Synthesis from Component Libraries*

Yoad Lustig** and Moshe Y. Vardi

Rice University, Department of Computer Science, Houston, TX 77251-1892, USA
{yoad,vardi}@cs.rice.edu
http://www.cs.rice.edu/~yoad,
http://www.cs.rice.edu/~vardi

Abstract. Synthesis is the automated construction of a system from its specification. In the classical temporal synthesis algorithms, it is always assumed 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 system, either in hardware or in 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 this work we define and study the problem of LTL synthesis from libraries of reusable components. We define two notions of composition: data-flow composition, for which we prove the problem is undecidable, and control-flow composition, for which we prove the problem is 2EXPTIME-complete. As a side benefit we derive an explicit characterization of the information needed by the synthesizer on the underlying components. This characterization can be used as a specification formalism between component providers and integrators.

1 Introduction

The design of almost every non-trivial commercial system, either hardware or software system, involves many sub-systems each dealing with different engineering aspects and each requiring different expertise. For example, a software application for an email client contains sub-systems for managing graphic user interface and sub-systems for managing network connections (as well as many other sub-systems). In practice, the developer of a commercial product rarely develops all the required sub-systems himself. Instead, many sub-systems can be acquired as collections of reusable components that can be integrated into the system. We refer to a collection of reusable components as a *library*.[1]

* Work supported in part by NSF grants CCR-0124077, CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by gift from Intel.

** Part of this research was done while this author was at the Hebrew University in Jerusalem.

[1] In the software industry, every collection of reusable components is referred to as a "library". In the hardware industry, the term "library" is sometimes reserved for collections of components of basic functionality (e.g., logical *and*-gates with fan-in 4), while reusable components with higher functionality (e.g., an ARM CPU) are sometimes referred to by other names (such as IP cores). In this paper we refer to any collection of reusable components as a library, regardless of the level of functionality.

L. de Alfaro (Ed.): FOSSACS 2009, LNCS 5504, pp. 395–409, 2009.

The exact nature of the reusable components in a library may differ. The literature suggest many different types of components. For example: IP cores (in hardware), function libraries (for procedural programming languages), object libraries (for object oriented programming languages), and aspect libraries (for aspect oriented programming languages). Web-services can also be viewed as reusable components used by an orchestrator.

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 construction. The first formulation of synthesis goes back to Church [1]; the modern approach to that problem was initiated by Pnueli and Rosner who introduced LTL (linear temporal logic) synthesis [2]. In LTL synthesis the specification is given in LTL and the system constructed is a finite-state transducer modeling a reactive system.

In the work of Pnueli and Rosner, and in the many works that followed, 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 system is constructed using libraries of reusable components. In fact, in many cases the use of reusable components is essential. This is the case when a system is granted access to a reusable component, while the component itself is not part of the system. For example, a software system can be given access to a hard-disk device driver (provided by the operating system), and a web-based system might orchestrate web services to which it has access, but has no control of. Even when it is theoretically possible to design a sub-system from scratch, many times it is desirable to use reusable components. The use of reusable components allows to abstract away most of the detailed behavior of the sub-system, and write a specification that mentions only the aspects of the sub-system relevant for the synthesis of the system at large.

We believe therefore, that one of the prerequisites of wide use of synthesis algorithms is support of synthesis from libraries. In this work, we define and study the problem of LTL synthesis from libraries of reusable components.

As a perquisite to the study of synthesis from libraries of reusable components, we have to define suitable models for the notions of reusable components and their composition. 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, etc. . As an example for the multitude of composition notions see [3], where Sifakis suggests an algebra of various composition forms.

In this work we approach the general problem by choosing two specific concrete notions of models and compositions, each corresponding to a natural facet of the problem. For components, we abstract away the precise details of the components and model a component as a *transducer*, i.e., a finite-state machine with outputs. Transducers constitute a canonical model for a reactive component, abstracting away internal architecture and focusing on modeling input/output behavior.

As for compositions, we define two notions of component composition. One relates to data-flow and is motivated by hardware, while the other relates to control-flow and is

motivated by software. We study synthesis from reusable components for these notions, and show that whether or not synthesis is computable depends crucially on the notion of composition.

The first composition notion, in Section 3, is *data-flow* composition, in which the outputs of a component are fed into the inputs of other components. In data-flow composition the synthesizer controls the flow of data from one component to the other. We prove that the problem of LTL synthesis from libraries is undecidable in the case of data-flow composition. In fact, we prove a stronger result. We prove that in the case of data-flow composition, the LTL synthesis from libraries is undecidable even if we restrict ourselves to pipeline architectures, where the output of one component is fed into the input of the next component. Furthermore, it is possible to fix either the formula to be synthesized, or the library of components, and the problem remains undecidable.

The second notion of composition we consider is *control-flow* composition, which is motivated by software and web services. In the software context, when a function is called, the function is given control over the machine. The computation proceeds under the control of the function until the function calls another function or returns. Therefore, it seems natural to consider components that gain and relinquish control over the computation. A control-flow component is a transducer in which some of the states are designated as final states. Intuitively, a control-flow component receives control when entering an initial state and relinquish control when entering a final state. Composing control-flow components amounts to deciding which component will resume control when the control is relinquished by the component that currently is in control.

Web-services orchestration is another context naturally modeled by control-flow composition. A system designer composes web services offered by other parties to form a new system (or service). When referring a user to another web service, the other service may need to interact with the user. Thus, the orchestrator effectively relinquishes control of the interaction with that user until the control is received back from the referred service. Web-services orchestration has been studied extensively in recent years [4,5,6]. In Subsection 1.1, we compare our framework to previously studied models.

