Synthesis from Recursive-Components Libraries*

Yoad Lustig†
Rice University
6100 Main Street
Houston, TX 77005-1892, USA
yoad.lustig@gmail.com

Moshe Y. Vardi‡
Rice University
6100 Main Street
Houston, TX 77005-1892, USA
vardi@cs.rice.edu

Synthesis is the automatic construction of a system from its specification. In classical synthesis algorithms it is always assumed that the system is "constructed from scratch" rather than composed from reusable components. This, of course, rarely happens in real life. In real life, almost every non-trivial commercial software system relies heavily on using libraries of reusable components. Furthermore, other contexts, such as web-service orchestration, can be modeled as synthesis of a system from a library of components.

In 2009 we introduced LTL synthesis from libraries of reusable components. Here, we extend the work and study synthesis from component libraries with "call and return" control flow structure. Such control-flow structure is very common in software systems. We define the problem of Nested-Words Temporal Logic (NWTL) synthesis from recursive component libraries, where NWTL is a specification formalism, richer than LTL, that is suitable for "call and return" computations. We solve the problem, providing a synthesis algorithm, and show the problem is 2EXPTIME-complete, as standard synthesis.

1 Introduction

The design of almost every non-trivial software system is based on using libraries of reusable components. Reusable components come in many forms: functions, objects, or others. Nevertheless, the basic idea of constructing systems from reusable components underlies almost all software construction. Indeed, almost every system involves many sub-systems, each dealing with different engineering aspects and each requiring different expertise. In practice, the developer of a commercial product rarely develops all the required sub-systems herself. For example, a software application for an email client contains sub-systems for managing graphic user interface (as well as many other sub-systems). Rarely will a developer of the email-client system develop the basic graphic-user-interface functionality as part of the project. Instead, basic sub-systems functionality is usually acquired as a *library*, i.e., a collection of reusable components that can be integrated into the system. The construction of systems from reusable components is extensively studied. Many examples for important work on the subject can be found in Sifakis' work on component-based construction [16] and de Alfaro and Henzinger's work on "interface-based design" [1]. Furthermore, other situations, such as web-service orchestration [8, 15], can be viewed as the construction of systems from libraries of reusable components.

Synthesis is the automated construction of a system from its specification. The basic idea is simple and appealing: instead of developing a system and verifying that it adheres to its specification, we would like to have an automated procedure that, given a specification, constructs a system that is correct by

*For a longer version of this paper see http://www.cs.rice.edu/∼vardi/papers.

†Current address: Yahoo! Labs Haifa, Matam Scientific Industries Center Building #3, Matam Park, Haifa, 31905 Israel, email: yoad@yahoo-inc.com

‡Work supported in part by NSF grants CCF-0728882, and CNS 1049862, by BSF grant 9800096, and by gift from Intel.

Giovanna D'Agostino, Salvatore La Torre (Eds.):
Proceedings of the Second International Symposium on
"Games, Automata, Logics and Formal Verification" (GandALF 2011)
EPTCS 54, 2011, pp. 1–16, doi:10.4204/EPTCS.54.1

construction. The modern approach to temporal synthesis was initiated by Pnueli and Rosner, who introduced LTL (linear temporal logic) synthesis [14]. In LTL synthesis, the specification is given in LTL and the system constructed is a finite-state transducer modeling a reactive system. In this setting of synthesis it is always assumed that the system is "constructed from scratch" rather than "composed" from reusable components. In [12], we introduced the study of synthesis from reusable components. We argued there that even when it is theoretically possible to design a sub-system from scratch, it is often desirable to use reusable components. The use of reusable components allows abstracting away most of the detailed behavior of the sub-system, and writing a specification that mentions only the aspects of the sub-system relevant for the synthesis of the system at large.

A major concern in the study of synthesis from reusable components is the choice of a mathematical model for the components and their composition. The exact nature of the reusable components in a software library may differ. The literature, as well as the industry, suggest many different types of components; for example, function libraries (for procedural programming languages) or object libraries (for object-oriented programming languages). Indeed, there is no one correct model encompassing all possible facets of the problem. The problem of synthesis from reusable components is a general problem to which there are as many facets as there are models for components and types of composition. Components can be composed in many ways: synchronously or asynchronously, using different types of communications, and the like [16].

As a basic model for a component, following [12], we abstract away the precise details of the component and model a component as a *transducer*, i.e., a finite-state machine with outputs. Transducers constitute a canonical model for reactive components, abstracting away internal architecture and focusing on modeling input/output behavior. In [12], two models of composition were studied. In *data-flow* composition the output of one component is fed as input to another component. The synthesis problem for data-flow composition was shown to be undecidable. In *control-flow* composition control is held by a single component at every point in time; the composition of components amounts to deciding how control is passed between components, by setting which component receives control when another component relinquishes it. Control-flow is motivated by software (and web services) in which a single function is in control at every point during the execution. In [12] we focused on "goto" control flow, and proved that LTL synthesis in that setting is 2EXPTIME-complete.

In this paper we extend that work and study a composition notion that relates to "call and return" control structure. "Call and return" control flow is very natural for both software and web services. An online store, for example, may "call" the PayPal web service, which receives control of the interaction with the user until it returns the control to the online store. To allow for "call and return" control-flow structure, we define a recursive component to be a transducer in which some of the states are designated as exit states. The exist states are partitioned into call states, and return states. Intuitively, a recursive component receives control when entering its initial state and relinquishes control when entering an exit state. When a call state is entered, the control is transferred from the component in control to the component that is being called by the component in control. When a return state is entered, the control is transferred from the component in control to the component that called it (i.e., control is returned). To model return values, each transducer has several return states. Each return state is associated with a re-entry state. Thus, each transducer has a single entry state, several re-entry states, several return states, and several call states. Composing recursive components amounts to matching call states with entry states and return states with re-entry states.[1]

[1] It is possible to consider more complex models, for example, models in which there are several call values. The techniques presented here can be extended to deal with such models.

Dealing with "call and return" control flow poses two distinct conceptual difficulties. The first is the technical difficulty of dealing with a "call and return" system that has a pushdown store. When adapting the techniques of [12], a run is no longer a path in a control-flow tree, but rather a traversal in a composition tree, in which a return corresponds to climbing up the tree. To deal with this difficulty we employ techniques used with 2-way automata [13]. A second difficulty has to do with the specification language. "Call and return" control-flow requires a richer specification language than LTL [5, 3]. For example, one might like to specify that one function is only called when another function is in the caller's stack; or that some property holds for the local computations of some function. In recent years an elegant theory of these issues was developed, encompassing suitable specification formalisms, as well as semantic, automata-theoretic, and algorithmic issues [5, 3, 6]. Here we use the specification language *nested-words temporal logic* (NWTL) [3], and the automata-theoretic tool of *nested words Büchi automata* (NWBA) [3, 6].

We define here and study the NWTL recursive-library-component realizability and synthesis problems. We show that the complexity of the problem is 2EXPTIME-complete (like standard synthesis and synthesis of "goto" components) and provide a 2EXPTIME algorithm for the problem. We use the composition-tree technique of [12], in which a composition is described as an infinite tree. The challenge here is that we need to find nested words in classical trees. While the connection between nested words and trees has been studied elsewhere, cf. [2], our work here is the first to combine nested-word automata with the classical tree-automata framework for temporal synthesis, using techniques developed for two-way automata [13, 17].

2 Preliminaries

Transducers: A *transducer* is a deterministic automaton with outputs; $\mathscr{T} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, F, L \rangle$, where: Σ_I is a finite input alphabet, Σ_O is a finite output alphabet, Q is a set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma_I \to Q$ is a transition function, F is a set of final states, and $L : Q \to \Sigma_O$ is an output function labeling states with output letters. For a transducer $\mathscr{T}$ and an input word $w = w_1 w_2 \dots w_n \in \Sigma_I^n$, a *run*, or a *computation* of $\mathscr{T}$ on w is a sequence of states $r = r_0, r_1, \dots r_n \in Q^n$ such that $r_0 = q_0$ and for every $i \in [n]$ we have $r_i = \delta(r_{i-1}, w_i)$.

