Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Synthesis from Component Libraries

Yoad Lustig and Moshe Y. Vardi[⋆]

Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.

Received: date / Revised version: date

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]

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

[⋆] 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.

[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.

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 [4–6]. 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 [4–6]. 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 $\Sigma^* \cup \Sigma^\omega$.

2.1 Linear temporal logic

Formulas of *linear-time propositional temporal logic* (LTL) are built from a set AP of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective $\mathbf{X}$ (next), and the binary temporal connective $\mathbf{U}$ (until) [13,14]. LTL is interpreted over *computations*. A computation is a function $\pi : \mathbb{N} \to 2^{AP}$, which assigns truth values to the elements of AP at each time instant (natural number). For a computation π and a point $i \in \mathbb{N}$, we have that:

- $\pi, i \models p$ for $p \in AP$ iff $p \in \pi(i)$.
- $\pi, i \models \xi \wedge \psi$ iff $\pi, i \models \xi$ and $\pi, i \models \psi$.
- $\pi, i \models \neg\phi$ iff not $\pi, i \models \phi$
- $\pi, i \models \mathbf{X}\,\phi$ iff $\pi, i+1 \models \phi$.
- $\pi, i \models \xi\,\mathbf{U}\,\psi$ iff for some $j \geq i$, we have $\pi, j \models \psi$ and for all k, $i \leq k < j$, we have $\pi, k \models \xi$.

Thus, the formula $\mathbf{true}\,\mathbf{U}\,\phi$, abbreviated as $\mathbf{F}\,\phi$, says that ϕ holds *eventually*, and the formula $\neg\,\mathbf{F}\,\neg\phi$, abbreviated $\mathbf{G}\,\phi$, says that ϕ holds *henceforth*. For example, the formula $\mathbf{G}\,(\neg\mathsf{request} \vee (\mathsf{request}\,\mathbf{U}\,\mathsf{grant}))$ says that whenever a request is made it holds continuously until it is eventually granted. We will say that π *satisfies* a formula ϕ, denoted $\pi \models \phi$, iff $\pi, 0 \models \phi$.

2.2 Alternating Tree Automata

A *tree* is a set $T \subseteq \mathbb{N}^*$ such that if $x \cdot c \in T$ where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, then also $x \in T$, and for all $0 \leq c' < c$, $x \cdot c' \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $x \in T$, the nodes $x \cdot c$ where $c \in \mathbb{N}$ are the *successors* of x. The number of successors of x is called the *degree* of x and is denoted by $d(x)$. A node is a *leaf* if it has no successors. 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 $c \in \mathbb{N}$ such that $x \cdot c \in \pi$. Given an alphabet Σ, a *Σ-labeled tree* is a pair $\langle T, V\rangle$ where T is a tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. Note that an infinite word in Σ^ω can be viewed as a Σ-labeled tree in which the degree of all nodes is 1. Of special interest to us are Σ-labeled trees in which $\Sigma = 2^{AP}$ for some set AP of atomic propositions. We call such Σ-labeled trees *computation trees*. Note that a computation tree can be viewed as a (possibly infinite) Kripke structure. Given a set $\mathcal{D} \subset \mathbb{N}$, a $\mathcal{D}$-tree is a computation tree in which all the nodes have degree in $\mathcal{D}$.

Automata over infinite trees (tree automata) run over Σ-labeled trees that have no leaves. *Alternating automata* generalize nondeterministic tree automata and were first introduced in [15]. For simplicity, we refer first to automata over binary trees (i.e., when $T = \{0,1\}^*$). Consider a nondeterministic tree automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F\rangle$. The transition relation $\delta : Q \times \Sigma \to 2^{Q^2}$ maps an automaton state $q \in Q$ and an input letter $\sigma \in \Sigma$ to a set of pairs of states. Each such pair suggests a nondeterministic choice for the automaton's next configuration. When the automaton is in a state q as it reads a node x labeled by a letter σ, it proceeds by first choosing a pair $\langle q_1, q_2\rangle \in \delta(q,\sigma)$, and then splitting into two copies. One copy enters the state q_1 and proceeds to the node $x \cdot 0$ (the left successor of x), and the other copy enters the state q_2 and proceeds to the node $x \cdot 1$ (the right successor of x).

For a given 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** and **false** and, as usual, $\wedge$ has precedence over $\vee$. 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 $X \setminus Y$ satisfies θ. We can represent the transition relation δ of a nondeterministic automaton on binary trees using $\mathcal{B}^+(\{0,1\} \times Q)$. For example, $\delta(q,\sigma) = \{\langle q_1, q_2\rangle, \langle q_3, q_1\rangle\}$ can be written as $\delta(q,\sigma) = (0,q_1) \wedge (1,q_2) \vee (0,q_3) \wedge (1,q_1)$, meaning that the automaton can choose between two possibilities. In the first, the copy that proceeds to direction 0 enters the state q_1 and the one that proceeds to direction 1 enters the state q_2. In the second, the copy that proceeds to direction 0 enters the state q_3 and the one that proceeds to direction 1 enters the state q_1.

In nondeterministic tree automata, each conjunction in δ has exactly one element associated with each direction. In alternating automata over binary trees, $\delta(q,\sigma)$ can be an arbitrary formula from $\mathcal{B}^+(\{0,1\} \times Q)$. We can have, for instance, a transition

$$\delta(q,\sigma) = (0,q_1) \wedge (0,q_2) \vee (0,q_2) \wedge (1,q_2) \wedge (1,q_3).$$

The above transition illustrates that several copies may go to the same direction and that the automaton is not required to send copies to all the directions. Formally, a finite alternating automaton over infinite binary trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F\rangle$ where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(\{0,1\} \times Q)$ is a transition function, $q_0 \in Q$ is an initial state, and F specifies the acceptance condition.