We show that the problem of LTL synthesis from libraries in the case of control-flow composition is 2EXPTIME-complete. One of the side benefits of this result is an explicit characterization of the information needed by the synthesis algorithm about the underlying control-flow components. The synthesis algorithm does not have to know the entire structure of the component but rather needs some information regarding the reachable states of an automaton for the specification when it monitors a component's run (the technical details can be found in Section 4). This characterization can be used to define the interface between providers and integrators of components. On the one hand, a component provider such as a web service, can publish the relevant information to facilitate the component use. On the other hand, a system developer, can publish a specification for a needed component as part of a commercial tender or even as an interface with another development group within the same organization.

1.1 Related Work

The synthesis problem was first formulated by Church [1] and solved by Büchi and Landweber [7] and by Rabin [8]. We follow the LTL synthesis problem framework

presented by Pnueli and Rosner in [2,9]. We also incorporate ideas from Kupferman and Vardi [10], who suggested a way to work directly with a universal automata for the specification. In [11], Krishnamurthi and Fisler suggest an approach to aspect verification that inspired our approach to control-flow synthesis.

While the synthesis literature does not address the problem of incorporating reusable components, extensive work studies the construction of systems from components. Examples for important work on the subject can be found in Sifakis' work on component based-construction [3], and de Alfaro and Henzinger's work on "interface-based design" [12].

In addition to the work done on the subject by the formal verification community, much work has been done in field of web-services orchestration [4,5,6]. The web-services literature suggests several models for web services; the most relevant to this work is known as the "Roman model", presented in [5]. In the Roman model web services are modeled, as here, by finite-state machines. The abstraction level of the modeling, however, is significantly different. In the Roman model, every interaction with a web-service is abstracted away to a single action and no distinction is made between the inputs of the web service and the outputs of the web service.

In our framework, as in the synthesis literature, there is a distinction between output signals, which the component controls, and input signals, which the component does not control. A system should be able to cope with any value of an input signal, while the output signals can be set to desired values [2]. Therefore, the distinction is critical as the quantification structure on input and output signals is different (see [2] for details). In the Roman model, since no distinction between inputs and outputs is made, the abstraction level of the modeling *must* leave each interaction abstracted as a single atomic action. The Roman model is suitable in cases in which all that is needed to ensure is the availability of web-services actions when these are needed. Many times, however, such high level of abstraction cannot suffice for complete specification of a system.

2 Preliminaries

For a natural number n, we denote the set $\{1, \ldots, n\}$ by $[n]$. For an alphabet Σ, we denote by Σ^* the set of finite words over Σ, by Σ^ω the set of infinite words over Σ, and by Σ^∞ the union $\Sigma^* \cup \Sigma^\omega$.

Given a set D of directions, a *D-tree* is a set $T \subseteq D^*$ such that if $x \cdot c \in T$, where $x \in D^*$ and $c \in D$, then also $x \in T$. For every $x \in T$, the words $x \cdot c$, for $c \in D$, are the *successors* of x. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $c \in D$ such that $x \cdot c \in \pi$. The *full* D-tree is D^*. Given an alphabet Σ, a *Σ-labeled D-tree* is a pair $\langle T, \tau \rangle$ where T is a tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ.

A *transducer*, (also known as a Moore machine [13]) is an deterministic finite automaton with outputs. Formally, a transducer is tuple $\mathcal{T} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, F, L \rangle$ where: Σ_I is a finite input alphabet, Σ_O is a finite output alphabet, Q is a finite 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 labelling states with output letters. For a transducer $\mathcal{T}$ and an input word $w = w_1 w_2 \ldots w_n \in \Sigma_I^n$, a *run*, or a *computation*

of $\mathcal{T}$ on w is a sequence of states $r = r_0, r_1, \ldots r_n \in Q^n$ such that $r_0 = q_0$ and for every $i \in [n]$ we have $r_i = \delta(r_{i-1}, w_i)$. The *trace* $tr(r)$ of the run r is the word $u = u_1 u_2 \ldots u_n \in \Sigma_O^n$ where for each $i \in [n]$ we have $u_i = L(r_{i-1})$. The notions of run and trace are extended to infinite words in the natural way.

For a transducer $\mathcal{T}$, 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 $\mathcal{T} = \langle \Sigma_I, \Sigma, Q, q_0, \delta, L \rangle$ such that for every $w \in \Sigma_I^*$, we have $\tau(w) = L(\delta^*(w))$.

A transducer $\mathcal{T}$ outputs a letter for every input letter it reads. Therefore, for every input word $w_I \in \Sigma_I^\omega$, the transducer $\mathcal{T}$ induces a word $w \in (\Sigma_I \times \Sigma_O)^\omega$ that combines the input and output of $\mathcal{T}$. A transducer $\mathcal{T}$ *satisfies* an LTL formula φ if for every input word $w_i \in \Sigma_I^\omega$ the induced word $w \in (\Sigma_I \times \Sigma_O)^\omega$ satisfies φ.

For a set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **True** (an empty conjunction) and **False** (an empty disjunction). For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning **True** to elements in Y and assigning **False** to elements in $X \setminus Y$ makes θ true. An *alternating tree automaton* is $\mathcal{A} = \langle \Sigma, D, Q, q_{in}, \delta, \alpha \rangle$, where Σ is the input alphabet, D is a set of directions, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$ is a transition function, $q_{in} \in Q$ is an initial state, and α specifies the acceptance condition (a condition that defines a subset of Q^ω; we define several types of acceptance conditions below). For a state $q \in Q$, we denote by $\mathcal{A}^q$ the automaton $\langle \Sigma, D, Q, q, \delta, \alpha \rangle$ in which q is the initial state.