For a transducer $\mathscr{T}$, we define $\delta^* : \Sigma_I^* \to Q$ in the following way: $\delta^*(\varepsilon) = q_0$, and for $w \in \Sigma_I^*$ and $\sigma \in \Sigma_I$, we have $\delta^*(w \cdot \sigma) = \delta(\delta^*(w), \sigma)$. A Σ_O-labeled Σ_I-tree $\langle \Sigma_I^*, \tau \rangle$ is *regular* if there exists a transducer $\mathscr{T} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, L \rangle$ such that for every $w \in \Sigma_I^*$, we have $\tau(w) = L(\delta^*(w))$. A transducer $\mathscr{T}$ outputs a letter for every input letter it reads. Therefore, for an input word $w_I \in \Sigma_I^\infty$, the transducer $\mathscr{T}$ induces a word $w \in (\Sigma_I \times \Sigma_O)^\infty$ that combines the input and output of $\mathscr{T}$. The *maximal computations* of $\mathscr{T}$ are those that exit at a final state in F or are of length ω.

Nested Words, NWTL and NWBA: When considering a run in the "call and return" control-flow model, the run structure should reflect both the linear order of the execution and the matching between calls and their corresponding returns. For example, when a programmer uses a debugger to simulate a run, and the next command to be executed is a call, there are two natural meanings to "simulate next command": first, it is possible to execute the next machine command to be executed (i.e. jump into the called procedure). In debugger terminology this is "step into", and this meaning reflects the linear order of machine commands being executed. On the other hand, it is possible to simulate the entire computation of the procedure being called, i.e. every machine command from the call to its corresponding return. In compiler terminology this is "step over", and this meaning reflects the matching between calls and their returns. Thus, the structure of a run, with the matching between calls and returns, is richer then the

sequence of commands that reflects only the linear order. Relating to this richer structure is crucial for reasoning about recursive systems, and it should be reflected in the mathematical model of a run, in the formalism by which formal claims on runs are made, i.e., in the specification formalism.

A run in a "call and return" model is a sequence of configurations, or *a word*, together with a matching relation that matches calls and their corresponding returns. The matching relation is *nested*, i.e. constrained to ensure that a return to an inner call appears before the return to an outer call. A formal definition appears below. The model of the run consists of both the word (encoding the linear order) and the matching relation. A word with nested matching is a *nested word* [6]. At the specification level, it should be possible to make formal claims regarding system that refer to the "call and return" structure [5, 3]. For example: one may want to argue about the value of some memory location as long as a function is in scope (i.e. during the subsequence of the computation between the call to the function and its corresponding return). Alternatively one may want to argue about the values of some local values whenever some function is in control (that may correspond to several continuous subsequences of commands). Another example is arguing about the call stack whenever some function is in control (such as "whenever f is in control either g or h are on the call stack"). Several specification formalisms were suggested to reason about "call and return" computations [5, 3, 6]. Here we use *Nested Words Temporal Logic*, (NWTL) [3], which is both expressive and natural to use. Finally, to reason about nested words, we use *nested words Büchi automata* (NWBA), which are a special type of automata that run on nested words [3, 6]. Intuitively, in a standard infinite word, each letter has a single successor letter. Therefore, automata on standard words can be seen as being in some state q, reading a letter σ and "sending" the next state q' to the successor letter σ'. In a nested word, however, a letter σ might have two "natural successors". First the letter σ' following it in the linear sequence of execution, and second another letter σ'' that is matched to it by the "call and return" matching. A NWBA not only "sends" a state to the successor letter σ, but also "sends" some information, named *hierarchical symbol*, to the matched letter σ''. The transition relation takes into account both the state and the hierarchical symbols. A formal definition of NWBA's is presented below.

We proceed with the formal definitions of nested words, the logic NWTL for nested words, and the automata NWBA running on nested words. The material presented below is taken from [3], which we recommend for a reader who is not familiar with nested words, their logic, or their automata.

A *matching* on $\mathbb{N}$ or an interval $[1,n]$ of $\mathbb{N}$ is a binary relation μ and two unary relations *call* and *ret*, satisfying the following: (1) if $\mu(i,j)$ holds then $call(i)$ and $ret(j)$ hold and $i<j$; (2) if $\mu(i,j)$ and $\mu(i,j')$ hold then $j=j'$ and if $\mu(i,j)$ and $\mu(i',j)$ hold then $i=i'$; (3) if $i\leq j$ and $call(i)$ and $ret(j)$ hold, then there exists $i\leq k\leq j$ such that either $\mu(i,k)$ or $\mu(k,j)$. Let Σ be a finite alphabet. A finite nested word of length n over Σ is a tuple $\overline{w}=\langle w,\mu,call,ret\rangle$, where $w=a_1\dots a_n\in\Sigma^*$, and $\langle\mu,call,ret\rangle$ is a matching on $[1,n]$. A nested ω-word is a tuple $\overline{w}=\langle w,\mu,call,ret\rangle$, where $w=a_1\dots\in\Sigma^\omega$, and $\langle\mu,call,ret\rangle$ is a matching on $\mathbb{N}$. We say that a position i in a nested word $\overline{w}$ is a call position if $call(i)$ holds; a return position if $ret(i)$ holds; and an internal position if it is neither a call nor a return. If $\mu(i,j)$ holds, we say that i is the matching call of j, and j is the matching return of i, and write $c(j)=i$ and $r(i)=j$. Calls without matching returns are pending calls. For a nested word $\overline{w}$, and two positions i,j of $\overline{w}$, we denote by $\overline{w}[i,j]$ the substructure of $\overline{w}$ (i.e., a finite nested word) induced by positions l such that $i\leq l\leq j$. If $j<i$ we assume that $\overline{w}[i,j]$ is the empty nested word. For nested ω-words $\overline{w}$, we let $\overline{w}[i,\infty]$ denote the substructure induced by positions $l\geq i$. When this is clear from the context, we do not distinguish references to positions in subwords $\overline{w}[i,j]$ and $\overline{w}$ itself, e.g., we shall often write $\langle\overline{w}[i,j],i\rangle\models\varphi$ to mean that φ is true at the first position of $\overline{w}[i,j]$.

Nested words temporal logic (*NWTL*) is a specification formalism suitable for "call and return" computations [3]. First we define a summary path between positions $i<j$ in a nested word $\overline{w}$. Intuitively, a

summary path skips from calls to returns on the way from i to j. The *summary path* between positions $i < j$ in a nested word $\overline{w}$ is a sequence $i = i_0 < i_1 < \ldots < i_k = j$ such that for all $p < k$ we have $i_{p+1} = r(i_p)$ if i_p is a matched call and $j \geq r(i_p)$; or $i_{p+1} = i_p + 1$ otherwise. Next, we define NWTL syntax. For an alphabet Σ, the letters of Σ, $\top$ (standing for true), *call*, and *ret* are NWTL formulas. NWTL has the operators: not $\neg$, or $\vee$, next $\bigcirc$, abstract next (that skips from a call to its return) $\bigcirc_\mu$, previous $\ominus$, abstract previous $\ominus_\mu$, summary until (to be defined below) $\mathbf{U}^\sigma$, and summary since $\mathbf{S}^\sigma$. For NWTL formulas φ_1, φ_2 the following are NWTL formulas: $\neg\varphi_1 | \varphi_1 \vee \varphi_2 | \bigcirc \varphi_1 | \bigcirc_\mu \varphi_1 | \ominus \varphi_1 | \ominus_\mu \varphi_1 | \varphi_1 \mathbf{U}^\sigma \varphi_2 | \varphi_1 \mathbf{S}^\sigma \varphi_2$. We proceed to define NWTL semantics. Let $w = w_1 \ldots w_n$ or $w_1 \ldots$ be a finite or infinite word over Σ. Let $\overline{w} = \langle w, call, ret, \mu \rangle$, and $i \geq 1$ be a number bounded by the length of w. Every nested word satisfies $\top$, in particular $(\overline{w}, i) \models \top$. For a letter $\sigma \in \Sigma$ we have $(\overline{w}, i) \models \sigma$ iff $\sigma = w_i$. (This is can be extended to alphabets of the type $\Sigma = 2^{AP}$, that consists of sets of atomic propositions, in the standard way, i.e., $(\overline{w}, i) \models p$ iff $p \in w_i$). Boolean operators semantics is standard $(\overline{w}, i) \models \neg\varphi$ iff $(\overline{w}, i) \not\models \varphi$; and $(\overline{w}, i) \models \varphi_1 \vee \varphi_2$ iff $(\overline{w}, i) \models \varphi_1$ or $(\overline{w}, i) \models \varphi_2$. We also have $(\overline{w}, i) \models \bigcirc\varphi$ iff $(\overline{w}, i+1) \models \varphi$ and $(\overline{w}, i) \models \ominus \varphi$ iff $(\overline{w}, i-1) \models \varphi$. We have $(\overline{w}, i) \models call$ iff i is a call, and $(\overline{w}, i) \models ret$ iff i is a return. We have $(\overline{w}, i) \models \bigcirc_\mu \varphi$ iff i is a call with a matching return j (i.e., $\mu(i, j)$ holds) and $(\overline{w}, j) \models \varphi$. Similarly, $(\overline{w}, i) \models \ominus_\mu \varphi$ iff i is a return with a matching call j (i.e., $\mu(j, i)$ holds) and $(\overline{w}, j) \models \varphi$. For summary until we have $(\overline{w}, i) \models \varphi_1 \mathbf{U}^\sigma \varphi_2$ iff there exists a $j \geq i$ for which $(\overline{w}, j) \models \varphi_2$, and for the summary path $i = i_0 < i_1 < \ldots < i_k = j$ between i and j we have for every $p < k$ that $(\overline{w}, i_p) \models \varphi_1$. Similarly, $(\overline{w}, i) \models \varphi_1 \mathbf{S}^\sigma \varphi_2$ iff there exists a position $j < i$ for which $(\overline{w}, j) \models \varphi_2$ and for the summary path $j = i_0 < i_1 < \ldots < i_k = i$ between j and i we have for every $p \in [k]$ that $(\overline{w}, i_p) \models \varphi_1$.

