of the VPA and ensures that each call to a C_i goes through a call to C_{stack_i} (with $i \in [n]$). The new components guess the stack symbols of the VPA via a choice of pl_1 and the alternation between original components and new components ensure the synchronization of the VPA stack and the call stack of the synthesized RSM. This reduction clearly applies also to component-based and single-instance VPA LMS. (See Appendix D.)

Theorem 4. *The considered VPA* LMS *problems are* EXPTIME*-complete.*

5 Discussion and conclusion

In this paper, we have introduced a modular synthesis problem that generalizes the synthesis of modular strategies studied in [2, 3, 6] as also observed in Section 4. In the synthesis from recursive-component libraries [11], the library of reusable components is modeled using a set of transducers with call-return structures. The related synthesis problem asks to find a composition such that the synthesized system fulfills a given LTL specification. This problem can be modeled as a component-based LMS where the library only has components with only vertices of pl_1, i.e., no game at the component level (see Appendix E). By rephrasing the LMS problem in terms of the problem from [11] we would have a library of infinitely many recursive-components formed of all the different instances of the components of the library in the LMS.

Looking at the LMS as a game-graph problem the winning strategies we compute are *modular* and thus the local strategy of an instance that is called several times is oblivious of previous calls, i.e., it does not keep the memory of previous invocations. It is known that non-oblivious modular games are undecidable (see [3]). If in the studied LMS problems we extend the class of solutions by allowing to resolve the internal nondeterminism of pl_0 by a global strategy, we can show that the resulting problem is still decidable and admits solutions also if the modular LMS does not (see Appendix E). Note that this does not contradict the previous undecidability result since the moves taken by pl_1 in an instance I are now observable by pl_0 in any other instance. Further, the global LMS problem can be reduced to a standard pushdown game (PDG) with an exponential blow-up and vice-versa a PDG can be polynomially translated to a global LMS with a total box-to-component map (see also [1, 3]).

Finally, we observe that the nondeterministic finite automata and nondeterministic VPA specifications can be handled via determinization (since these classes are determinizable). Other common classes of specifications such as deterministic/universal Büchi/coBüchi automata and temporal logic formulas can also be allowed in our settings by retaining decidability. Details on these will be given in the extended version of this paper.

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: CAV. LNCS, vol. 2725, pp. 67–79. Springer (2003)
3. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for recursive game graphs. Theor. Comput. Sci. 354(2), 230–249 (2006)
4. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56(3) (2009)
5. Aminof, B., Mogavero, F., Murano, A.: Synthesis of hierarchical systems. Sci. Comput. Program. 83, pp. 56-79 (2014)
6. De Crescenzo, I., La Torre, S., Velner, Y.: Visibly pushdown modular games. In: GandALF. EPTCS, vol.161, pp. 260–274 (2014).
7. De Crescenzo, I., La Torre, S.: Modular Synthesis with Open Components. In: RP. LNCS, vol. 8169, pp. 96–108. Springer(2013)
8. Löding, C., Madhusudan, P., Serre, O.: Visibly pushdown games. In: FSTTCS. LNCS, vol. 3328, pp. 408–420. Springer (2004)
9. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ACM/IEEE ICSE, pp. 215–224 (May 2010)
10. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: FOSSACS. LNCS, vol. 5504, pp. 395–409. Springer (2009)
11. Lustig, Y., Vardi, M.Y.: Synthesis from recursive-components libraries. In: 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. Salvati, S., Walukiewicz, I.: Evaluation is MSOL-compatible. In: FSTTCS. LIPIcs, vol. 24, pp. 103-114 (2013)

14. Thomas, W.: Automata on Infinite Object. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–191. MIT press (1990)
15. Thomas, W.: Infinite games and verification (extended abstract of a tutorial). In: CAV. LNCS, vol. 2404, pp. 58–64. Springer (2002)
16. Walukiewicz, I.: Pushdown processes: Games and model-checking. Inf. Comput. 164(2), 234–263 (2001)

A Definitions

Tree automata. Let $d \in \mathbb{N}$ and Ω be a finite alphabet. A *Ω-labeled d-tree* is a pair $([d]^*, \nu)$ where the set $[d]^*$ denotes the tree-nodes and $\nu : [d]^* \to \Omega$ is a labeling function. The symbol ϵ (denoting as usual the empty word) is the root and for every tree-node $x \in [d]^*$, the tree-node $x.i$ is the i^{th} child of x.