The alternating automaton $\mathcal{A}$ runs on Σ-labeled full D-trees. A *run* of $\mathcal{A}$ over a Σ-labeled D-tree $\langle T, \tau \rangle$ is a $(T \times Q)$-labeled $\mathbb{N}$-tree $\langle T_r, r \rangle$. Each node of T_r corresponds to a node of T. A node in T_r, labeled by (x, q), describes a copy of the automaton that reads the node x of T and visits the state q. Note that many nodes of T_r can correspond to the same node of T. The labels of a node and its successors have to satisfy the transition function. Formally, $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = \langle \varepsilon, q_{in} \rangle$.
2. Let $y \in T_r$ with $r(y) = \langle x, q \rangle$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S = \{(c_0, q_0), (c_1, q_1), \ldots, (c_{n-1}, q_{n-1})\} \subseteq D \times Q$, such that S satisfies θ, and for all $0 \leq i \leq n-1$, we have $y \cdot i \in T_r$ and $r(y \cdot i) = \langle x \cdot c_i, q_i \rangle$.

A run $\langle T_r, r \rangle$ is accepting if all its infinite paths satisfy the acceptance condition. Given a run $\langle T_r, r \rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in T \times \{q\}$. We consider *Büchi* acceptance in which a path π is accepting iff $inf(\pi) \cap \alpha \neq \emptyset$, and *co-Büchi* acceptance in which a path π is accepting iff $inf(\pi) \cap \alpha = \emptyset$. An automaton accepts a tree iff there exists a run that accepts it. We denote by $L(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts.

The alternating automaton $\mathcal{A}$ is *nondeterministic* if for all the formulas that appear in δ, if (c_1, q_1) and (c_2, q_2) are conjunctively related, then $c_1 \neq c_2$. (i.e., if the transition is rewritten in disjunctive normal form, there is at most one element of $\{c\} \times Q$, for each $c \in D$, in each disjunct). The automaton $\mathcal{A}$ is *universal* if all the formulas that

appear in δ are conjunctions of atoms in $D \times Q$, and $\mathcal{A}$ is *deterministic* if it is both nondeterministic and universal. The automaton $\mathcal{A}$ is a *word* automaton if $|D| = 1$.

We denote each of the different types of automata by three-letter acronyms in $\{D, N, U\} \times \{B, C\} \times \{W, T\}$, where the first letter describes the branching mode of the automaton (deterministic, nondeterministic, or universal), the second letter describes the acceptance condition (Büchi or co-Büchi), and the third letter describes the object over which the automaton runs (words or trees). For example, NBT are nondeterministic tree automata and UCW are universal co-Büchi word automata.

Let I be a set of input signals and O be a set of output signals. For a 2^O-labeled full-2^I tree $T = \langle (2^I)^*, \tau \rangle$ we denote by T' the $2^{I \cup O}$-labeled full 2^I-tree in which ε is labeled by $\tau(\varepsilon)$ and for every $x \in (2^I)^*$ and $i \in 2^I$ the node $x \cdot i$ is labeled by $i \cup \tau(x \cdot i)$.

The LTL *realizability* problem is: given an LTL specification φ (with atomic propositions from $I \cup O$), decide whether there is a tree T such that the labelling of every path in T' satisfies φ. It was shown in [7] that if such a tree exists, then a regular such tree exists. The *synthesis* problem is to find the transducer inducing the tree if such a transducer exists [2].

3 Data-Flow Composition

Data-flow composition is the form of composition in which the outputs of a component are fed into other components as inputs. In the general case, each component might have several input and output channels, and these may be connected to various other components. For an exposition of general data-flow composition of transducers we refer the reader to [14]. In this paper, however, the main result is a negative result of undecidability. Therefore, we restrict ourselves to a very simple form of data-flow decomposition: the pipeline architecture. To that end, we model each component as a transducer with a single input channel and single output channel. The composition of such components form the structure of a pipeline. We prove that even for such limited form of data-flow composition the problem remains undecidable.

A *data-flow component*, is a transducer in which the set of final states plays no role. We denote such a component by $C = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, L \rangle$. For two data-flow components: $C^i = \langle \Sigma_I^i, \Sigma_O^i, Q^i, q_0^i, \delta^i, L^i \rangle$, $i = 1, 2$, where $\Sigma_O^1 \subseteq \Sigma_I^2$, the composition of C^1 and C^2 is the data-flow component $C^1 \circ C^2 = \langle \Sigma_I^1, \Sigma_O^2, Q^1 \times Q^2, \langle q_0^1, q_0^2 \rangle, \delta, L \rangle$ where $\delta(\langle q_1, q_2 \rangle, \sigma) = \langle \delta^1(q_1, \sigma), \delta^2(q_2, L^1(q_1)) \rangle$, and $L(\langle q_1, q_2 \rangle) = L^2(q_2)$. It is not hard to see that the trace of the composition on a word w is the same as the trace of the run of C^2 on the trace of the run of C^1 on w.

A library $\mathcal{L}$ of data-flow component is simply a set of data-flow components. Let $\mathcal{L} = \{C_i\}$ be a collection of data-flow components. A data-flow component C is a *pipeline composition* of $\mathcal{L}$-components if there exists $k \geq 1$ and $C_1, \ldots, C_k \in \mathcal{L}$, not necessarily distinct, such that $C = C_1 \circ C_2 \circ \ldots \circ C_k$. When the library $\mathcal{L}$ is clear from the context, we abuse notation and say that C is a pipeline.

The *data-flow library LTL realizability problem* is: Given a data-flow components library $\mathcal{L}$ and an LTL formula φ, is there a pipeline composition of $\mathcal{L}$-components that satisfies φ.