Rather than use NWTL directly, we use here *nested-word Büchi automata* (NWBA), which are known to be at least as expressive as NWTL; in fact, there is an exponential translation from NWTL to NWBA [3], analogous to the exponential translation of linear temporal logic to Büchi automata [18]. A *nondeterministic nested word Büchi automaton* (*NWBA*) is a tuple $\mathscr{A} = \langle \Sigma, Q, Q_0, Q_f, P, P_0, P_f, \delta_c, \delta_i, \delta_r \rangle$, consisting of a finite alphabet Σ, finite set Q of states, a set $Q_0 \subseteq Q$ of initial states, a set $Q_f \subseteq Q$ of accepting states, a finite set P of hierarchical symbols, a set $P_0 \subseteq P$ of initial hierarchical symbols, a set $P_f \subseteq P$ of final hierarchical symbols, a call-transition relation $\delta_c \subseteq Q \times \Sigma \times Q \times P$, an internal transition relation $\delta_i \subseteq Q \times \Sigma \times Q$, and a return-transition relation $\delta_r \subseteq Q \times P \times \Sigma \times Q$. The automaton $\mathscr{A}$ starts in an initial state and reads the nested word from left to right. A run r of the automaton $\mathscr{A}$ over a nested word $\overline{w} = \langle a_1 a_2 \ldots, \mu, call, ret \rangle$ is a sequence $q_0, q_1, \ldots$ of states, and a sequence $p_{i_1}, p_{i_2}, \ldots$ of hierarchical symbols, corresponding to the call positions $i_1, i_2, \ldots$, such that $q_0 \in Q_0$, and for each position i, if i is a call then $\langle q_{i-1}, a_i, q_i, p_i \rangle \in \delta_c$; if i is internal, then $\langle q_{i-1}, a_i, q_i \rangle \in \delta_i$; if i is a return such that $\mu(j, i)$, then $\langle q_{i-1}, p_j, a_i, q_i \rangle \in \delta_r$; and if i is an unmatched return then $\langle q_{i-1}, p, a_i, q_i \rangle \in \delta_r$ for some $p \in P_0$. Intuitively, in a run r, the hierarchical symbol associated with a matched return position i, is the hierarchical symbol p_j, associated with the call position j that is matched to i. The run r is accepting if (1) for all pending calls i, $p_i \in P_f$, and (2) if $\overline{w}$ is a finite word of length l then the final state q_l is accepting (i.e., $q_l \in Q_f$), and if $\overline{w}$ is an ω-word then for infinitely many positions i, we have $q_i \in Q_f$. The automaton $\mathscr{A}$ accepts the nested word $\overline{w}$ if it has an accepting run over $\overline{w}$.

3 The computational model

Recursive Components and their composition: To reason about recursive components one has to choose a mathematical model for components. The choice of model has to balance the need for a rich modeling formalism, for which computationally powerful models are preferred, and the need to avoid the pitfall of undecidability, for which simpler models are preferred.

A successful sweet spot in this trade off is the computational model of finite-state transducers, i.e. finite-state machines with output. A common approach to reasoning about real world systems, is abstracting away the data-intensive aspects of the computation and model the control aspects of the computation by a finite-state transducer. Using this approach, the transducers model is rich enough to model real world industrial designs [9, 7]. For that reason, transducers are widely used in both theory [18, 14, 4] and practice [9, 7], and are prime candidates as a model for "call and return" components.

To model "call and return" control-flow by transducers, we introduce a small variation on the basic transducer model. Essentially, we use transducers in which some states are "call states", where a transition to one of these states stands for a call to another component; some states are "return" states, where a transition to one of these states stands for a return to the component that called this component; and some states are re-entry states, i.e., states to which the component enters upon return from a call to another component. Similar models can be found in [4]. Different return values, are modeled here by having different re-entry states. The model is somewhat simplified in the sense that a return is not constrained in terms of the call state through which the call was made. In software, for example, the return is constrained to the instruction following the call instruction (although several return values may be permitted). Nevertheless, the model is rich enough to deal with the essence of "calls and returns", and the techniques we present can be used to deal with richer models (e.g. each call may be associated with a mapping between return states and re-entry states capturing constrained returns as above). We chose this simpler model as it allows for simpler notation and clearer presentation of the underlying ideas.

To simplify the notation, we fix a number n_C and assume every component in the library has exactly n_C calls. Similarly, we fix a number n_R and assume every component in the library has exactly n_R return points, as well as exactly n_R points to which the control is passed upon return.

A *Recursive Library Component* (*RLC*) is a finite transducer with call, return and re-entry states. Formally, an RLC is a tuple $M = \langle \Sigma_I, \Sigma_O, S, s_0, s_e^R, S_C, S_R, \delta, L \rangle$ where: (1) Σ_I and Σ_O are finite input and output alphabets. (2) S is a finite set of states. (3) $s_0 \in S$ is an initial state. When called by another component, the component M enters s_0. (4) $s_e^R \subseteq S$ is a set of re-entry states. When the control returns from a call to another component, M enters one of the re-entry states in s_e^R. We denote $s_e^R = \{s_e^1, \ldots, s_e^{n_R}\}$ (5) $S_C \subseteq S$ is a set of call states. When M enters a state in S_C, another component M' is called, and the control is transferred to M' until control is returned. We denote $S_C = \{s_C^1, \ldots, s_C^{n_C}\}$ (6) $S_R \subseteq S$ is a set of return states. When M enters a return state, the control is returned to the component that called M. We denote $S_R = \{s_R^1, \ldots, s_R^{n_R}\}$. When the i-th return state, i.e. s_R^i, is entered, control is returned to the caller component M', which is entered at his i-th re-entry state (i.e., M''s state s_e^i). (7) $\delta : S \times \Sigma_I \to S$ is a transition function. (8) $L : S \to \Sigma_O$ is an output function, labeling each state by an output symbol.

The setting we consider is the one in which we are given a library $\mathscr{L} = \{C_1, \ldots, C_l\}$ of RLC components. A *composition* over $\mathscr{L}$ is a tuple $\langle (1, C_1, f_1), (2, C_2, f_2), \ldots, (k, C_k, f_k) \rangle$ of k composition elements, in which each composition element is a triple composed of an index i, an RLC $C_i \in \mathscr{L}$, and an interface function $f_i : S_C \to [k]$ that maps each of C_i's call states into the composition element that is called upon entry to the call state. Note that the same RLC can be instantiated in different elements of the composition, but with different interface functions, and the size of the composition is a priori unbounded.[2] While we consider here only finite compositions, we could have considered, in principle, also infinite compositions. As we shall see, for NWBA specifications, finite compositions are sufficient.

A run of the system begins in state s_0 of C_1. When the run is in a state of the component C we say

[2] If we had bounded the number of elements in a composition, then the number of ways in which these elements can be composed would have been finite and the search for a composition that satisfies some specification would have turned into a combinatorial search, analogously, for example, to *bounded synthesis* [10].

that the component C is in control. For example, a run begins when the component C_1 is in control. For every $i \leq k$, as long as a component C_i is in control, the system behaves as C_i until an exit state (i.e. a call state or a return state) is entered. If a call state $s_C^j \in S_C$ of C_i is entered then the component $C_{f_i(j)}$ is called. That is, the control is passed to the $f_i(j)$-th component in the composition. The run proceeds from the start state of $C_{f_i(j)}$. If a return state $s_R^j \in S_R$ of C_i is entered (when C_i is in control), then C_i returns the control to the component that called C_i. If, for example, C_i was called by C_j then when s_R^m is entered, the run proceeds from the re-entry state s_e^m of C_j. We now define the composition formally.