We now generalize alternating automata to D-trees for a finite set of directions D. The transition function is $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$. When the automaton is in a state q as it reads a node that is labeled by a letter σ and has $|D|$ successors, it applies the transition $\delta(q,\sigma)$. We define the *size* $\|\mathcal{A}\|$ of an automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F\rangle$ as $|Q| + |F| + \|\delta\|$, where $|Q|$ and $|F|$ are the respective cardinalities of the sets Q and F, and where $\|\delta\|$ is the sum of the lengths of the nonidentically false formulas that appear as $\delta(q,k)$ for some $q \in Q$ and $\sigma \in \Sigma$ (note that the restriction to nonidentically false formulas is to avoid an unnecessary $|Q| \cdot |\Sigma|$ minimal size for δ).

A run of an alternating automaton $\mathcal{A}$ over a tree $\langle T, V\rangle$, where T is the tree D^*, is a tree $\langle T_r, r\rangle$ in which the root is labeled by q_0 and every other node is labeled by an element of $T \times Q$. 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; in contrast, in a run of a nondeterministic automaton over $\langle T, V\rangle$ there is a one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r\rangle$ is a Σ_r-labeled tree where $\Sigma_r = T \times Q$ and $\langle T_r, r\rangle$ satisfies the following:

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

For example, if $\langle T, V\rangle$ is a binary tree with $V(\varepsilon) = a$ and $\delta(q_0, a) = ((0,q_1) \vee (0,q_2)) \wedge ((0,q_3) \vee (1,q_2))$, then, at level 1, $\langle T_r, r\rangle$ includes a node labeled $(0,q_1)$ or a node labeled $(0,q_2)$, and includes a node labeled $(0,q_3)$ or a node labeled $(1,q_2)$. Note that if, for some y, δ has the value **true**, then y need not have successors. Also, δ can never have the value **false** in a run.

A run $\langle T_r, r\rangle$ is accepting if all its infinite paths satisfy the acceptance condition. We consider here Büchi and co-Büchi acceptance conditions. 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 $V_r(y) \in T \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies a *Büchi* acceptance condition $F \subseteq Q$ if and only if $inf(\pi) \cap F \neq \emptyset$; it satisfies a *co-Büchi* acceptance condition $F \subseteq Q$, if and only if $inf(\pi) \cap F = \emptyset$

An automaton accepts a tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. Note that an alternating automaton over infinite words is simply an alternating automaton over infinite trees with $|D| = 1$. Formally, we define an alternating automaton over infinite words as $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F\rangle$ where $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$.

In [16], Muller et al. introduce *weak alternating automata* (WAAs). In a WAA, we have a Büchi acceptance condition $F \subseteq Q$ and there exists a partition of Q into disjoint sets, Q_i, such that for each set Q_i, either $Q_i \subseteq F$, in which case Q_i is an *accepting set*, or $Q_i \cap F = \emptyset$, in which case Q_i is a *rejecting set*. In addition, there exists a partial order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q,\sigma,k)$, for some $\sigma \in \Sigma$ and $k \in \mathcal{D}$, we have $Q_j \leq Q_i$. Thus, transitions from a state in Q_i lead to states in either the same Q_i or a lower one. It follows that every infinite path of a run of a WAA ultimately gets "trapped" within some Q_i. The path then satisfies the acceptance condition if and only if Q_i is an accepting set. Indeed, a run visits infinitely many states in F if and only if it gets trapped in an accepting set.

Note that an 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). An alternating automaton $\mathcal{A}$ is *universal* if all the formulas that appear in δ are conjunctions of atoms in $D \times Q$ (i.e., no disjunctions appear in the transitions). An alternating automaton $\mathcal{A}$ is *deterministic* if it is both nondeterministic and universal (i.e. no disjunctions appear in the transitions and in each transition there is at most one element of $\{c\} \times Q$, for each $c \in D$).

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.

2.3 *Transducers*

A *transducer*, (also known as a Moore machine [17]) is an deterministic finite automaton with outputs. Formally, a transducer is a 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 $L : Q \to$

Σ_O is an output function labelling states with output letters. For a transducer $\mathcal{T}$ and an input word $w = w_1 w_2 \dots w_n \in \Sigma_I^n$, a *run*, or a *computation* of $\mathcal{T}$ on w is a sequence of states $r = r_0, r_1, \dots r_n \in Q^{n+1}$ such that $r_0 = q_0$ and for every $i \in [n]$ we have $r_i = \delta(r_{i-1}, w_i)$. The *trace*, denoted $tr(r)$, of the run r is the word $u = u_1 u_2 \dots u_n \in \Sigma_O^n$ 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 transducer $\mathcal{T}$ induces the Σ_O-decorated Σ_I tree $\tau = \langle \Sigma_I^*, \delta^* \rangle$ in which every node w in the Σ_I^* tree, which is a word $w \in \Sigma_I^*$ is labeled by $\delta^*(w)$. A Σ_O-labeled Σ_I-tree $\langle \Sigma_I^*, \tau \rangle$ is *regular* if there exists a transducer $\mathcal{T} = \langle \Sigma_I, \Sigma, Q, q_0, \delta, L \rangle$ that induces it, i.e., 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.