Theorem 1. *Data-flow library LTL realizability is undecidable.*[2] *Furthermore, the following hold:*

1. *There exists a library $\mathcal{L}$ such that the data-flow library LTL realizability problem with respect to $\mathcal{L}$ is undecidable.*
2. *There exists an LTL formula φ such that the data-flow library φ-realizability is undecidable.*

The standard way to prove undecidability of some machine model is to show that the model can simulate Turing machines. Had we allowed a more general way of composing transducers, for example, as in [14], such an approach, indeed, could have been used. Indeed, the undecidability proof technique in [16] can be cast as an undecidability result for data-flow library realizability, where the component transducers are allowed to communicate in a two-sided manner, each simulating a tape cell of a Turing machine. Here, however, we are considering a pipeline architecture, in which information can be passed only in one direction. Such an architecture seems unable to simulate a Turing machine. In fact, in the context of *distributed* LTL realizability, which is undecidable in general [9], the case of a pipeline architecrure is the decidable case [9].

Nevertheless, data-flow library LTL realizability is undecidable even for pipeline archirecture. We prove undecidability by leveraging an idea used in the undecidability proof in [9] for non-pipeline architectures. The idea is to show that our machine model, though not powerful enough to *simulate* Turing machines, is powerful enough to *check* computations of Turing machines. In this approach, the environment produces an input stream that is a candidate computation of a Turing machine, and the synthesized system checks this computation.

We now proceed with details. Let M be a Turing machine with an RE-hard language. We reduce the language of M to a data-flow library realizability problem. Given a word w, we construct a library of components $\mathcal{L}_w$ and a formula φ, such that φ is realizable by a pipeline of $\mathcal{L}_w$-components iff $w \in L(M)$.

The gist of the proof, is that given w, we construct a pipeline that checks whether its input is an encoding of an accepting computation of M on w. Each component in the pipeline is checking a single cell in M's tape. The detailed proof can be found in the full version below we present an overview of the proof.

Intuitively, the pipeline C checks whether its input is an encoding of an accepting computation of M on w. (To encode terminating computations by infinite words, simply iterate the last configuration indefinitely). The pipeline C produces the signal ok either if it succeeds to verify that the input is an accepting computation of M on w, or if the input is not a computation of M on w. That way, if $w \in L(M)$ then on every word ok is produced, while if $w \notin L(M)$ then on the computation of M on w, the signal ok is never produced.

The input to the transducer is an infinite word u over some alphabet Σ_{tape} defined below. Intuitively, u is broken into chunks where each chunk is assumed to encode a single configuration of M. The general strategy is that every component in the pipeline tracks a single tape cell, and has to verify that letters that are supposed to correspond to the content of the cell "behave properly" throughout the computation.

[2] It is not hard to see that the problem is computationally enumerable, since it is computationally possible to check whether a specific pipeline composition satisfies φ [15].

The input alphabet Σ_{tape} is used to encode configurations in a way that allows the verification of the computation. The content of one cell is either a letter from M's tape alphabet Γ, or both a letter and the state of M encoding the fact that the head of M is on the cell. Each letter in Σ_{tape} encodes the content of a cell and the content of its two adjacent cells. The reason for this encoding is that during a computation the content of a cell can be predicted by the content of the cell and its neighbors in the previous cycle. In addition to letters from Γ, we allow a special separator symbol $\#$ to separate between configurations. For simplicity, assume that all the configurations of the computation are encoded by words of the same length. (In the full version we deal with the general case.)

The library $\mathcal{L}_w$ contains only two types of components C_f and C_s. In the interesting pipelines a single C_f component is followed by one or more C_s components. Intuitively, each C_s component tracks one cell tape (e.g., the third cell) and checks whether the input encodes correctly the content of the tracked cell throughout the computation. The C_f component drives the C_s components.

The alphabet Σ_{tape} is the input alphabet of the C_f component. The output alphabet of C_f as well as the input and output alphabet of C_s is more complicated. We describe this alphabet as Cartesian product of several smaller alphabets. The first of these is Σ_{tape} itself, and both C_f and C_s produce each cycle the Σ_{tape} letter they read (thus the content is propagated through the pipeline).

In order to make sure each component tracks one specific cell (e.g., the third cell), we introduce another alphabet $\Sigma_{clock} = \{pulse, \neg pulse\}$. The components produces a pulse signal as follows: A C_f component produces $pulse$ one cycle after it sees a letter encoding a separator symbol #, and a C_s component produces a $pulse$ signal two cycles it reads a $pulse$. On other cycles $\neg pulse$ is produced. Note that one cycle delay is implied by the definition of transducers. Thus, a C_s component delays the pulse signal for one additional cycle. In the full version we show that this timing mechanism allows each C_s transducer to identify the letters that encode the content of "his" cell.

As for the tracking itself, the content of a tape cell (and the two adjacent cells) in one configuration contains the information needed to predict the content of the cell in the following configuration. Thus, whenever a clock $pulse$ signal is read, each C_s component compares the content of the cell being read to the expected content from the previous cycle in which $pulse$ was read. If the content is different from the expected content a special signal is sent. The special signal $junk$, sent is part of another alphabet $\Sigma_{junk} = \{junk, \neg junk\}$. When $junk$ is produced it is propagated throughout the pipeline. The C_f component is used to check the consistency of adjacent Σ_{tape} letters, as well as that the first configuration is an encoding of the initial configuration. If an inconsistency is found $junk$ is produced.

To discover accepting computations we introduce another signal, acc, that is produced by a C_s if M enters the accepting state and is propagated throughout the pipeline. Finally, we introduce the signal ok. A C_s component produces ok if it never saw M's head and either it reads $junk$ (i.e., the word does not encode a computation), or it reads acc (i.e., the encoded computation is accepting). Note that the signal ok is never produced if the pipeline is too short to verify the computation.