Formally, a composition $C = \langle (1,C_1,f_1),(2,C_2,f_2),\ldots,(k,C_k,f_k)\rangle$, where $C_i = \langle \Sigma_I, \Sigma_O, S[i], s_0[i], s_e^R[i], S_C[i], S_R[i], \delta[i], L[i]\rangle$, induces a (possibly infinite) transducer $M = \{\Sigma_I, S_O, s_0^M, \delta^M, L^M\}$, where:

1. The input alphabet is Σ_I and the output alphabet is Σ_O.
2. The states of M are finite sequences of the form $\langle i_1, i_2, \ldots, i_m, s\rangle$, where for every $j \leq m$ we have $i_j \in [k]$, and the final element is a state $s \in S[i_m]$ of C_{i_m}. Intuitively, such a state stands for the computation being in the state s of the RLC C_{i_m}, where the computation call stack is $i_1, i_2, \ldots, i_m$. The initial state of M is $\langle 1, s_0[1]\rangle$ where $s_0[1]$ is the initial state of C_1. Formally, $S_M = [k]^* \cdot (\bigcup_{i\in[k]} i \cdot S[i])$.
3. Next, we define the transition function δ^M. Let $v = \langle i_1, i_2, \ldots, i_m, s\rangle$ be a state of M. Then, $\delta^M(v,\sigma) = v'$ if one of the following holds:
   (a) **internal transition**: If $\delta[i_m](s,\sigma) = s'$ for some state $s' \in S[i_m] \setminus (S_C[i_m] \cup S_R[i_m])$ of C_{i_m}, then $v' = \langle i_1, \ldots, i_r, s'\rangle$, where
   (b) **call transition**: If $\delta[i_m](s,\sigma) = s'$ where $s' \in S_C[i_m]$ is the j-th call state of C_{i_m} (i.e., $s' = s_C^j[i_m]$), then $v' = \langle i_1, \ldots, i_m, f_{i_m}(j), s_0[f_{i_m}(j)]\rangle$,
   (c) **return transition**: If $\delta[i_m](s,\sigma) = s'$ where $s' \in S_R[i_m]$, is the j-th return state of C_{i_m} (i.e., $s' = s_R^j[i_m]$), then $v' = \langle i_1, \ldots, i_{m-1}, s_e^j[i_{m-1}]\rangle$.
4. The final state set $F^M = \langle 1, S_R[1]\rangle$. Intuitively, the computation terminates when the first component returns.
5. The output function L^m is defined by $L^m(\langle i_1, \ldots, i_m, s\rangle) = L[i_m](s)$.

For an input word $w^I = w_0^I, w_1^I \ldots \in \Sigma_I^\infty$, the transducer M induces an output word $w^O = w_0^O, w_1^O, \ldots \in \Sigma_O^\infty$. We denote by $w = (w_0^I, w_0^O), (w_1^I, w_1^O) \ldots$ the combined input-output sequence induced by w^I. Furthermore, on the input word w^I, the composition C induces a nested word $\overline{w} = \langle w, call, ret, \mu\rangle$ in which w is the input-output induced word, *call* holds in positions in which a component made a call, *ret* holds in positions in which a component returned, and μ maps each call to its return. We sometime abuse notation and refer to the word w rather than the nested word $\overline{w}$. Similarly we might refer to a *computation* of, or in, a composition meaning a nested word induced by the composition. Similarly, we may refer to a *computation segment* meaning a substructure $\overline{w}[i,j]$, for some positions i, j, of a computation.

A composition C *realize* an NWTL specification φ if all computations induced by C satisfy φ. The *recursive-library-components realizability problem* is: given a library of RLCs $\mathscr{L} = \{M_j\}_{j=1}^n$ and an NWTL specification φ, decide whether there exists a composition of components from the library that realize φ. The *recursive-library-components-synthesis problem* is: given a library of RLCs $\mathscr{L} = \{M_j\}_{j=1}^n$ and an NWTL specification φ, decide whether φ is realizable by a composition of RLCs from $\mathscr{L}$ and if so, output a composition realizing φ.

Composition trees Next, we define the notion of a *composition tree*, which is the analog of a control-flow tree in [12]. Fixing a library $\mathscr{L}$ of RLCs, composition trees represent compositions. A composition tree is labeled tree $\tau = \langle T, \lambda\rangle$, where T, the tree structure, is the set $[n_C]^*$, and $\lambda : T \to \mathscr{L}$ is a mapping of the tree vertexes into $\mathscr{L}$. Every composition $C = \langle (1,C_1,f_1),(2,C_2,f_2),\ldots,(k,C_k,f_k)\rangle$, induces an $\mathscr{L}$-labeled composition tree τ_C. We first show that C induces a $[k]$-labeled tree that we call *intermediate tree*.

A labeled tree $\langle [n_C]^*, \kappa \rangle$, where $\kappa : [n_C]^* \to [k]$, is the intermediate mapping induced by C, if $\kappa(\varepsilon) = 1$, and, for every $v \in [n_C]^*$ and $j \in [n_C]$, we have that $\kappa(v \cdot j) = f_{\kappa(v)}(j)$. The *composition tree* induced by C is $\langle [n_C]^*, \lambda \rangle$ where for every $v \in [n_C]^*$ we have that $\lambda(v) = C_{\kappa(v)}$. A node $v = i_1 \cdots i_k$ represents a call-stack configuration. The node's label $\lambda(v)$ is the component in control, while the labels of the node's successors, i.e., $\lambda(v \cdot 1), \ldots, \lambda(v \cdot n_C)$, stand for the components that are called if a call state is entered. Intuitively, the control flow of an actual computation is a represented by a traversal in a composition tree. The control is first given to the component labeled by the root. For a node v, a call corresponds to a descent to a successor (where a call from the i-th call state corresponds to a descent to the i-th successor). Similarly, a return from a node v corresponds to an ascent to the predecessor of v.

Thus, a composition induces a composition tree. On the other hand, a composition tree can be seen as an "infinite composition" in which each node v stands for a composition element in which the component is the label of v, and the interface function f_v maps the call states to the successors (i.e., for every $v \in [n_C]^*$ and $i \in [n_C]$ we have $f_v(i) = v \cdot i$). So a composition tree induces an infinite composition. We abuse terminology and refer to computations of a composition tree, where we mean to refer to computations of the induced infinite composition. Furthermore, in Theorem 4.2 we show how a finite composition can be extracted from a *regular* composition tree. Another abuse of terminology we make is to refer to a labeled subtree of a composition tree as a composition tree.

4 Recursive-library-components synthesis algorithm

Our approach to the solution of the RLC synthesis problem, is first to construct a tree-automaton $\mathscr{A}_b$ that accepts composition trees that do *not* satisfy the specification. Once that is achieved, $\mathscr{A}_b$ can be complemented to get an automaton $\mathscr{A}$ which is accepts composition trees that *do* satisfy the specification. Finally, $\mathscr{A}$'s language can be checked for emptiness and if not empty, a system can be extracted from a witness (similar to the algorithm in [12]). Thus, the main ingredient in the solution is the following theorem (that allows the construction of $\mathscr{A}_b$).

Theorem 4.1: *Let $\mathscr{L}$ be a library of RLC components, each with n_R return states, and let $\mathscr{A}_\varphi$ be a NWBA. There exists an alternating Büchi automaton on trees (ABT) $\mathscr{A}$, with at most $O(|\mathscr{A}_\varphi|^2 \cdot n_R)$ states, whose language is the set of composition trees for which there exists a computation in the language of $\mathscr{A}_\varphi$.*

Our main result follows from Theorem 4.1.