A *nondeterministic Büchi* (resp. *co-Büchi*) *tree automaton* over Ω-labeled d-trees is $\mathcal{A} = (Q, Q_0, \delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta \subseteq Q \times \Omega \times Q^d$ is the transition relation and $F \subseteq Q$ is a Büchi (resp. co-Büchi) acceptance condition.

A *run* R of $\mathcal{A}$ on a Ω-labeled d-tree $T = ([d]^*, \nu)$ is a Q-labeled d-tree $([d]^*, \tau)$ such that $\tau(\varepsilon) \in Q_0$ and for each $x \in [k]^*$, $(\tau(x), \nu(x), \tau(x.1), \ldots, \tau(x.k)) \in \delta$.

A *path* in a d-tree is $x_1 x_2 \ldots$ where for all $i \in \mathbb{N}$, $x_i = x_{i-1}.j_i$ for $j_i \in [d]$. In a Büchi (resp. co-Büchi) tree automaton a run R is *accepting* if each path is accepting, i.e., for every *infinite* path $x_1 x_2 \ldots, \tau(x_i) \in F$ for infinitely many $i \in \mathbb{N}$ (resp., there is a $j \in \mathbb{N}$ s.t. $\tau(x_i) \notin F$ for every $i > j$).

Component and library trees. Consider a library $\mathcal{L}ib = \langle \{C_i\}_{i \in [0,n]}, Y_{\mathcal{L}ib} \rangle$ and be $B_{\mathcal{L}ib} = \bigcup_{i \in [0,n]} B_{C_i}$ the set of all boxes and $V_{\mathcal{L}ib} = \bigcup_{i \in [0,n]} V_{C_i}$ the set of all vertices (i.e. nodes,calls and returns) of the library components.

Let d the maximum over the number of exits, the number of boxes in each component and the out-degree of the vertices of the $\mathcal{L}ib$-components.

Denote with $\widehat{\Omega}$ the set $\{dummy, C_0, ..., C_n\} \cup B_{\mathcal{L}ib} \cup V_{\mathcal{L}ib}$, we say that a *component tree* is $\widehat{\Omega}$-labeled d-tree that encodes the unrolling of a component of $\mathcal{L}ib$. If a component tree encodes a component C_i, the label C_i is associated with the root of the tree. The first child of the root is labeled with the entry of the component C_i and the subtree rooted in such tree-node encodes its unrolling. If a tree-node is labeled with a call, its successors correspond to the returns of the corresponding box. If a tree-node is labeled with an exit, the tree-node has no children. Let h be the number of boxes in the component C_i. Each child of the root, from the second to the $(h+1)^{th}$, is labeled with the name of a box in C_i and has no children. We refer to $\overline{b} = \{b_1, b_2, ..., b_h\}$ with $b_i \in B_C$ as the *frontier* of a component tree T_C. The *dummy* nodes are used to complete the d-tree.

Formally, an $\widehat{\Omega}$-labeled d-tree T_{C_i} is a *component tree* of C_i in $\mathcal{L}ib$, if:

- the root of T_{C_i} is labeled with C_i;
- The subtree $T^1_{C_i}$, rooted at the first child of the root corresponds to the unrolling of the component C_i; the nodes of $T^1_{C_i}$ are labeled with the corresponding vertices of the component C_i; thus, in particular, the root of $T^1_{C_i}$ is labeled with e_{C_i} and the calls have as children the matching returns; a tree-node labeled with an exit has no children; in $T^1_{C_i}$ all the nodes that do not correspond to a vertex in the unrolling of C_i are labeled with *dummy*, meaning that they are not meaningful in the encoding;
- for $i \in [2, h+1]$, the j^{th} child of the root is labeled with b with $b \in B_{C_i}$ and for any $j, z \in [2, h+1]$ with $j \neq z$ the labels of the j^{th} child and the z^{th} child must be different;

– the tree-nodes labeled with $b \in B_{C_i}$ have no children;
– the remaining tree-nodes are labeled with *dummy*.

In Fig. 2 we depict a fragment of the component tree for the component C_1 from the library in Fig. 1 (a).