2.4 Realizability and synthesis

Consider finite sets I and O of input and output signals, respectively. We model finite-state reactive systems with inputs in I and outputs in O by *transducers* with input alphabet $\Sigma_I = 2^I$ and output alphabet $\Sigma_O = 2^O$.

Let $\mathcal{T} = \langle 2^I, 2^O, Q, q_0, \delta, L \rangle$ be a transducer modeling such a reactive system, and consider an infinite sequence $w = i_0, i_1, i_2, i_3, \dots \in (2^I)^\omega$ of input letters. The computation of $\mathcal{T}$ on w, denoted $\mathcal{T}(w)$, is $\rho = (o_0 \cup i_0), (o_1 \cup i_1), (o_2 \cup i_2), \dots \in (2^{I \cup O})^\omega$ such that for all $j \geq 0$, we have $o_j = \delta^*(i_0 \cdot i_1 \cdots i_{j-1})$.

For an LTL specification φ over $I \cup O$ we would like to ask is there a reactive system $\mathcal{T}$ such that for every possible input sequence, the computation of $\mathcal{T}$ satisfies φ?

To define this problem in terms of trees and transducers, note that the full 2^I tree captures all possible input sequences. For a transducer $\mathcal{T} = \langle 2^I, 2^O, Q, q_0, \delta, L \rangle$, the output of the transducer on every possible input sequence is encoded in the Σ_O-labeled Σ_I-tree $\langle 2^I, \delta^* \rangle$ induced by $\mathcal{T}$. Technically, however, we would like to consider a labeled tree in which the labels capture both the inputs and the outputs. To that end we introduce a the following notion: 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 $\tau = \langle (2^I)^*, \tau \rangle$ we denote by $Dir(\tau)$ 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 exists a regular tree τ such that the labeling of every path in $Dir(\tau)$ 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

3.1 Data-flow definition

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 [18]. 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 forms 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 $C = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \emptyset, L \rangle$. For two data-flow components C^1, C^2, the composition of C_2 on C_1, denoted $C^1 \circ C^2$, intuitively captures running C^2 on the output of C^1. Formally, for two data-flow components: $C^i = \langle \Sigma_I^i, \Sigma_O^i, Q^i, q_0^i, \delta^i, \emptyset, L^i \rangle$, $i = 1, 2$, where $\Sigma_O^1 \subseteq \Sigma_I^2$, the composition of C^1 and C^2 is the data-flow component $C^1 \circ C^2$ where:

1. the input alphabets is Σ_I^1,
2. the output alphabet is Σ_O^2,
3. the set of states is $Q^1 \times Q^2$,
4. the initial state is $\langle q_0^1, q_0^2 \rangle$,
5. the transition relation is defined as: $\delta(\langle q_1, q_2 \rangle, \sigma) = \langle \delta^1(q_1, \sigma), \delta^2(q_2, L^1(q_1)) \rangle$, and
6. the output function is defined as $L(\langle q_1, q_2 \rangle) = L^2(q_2)$.

Intuitively, to run C^2 on the trace of C^1 one has to keep track of both C^1 state, which we denote by q^1, and C^2 state, which we denote by q^2. Whenever an input letter σ is read, q^1 is updated as if C^1 reads σ, and q^2 is updated as if C^2 reads C^1's output which is $L^1(q^1)$. The definition captures this intuition. 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 $C_1, \dots, C_k \in \mathcal{L}$, not necessarily distinct, such that $C = C_1 \circ C_2 \circ \dots \circ C_k$. 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 component library $\mathcal{L}$ and an LTL formula φ, is there a pipeline composition of $\mathcal{L}$-components that satisfies φ.

3.2 Example

For example, let C_1 be the transducer $\langle \Sigma_I^1, \Sigma_O^1, Q^1, q_0^1, \delta^1, \emptyset, L^1\rangle$ where:

1. $\Sigma_I^1 = \{a_1, b_1\}$.
2. $\Sigma_O^1 = \{a_2, b_2\}$.
3. $Q^1 = \{s_0, s_1\}$.
4. q_0^1 is s_0.
5. The transitions are:
   - $\forall q \in Q^1\ \delta^1(q, a_1) = s_0$, and
   - $\forall q \in Q^1\ \delta^1(q, b_1) = s_1$.
6. $L^1(s_0) = b_2$ and $L^1(s_1) = a_2$.

Intuitively, C^1 "flips" it's output (between a's and b's). Formally, C^1 realizes the specification $\mathbf{G}\ [(a_1 \rightarrow \mathbf{X}\, b_1) \wedge (b_1 \rightarrow \mathbf{X}\, a_1)]$.

Let C^2 be the transducer $\langle \Sigma_I^2, \Sigma_O^2, Q^2, q_0^2, \delta^2, \emptyset, L^2\rangle$ where:

1. $\Sigma_I^2 = \{a_2, b_2\}$.
2. $\Sigma_O^2 = \{a_3, b_3\}$.
3. $Q^2 = \{t_0, t_1\}$.
4. q_0^2 is t_0.
5. The transitions are:
   - $\delta^2(t_0, a_2) = t_0$,
   - $\delta^2(t_0, b_2) = t_1$, and
   - $\forall \sigma \in \{a_2, b_2\}\ \delta(t_1, \sigma) = s_1$.
6. $L(t_0) = a_3$ and $L(s_1) = b_3$.

Intuitively, C^2 treats b's as a "sticky bit", once a b is seen, C^2 continues to output b's. Formally, C^2 realize the specification $a_3 \wedge [(\neg a_3)\ \mathbf{U}\ (b_2 \wedge\ \mathbf{X}\ \ \mathbf{G}\, b_3)]$.