In the full version we prove the following: if $w \in L(M)$ then there exists a (long enough) pipeline in which ok is produced on every word, while if $w \notin L(M)$ then ok

is never produced (by any pipeline) on the word that encodes M's computation on w. The above shows Theorem 1 for the fixed formula Fok. The proof for a fixed library is similar.

4 Control-Flow Composition

In the case of software systems, another model of composition seems natural. In the software context, when a function is called, the function is given control over the machine. The computation proceeds under the control of the function until the function calls another function or returns. Therefore, in the software context, it seems natural to consider components that gain and relinquish control over the computation.

In our model, during a phase of the computation in which a component C is in control, the input-output behavior of the entire system is governed by the component. An intuitive example is calling a function from a GUI library. Once called, the function governs the interaction with user until it returns. Just as a function might call another function, a component might relinquish control at some point. In fact, there might be several ways in which a component might relinquish control (such as taking one out of several exit points).

The composition of such components amounts to deciding the flow of control. This means that the components have to be composed in a way that specifies which component receives control in what circumstances. Thus, the system synthesizer provides an interface for each component C, where the next component to receive control is specified for every exit point in C (e.g., if C exits in one way then control goes to C_2, if C exists in another way control goes to C_3, etc.). An intuitive example of such interface in real life would be a case statement on the various return values of a function f. In case f returns 1: call function g, in case f returns 2: call function h, and so on.[3]

Below we discuss a model in which the control is passed explicitly from one component to another, as in goto. A richer model would consider also control flow by calls and returns; we leave this to future work. In our model each component is modeled by a transducer and relinquishing control is modeled by entering a final state. The interface is modeled by a function mapping the various final states to other components in the system.

Let Σ_I be an input alphabet, Σ_O be an output alphabet and, $\Sigma = \Sigma_I \cup \Sigma_O$. A *control-flow component* is a transducer $M = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, F, L \rangle$. Unlike the data-flow component case, in control-flow components the set F of final states is important. Intuitively, control-flow components receives control when entering the initial state and relinquishes control when entering a final state. When a control-flow component is in control, the input-output interaction with the environment is done by the component. For that reason, control-flow components in a system (that interact with the same environment) must share input and output alphabets. A *control-flow components library*

[3] At first sight, it seems that the synthesizer is not given realistic leeway. In real life, systems are composed not only from reusable components but also from some code written by the system creator. This problem is only superficial, however, since one can add to the component library a set of components with the same functionality as the basic commands at the disposal of the system creator.

is a set of control-flow components that share the same input and output alphabets. We assume w.l.o.g. all the final sets in the library are of the same size n_F. We denote the final set of the i-th component in the library by $F_i = \{s^i_1, \ldots s^i_{n_F}\}$.

Next, we discuss a notion of composition suitable for control-flow components. When a component is in control the entire system behaves as the component and the system composition plays no role. The composition comes into play, however, when a component relinquishes control. Choosing the "next" component to be given control is the essence of the control-flow composition. A control-flow component relinquishes control by entering one of several final states. A suitable notion of composition should specify, for each of the final states, the next component the control will be given to. Thus, a control-flow composition is a sequence of components, each paired with an interface function that maps the various final states to other components in system. We refer to these pairs of a component coupled with an interface function as *interfaced component*. Note that a system synthesizer might choose to reuse a single component from the library several times, each with a different interface. Therefore, the number of interfaced components might differ from the number of components is the library. Formally, a *composition* of components from a control-flow components library $\mathcal{L}$ is a finite sequence of pairs $\langle C_1, f_1\rangle, \langle C_2, f_2\rangle, \ldots, \langle C_n, f_n\rangle$ where the first element in each pair is a control-flow component $C_i = \langle \Sigma_I, \Sigma_O, Q_i, q^i_0, \delta_i, F_i, L_i\rangle \in \mathcal{L}$ and the second element in each pair is an *interface function* $f_i : F_i \to \{1, \ldots, n\}$. Each of the pairs $\langle C_i, f_i\rangle$ is an *interfaced component*.

Intuitively, for an interfaced component $\langle C_i, f_i\rangle$, when C_i is in control and enters a final state $q \in F_i$, the control is passed to the interfaced component $\langle C_{f_i(q)}, f_{f_i(q)}\rangle$. Technically, this amounts to moving out of the state as if the state is not the final state q of C_i but rather the initial state $q_0^{f_i(q)}$ of $C_{f_i(q)}$. For technical reasons, we restrict every interface function $f_i : F_i \to \{1, \ldots, n\}$ in the composition to map every final state to a component whose initial state agrees with the final state on the labelling.[4] Thus, f_i is an an interface function if for every $j \leq |F_i|$ we have $L_i(s^i_j) = L_{f_i(s^i_j)}(q_0^{f_i(s^i_j)})$.

The fact that a control-flow component C might appear in more than one interfaced component means that each component in the composition can be referred to in more than one way: first, as the i-th component in the library, and second, as the component within the j-th interfaced component in the composition. To avoid confusion, whenever we refer to a component from the library (as in the i-th component from the library) we denote it by $M^i \in \mathcal{L}$, while whenever we refer to a component within an interfaced component in the composition (as in the component within the j-th interfaced component) we denote it by C_j. We denote by $type(j)$ the index, in the library, of the component C_j which is the component within the j-th interfaced component. Thus, C_i is the same reusable component as $M^{type(i)}$.