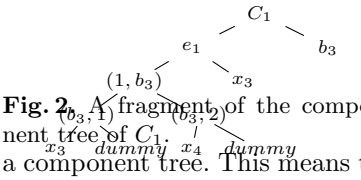

Fig. 2. A fragment of the component tree of C_1.

A *library tree* is an $\widehat{\Omega}$-labeled d-tree that encodes a system synthesized from $\mathcal{L}ib$. The root of the library tree is the root of the component tree of the main component. The leaves labeled with the names of the boxes must be unrolled and they are replaced with a component tree. This means that, when the a call is executed using a box, the component encoded using that component tree will be invoked.

Formally, a library tree t is a ω-fold concatenation of component trees, starting from the component tree of C_0 and replacing each tree-node in the frontier with a component tree, i.e. $t = T_{C_0} \cdot_{\overline{b}} (T_{C_1}, ...T_{C_n})^{\omega \overline{b}}$.

In Fig. 3 there is the initial fragment of the RSM depicted in Fig. 1 (b). Note that the second and the third child of the root, that were labeled with b_1 and b_2 respectively, are replaced by two component trees of C_1.

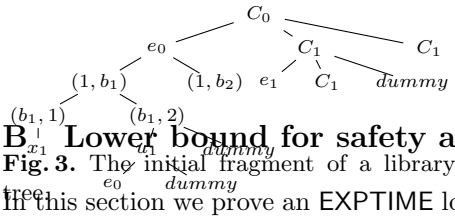

Fig. 3. The initial fragment of a library tree.

B Lower bound for safety automata specifications

In this section we prove an EXPTIME lower bound for solving the LMS problem. We adapt the standard reduction for [2] to our setting.

Theorem 5. *Deciding the* LMS *problem for safety automaton specifications is* EXPTIME*-complete.*

Proof. By Theorem 1, we only need to prove EXPTIME-hardness. The reduction is from the membership problem for alternating linear-space Turing machines.

An alternating Turing machine is $M = (\Sigma, Q, Q_\exists, Q_\forall, \delta, q_0, q_f)$, where Σ is the alphabet, Q is the set of states, $(Q_\exists, Q_\forall)$ is a partition of Q, $\delta : Q \times \Sigma \times \{D_1, D_2\} \longrightarrow Q \times \Sigma \times \{L, R\}$ is the transition function, and q_0 and q_f are respectively the initial and the final states. (We assume that for each pair $(q, \sigma) \in Q \times \Sigma$, there are exactly two transitions that we denote respectively as the D_1-transition and the D_2-transition.) A d-transition of M is $\delta(q, \sigma, d) = (q', \sigma', L/R)$ meaning that if q is the current state and the tape head is reading the symbol σ on cell i, M writes σ' on cell i, enters state q' and moves the read head to the left/right on cell $(i-1)/(i+1)$. Let n be the number of cells used by M on an input word w. A configuration of M is a word $\sigma_1 \dots \sigma_{i-1}(q, \sigma_i) \dots \sigma_n$

where $\sigma_1 \dots \sigma_n$ is the content of the tape cells and q is a state of M. The *initial configuration* contains the word w and the initial state. An *outcome* of M is a sequence of configurations, starting from the initial configuration, constructed as a play in the game where the $\exists$-player picks the next transition when the play is in a state of $Q_\exists$, and the $\forall$-player picks the next transition when the play is in a state of $Q_\forall$. A *computation* of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation that reaches a configuration with state q_f. A polynomial-space alternating Turing machine M is an alternating Turing machine that on an input word w uses a number of tape cells that is at most polynomial in $|w|$.