Having a transducer that intuitively flips a's and b's, and a transducer that intuitively treats b's as a sticky bit, we would like to build a composed system that outputs a's as long as it reads b's, and output b's from the point an a is read onwards. Formally, we would like a system that realizes: $a_3 \wedge [(\neg a_3)\ \mathbf{U}\ (a_1 \wedge\ \mathbf{X}\ \ \mathbf{G}\, b_3)]$.

The composition $C^1 \circ C^2$, is $C^\circ = \langle \Sigma_I^\circ, \Sigma_O^\circ, Q^\circ, q_0^\circ, \delta^\circ, \emptyset, L^\circ\rangle$ where:

1. $\Sigma_I^\circ = \{a_1, b_1\}$.
2. $\Sigma_O^\circ = \{a_3, b_3\}$.
3. $Q^\circ = \{\langle s_0, t_0\rangle, \langle s_0, t_1\rangle, \langle s_1, t_0\rangle, \langle s_1, t_1\rangle\}$.
4. $q_0^\circ = \langle s_0, t_0\rangle$.
5. The transitions are:
   - $\delta^\circ(\langle s_0, t_0\rangle, a_1) = \langle \delta^1(s_0, a_1), \delta^2(t_0, b_2)\rangle = \langle s_0, t_1\rangle$,
   - $\delta^\circ(\langle s_0, t_0\rangle, b_1) = \langle \delta^1(s_0, b_1), \delta^2(t_0, b_2)\rangle = \langle s_1, t_0\rangle$,
   - $\delta^\circ(\langle s_0, t_1\rangle, a_1) = \langle \delta^1(s_0, a_1), \delta^2(t_1, b_2)\rangle = \langle s_0, t_1\rangle$,
   - $\delta^\circ(\langle s_0, t_1\rangle, b_1) = \langle \delta^1(s_0, b_1), \delta^2(t_1, b_2)\rangle = \langle s_1, t_1\rangle$,
   - $\delta^\circ(\langle s_1, t_0\rangle, a_1) = \langle \delta^1(s_1, a_1), \delta^2(t_0, a_2)\rangle = \langle s_0, t_1\rangle$,
   - $\delta^\circ(\langle s_1, t_0\rangle, b_1) = \langle \delta^1(s_1, b_1), \delta^2(t_0, a_2)\rangle = \langle s_1, t_0\rangle$,
   - $\delta^\circ(\langle s_1, t_1\rangle, a_1) = \langle \delta^1(s_1, a_1), \delta^2(t_1, a_2)\rangle = \langle s_0, t_1\rangle$, and
   - $\delta^\circ(\langle s_1, t_1\rangle, b_1) = \langle \delta^1(s_1, b_1), \delta^2(t_1, a_2)\rangle = \langle s_1, t_1\rangle$.
6. $\forall q \in Q^1\ L^\circ(\langle q, t_0\rangle) = a_3$ and $\forall q \in Q^1\ L^\circ(\langle q, t_1\rangle) = b_3$.

It is not hard to verify that $C^1 \circ C^2$ realizes $a_3 \wedge [(\neg a_3)\ \mathbf{U}\ (a_1 \wedge\ \mathbf{X}\ \ \mathbf{G}\, b_3)]$.

3.3 Data-flow synthesis

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.*

Proof: 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 [18], such an approach could have been used. Indeed, the undecidability proof technique in [20] 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 architecture. 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. Fix some Turing machine M with a computationally enumerable complete language $L(M)$ (e.g., a universal machine). We reduce $L(M)$ to the data-flow library LTL pipeline 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 $w \in L(M)$. Furthermore, the formula φ does not depend on the word w, it is the LTL formula eventually *ok* (i.e., $\mathbf{F}$ *ok*) where *ok* is proposition appearing in the construction.

A technical note, the data-flow components we construct read letters from the input alphabet and output letters from the output alphabet. Nevertheless, it is easier to think of the input and output alphabets as Cartesian products of smaller alphabets, and describe the components actions in terms of these smaller alphabets. We abuse notation in that way. We also sometimes refer to a "signal" rather than a "letter" the notions are interchangeable.

[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 [19].

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 every word is either an accepting computation or not a computation at all. In any case, C produces ok on every word. If, on the other hand, $w \notin L(M)$ the computation of M on w is a word on which the pipeline never produces ok.

The input to the transducer is an infinite word u over some alphabet Σ_{tape} we define 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.

A technical note: For now, assume that all the configurations of the computation are encoded by words of the same length. (This can only work if the computation of M on w uses only a bounded number of cells, which is always the case if $w \in L(M)$).

The library $\mathcal{L}_w$ contains only two types of components C_f and C_s. Below, it will become clear that the interesting pipelines are of the structure $C_f \circ C_s^+$ in which a single C_f component is followed by one or more C_s components.

We now turn to define the various alphabets used and explain their usage. The input alphabet Σ_{tape} is the input to the entire composition and is used to encode configurations in a way that allows the verification of the computation. We denote by Γ be the tape alphabet of M and by Q the state set of M. That way, a configuration of M can be encoded by a word over $\Gamma \cup (\Gamma \times Q)$. Introducing a separator symbol $\#$, a computation, as a sequence of configurations, can be encoded over $\Gamma \cup (\Gamma \times Q) \cup \{\#\}$. (See [17] for details of similar encodings.) In the following we abuse notation and regard the question whether the head is on a cell and if so, what is machine's state, as part of the cell's content.