Theorem 4.2: *The recursive library components realizability problem and the recursive library components synthesis problem are 2EXPTIME-complete.*

Proof: The algorithm proceeds as follows. We first translate $\neg\varphi$ into an equivalent NWBA $\mathscr{A}_{\neg\varphi}$, with an exponential blow-up [3]. We then construct an ABT $\mathscr{A}$ for $\mathscr{A}_{\neg\varphi}$ according to Theorem 4.1, dualize $\mathscr{A}$ into an an alternating co-Büchi automaton on trees (ACT) $\mathscr{A}'$, and check $\mathscr{A}'$'s language for nonemptiness as in [11]. If the specification is realizable, then the language of $\mathscr{A}'$ contains a regular composition tree, for which all computations satisfy φ. Otherwise, the language of $\mathscr{A}'$ is empty. Given a regular composition tree $\langle [n_C]^*, \tau \rangle$, it is induced by a transducer (without final states) $T = \langle [n_C], \mathscr{L}, Q, q_0, \delta, L \rangle$, such that for every $w \in [n_C]^*$, we have $\tau(w) = L(\delta^*(w))$. We assume, w.l.o.g. that the set Q is the set $[|Q|]$ of natural numbers, and that q_0 is the number 1. A finite composition can now be constructed in the following way: For every state $q \in Q$ there is a composition element $\langle q, C_q, f_q \rangle$ in which $C_q = L(q)$, and for every $j \in [n_C]$ we have $f_i(j) = \delta(i, j)$. It can then be shown that the constructed composition induces the same infinite-state transducer as the regular composition tree (up to component names) and therefore satisfies φ.

As for complexity, $\mathscr{A}$'s number of states is quadratic in $|\mathscr{A}_\varphi|$ and linear in n and b (upper bounding n_R by b). (Note that quadratic in $|\mathscr{A}_\varphi|$ is exponential in $|\varphi|$). The complementation of $\mathscr{A}$ into $\mathscr{A}'$ incurs no complexity cost. Finally, checking $\mathscr{A}'$ for emptiness is exponential in its number of states. This provides a 2EXPTIME upper bound. For a lower bound, note that a "goto" can be seen as a call without a return and LTL is a fragment of NWTL. Thus, a 2EXPTIME lower bound follows from the 2EXPTIME lower bound in [12]. □

We now prove Theorem 4.1. There are two sources of difficilty in the construction. First, we have to handle here call-and-return computations in composition trees. While computations in composition trees in [12] always go down the tree, computations here go up and down the tree. Second, here we have to emulate NWBA on the computations of composition trees, but we want to end up with standard tree automata, rather then nested-word automata.

Intuitively, given a computation tree as input, our construction would guess a computation of the input tree, in the language of $\mathscr{A}_\varphi$, together with an accepting run of $\mathscr{A}_\varphi$, on the guessed computation. As mentioned in in the discussion of Composition trees, however, a computation of the composed system corresponds to a traversal in the composition tree. Therefore, to guess the computation, i.e., the traversal in the input tree, and the computation of $\mathscr{A}_\varphi$ on it, we employ 2-way-automata techniques.

Let $\mathscr{A}_\varphi = \langle Q, Q_0, Q_f, P, P_0, P_f, \delta_c, \delta_i, \delta_r \rangle$. The construction of $\mathscr{A}$ is quite technical. Below we present the construction of $\mathscr{A}$, where the introduction of each part begins in an informal/intuitive discussion and ends in a formal definition.

The states of $\mathscr{A}$: Intuitively, $\mathscr{A}$ reads an input tree τ and guesses an accepting run of $\mathscr{A}_\varphi$ on a computation of that input tree. The difficulty is that a computation cannot be guessed node by node, since when a computation enters a call node, we need to consider the return to that node. Thus, when reading a node v labeled by component C, the ABT $\mathscr{A}$ guesses an *augmented computation* of C in which there are *call transitions* from call states to re-entry states, and a corresponding *augmented run* of $\mathscr{A}_\varphi$ (in which $\mathscr{A}_\varphi$'s state changes at the end of a call transition of C). Of course, when $\mathscr{A}$ guesses a call transition it should also verify that there exists a computation segment and a run segment of $\mathscr{A}_\varphi$, corresponding to that call transition. To verify a call transition from s_C^j to s_R^k, the ABT $\mathscr{A}$ sends a copy of itself, in an appropriate state, to j-child son of the component being read.

In general, $\mathscr{A}$ has two types of states: states for verifying call transition (i.e. computations segments between a call and its return), and states for verifying the existence of computation suffixes that do not return. An example of a computation suffix that does not return is a computation that follows a pending call. States of the first type verify the feasibility of a computation segment, and there exists such a state every triple $\langle q, q', i \rangle \in Q^2 \times [n_R]$. If $\mathscr{A}$ reads a tree node v in state $\langle q, q', i \rangle$ it has to verify the existence of a computation in which a call was made to v's component when $\mathscr{A}_\varphi$ was in state q, and the first return from v's component is from the i-th return state s_R^i, when $\mathscr{A}_\varphi$ is in state q'. States of the second type exist for every state $q \in Q$. If $\mathscr{A}$ reads a tree node v in state q it has to verify the existence of a computation suffix in which a call was made to v's component when $\mathscr{A}_\varphi$ was in state q, and $\mathscr{A}_\varphi^q$ has an accepting run on that suffix. The initial state of $\mathscr{A}$ is of the second type: the initial state q_0 of $\mathscr{A}_\varphi$.

In fact, the state space of $\mathscr{A}$ must reflect one more complication. The ABT $\mathscr{A}$ not only has to guess a computation of a system and a run of $\mathscr{A}_\varphi$ on it, the run of $\mathscr{A}_\varphi$ must be *accepting*. For that reason we also need to preserve information regarding $\mathscr{A}_\varphi$'s passing through an accepting state during a run segment. In particular, when considering a call transition that stand for a computation segment during which $\mathscr{A}_\varphi$ moved from q to q', it is sometimes important whether during that run segment $\mathscr{A}_\varphi$ passed through an accepting state. For that reason, states of the first type (that verify call transitions) come in two flavors: First, states $\langle q, q', i, 0 \rangle$ that retain the meaning explained above. Second, states $\langle q, q', i, 1 \rangle$ in which $\mathscr{A}$ has

to verify that in addition to the existence of a computation segment and an $\mathscr{A}_\varphi$ run segment as above, the run segment of $\mathscr{A}_\varphi$ *must* pass through an accepting state. Similarly, when $\mathscr{A}$ reads a component C while in state q, it has to verify there is a computation that does not return on which $\mathscr{A}_\varphi^q$ has an accepting run. One of the ways this might happen, is that the C would make a pending call to some other component C'. If this is the case, we need to keep track of whether an accepting state was seen from the entrance to C until the call to C'. For that reason, states of the type q also have two flavors: $\langle q,0\rangle$ and $\langle q,1\rangle$ (where the second type stands for the constrained case in which an accepting state must be visited). Thus, the formal definition of $\mathscr{A}$'s states set is $Q_\mathscr{A} = Q^2 \times [n_R] \times \{0,1\} \bigcup Q \times \{0,1\}$.

The transitions of $\mathscr{A}$: Intuitively, when $\mathscr{A}$ reads an input-tree node v and its labeling component C, the ABT $\mathscr{A}$ guesses an augmented computation and a corresponding augmented run that take place in C. Furthermore, for every call transition in the guessed augmented computation, the ABT $\mathscr{A}$ sends a copy of itself to the direction of the call to ensure the call transition corresponds to an actual computation segment. Thus, if the call transition is from s_C^j to s_R^k and $\mathscr{A}_\varphi$ is moves from q to q' on that transition, then for some $b \in \{0,1\}$ the ABT $\mathscr{A}$ sends a state $\langle q,q',k,b\rangle$ to the j-th direction (how b is chosen is explained below). The transition relation, therefore, has the following high level structure: a disjunction over possible augmented computations and runs, where for each augmented run a conjunction over all call transitions sending the corresponding $\mathscr{A}$'s states to the correct directions.

Before going into further detail, we introduce some notation: Given an augmented computation of C that begins in state s and ends in state s' and an augmented run of $\mathscr{A}_\varphi$ on it that begins in state q and ends in state q' we say that the beginning *configuration* is (s,q) and the final *configuration* is (s',q'). Transitions of $\mathscr{A}_\varphi$ that have to do with calls or returns have a hierarchical symbol associated with them. If the composition C is in state s, the ABT $\mathscr{A}$ is in state q and a hierarchical symbol p is associated then the *configuration* is (s,q,p). Given two configuration c_1 and c_2 then c_2 is *reachable in* C from c_1 if there exists computation segment of C, that contain no call transitions, that begins in c_1 and ends in c_2. The configuration c_2 is *reachable through accepting state in* C from c_1 if there exists computation segment of C, that contain no call transitions, that begins in c_1 and ends in c_2, and on which $\mathscr{A}_\varphi$ visits an accepting state.