Let $M = (\Sigma, Q, Q_\exists, Q_\forall, \delta, q_0, q_f)$ be polynomial-space alternating Turing machine, and n be the number of cells used by M on an input word w. In the following, we define a library of recursive component $\mathcal{L}ib_{TM}$ and a safety automaton A_{TM} such that an RSM S from $\mathcal{L}ib_{TM}$ exists and each its possible run is winning according to A_{TM} if and only if M accepts w. The library $\mathcal{L}ib_{TM}$ has two components: C_0 and C_1. Let Σ' be $\Sigma \cup (Q \times \Sigma) \cup \{D_1, D_2\}$, C_0 generates sequences from $(\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \cdot \{D_1, D_2\})^* \cdot \Sigma'^\omega$ interspersed with a new symbol \$. [1] C_0 has a cyclic structure, that repeatedly executes a call using a box b. The box-to-component map of $\mathcal{L}ib_{TM}$ relates such box b to C_1. After the execution of each call, the component C_0 enters a node labeled with a symbol of $\Sigma \cup (Q \times \Sigma)$. Also, it ensures that symbols from $\{D_1, D_2\}$ are selected according to a move of pl_0 (resp. pl_1) if the last pair from $Q \times \Sigma$ which is generated on the current play has a state from $Q_\exists$ (resp. $Q_\forall$). In C_1, there is only one exit, each node is controlled by pl_1 and two new labels *ok* and *obj* are used. Intuitively pl_1 simply chooses between "let the play continue" (*ok* is generated) or "raise an objection" (*obj* is generated), then the control is returned to C_0 that generates the next symbol in the sequence. The objection *obj* along with the specification automaton A_{TM} is used to check that on each run according to the local strategies for pl_0, if we delete all the occurrences of *ok*, *obj* and \$ we obtain a sequence $w \cdot w'$ such that $w \in (\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \cdot \{D_1, D_2\})^*$, $w' \in \Sigma'^\omega$, and w encodes an outcome of a halting computation of M on w. In particular, A_{TM} is a safety automaton that checks the following:

1. the first n symbols of w encode the starting configuration;
2. each subsequence w'' of w such that $d' \cdot w'' \cdot d''$, for $d', d'' \in \{D_1, D_2\}$, and $w'' \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$, contains exactly n symbols;
3. while generating a configuration c, if the content of cell i is generated right after objection *obj* is raised, and on the current play this is the first time that *obj* is raised, then the $(i+1)^{th}$ cell of the next configuration is consistent with the transition selected after generating c and the contents of cells i, $(i+1)$ and $(i+2)$ of c. [2]

[1] This symbol is only needed for labeling entry and exit nodes, and has no particular meaning in this encoding.

[2] Here, we skip the description of the limit cases concerning the cases when there are only two or one cells left to the end of the current configuration.

Whenever A_{TM} finishes reading the configuration containing the final state q_f, if the above part 2 holds, it enters a state where it stays on each input (and thus accepts). Also when *obj* is raised, if the above part 3 holds then A_{TM} accepts. If one of the above checks fails A_{TM} halts, thus rejects all the the plays that are obtained as a continuation of the word it has read. The library tree automaton also checks whether the configuration sequence ends at a halting configuration.

Due to the box-to-component map and the lack of boxes in C_1, an RSM S from $\mathcal{L}ib_{TM}$ is forced to have only two instances, I_0 that is an instance of C_0, and I_1 that is an instance of C_1. The choices done by pl_1 can be done only in I_1 and, consequently, they are hidden to the other player. Since strategies of pl_0 are local and the strategy for I_0 is oblivious to the objection being raised in I_1, pl_0 needs to generate an accepting run of TM in order to win. Hence, each run of S is winning according to A_{TM} if and only if M accepts its input, and the theorem holds. □

C Solving LMS and component-based LMS problems with reachability winning conditions

Reachability LMS *problem:* In [7] an exponential-time fixed-point algorithm is proposed that solves the LMS problem with reachability winning conditions. However, the model considered in [7] does not provide the box-to-component map. In this section, we present the algorithm in [7], slightly modified to handle the partial mapping function of our model.

Fix a library of components $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$ and a target set $\mathcal{T}$ of C_0 exits. The algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b\in B_C})$ where u is a vertex of a component C, E is a set of C exits, B_C is the set of C boxes and for each box $b \in B_C$, μ_b is either a set of exits of a component or undefined (we use $\perp$ 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 $b \in B_C$, 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 $(u, E, \{\mu_b\}_{b\in B_C})$ summarizes for vertex u a reachable *local target* E and a set of *assumptions* $\{\mu_b\}_{b\in B_C}$ 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 $\mu' = \perp$, or $\mu = \perp$ (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 $\perp \cup X = X \cup \perp = X$ holds for each set X).

The set Φ is initialized with all the tuples of the form $(u, \mathcal{T}, \{\perp\}_{b\in B_{C_0}})$ where $u \in \mathcal{T}$ and B_{C_0} is the set of boxes of C_0. 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. Moreover the algorithm must verify that the association between a box and an instance is done according to the partial local function of the library $Y_{\mathcal{L}ib}$.