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. To that end, each C_s needs to predict the content of the cell it tracks based on the cell's content in the previous configuration. In order to predict the contents of a cell in some configuration, one needs to know the content of the cell and the content of the two adjacent cells in the preceding configuration. We therefore use a redundant encoding in which a single letter actually encodes the contents of three cells, the cell in question, and its two adjacent cells. To that end we define $\Sigma_{tape} = (\Gamma \cup (\Gamma \times Q) \cup \{\#\})^3$. for As each Σ_{tape} letter encodes three cells, adjacent letters should to be checked for consistency, we take care of this point below, at this point assume the letters are consistent.

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 alphabets of C_s contain some other alphabets, defined below, in addition to Σ_{tape}. We can, at this stage, specify the behaviour of C_f and C_s on the Σ_{tape} letters read. Both C_f and C_s read a Σ_{tape} letter every cycle, and produce the same Σ_{tape} letter read (thus the content is propagated throughout the pipeline). Note, however, that due to the inherent delay implicit in the composition of transducers, at a given cycle each component reads the letter read by the predecessor component in the preceding cycle. The first letter produced by any component is always a letter encoding three blanks, hence denoted the blank letter. Thus, on the 8-th cycle C_f reads the 8-th letter read by the entire composition C, while the first C_s component reads the 7-th letter C read. Similarly, the second C_s reads the 6-th letter C read, and so on until the 7-th C_s reads the first letter C read. If C is longer, then the following C_s components read the blank letter as this is the first letter produced by all components.

For $n \geq 0$ we denote by C_s^n the composition $C_s \circ \ldots \circ C_s$ of n components of type C_s.

Observation 2 For a pipeline C of the structure $C_f \circ C_s^n$, an input word $\sigma = \sigma_1 \sigma_2 \ldots \in \Sigma_{tape}^{\omega}$, and $j \leq n$, the j-th component reads σ_i, the i-th letter read by C, on the $i + j - 1$ cycle of C's run. $\square$

This way of propagating the tape-contents means that each component sees (an encoding of) the content of all tape cells. 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 # (formally, a letter from $(\Gamma \cup (\Gamma \times Q) \cup \{\#\}) \times \{\#\} \times (\Gamma \cup (\Gamma \times Q) \cup \{\#\})$, hence denoted a separator letter). On other cycles, C_f produces $\neg pulse$.

A C_s component produces a $pulse$ signal two cycles it reads a $pulse$, and produces $\neg pulse$ on other cycles. 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.

Observation 3 For a pipeline C of the structure $C_f \circ C_s^n$, and $i \leq n$, the i-th C_s component reads a $pulse$ signal i cycles after it reads a separator letter. $\square$

Intuitively, each component assumes that the content of the cell it is tracking is read exactly at the cycles in which $pulse$ is read. Since for different components the $pulse$ signal coincides with different Σ_{tape} letters, we get that different components track different tape cells.

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 is sent over another alphabet $\Sigma_{junk} = \{junk, \neg junk\}$. The intuitive meaning of a $junk$ signal is that an inconsistency was found and that the word read by C is not an encoding of a computation. We would like every $junk$ signal to be propagated to the end of the pipeline. Therefore a C_s component produces $junk$ according to the following rules: First, if $junk$ is read then $junk$ is produced. In addition, if $junk$ is produced it is continued to be produced on all subsequent cycles. Otherwise, $junk$ is produced on cycles in which $pulse$ is read and the Σ_{tape} letter read is not compatible with the expected content (according to the Σ_{tape} letter read on the previous cycle in which $pulse$ was read). On all other cycles $\neg junk$ is produced.

The C_f component is used to check the two aspects of a computation encoding that are not checked by the C_s components. First, the consistency of adjacent Σ_{tape} letters. As each Σ_{tape} letter encodes three tape cells every two adjacent Σ_{tape} letters should agree on the content of the cells encoded by both. The second aspect is to check that the first configuration encodes an initial configuration of the computation of M on w[3]. Therefore, C_f produces Σ_{junk} letter according to the following rules: Every Σ_{tape} letter is compared to the preceding Σ_{tape} letter. If incompatibility is found then $junk$ is produced and is continued to be produced on all subsequent cycles.

In addition, at the beginning of the run the first configuration is checked to be an encoding of an initial configuration of M on w. Note that the initial configuration is defined uniquely up to the number of trailing blanks. Therefore C_f can check that the prefix of the input word, up to the first separator letter, encodes an initial configuration. Thus, a $junk$ is produced if the prefix of the input word is not an encoding of the initial configuration of M on w, or if at any cycle along the run, two adjacent Σ_{tape} letters are incompatible. If $junk$ is produced it is continued to be produced on all subsequent cycles. Otherwise, $\neg junk$ is produced.

Up to this point we developed mechanisms that allow a composition C of type $C_f \circ C_s^n$, for some $n \geq 1$, to check that the letters corresponding to the first n tape cells in the input word encode the prefixes of the configurations that appear in the computation. To formalize the intuition above we introduce a notion of a word agreeing-to-n with the (prefix) of the computation of M on w. For a word $u \in \Sigma_{tape}^{\omega}$, we look at u as a concatenation of separator letters and subwords in which no separator letter appears. Thus, $u = u_1\sigma_1u_2\sigma_2u_3\ldots$ where for every $i \geq 1$ the letter σ_i is separator letter, while u_i encodes a subword containing no separator letters. (Note that the sequence may be finite if u contains only finitely many separator letters.) Let $v \in \Sigma_{tape}^{\infty}$ be an encoding of a prefix of the computation of M on w. A word $u \in \Sigma_{tape}^{\infty}$ *agrees-to-n* with v if the n-long prefixes of the sequence $u_1, u_2\ldots$ encode n-long prefixes of the configurations in v. A pipeline $C = C_f \circ C_s^n$ checks only the first n cells of the tape. Therefore C can only verify computations that use n cells or less. A word $u \in \Sigma_{tape}^{\infty}$ is *n-bounded* if in all the separator-free subwords u_i all the letters encoding the location of the head appear before the n-th letter.