Next, we describe the transitions out of a state $\langle q,q',k,0\rangle$. This is the simplest case as it does not involve analyzing whether an accepting state of $\mathscr{A}_\varphi$ is visited. Assume $\mathscr{A}$ is in state $\langle q,q',k,0\rangle$ when it reads a component C. Intuitively, this means that $\mathscr{A}$ has to guess an augmented computation of C that begins at C's initial state, and ends in C's k-th return state, and an augmented run of $\mathscr{A}_\varphi$ on that computation that begins in state q and ends in state q'. In fact, instead of explicitly guessing the entire augmented computation and run, what $\mathscr{A}$ actually guesses are only the call transitions appearing in the computation, and the state transitions of $\mathscr{A}_\varphi$ corresponding to these call transitions. These are needed as they define the states of $\mathscr{A}$ that will be sent in the various directions down the tree. The computation begins when C is in its initial state s_0, and $\mathscr{A}_\varphi$ is in state q. Thus the beginning configuration is (s_0,q). The first call transition source is some call state $s_C^{j_1}$ of C, some state q_1 of $\mathscr{A}_\varphi$ and a hierarchical symbol p_1 of $\mathscr{A}_\varphi$. Thus the first computation segment ends in configuration $(s_C^{j_1},q_1,p_1)$. Note that it must be the case that the configuration $(s_C^{j_1},q_1,p_1)$ is reachable in C from (s_0,q). The target of the call transition is some configuration $(s_R^{k_1},q_1',p_1)$. At this stage, i.e. when $\mathscr{A}$ reads C, the target configuration is only constrained by sharing the hierarchical symbol with the call transition source. The constraints on the possible states in the target configurations depend on components down the tree that $\mathscr{A}$ will read only at a later stage of its run. The configuration which is the source of the next call transition, however, again has to be reachable from $(s_R^{k_1},q_1',p_1)$.

Our approach, therefore is to define a graph G_C whose vertexes are configurations, and there exists an edge from a source configuration to a target configuration if it is possible to reach the target from the source (see earlier discussion of configurations). Recall the notation $C = \langle \Sigma_I, \Sigma_O, S, s_0, s_e^R, S_C, S_R, \delta, L \rangle$, where $s_e^R = \{s_e^i\}_{i=1}^{n_R}$, $S_C = \{s_C^i\}_{i=1}^{n_C}$, and $S_R = \{s_R^i\}_{i=1}^{n_R}$. The vertex set V_C of G_C is the union of four sets: (1) Initial configurations $\{s_0\} \times Q$. (2) Call configurations $S_C \times Q \times P$. (3) Re-entry configurations $s_e^R \times Q \times P$. (4) Final configurations $\{s_R^k\} \times Q$.

There are two types of edges in G_C. *Component edges* reflect reachability in C. There is a component edge in G_C from configuration c_1 to configuration c_2 iff c_2 is reachable in C from c_1. *Call edges* capture call transitions and the corresponding state changes in $\mathscr{A}_\varphi$. There is a call edge in G_C between $c_1 = (s,q,p)$ and $c_2 = (s',q',p')$ if s is a call state, s' is a re-entry state, and $p = p'$.

An augmented computation and run of $\mathscr{A}_\varphi$ on it, correspond to a path in G_C. When $\mathscr{A}$ is in state $\langle q,q',k,0 \rangle$ and reads a component C it guess a path in G_C from $\langle s_0,q \rangle$ to $\langle s_R^k,q' \rangle$. If there exists such a path in G_C there exists a short path of length bounded by $|V_C|$, i.e. the number of vertexes in G. We denote by $Path(q,q',s_R^k)$ the set of paths from (s_0,q) to (s_R^k,q') of length bounded by $|V_C|$. For each path $\pi \in Path(q,q',s_R^k)$, we denote by $E_C(\pi)$ the set of call edges appearing in π. For a call edge $e = \langle (s_C^i,q,p),(s_e^j,q',p) \rangle$, we denote $s_C(e) = i$, $s_0(e) = j$, $q(e) = q$, and $q'(e) = q'$. The transitions from $\langle q,q',k,0 \rangle$ are defined:

$$\delta(\langle q,q',k,0 \rangle, C) = \bigvee_{\pi \in Path(q,q',s_R^k)} \bigwedge_{e \in E_C(\pi)} (s_C(e), \langle q(e), q'(e), s_0(e), 0 \rangle).$$

Intuitively, a path in G_C is guessed and for each call edge e, the state $\langle q(e),q'(e),s_0(e),0 \rangle$ is sent in the direction of the call, i.e. $s_C(e)$.

Next, we describe the transitions out of a state $\langle q,q',k,1 \rangle$. This case a very similar to the case of transitions out of $\langle q,q',k,0 \rangle$ outlined above. The difference is that in this case $\mathscr{A}_\varphi$ must visit an accepting state during its augmented run. There is no restriction, however, that the accepting state will be visited when the control is held by the component C. It is possible that the accepting state will be visited when some other (called) component is in control. Intuitively, as in the $\langle q,q',k,0 \rangle$ case, the ABT $\mathscr{A}$ guesses a path in G_C from the initial to the final configuration, in addition, $\mathscr{A}$ guesses an edge from the path in which an accepting state should be visited. For component edges, it is possible to make sure that guessed edges represent computations on which $\mathscr{A}_\varphi$ visits an accepting state. For call edges, the task of verifying that an accepting state is visited, is delegated to the state of $\mathscr{A}$ that is sent in the direction of the call (by sending a state whose last bit b is 1).

Formally, a component edge in G_C from configuration c_1 to configuration c_2 is an *accepting edge* iff c_2 is reachable in C through an accepting state from c_1. Note that if there exists a path from a configuration c_1 to configuration c_2 that visits an accepting edge, then there exists one of length at most $2|V_G|$ (a simple path to the accepting edge and a simple path from it). For $q,q' \in Q$, $s_R^k \in S_R$, we denote by $Path_a(q,q',s_R^k)$ a set of pairs in which the first element is a path π of length at most $2|V_C|$ from (s_0,q) to (s_R^k,q'), and the second element is a function f mapping the edges in π into $\{0,1\}$ such that:

1. Exactly one edge is mapped to 1, and
2. If the edge mapped to 1 is a component edge then it is also an accepting edge.

Finally,

$$\delta(\langle q,q',k,1 \rangle, C) = \bigvee_{(\pi,f) \in Path_a(q,q',s_R^k)} \bigwedge_{e \in E_C(\pi)} (s_C(e), \langle q(e), q'(e), s_0(e), f(e) \rangle).$$

Next, we describe the transitions out of a state $\langle q,b \rangle$, for $b \in \{0,1\}$, in which $\mathscr{A}$ has to verify there exists an accepting augmented computation of C that does not return, and a run of $\mathscr{A}_\varphi^q$ on it. There are

three distinct forms such a computation might take. (1) First, it is possible that the computation has a infinite suffix in which C remains in control. (2) Second, it is possible that the eventually the component makes some pending call. (3) Finally, it is possible that the computation contains infinitely many calls to, and returns from, other components. We deal with each of the case separately, we construct a partial transition relation for each case, the transition relation itself is the disjunction of these three parts.

First, to deal with infinite (suffixes) of computations that never leave the component, we modify the graph G_C to consider such runs. We introduce a new vertex $\perp$ that intuitively stand for "an infinite (suffix) of a computation in C, and an accepting run of $\mathscr{A}_\varphi$ on it". There is an edge from a configuration c to $\perp$, if there is an exists an infinite computation of C that begins in configuration c, never enters an exit state, and there exists an accepting run of $\mathscr{A}_\varphi$ on it. There are no edges from $\perp$.