The *result* of the composition is the transducer $M = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, L\rangle$ where:

1. The state space Q is $\bigcup_{i=1}^{n}(Q_i \times \{i\})$.
2. The initial state q_0 is $\langle q^1_0, 1\rangle$.
3. The transition function δ is defined as follows:

[4] This restriction is only a technicality chosen to simplify notation in proofs.

(a) For a state $\langle q, i\rangle$ in which $q \in Q_i \setminus F_i$, we set $\delta(\langle q, i\rangle, \sigma) = \langle \delta_i(q, \sigma), i\rangle$.
(b) For $\langle q, i\rangle$, where $q \in F_i$ we set $\delta(\langle q, i\rangle, \sigma) = \delta_{f_i(q)}(\langle q_0^{f_i(q)}, f_i(q)\rangle, \sigma)$.

4. The labelling function L is defined $L(\langle q, i\rangle) = L_i(q)$.

The control-flow library LTL realizability problem is: Given a control-flow components library $\mathcal{L}$ and an LTL formula φ, decide whether there exists a composition of components from $\mathcal{L}$ that satisfies φ. The control-flow library LTL synthesis problem is similar, given a $\mathcal{L}$ and φ, find the composition realizing φ if one exists.

Theorem 2. *The control-flow library LTL synthesis problem can is 2EXPTIME-complete.*

Proof. For the lower bound, we reduce classical synthesis to control-flow library synthesis. Thus, a 2EXPTIME complexity lower bound follows from the classical synthesis lower bound [17]. We proceed to describe this reduction in detail.

As described earlier, the problem of classical synthesis is to construct a transducer such that for every sequence of input signals, the sequence of input and output signals induced by the transducer computation satisfies φ. The reduction from classical synthesis is simply to provide a library of control-flow components of basic functionality, such that every transducer can be composed out of this library.

An *atomic* transducer, is a transducer that has only an initial state and final states. Furthermore, every transition from the initial state enters a final state. Thus, in an atomic transducer we have state set $Q = \{q_0, q_1, \ldots, q_m\}$, where $m = |\Sigma_I|$, final state set $F = \{q_1, \ldots, q_m\}$, and transition function $\delta(q_0, a_i) = q_i$. The different atomic transducers differ only in their output function L.

Consider now the library of all possible atomic transducers. It is not hard to see that every transducer can be composed out of this library (where every state in the transducer has its own interfaced component in the composition). Therefore synthesis is possible from this library of atomic control-flow components iff synthesis is possible at all. This concludes the reduction.

We proceed to prove the upper bound. Before going into further details, we would like to give an overview of the proof and the ideas underlying it. The classical synthesis algorithm [2] considers execution trees. An execution tree is an infinite labelled tree where the structure of the tree represents the possible finite input sequences (i.e., for input signal set I the structure is $(2^I)^*$) and the labelling represents mapping of inputs to outputs. Every transducer induces an execution tree, and every regular execution tree can be implemented by a transducer. Thus, questions regarding transducers can be reduced to questions regarding execution trees. Questions regarding execution trees can be solved using tree automata. Specifically, it is possible to construct a tree automaton whose language is the set of execution trees in which the LTL formula is satisfied, and the realizability problem reduces to checking emptiness of this automaton.

Inspired by the approach described above, we should ask what is the equivalent of an execution tree in the case of control-flow components synthesis? Fixing a library $\mathcal{L}$ of components, we would like to have a type of labelled trees, representing compositions, such that every composition would induce a tree, and every regular tree would induce a composition. To that end, we define control-flow trees. Control-flow trees represent

the possible flows of control during computations of a composition. Thus, the structure of control flow trees is simply $[n_F]^*$, each node representing a finite sequence of control-flow choices. (Where each choice picks one final state from which the control is relinquished.) A control-flow tree is labelled by components from $\mathcal{L}$. Thus, a path in a control-flow tree represents the flow of control between components in the system. Note that a control-flow tree also implicitly encodes interface functions. For every node $v \in [n_F]^*$ in the tree, both v and v's sons are labelled by components from $\mathcal{L}$. We denote the labelling function by $\tau : [n_f]^* \to \mathcal{L}$. For a direction $d \in [n_F]$, the labelling $\tau(v \cdot d) \in \mathcal{L}$ of the son of v in direction d, implicitly the flow of control. (Formally, $\tau(v \cdot d)$ defines the component from $\mathcal{L}$, within the interfaced component, to which the control is passed.) Thus, a regular control-flow tree can be used to define a composition of control-flow components from $\mathcal{L}$.

Each path in a control-flow tree stands for many possible executions, all of which share the same control-flow. It is possible , however, to analyse the components in $\mathcal{L}$ and reason about the possible executions represented by a path in the control-flow tree. This allows us to construct a tree automaton that runs on control-flow trees and accept the control-flow trees in which all executions satisfy the specification. Once we have such tree automaton we can take the classical approach to synthesis.

An *infinite tree composition* $\langle [n_F]^*, \tau \rangle$ is an $[|\mathcal{L}|]$-labeled $[n_F]^*$-tree in which $\tau(\varepsilon) = 1$. Intuitively, an infinite tree composition represents possible flow of control in a composition. The root is labeled 1 since the run begins when C_1 is in control. The j-th successor of a node is labeled by $i \in |\mathcal{L}|$ if on arrival to the j-th final state, the control passed to M^i. Every finite composition $\langle C_1, f_1 \rangle, \langle C_2, f_2 \rangle, \ldots, \langle C_n, f_n \rangle$ can be unfolded to an infinite composition tree in the following way: $\tau(\varepsilon) = 1$, and for $x \in [n_F]^*$, and $i \in [n_F]$ we set $\tau(x \cdot i) = f_{\tau(x)}(s_i^{\tau(x)})$. In the proof we construct a tree automaton $\mathcal{A}$ that accepts the infinite tree compositions that satisfy φ. As we show below, if the language of $\mathcal{A}$ is empty then φ cannot be satisfied by any control-flow composition. If, on the other hand, the language of $\mathcal{A}$ is not empty, then there exists a regular tree in the language of $\mathcal{A}$, from which we can extract a finite composition.