[3] Note, that this is the only dependence on w in the entire construction of $\mathcal{L}_w$.

Observation 4 Let $C = C_f \circ C_s^n$ be a pipeline. For an n-bounded input word $u \in \Sigma_{tape}^{\omega}$ we have:

1. If the u agrees-to-n with an n-bounded computation then $junk$ is never produced by C.
2. If u does not agree-to-n with any n-bounded computation, then when C is run on u it will eventually produce the signal $junk$ and continue to do so for the remainder of its computation.

□

Proof: The interesting case is when u does not agree-to-n with an n-bounded computation.

Then, u does not encode a computation of M. We denote $u = u_1\sigma_1u_2\ldots$ as above. Either u_1 does not encode the initial configuration, or there exists a minimal $i \geq 1$ for which the subword u_{i+1} does not encode the successor configuration of u_i.

In the first case, once the first letter diverging from the initial configuration is read, C_f will start producing the $junk$ signal and continue to do so indefinitely. The $junk$ signal will propagate to the end of the pipeline C which will hence produce $junk$.

Otherwise, u_{i+1} does not encode the successor configuration of u_i. In particular, there must exist a $j \leq n$ for which the j-th letter in u_{i+1} does not encode the content of the j-th cell in the successor configuration of u_i. By Observation 3, the j-th C_s will read a two consecutive $pulse$ signals exactly when reading the j-th letters in u_i and u_{i+1}. Therefore, assuming the Σ_{tape} letters are consistent, after reading the j-th letter in u_i the j-th C_s expects to see the content of the j-th cell in the successor configuration to u_i. Since the j-th letter read from u_{i+1} is different than the expectation, C_s will start producing $junk$ signals.

Finally, if our assumption that the Σ_{tape} letters are consistent fails, this will be discovered by C_f. In any case, C will eventually produce $junk$ signals and continue to do so indefinitely. □

To allow the pipeline to discover if it agrees with an *accepting* n-bounded computation we introduce another signal $\Sigma_{acc} = \{acc, \neg acc\}$. A component C_s produces acc if it detects the accepting state on the cell it is tracking, or if it reads acc from the preceding component. Thus, the signal acc is produced according to the following rules: C_s produces acc either if it reads acc, or if it reads a letter encoding the accepting state of M on the same cycle it reads $pulse$. If an acc is produced it is continued to be produced on all subsequent cycles. Otherwise, $\neg acc$ is produced. The first component C_f always produces $\neg acc$.

Observation 5 Let $C = C_f \circ C_s^n$ be a pipeline, and let u be an input word that agrees-to-n with an n-bounded computation. Then, the n-th C_s in C reads acc in a suffix of C's run, iff u agrees-to-n with an accepting computation. □

Finally, we introduce the signal $\Sigma_{ok} = \{ok, \neg ok\}$ and the rules for producing it. The C_f transducer always produces $\neg ok$. A C_s transducer produces ok if:

1. the same C_s transducer produces $junk$ or,
2. (a) the transducer never read both a $pulse$ and a letter indicating encoding the head's presence at the same cycle, and
   (b) the transducer reads acc.

Claim 6 If $w \in L(M)$ then there exists an $n \geq 0$ for which for every input word $u \in \Sigma_{tape}$ the run of $C = C_f \circ C_s^n$ on u satisfies $\mathbf{F}\ ok$. □

Proof: If $w \in L(M)$ then there is a bound on the number of cells used in the computation of M on w. Let n be bigger than that bound. Thus, the computation of M on w is n-bounded.

For an input word $u \in \Sigma_{tape}^n$, if u agrees-to-n with the accepting computation, then:

1. Since the computation is n-bounded the n-th C_s never reads both a $pulse$ and a letter indicating encoding the head's presence at the same cycle, and
2. by Observation 5 there exists a suffix of the run in which the n-th C_s reads acc.

Therefore there exists a suffix of the run in which C produces ok.

If, on the other hand, u does not agree-to-n with the accepting computation, look at the first prefix of u that does not agree-to-n with the accepting computation. Since the real computation is n-bounded, this prefix must still be n-bounded. Thus, by Observation 4 there exists a suffix of the run of C in which the n-th C_s reads $junk$ and therefore produces ok. □

Claim 7 If $w \notin L(M)$ then for every pipeline composition of L_w-components C, there exists a word $u \in \Sigma_{tape}$ such that the run of C on u does not satisfy $\mathbf{F}\ ok$. □

Proof: The first transducer must be C_f as it is the only transducer with the correct input alphabet. As the output alphabet of both C_f and C_s differ from Σ_{tape}, the only possible constructions are of the structure $C_f \circ C_s^n$ for $n \geq 0$. Furthermore, since C_f never produces ok then the composition C_f trivially satisfies the claim.

Let $n > 0$ be a number and $C = C_f \circ C_s^n$ be the pipeline composition. Let $u \in \Sigma_{tape}^\omega$ be a word encoding the computation of M on w, in which all configurations are encoded by words of length at least n (this is always possible as a configuration can always be padded by blanks).