Below, we denote with b_x the return of a box b corresponding to an exit x (recall that all 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 $(v, E, \{\mu_b\}_{b\in B_C})$ provided that there is a transition from v to u and $(u, E, \{\mu_b\}_{b\in B_C}) \in \Phi$ (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 $b \in B_C$: (1) there is a $(u_i, E_i, \{\mu_b^i\}_{b\in B_C}) \in \Phi$, (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 $(u, E, \{\bot\}_{b\in B_{C'}})$ to Φ provided that $u \in E$ and for a box b' it holds that there are tuples $(b'_x, E_x, \{\mu_b^x\}_{b\in B_C}) \in \Phi$, one for each $x \in E$, such that for all $x, y \in E$ and $b \in B_C$, (1) $\mu_{b'}^x = \bot$, (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 $(u, E_u, \{\mu_b^u\}_{b\in B_C})$ to Φ provided that (i) there is an entry e s.t. $(e, E_e, \{\mu_b^e\}_{b\in B_{C'}}) \in \Phi$, (ii) for each return b'_x, $x \in E_e$, there is a tuple $(b_x, E, \{\mu_b^x\}_{b\in B_C}) \in \Phi$ 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 to the boxes $b \neq b'$ of the returns b'_x and make an assumption E_e on box b'), (iiii) if $Y_{\mathcal{L}ib}(b') = C''$, then $e = e_{C''}$.

We compute Φ as the fixed-point of the recursive definition given by the above rules and outputs "YES" iff $(e, \mathcal{T}, \{\mu_b\}_{b\in B_{C_0}}) \in \Phi$ for the entry e of C_0.

Observe that, the total number of tuples of the form $(u, E, \{\mu_b\}_{b\in B_C})$ is bounded by $|\mathcal{L}ib|\, 2^{O(k\beta)}$ where k is the number of exits of each component in $\mathcal{L}ib$

and β is the maximum over the number of boxes of each component. Therefore, the algorithm always terminates and takes at most time exponential in k and β, and linear in the size of $\mathcal{L}ib$.

Reachability component-based Lms *problem:* For an instance of Lms problem, if there is a set of winning local strategies for a reachability condition, then a modular memoryless strategy exists such that it is winning according to the same reachability condition. Intuitively, if two moves of a winning local strategy are executed in a same vertex, this means that such vertex is reachable with two different local histories. We have two cases. One local history is a prefix of the other. This means that, after visiting the vertex the first time, the run does a local loop and reaches again the vertex. The modular memoryless strategy chooses always this second move, avoiding to execute the cycle, and all the resulting runs are still winning (acceptance does not depend on the specific sequence of vertices to the target). The second case is that the different moves are done according to two different and incomparable local histories. In such case, the modular memoryless strategy chooses one of these moves and such strategy is still winning, because, in both cases, all the resulting runs will reach the target (acceptance does not depend on the context where the instance was invoked). Therefore, we guess a modular memoryless strategy $\bar{f}_C$ for each component and, then, we change the Update 1 rule to:
Update 1: For a pl_0-vertex $v \in V_C$, we add $(v, E, \{\mu_b\}_{b\in B})$ provided that $\bar{f}_C(v) = u$ and $(u, E, \{\mu_b\}_{b\in B}) \in \Phi$.

This new rule guarantees that, even if we have instances with a different mapping on boxes, they are still controlled in the same way.

D Deterministic VPA specification

In this paragraph, we consider library games where the winning conditions are given as a deterministic VPA automaton. First, we prove that:

Theorem 6. *Solving the* Lms *problem for deterministic VPA winning conditions is* EXPTIME*-complete*

We propose a reduction from library games with VPA specifications to a library game with specifications given as a finite state automata. Let $(\mathcal{L}ib, \mathcal{W}_{\mathcal{A}_v})$ be a library game where $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$ and $\mathcal{W}_{\mathcal{A}_v}$ is the language recognized by the deterministic VPA $\mathcal{A}_v$.

The idea is to achieve the synchronization on the stack symbols between automaton and specification using the mechanics of the game, such that the specification can be considered as a finite state automaton. The adversary pl_1 guesses a symbol of the stack and then he challenges pl_0 to produce the same symbol. The pl_1 guess must be kept invisible to pl_0, to avoid that the local memory could help pl_0 in his choices.