The first part of the transition relation is

$$\delta_1(\langle q,b\rangle,C) = \bigvee_{\pi \in Path(q,\perp)} \bigwedge_{e \in E_C(\pi)} (s_C(e), \langle q(e), q'(e), s_0(e), 0\rangle)$$

Second, we have to deal with computation segments that end in a pending call. These types of computations are easily dealt with in terms of paths in G_C to a configuration in which the state is a call state. We would like to note two details. First, note that by the definition of an accepting run of an NWBA, the hierarchical symbols associated with pending calls must be from the set P_f. Second, note the difference between states of type $\langle q,0\rangle$ and type $\langle q,1\rangle$. In the $\langle q,0\rangle$ case there is no constraint that has to do with $\mathscr{A}_\varphi$'s accepting states. Therefore, the second part of the transition relation is

$$\delta_2(\langle q,0\rangle,C) = \bigvee_{\substack{s_C^k \in S_C,\\ q' \in Q,\\ p \in P_f}} \bigvee_{\pi \in Path(s_c^k,q,q',p)} \bigvee_{b \in \{0,1\}} ((k,\langle q,b\rangle) \wedge \bigwedge_{e \in E_C(\pi)} (s_C(e), \langle q(e), q'(e), s_0(e), 0\rangle))$$

In the $\langle q,1\rangle$, case an accepting state of $\mathscr{A}_\varphi$ must be visited, therefore the second part of the transition relation is

$$\delta_2(\langle q,1\rangle,C) = \bigvee_{\substack{s_C^k \in S_C,\\ q' \in Q,\\ p \in P_f}} \bigvee_{(\pi,f) \in Path_a(s_c^k,q,q',p)} \bigvee_{b \in \{0,1\}} ((k,\langle q,b\rangle) \wedge \bigwedge_{e \in E_C(\pi)} (s_C(e), \langle q(e), q'(e), s_0(e), f(e)\rangle))$$

We have to deal with suffixes of computation that contain infinitely many call to, and return from, other components. Such computations must contain a configuration that appears twice. A *ρ-path* in G_C is a path in G_C in which the last vertex is visited more then once along the path (intuitively, closing a cycle). The part of the path between the first and last occurrences of the last vertex is the *cycle*. As we require $\mathscr{A}_\varphi$'s run to accept, an accepting state from Q_f should be visited during a segment of a computation that correspond to an edge on the cycle. An *accepting ρ-path* is a path in which one of the edges along the cycle is accepting. There exists an accepting ρ-path iff there exists an accepting ρ-path of length at most $3|V_C|$ (a simple path to the cycle, and a cycle of length at most $2|V_C|$).

For $q, \in Q$ we denote by ρ-$Path(q)$ a set of pairs in which: (1) the first element π is a ρ-path of length at most $3V_C$ starting at (s_0,q); (2) the second element is a function f mapping the edges in π into $\{0,1\}$ such that: (1) exactly one edge is mapped to 1, this edge is on the cycle, and (2) if the edge mapped to 1

is a component edge then it is also an accepting edge. The third part of the transition relation is

$$\delta_3(\langle q,b\rangle,C) = \bigvee_{(\pi,f)\in \rho\text{-}Path(q)} \bigwedge_{e\in E_C(\pi)} (s_C(e),\langle q(e),q'(e),s_0(e),f(e)\rangle)$$

Finally, for a state $\langle q,b\rangle$ the transition relation is

$$\delta(\langle q,b\rangle,C) = \delta_1(\langle q,b\rangle,C) \vee \delta_2(\langle q,b\rangle,C) \vee \delta_3(\langle q,b\rangle,C)$$

This concludes the definition of the transition relation

Accepting states of $\mathscr{A}$: Finally, the set F of $\mathscr{A}$'s accepting states is the set $Q\times\{1\}$. Intuitively, in an accepting run tree of $\mathscr{A}$, each path is either finite, i.e. ends a nodes whose transition relation is **true**, or an infinite path of states that correspond to pending calls. For the run to be accepting, an accepting state must be visited infinitely often along such infinite path of pending calls. As we defined the accepting-states set to be $Q\times\{1\}$, an infinite path of pending calls is accepted iff in the run of $\mathscr{A}_\varphi$ visits an $\mathscr{A}_\varphi$ accepting state infinitely often. This concludes the main construction,

We now prove the correctness in several stages. First, we prove a claim regarding states of the form $\langle q,q',i,b\rangle$.

Claim 4.3 For a composition tree T, there exists a finite accepting run tree of $A^{\langle q,q',i,b\rangle}$ on T iff there exits a computation π of the composition induced by T, such that:

1. π ends by returning from the i-th return state s^i_R of T's root.
2. there exists a run r of $\mathscr{A}^q_\varphi$ on the word induced by π that ends in q'.

Furthermore, for states $\langle q,q',i,1\rangle$ the iff statement is true for a run r that visits an accepting state from Q_f. □

Proof: Assume first that there exist computation π and run r as claimed. We prove that there exists a finite accepting run tree of $\mathscr{A}^{\langle q,q',i,b\rangle}$ on T. As the computation π returns from the root, the depth h of the subtree traversed by π in T is bounded. The proof is by induction on the depth h. The base case is a depth 1, i.e., only the root component is traversed. Then, the existence π implies there exists an edge in G_C from $\langle s_0,q\rangle$ to $\langle s^i_R,q'\rangle$. Therefore, there exists a path in G_C, between these vertexes, that does not contain any call edges. Thus, the transition relation evaluates to **true**, implying that there exists a finite accepting run of $\mathscr{A}^{\langle q,q',i,0\rangle}$ on T. Furthermore, if r visits a state from Q_f then the relevant edge is an accepting edge and there is an accepting run of $\mathscr{A}^{\langle q,q',i,1\rangle}$ on T. Assume now, the induction hypothesis for traversal of maximal depth h, we prove it for traversals of maximal depth $h+1$. The computation π can be broken into segments in which the control is in the root component and segments in which some other (called) components are in control. Each segment corresponds to an edge of G_C, where segments of computation in which the root is in control, correspond to component edges, and the rest of the segments correspond to call edges. Each call edge, correspond to a successor of the root in the composition tree, and for each call edge, the induction hypothesis implies the existence of accepting run tree on the corresponding composition subtree. Thus there exists an accepting run tree as claimed. Furthermore, if r visits Q_f then the visit is made during some computation segment. The edge corresponding to that computation segment can be mapped to 1 by the function f from the definition of the transition relation for $\langle q,q',i,1\rangle$. It follows that if r visits a state from Q_f then there exists a an accepting run of $\mathscr{A}^{\langle q,q',i,1\rangle}$ on T.

Assume now a finite accepting run tree of $\mathscr{A}^{\langle q,q',i,b\rangle}$ exists, we prove the existence of a computation π and a run r as needed. The proof is by induction on the height h of the accepting run tree. The base case is a run tree of height 1. Then, the transition relation δ must evaluate to **true** on the root. Thus, the path in G_C contains no call edges, and therefore by the definition of δ there exist π, and r as claimed.

Furthermore, if $b = 1$, the component edge must be an accepting edge implying that r visits Q_f. Assume now, the induction hypothesis for run trees of height h, we prove it for run trees of height $h+1$. The run-tree root is labeled by some set S of pairs of directions and $\mathscr{A}$-states that satisfy δ. This choice of states and directions corresponds to a path in G_C, in which some edges are call edges and some are component edges. By the definition of δ there exist computation segments corresponding to component edges, and by the induction hypothesis there exist computation segments corresponding to call edges. Splicing these computation segments together we get the a computation π, and r as claimed. Furthermore, if $b = 1$ then one of the edges is an accepting edge and therefore, r visits Q_f. □

By very similar reasoning, we can show that there exists a finite accepting run tree of $A^{\langle q,b\rangle}$ on a composition tree T, iff there exists a computation π of T such that: (1) π's traversal is bounded in a finite subtree of the composition tree ; (2) π never returns from the root of T; and (3) there exists an accepting run r of $\mathscr{A}_\varphi$ on π. Unlike, the $\langle q,q',i,b\rangle$, however, we have also to consider runs that are not bounded in a finite subtree of T. Next, we show that it is enough to consider computations that make infinitely many pending calls.

Observation 4.4 *For a library L, an NWBA $\mathscr{A}_\varphi$ and a composition tree T if there exists a computation π of T, in $L(\mathscr{A}_\varphi)$, in which a node $v \in T$ is visited infinitely often then there exists computation π' of T, in $L(\mathscr{A}_\varphi)$, that only traverses a finite subtree of T.*

Proof: First, note that it is enough to show that there exists a computation π' of T, in $L(\mathscr{A}_\varphi)$, such that π' only traverses a finite subtree of the subtree rooted at v (regardless of what happens outside that subtree). The reason is w.l.o.g. v can be assumed to be a node of minimal depth that is visited infinitely often by π. As such, the computation must eventually remain in the subtree rooted at v (since if v is not the root, v's predecessor is visited only finitely often).

Next, let π_1, π_2 be two computation segments of π, and r_1, r_2 be the corresponding parts of $\mathscr{A}_\varphi$'s accepting run on π such that:

1. π_1 and π_2 begin by entering the same call state s_C^i.
2. r_1 and r_2 begin by the same $\mathscr{A}_\varphi$ state q.
3. π_1 and π_2 end when the control is returned to v by the same re-entry state s_e^j.
4. r_1 and r_2 end in the same $\mathscr{A}_\varphi$ state q'.
5. r_1 visits Q_f iff r_2 visits Q_f.