The key to understanding the construction, is the observation that the effect of passing the control to a component is best analyzed in terms of the effect on an automaton for the specification. The specification φ has a UCW automaton $\mathcal{A}_\varphi = \langle \Sigma, Q_\varphi, q_0, \delta, \alpha \rangle$ that accepts exactly the words satisfying φ. To construct $\mathcal{A}_\varphi$ we construct an NBW $\mathcal{A}_{\neg\varphi}$ as in [18] and dualize it [10]. The effect of giving the control to a component M^i, with regard to satisfying φ, can be analyzed in terms of questions of the following type: assuming $\mathcal{A}_\varphi$ is in state q when the control is given to M^i, what possible states $\mathcal{A}_\varphi$ might be in when M^i relinquishes control by entering final state s, and whether $\mathcal{A}_\varphi$ visits an accepting state on the path from q to s.

Our first goal is to develop notation for the formalization of questions of the type presented above. For a finite word $w \in \Sigma$, we denote $\delta^*_\varphi(q, w) = \{q' \in Q_\varphi \mid$ there exists a run of $\mathcal{A}^q_\varphi$ on w that ends in $q'\}$. For $q \in Q_\varphi$ and $q' \in \delta^*_\varphi(q, w)$ we denote by $\alpha(q, w, q')$ the value of 1 if there exists a path in the run of $\mathcal{A}^q_\varphi$ on w that ends in q' and traverses through a state in α. Otherwise, $\alpha(q, w, q')$ is 0.

For a word $w \in \Sigma_I^*$ and a component $C = \langle \Sigma_I, \Sigma_O, Q_C, q_0^C, \delta_C, F_C, L_C \rangle$, we denote by $\delta^*_C(w)$ the state C reaches after running on w. We denote by $\Sigma(w, C)$ the word

from Σ induced by w and the run of C on w. For $w \in \Sigma_I^*$, we denote by $\delta^*_\varphi(q, C, w)$ the set $\delta^*_\varphi(q, \Sigma(w, C))$ and by $\alpha(q, w, q')$ the bit $\alpha(q, \Sigma(w, C), q')$. Finally, we define $e_C : Q_\varphi \times F_C \to 2^{Q_\varphi \times \{0,1\}}$ where $e_C(q, s) = \{\langle q', b\rangle \mid \exists w \in \Sigma_I \text{ s.t. } s = \delta^*_C(w) \text{ and } q' \in \delta^*_\varphi(q, C, w) \text{ and } b = \alpha(q, w, q')\}$. Thus, $\langle q', b\rangle$ is in $e_C(q, s)$ if there exists a word $w \in \Sigma_I^*$ such that when C is run on w it relinquishes control by entering s, and if at the start of C's run on w the state of $\mathcal{A}_\varphi$ is q then at the end of C's run the state of $\mathcal{A}_\varphi$ is q'. Furthermore, b is 1 iff there is a run of $\mathcal{A}^q_\varphi$ on $\Sigma(C, w)$ that ends in q' and traverses through an accepting state.

Note, that it also possible that for some component C and infinite input word $w \in \Sigma_I^\omega$ the component C never relinquish control when running on w. For an $\mathcal{A}_\varphi$-state $q \in Q_\varphi$, the component C is a *dead end* if there exists a word $w \in \Sigma_I^\omega$ on which C never enters a final state, and on which $\mathcal{A}^q_\varphi$ rejects $\Sigma(C, w)$.

Next, we define a UCT $\mathcal{A}$ whose language is the set of infinite tree compositions realizing φ.

Let $\mathcal{A} = \langle \mathcal{L}, Q, \Delta, \langle q_0, 1\rangle, \alpha\rangle$ where:

1. The state space Q is $(Q_\varphi \times \{0, 1\}) \cup \{q_{rej}\}$. Where q_{rej} is a new state (a rejecting sink).
2. The transition relation $\Delta : Q \times \mathcal{L} \to \mathcal{B}^+([n_F] \times Q)$ is defined as follows:
   (a) For every $C \in \mathcal{L}$ and $i \in [n_F]$ we have $\Delta(q_{rej}, C) = \bigwedge_{i \in [n_F]}(i, q_{rej})$.
   (b) For $\langle q, b\rangle \in Q_\varphi \times \{0, 1\}$ and $C \in \mathcal{L}$:
      – If C is a dead end for q, then $\Delta(\langle q, j\rangle, C) = \bigwedge_{i \in [n_F]}(i, q_{rej})$.
      – Otherwise, for every $j \in [n_F]$ the automaton $\mathcal{A}$ sends in direction j the states in $e_C(q, s^i_j)$. Formally, $\Delta(\langle q, b\rangle, C) = \bigwedge_{i \in [n_F]} \bigwedge_{\langle q_i, b_i\rangle \in e_C(q, s^i_j)} (i, \langle q_i, b_i\rangle)$.
3. The initial state is $\langle q_0, 1\rangle$ for q_0 the initial state of $\mathcal{A}_\varphi$.
4. The acceptance condition is $\alpha = \{q_{rej}\} \cup \{Q_\varphi \times \{1\}\}$.

Claim 3. $L(\mathcal{A})$ is empty iff φ is not realizable from $\mathcal{L}$ □

The proof can be found in the full version.