We define a new library game $(\widehat{\mathcal{L}ib}, \mathcal{W}_{\mathcal{A}})$, where $\mathcal{W}_{\mathcal{A}}$ is the language recognized by a finite state automaton $\mathcal{A}$. The new library is $\widehat{\mathcal{L}ib} = \langle \{\{C_i\}_{i\in[0,n]} \cup \{C_{stack_i}\}_{i\in[0,n]}\},\ Y_{\widehat{\mathcal{L}ib}}\rangle$.

The structure of a component C_{stack_i} with $i \in [1, n]$ is depicted in Fig. 4. The component C_{stack_0} has the same structure, but its boxes are unmapped. In the figure, we denote with g the number of stack symbols. Remember that we denote with k the number of exits of any possible component C. Note that all the vertices are controlled by pl_1. In C_{stack_i}, pl_1 guesses a stack symbol pushed by the specification pushdown automaton.

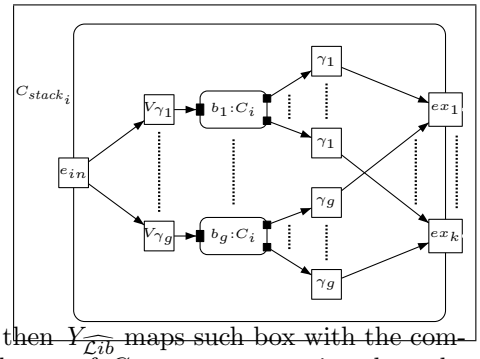

Fig. 4. The prototype of a component C_{stack_i} with $i \in [n]$

The box-to-component $Y_{\widehat{\mathcal{L}ib}}$ is defined to preserve the original box-to-component of the input library and to partially guarantee the alternation of components and stack components. If $Y_{\mathcal{L}ib}$ maps a box b to a component C_i, then $Y_{\widehat{\mathcal{L}ib}}$ maps such box with the component C_{stack_i}, with $i \in [n]$. All the boxes of C_{stack_i} are preassigned to the component C_i, with $i \in [n]$. The boxes of C_{stack_0} are unmapped. If $Y_{\mathcal{L}ib}$ is undefined for a box b, then $Y_{\widehat{\mathcal{L}ib}}$ maps the box b to the component C_{stack_0}. Formally, the new box-to-component $Y_{\widehat{\mathcal{L}ib}}$ is defined according to the following rules:

- $Y_{\widehat{\mathcal{L}ib}}(b) = C_i$ iff $b \in B_{C_{stack_i}}$ with $i \in [n]$ (*assign the boxes of* C_{stack_i} *to* C_i)
- $Y_{\widehat{\mathcal{L}ib}}(b) = C_{stack_i}$ iff $\exists i \in [n]$ such that $Y_{\mathcal{L}ib}(b) = C_i$ (*assign the boxes mapped with* C_i *to* C_{stack_i})
- $Y_{\widehat{\mathcal{L}ib}}(b) = C_{stack_0}$ iff $b \in B_{\mathcal{L}ib}$ and $Y_{\mathcal{L}ib}(b)$ is undefined (*assign the unmapped boxes of* $\mathcal{L}ib$ *to* C_{stack_0})
- $Y_{\widehat{\mathcal{L}ib}}(b)$ is undefined iff $b \in B_{C_{stack_0}}$ (*the boxes of* C_{stack_0} *are unmapped*)

The winning condition is given as a finite state automaton $\mathcal{A}$ that is the intersection of two finite state automata, $\mathcal{A}_1$ and $\mathcal{A}_2$. We embed the top stack symbol of the deterministic VPA in the states of $\mathcal{A}_1$. Moreover, the states of $\mathcal{A}_1$ simulate the corresponding states of $\mathcal{A}_v$, and we will get that the winning condition is equivalent. On calls, $\mathcal{A}_1$ must mimic a push transition t from the current state, by first storing in the control the pushed symbol γ and the next control state according to t, then, if the next input is γ, it continues, otherwise it enters an the accepting sink state. Returns are handled similarly (the popped symbol occurs after the return, and the fact that this corresponds to the symbol actually pushed in the current run by on the matching call is ensured by the instance of $\{C_{stack_i}\}_{i\in[0,n]}\}$).

As we said, the alternation of calls to instances of stack synchronization and calls to instances of the input library is guaranteed by the box-to-component map, with the exception of C_{stack_0}. When an instance of C_{stack_0} is entered, the alternation is guaranteed in this case by the second finite state automaton $\mathcal{A}_2$.