Since u encodes the computation of M on w in particular it agrees-to-n with the computation. It is true that the computation might not be n-bounded, however, even in this case, u agrees-to-n with the prefix of the computation to the point in which the head of M reaches the n-th cell. Up to this point, ok is never produced by Observation 4. After this point (if such a point exists) the last C_s component will never produce ok since it saw the head of the machine. In any case, ok is never produced. □

It is not hard to see that (given M and w) both C_f and C_s can be as finite-state transducers. Therefore the library $\mathcal{L}_w = \{C_f, C_s\}$ and $\varphi = \mathbf{F}\ ok$ proves the undecidability of data-flow library LTL realizability problem. This concludes the proof of Theorem 1.

Note that the only use of the input word w in the proof of Theorem 1 is by making sure that the first component C_f checked the first configuration to be the initial configuration of M on w. Another way to go, would be to relieve C_f of this duty, and instead construct a formula ψ_w that is true only on words, in which the first configuration is the initial configuration. In this case, the specification should be $\varphi_w = \psi_w \rightarrow \mathbf{F}\ ok$.

In this approach, the library $\mathcal{L}$ does not depend on the word but specification formula does. □

4 Control-flow composition

4.1 Control-flow definitions

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.[4]

Below we discuss a model in which the control is passed explicitly from one component to another, as in goto. A richer

[4] 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.

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* 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. Note that a system synthesizer might choose to reuse a single component from the library several times, each with a different interface function that maps the various final states to other components in system. Thus, a control-flow composition is a set of "interfaced components", where each "interface component" contains information on both the library-component to be in control, and the interface function of which component receive control if and when the "interfaced component" relinquishes control. (The number of interfaced components might differ from the number of components is the library.)

Formally, a composition can be modeled as a transducer $\mathcal{C} = \langle [n_F], \mathcal{L}, [m], i_0, \rho, \emptyset, L_{\mathcal{C}}\rangle$ Where:

1. The input alphabet is $[n_F]$ (intuitively an input letter specifies through which final state the component in control relinquished control).
2. The output alphabet is the component library $\mathcal{L}$.
3. The set of states is $[m]$ for some $m > 0$ (intuitively each state stands for some "interfaced component").
4. The initial state is $i_0 \in [m]$ (intuitively the interfaced component that holds control at the beginning of the computation)
5. $\rho : [m] \times [n_F] \to [m]$ is a deterministic transition function mapping a state (standing for an "interfaced component") and an input letter (standing for a final state) into the next state (intuitively the "interfaced component" that should receive control).
6. $L_{\mathcal{C}} : [m] \to \mathcal{L}$ is a labeling function that maps states into components from the library (intuitively, mapping "interfaced components" into the components from the library they relate to).

Intuitively, the computation starts when the interfaced component i_0 is in control. For every $i \in [m]$, when the interfaced component i is in control, the system behaves as the library component $L_{\mathcal{C}}(i)$. When $L_{\mathcal{C}}(i)$ relinquishes control by entering final state s_j (for $j \in [n_F]$) the control is transfered to the interfaced component $\rho(i, j)$ (whose computation begins at its initial state).

For the formal definition, we denote the component library by $\mathcal{L} = \{C_i\}_{i \in I}$ where $C_i = \langle \Sigma_I, \Sigma_O, Q_i, q^i_0, \delta_i, F_i, L_i\rangle$, and the composition by $\mathcal{C} = \langle [n_F], \mathcal{L}, [m], i_0, \rho, \emptyset, L_{\mathcal{C}}\rangle$. For $i \in [m]$ for which $L_{\mathcal{C}}(i) = C_j$ we denote $Q(i) = Q_j$, $q_0(i) = q^j_0$, $\delta(i) = \delta_j$, $F(i) = F_j$, and $L(i) = L_j$. We can now define the composed system as a transducer $\mathcal{C}^{\mathcal{L}} = \langle \Sigma_I, \Sigma_O, Q^{\mathcal{C}\mathcal{L}}, q^{\mathcal{C}\mathcal{L}}_0, \delta^{\mathcal{C}\mathcal{L}}, \emptyset, L^{\mathcal{C}\mathcal{L}}\rangle$ where:

1. The set of states $Q^{\mathcal{C}}$ is $\bigcup_{i \in [m]} Q(i) \times \{i\}$.
2. The initial state $q^{\mathcal{C}\mathcal{L}}_0$ is $\langle q_0(i_0), i_0\rangle$.
3. The transition relation $\delta^{\mathcal{C}\mathcal{L}}$ is defined in the following way: for a letter $\sigma \in \Sigma_I$ and a state $\langle q, i\rangle$
   (a) if $\delta(i)(q, \sigma) \in Q(i) \setminus F(i)$ is not a final state then we define $\delta^{\mathcal{C}\mathcal{L}}(\langle q, i\rangle, \sigma) = \langle \delta(i)(q, \sigma), i\rangle$ (i.e, the transition is taking place within the component $L_{\mathcal{C}}(i)$).
   (b) if $\delta(i)(q, \sigma) \in F(i)$ is a final state $s^{L_{\mathcal{C}}(i)}_j$ we define $\delta^{\mathcal{C}\mathcal{L}}(\langle q, i\rangle, \sigma) = \langle q_0(\rho(i, j)), \rho(i, j)\rangle$ (i.e., the control is transfered to the initial state of the interfaced component corresponding to $\rho(i, j)$).
4. There are no final states.
5. The labeling function $L^{\mathcal{C}\mathcal{L}}$ is defined : $L^{\mathcal{C}\mathcal{L}}(\langle q, j\rangle) = L_{\mathcal{C}}(j)(q)$.

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.