Note that while Claim 3 is phrased in terms realizability, the proof actually yields a stronger result. If the language of $\mathcal{A}$ is not empty, then one can extract a composition realizing φ from a regular tree in the language of $\mathcal{A}$. To solve the emptiness of $\mathcal{A}$ we transform it into an "emptiness equivalent" NBT $\mathcal{A}'$ by the method of [10]. By [10], the language of $\mathcal{A}'$ is empty iff the language of $\mathcal{A}$ is empty. Furthermore, if the language of $\mathcal{A}'$ is not empty, then the emptiness test yields a witness that is in the language of $\mathcal{A}$ (as well as the language of $\mathcal{A}'$). From the witness, which is a transducer labelling $[n_f]^*$ trees with components from $\mathcal{L}$, it is possible to extract a composition.

This concludes the proof of correctness for Theorem 2 and all that is left is the complexity analysis. The input to the problem is a library $\mathcal{L} = \{M^1, \ldots, M^{|\mathcal{L}|}\}$ and a specification φ. The number of states of the UCW $\mathcal{A}_\varphi$ is $2^{O(|\varphi|)}$. The automaton $\mathcal{A}_\varphi$ can be computed in space logarithmic in $2^{O(|\varphi|)}$ (i.e., space polynomial in $|\varphi|$). The main hurdle in computing the UCT $\mathcal{A}$ is computing the transitions by computing the e_C functions for the various components. For a component $M^i \in \mathcal{L}$, an $\mathcal{A}_\varphi$-state $q \in Q_\varphi$,

and a final state $s^i_j \in F_i$ the value of $e_C(q, s^i_j)$ can be computed by deciding emptiness of small variants of the product of $\mathcal{A}_\varphi$ and M^i. Thus, computing $e_C(q, s^i_j)$ is nondeterministic logspace in $2^{O(|\varphi|)} \cdot |M^i|$. The complexity of computing $\mathcal{A}$ is nondeterministic logspace in $2^{O(|\varphi|)} \cdot n_F \cdot (\sum_{i=1}^{|\mathcal{L}|} |M^i|)$. The number of states of $\mathcal{A}$ is twice the number of states of $\mathcal{A}_\varphi$, i.e. $2^{O(|\varphi|)}$, and does not depend on the library.

To solve the emptiness of $\mathcal{A}$ we use [10] to transform it into an "emptiness equivalent" NBT $\mathcal{A}'$. The size of $\mathcal{A}'$ is doubly exponential in $|\varphi|$ (specifically, $2^{2^{|\varphi|^2 \cdot \log(|\varphi|)}}$) and the complexity of its computation is polynomial time in the number of its states. Finally, the emptiness problem of an NBT can be solved in quadratic time (see [19]). Thus, the overall complexity of the problem is doubly exponential in $|\varphi|$ and polynomially dependent on the size of the library. □

An interesting side benefit the work presented so far, is the characterization of the information needed by the synthesis algorithm on the underlying components. The only dependence on a component C is by its corresponding e_C functions. Thus, given the e_C functions it is possible to perform synthesis without further knowledge of the component implementation. This suggest that the e_C functions can serve as a specification formalism between component providers and possible users.

5 Discussion

We defined two notions of component composition. Data-flow composition, for which we proved undecidability, and control-flow composition for which we provided a synthesis algorithm.

Control-flow composition required the synthesized system to be constructed only from the components in the library. In real life, system integrators usually add some code, or hardware circuitry, of their own in addition to the components used. The added code is not intended to replace the main functionality of the components, but rather allows greater flexibility in the integration of the components into a system. At first sight it might seem that our framework does not support adding such "integration code". This is not the case, as we now explain.

Recall, from the proof of Theorem 2, that LTL synthesis can be reduced to our framework by providing a library of atomic components. Every system can be constructed from atomic components. Thus, by including atomic components in our library, we enable the construction of integration code.

Note, however, that if *all* the atomic components are added to the input library, then the control-flow library LTL synthesis becomes classical LTL synthesis, as explained in the proof of Theorem 2. Fortunately, integration code typically supports functionality that can be directly manipulated by the system, as opposed to functionality that can only accessed through the components in the library. Therefore, it is possible to add to the input library only atomic components that manipulate signals in direct control of the system. This allows the control-flow library LTL synthesis of systems that contain integration code.

References

1. Church, A.: Logic, arithmetics, and automata. In: Proc. Int. Congress of Mathematicians, 1962, Institut Mittag-Leffler, pp. 23–35 (1963)
2. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)
3. Sifakis, J.: A framework for component-based construction extended abstract. In: Proc. 3rd Int. Conf. on Software Engineering and Formal Methods (SEFM 2005), pp. 293–300. IEEE Computer Society, Los Alamitos (2005)
4. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer, Heidelberg (2004)
5. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 43–58. Springer, Heidelberg (2003)
6. Sardiña, S., Patrizi, F., Giacomo, G.D.: Automatic synthesis of a global behavior from multiple distributed behaviors. In: AAAI, pp. 1063–1069 (2007)
7. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS 138, 295–311 (1969)
8. Rabin, M.: Automata on infinite objects and Church's problem. Amer. Mathematical Society (1972)
9. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. 31st IEEE Symp. on Foundations of Computer Science, pp. 746–757 (1990)
10. Kupferman, O., Vardi, M.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, pp. 531–540 (2005)
11. Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Transactions on Software Engineering Methods 16(2) (2007)
12. de Alfaro, L., Henzinger, T.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, C. (eds.) Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, pp. 83–104. Springer, Heidelberg (2005)
13. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
14. Nain, S., Vardi, M.Y.: Branching vs. Linear time: Semantical perspective. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 19–34. Springer, Heidelberg (2007)
15. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
16. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307–309 (1986)
17. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science (1992)
18. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)
19. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)