The automaton cycles on a state q_{in} until it reads a node labeled with $e_{C_{stack_0}}$. When such state is read, the automaton enters a state q_{wait} and cycles on it, waiting to read an entry e. If e is an entry of a component C_i with $i \in [n]$, the automaton enters again q_{in}, otherwise it enters a rejecting sink state. Due to the fact that pl_0 can not see the moves done in the C_{stack} instances (because their vertices are controlled by pl_1), the winning local strategies for pl_0 in the original game are exactly the same in the new game.

Denoting with k the number of exits of $\mathcal{L}ib$, with g the number of stack symbols, and z the number of states of the specification, we get that the resulting library $\widetilde{\mathcal{L}ib}$ has $2k$ exits and the resulting automaton $\mathcal{A}$ has $O(zg)$ states. Combining with the complexity to solve LMS problem, we have:

Theorem 6. *Solving the* LMS *problem for deterministic VPA winning conditions is* EXPTIME*-complete.*

This reduction clearly applies also to component-based and single-instance VPA LMS, and we say:

Theorem 4. *The considered VPA* LMS *problems are* EXPTIME*-complete.*

E Comparing other problems to LMS problems

Synthesis from recursive-component libraries. In the synthesis from recursive-component libraries (see [11]), the library of reusable components is modeled using a set of transducers with call-return structures. The related synthesis problem asks to find a composition of these elements such that the synthesized system fulfills a given LTL specification. This problem is strictly related to our setting and, in particular, it is a specific restriction of our problem. In fact, an instance of the problem proposed in [11] can be considered as an instance of the component-based LMS problem, where the library only has standard (non-game) components, i.e. components without the internal game between two players. In this case, a potential solution must determine only the correct external composition, as in [11]. However, note that the problem solved in [11] is a model checking problem, i.e. it asks that all the possible runs of the resulting system must be winning. This means that ,if we want to have the equivalence, the input library for the component-based LMS library game must be formed by components with all pl_1 vertices.

Rephrasing an LMS problem in terms of synthesis from library of recursive-components determines a significant difference. Consider a node where the local strategy, according to two different local histories, can choose two different moves. If we want to model such behavior with the recursive (non-game) components presented in [11], we must solve the choices of pl_0. In this particular example, we should define two distinct recursive components, one that models the first choice, and, an other that models the second choice, splitting the considered node in two distinct nodes. This means that, if we want to use a library

of recursive-components to model a library of components , we could need to generate a library with a potentially infinite number of transducers.

Global vs. modular synthesis. Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$. A *global RSM* $S = \langle \{G_i\}_{i\in[0,m]}, Y_S, f_g\rangle$ where:

- for $i \in [0, m]$, $G_i \overset{\text{iso}}{\equiv} C_{j_i}$ holds,
- $G_0 \overset{\text{iso}}{\equiv} C_0$ holds,
- the *box-to-instance* map $Y_S : \bigcup_{i\in[0,m]} B_{G_i} \to \{G_i\}_{i\in[m]}$ is a total function that is consistent with $Y_{\mathcal{L}ib}$, i.e. for each $i \in [0, m]$ and $b \in B_{G_i}$, denoting with b' the box of C_{j_i} that is isomorphic to b, it holds that if $Y_{\mathcal{L}ib}(b') = C_{j_i}$ then $Y_S(b) = G_i$
- f_g is a *global strategy*, i.e. $f_g : V_S^*.P_S^0 \to Calls_S \cup N_S$ such that $f_g(w.u) \in \delta_S(u)$ and f_g is computable by a pushdown automaton.

Given a library and an ω-regular specification, the global Lms (Lgs for short) problem asks to construct a global RSM S such that S satisfies the given specification.

We introduce the Lgs problem to show that, even if the Lms problem allows to generate a solution with a potentially unbounded number of instances and, consequently, new local strategies for each of them, the constructed system can not act as it knew the complete history of the plays (as using a global strategy). To support this claim, we present an example (Fig.5).