Then, π_1 and π_2 are interchangeable while the resulting computation still satisfies φ. Thus, while v is returned to infinitely often, there are only finitely many equivalence class of interchangeable computation segments. Choosing a single representative from each equivalence class, we can splice a computation whose traversal depth is bounded by the traversal depths of the representatives. □

Observation 4.4 implies that if there is a computation, in $L(\mathscr{A}_\varphi)$, that traverses an unbounded subtree of the composition tree, and does not make infinitely many pending calls, then there is also a computation, in $L(\mathscr{A}_\varphi)$, that traverses a finite subtree of the composition tree. Therefore, when considering computations that traverse an unbounded depth subtree of a composition tree, it is enough to consider compositions in which the computation, whose word is in $L(\mathscr{A}_\varphi)$, has infinitely many pending calls. The definition of $\mathscr{A}$'s accepting states set ensures correctness with respect to such computations. An accepting run tree, of $\mathscr{A}_\varphi$ on T, with an infinite path, must visit infinitely often an accepting state (i.e., a state $\langle q,1\rangle$) which means it is possible to construct a computation of T that makes infinitely many pending calls, and on which $\mathscr{A}_\varphi$ would have an accepting run. On the other hand, an accepting computation of $\mathscr{A}_\varphi$ that makes infinitely many pending calls, implies the existence of an accepting run tree of $\mathscr{A}$, with an infinite path that visits an accepting state infinitely often.

Finally we provide a complexity analysis. For a NWBA $\mathscr{A}_\varphi$, with $n_{\mathscr{A}_\varphi}$ states, and a library $\mathscr{L}$ with m_L components in which the components are of size m_C, the construction presented here, creates an ABT $\mathscr{A}$ with at most $O(n^2_{\mathscr{A}_\varphi} \cdot m_C)$ states. Note, however, that the number of states does not tell the entire story. First, the computation of δ involves reachability analysis of the components. Luckily, the reachability analysis is done separately on each component (in fact, the Cartesian product of each component with $\mathscr{A}_\varphi$) and therefore the complexity is $O(n_{\mathscr{A}_\varphi} \cdot m_C \cdot m_L)$. On the other hand, $\mathscr{A}$ is an alternating automaton with a transition relation that may be exponential in the size of the its state space. Thus, $\mathscr{A}$ can *not* be computed in space polynomial in the parameters. The computation of $\mathscr{A}$ involves an analysis of the paths in G_C and requires space polynomial in $n_{\mathscr{A}_\varphi}$ and m_C.

5 Discussion

We defined the problem of NWTL synthesis from library of recursive components, solved it, and shown it to be 2EXPTIME-complete. We now note that the ideas presented above are quite robust with respect to possible variants of the basic problem.

The model was chosen for simplicity rather than expressiveness, and can be extended and generalized. First, we can consider several call values per component. This translates to each component having a set $S_0 \subseteq S$ of initial states (rather than a single initial state $s_0 \in S$). Next, we can add greater flexibility with respect to return values. A single return value may have different meanings on different calls. Therefore, compositions might be allowed to perform some "return-value translation"; matching return states to re-entry states per call, rather than matching return states to re-entry states uniformly. This can be modeled by augmenting each composition element $\langle i, C_i, f_i \rangle$ by another function $r_i : S_C \to ([n_R] \to s_e^R)$ that maps each call state into a matching of return values to re-entry states. The synthesis algorithm, for the augmented model, remains almost the same. In the augmented model, a component implementation depends on the call value $s_0 \in S_0$ and the r_i function. Therefore, instead of working with composition trees, labeleded by $\mathscr{L}$, we'd work with augmented composition trees, labeled by tuples $\langle C, s_0, r_i \rangle$. Our algorithm and analysis can then be extended appropriately.

Another possible extension might be to consider bounded call stacks. Theoretically, "call and return" models allow for unbounded call stacks. Real life systems, however, have bounded call stacks. One can consider a variant of the synthesis problem, in which the output must have bounded call stack, where the bound is an output of the synthesis algorithm, rather then an apriori given input. To adapt the algorithm to this case, we have to find a finite composition tree in which all computations satisfy φ, as well as no computation makes a call from a leaf (ensuring bounded stack). To that end, we construct two alternating automata on finite trees (AFTs). First, an AFT $\mathscr{A}_1$ for finite composition trees in which there exists a computation violating φ. The AFT $\mathscr{A}_1$ is simply the ABT from Theorem 4.1, when considered as an AFT, and in which no state is considered accepting. In addition, we construct an AFT $\mathscr{A}_2$ that accepts trees that may perform a call from one of the leaves (see longer version of this paper.) The union of the languages of $\mathscr{A}_1$ and $\mathscr{A}_2$ contain all finite composition trees that do not realize φ. An AFT for the union can then be complemented and checked for emptiness as in the infinite case. Thus, the solution techniques presented in this paper are quite robust and extend to natural variants of the basic model.

Acknowledgements Work supported in part by NSF grants CCF-0728882, and CNS 1049862, by BSF grant 9800096, and by gift from Intel.

References

[1] L. de Alfaro & T.A. Henzinger (2005): *Interface-based design*. In M. Broy, J. Grünbauer, D. Harel & C.A.R. Hoare, editors: *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, Springer, pp. 83–104.

[2] R. Alur (2007): *Marrying words and trees*. In: *Proc. 26th ACM Symp. on Principles of Database Systems*, pp. 233–242, doi:10.1007/978-3-540-74510-5_3.

[3] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman & L. Libkin (2008): *First-Order and Temporal Logics for Nested Words*. *Logical Methods in Computer Science* 4(4).

[4] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps & M. Yannakakis (2005): *Analysis of recursive state machines*. *ACM Transactions on Programming Languages and Systems* 27(4), pp. 786–818, doi:10.1145/1075382.1075387.

[5] R. Alur, K. Etessami & P. Madhusudan (2004): *A temporal logic of nested calls and returns*. In: *Proc. 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science* 2725, Springer, pp. 67–79.

[6] R. Alur & P. Madhusudan (2009): *Adding nesting structure to words*. *Journal of the ACM* 56(3), pp. 1–43, doi:10.1007/11779148_1.

[7] T. Ball, B. Cook, V. Levin & S.K. Rajamani (2004): *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft*. In: *Integrated Formal Methods*, pp. 1–20, doi:10.1007/978-3-540-24756-2_1.

[8] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini & M. Mecella (2003): *Automatic Composition of E-services That Export Their Behavior*. In: *ICSOC*, pp. 43–58, doi:10.1007/978-3-540-24593-3_4.

[9] G.J. Holzmann (1997): *The Model Checker SPIN*. *IEEE Transactions on Software Engineering* 23(5), pp. 279–295.

[10] O. Kupferman, Y. Lustig, M.Y. Vardi & M. Yannakakis (2011): *Temporal Synthesis for Bounded Systems and Environments*. In: *Proc. 28th Symp. on Theoretical Aspects of Computer Science*, pp. 615–626.

[11] O. Kupferman & M.Y. Vardi (2005): *Safraless Decision Procedures*. In: *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pp. 531–540, doi:10.1109/SFCS.2005.66.

[12] Y. Lustig & Moshe Y. Vardi (2009): *Synthesis from Component Libraries*. In: *Proc. 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS), Lecture Notes in Computer Science* 5504, Springer, pp. 395 – 409, doi:10.1007/978-3-642-00596-1_28.

[13] N. Piterman & M. Vardi (2001): *From Bidirectionality to Alternation*. In: *26th Int. Symp. on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science* 2136, Springer, pp. 598–609, doi:10.1016/S0304-3975(02)00410-3.

[14] A. Pnueli & R. Rosner (1989): *On the Synthesis of a Reactive Module*. In: *Proc. 16th ACM Symp. on Principles of Programming Languages*, pp. 179–190.

[15] S. Sardiña, F. Patrizi & G. De Giacomo (2007): *Automatic Synthesis of a Global Behavior from Multiple Distributed Behaviors*. In: *AAAI*, pp. 1063–1069.

[16] J. Sifakis (2005): *A Framework for Component-based Construction Extended Abstract*. In: *Proc. 3rd Int. Conf. on Software Engineering and Formal Methods*, IEEE Computer Society, pp. 293–300, doi:10.1109/SEFM.2005.3.

[17] M.Y. Vardi (1998): *Reasoning about the past with two-way automata*. In: *Proc. 25th Int. Colloq. on Automata, Languages, and Programming, Lecture Notes in Computer Science* 1443, Springer, Berlin, pp. 628–641.

[18] M.Y. Vardi & P. Wolper (1994): *Reasoning about Infinite Computations*. *Information and Computation* 115(1), pp. 1–37.