Consider the depicted library and a winning condition that asks to see alternatively often the symbols α and β. It is easy to see that a solution S exists if we admit a global strategy: S is composed of two instances, one from C_0 and one from C_1, and, each time the play reaches e_1, looking at the global memory of the play, the strategy can see if x_1 was the last visited exit, and then x_2 is selected, or it is x_2, and then x_1 is selected. On the other hand, if only local strategies can be used, there is no solution for the proposed problem: in fact, the box b can be assigned only one time to a fixed instance of C_1. In this case, each time the play reaches e_1, the strategy is forced to move always ~~on x_1 or always~~ on x_2, because at the entry point, the instance has no memory about the previous moves/calls of the play.

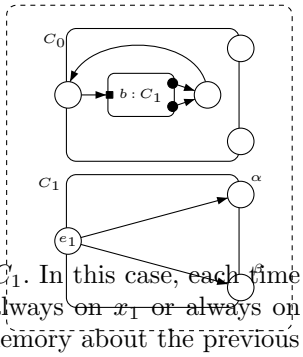

Fig. 5. A sample library.

The Lgs problem is decidable and can be solved introducing some changes to the algorithm for the Lms problem. We give only the idea for this solution. The key intuition is that in an RSM synthesized using the Lms restrictions, if in an activation of an instance two different paths reach a same box, they must call the same instance. The behavior of the called instance is summarized using a pre-post condition. Using the global strategy, even if in an activation of an instance two paths reach a same box, the invoked instance can implement different behaviors according to the different global histories. This means that

an instance can implement a set of behaviors that can be modeled by a set of single pre-post conditions. Each single pre-post condition is independent and it can be verified using a different strategy.

A *single pre-post condition* $\bar{p}$ is a tuple of the form $(q, [Q_1, \ldots, Q_k])$ where $q \in Q$, for each $i \in [k]$, $Q_i \subseteq Q$.

A *global pre-post condition* is a set $\bar{\mathcal{P}} = \{\bar{p}_i\}_{i \in [z]}$ of single pre-post conditions such that for any $i, j \in [z]$ with $i \neq j$ then $\bar{p}_i \neq \bar{p}_j$. Due to the fact that the set Q is finite and, fixed a precondition q, there are $2^{|Q|k}$ distinct postconditions, this means that $z \leq 2^{|Q|^2 k}$.

A *global box summary* of an instance of C is a tuple $\bar{\mathcal{B}_C} = \langle \hat{Y}_C, \{\bar{\mathcal{P}}_{\hat{Y}_C(b)}\}_{b \in B_C} \rangle$, where $\hat{Y}_C$ is a a box mapping and $\{\bar{\mathcal{P}}_{\hat{Y}_C(b)}\}_{b \in B_C}$ is a set of global pre-post conditions, one for each box.

If we consider safety winning condition, we slightly change the structure of the solution for the LMS problem to solve the LGS problem, handling these different assumptions. We introduce a set of automata of the form $\mathcal{A}^C_{\bar{\mathcal{P}},\bar{\mathcal{B}}}$ that starts an automaton $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ for each $\bar{p} \in \bar{\mathcal{P}}$. An automaton of the form $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ is a simplified form of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$. In fact, the states are: an initial state q_s, an accepting sink state q_a, a rejecting sink state q_r, a state q_e, a state q_b for each box b of C, and states of the form (R) where $R \subseteq Q$. If $\bar{p} = (q, [Q_1, \ldots, Q_k])$ from q_e, it behaves as from $(\{q\})$ if the current node corresponds to the entry of C (remember that q is the precondition of $\bar{p}$). On the update on tree-nodes corresponding to a call $(1, b)$, $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ must nondeterministically choose a particular single pre-post condition $\bar{p}$ in $\bar{\mathcal{P}}_{\hat{Y}_C(b)}$ and, then, use it to execute the update of its state (R). The choice of the single pre-post condition can change each time a tree-nodes corresponding to a call is read. In the remaining cases, the automaton $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ acts as $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$, with the simplification that it works only on the single set R.

In $\mathcal{A}'$ we extend the alphabets such that $\mathcal{A}^C_{\bar{\mathcal{P}},\bar{\mathcal{B}}}$ accepts each tree that differs from the component tree T_C of C only for the labeling of the leaves corresponding to the boxes of C. The labels are of the form $(C, \bar{\mathcal{P}}, \bar{\mathcal{B}})$. The working of $\mathcal{A}'$ is exactly the same, up to a renaming from $\mathcal{P}/\mathcal{B}$ to $\bar{\mathcal{P}}/\bar{\mathcal{B}}$