4.2 Example

To illustrate control flow composition, we consider the following library: The library $\mathcal{L}$ contains only two control-flow transducers C_a, C_b. The input alphabet is $\Sigma_I = \{0, 1, 2\}$ and the output alphabet is $\Sigma_O = \{a, b, c\}$. The first transducer $C_a = \langle \Sigma_I, \Sigma_O, Q_a, q^a_0, \delta_a, F_a, L_a\rangle$, is defined in the following way:

1. The set of states is $Q_a = \{s_0, s_1, s_2, s_3\}$.
2. The initial state q^a_0 is s_0
3. The final states set is $F = \{s_2, s_3\}$.
4. The transition function δ_a is:
   – $\delta_a(s_0, 0) = s_1$

   - $\delta_a(s_0, 1) = s_2$
   - $\delta_a(s_0, 2) = s_3$
   - $\delta_a(s_1, 0) = s_1$
   - $\delta_a(s_1, 1) = s_2$
   - $\delta_a(s_1, 2) = s_3$

   Note: transitions out of final states are omitted as they do not influence the composed system.
5. The labeling function L_a is
   - $L_a(s_0) = a$
   - $L_a(s_1) = c$

   Note: the labels of final states are omitted as they do not influence the composed system.

In simple terms, C_a initially outputs a, as long it reads 0's it outputs c's. If it reads either 1 or 2 it relinquishes control.

The second transducer C_b is almost identical to C_a, the only difference is that in L_b is the output in the initial state is b (rather than a). Thus, $C_b = \langle \Sigma_I, \Sigma_O, Q_b, q_0^b, \delta_b, F_b, L_b \rangle$, is defined in the following way:

1. The set of states is $Q_b = \{t_0, t_1, t_2, t_3\}$.
2. The initial state q_0^b is t_0
3. The final states set $F = \{t_2, t_3\}$.
4. The transition function δ_b is defined:
   - $\delta_b(t_0, 0) = t_1$
   - $\delta_b(t_0, 1) = t_2$
   - $\delta_b(t_0, 2) = t_3$
   - $\delta_b(t_1, 0) = t_1$
   - $\delta_b(t_1, 1) = t_2$
   - $\delta_b(t_1, 2) = t_3$

   Note: transitions out of final states are omitted as they do not influence the composed system.
5. The labeling function L_b is
   - $L_a(t_0) = b$
   - $L_a(t_1) = c$

   Note: the labels of final states are omitted as they do not influence the composed system.

Note that n_F, i.e. the number of final states of transducers in $\mathcal{L}$, is 2.

Having one transducer the outputs a and c, and one transducer that outputs b and c, we would like to build a composed system in which throughout the computation an input of 0 is followed by an output of c, an input of 1 is followed by an output of a, and an input of 2 is followed by an output of b. In LTL terms, we would like a system that realizes $\mathbf{G}\,[(0 \to \mathbf{X}\,c) \wedge (1 \to \mathbf{X}\,a) \wedge (2 \to \mathbf{X}\,b)]$.

Intuitively, The composition is very simple, whenever a 1 is read, the control should be passed to C_a, and whenever a 2 is read, the control should be passed to C_b. Consider the composition $\mathcal{C}_1 = \langle [n_F], \mathcal{L}, [m], i_0, \rho, \emptyset, L_{\mathcal{C}} \rangle$, where:

1. The input alphabet is $[n_F] = 2$.
2. The output alphabet $\mathcal{L} = \{C_a, C_b\}$.
3. The set of states is $[m] = \{1, 2\}$.
4. The initial state is $i_0 = 1$.
5. The transition relation ρ is defined:
   - for $i = 1, 2$ $\rho(i, 1) = 1$.
   - for $i = 1, 2$ $\rho(i, 2) = 2$.
6. The output function is $L_{\mathcal{C}}(1) = C_a$ and $L_{\mathcal{C}}(2) = C_b$.

The composed system is $\mathcal{C}^{\mathcal{L}} = \langle \Sigma_I, \Sigma_O, Q^{\mathcal{CL}}, q_0^{\mathcal{CL}}, \delta^{\mathcal{CL}}, \emptyset, L^{\mathcal{CL}} \rangle$ where:

1. The composed system input alphabet is the same as the components input alphabet $\Sigma_I = \{0, 1, 2\}$.
2. The composed system output alphabet is the same as the components output alphabet $\Sigma_O = \{a, b, c\}$.
3. The composed system states are "indexed versions" of the components states. Here, $Q^{\mathcal{CL}} = \{\langle s_0, 1\rangle, \langle s_1, 1\rangle, \langle s_2, 1\rangle, \langle t_0, 2\rangle, \langle t_1, 2\rangle, \langle t_1, 2\rangle\}$.
4. The initial state is $\langle s_0, 1\rangle$.
5. The transition relation $\delta^{\mathcal{CL}}$ is defined in the following way:
   - $\delta^{\mathcal{CL}}(\langle s_0, 1\rangle, 0) = \langle s_1, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle s_0, 1\rangle, 1) = \langle s_0, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle s_0, 1\rangle, 2) = \langle t_0, 2\rangle$.
   - $\delta^{\mathcal{CL}}(\langle s_1, 1\rangle, 0) = \langle s_1, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle s_1, 1\rangle, 1) = \langle s_0, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle s_1, 1\rangle, 2) = \langle t_0, 2\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_0, 2\rangle, 0) = \langle t_1, 2\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_0, 2\rangle, 1) = \langle s_0, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_0, 2\rangle, 2) = \langle t_0, 2\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_1, 2\rangle, 0) = \langle t_1, 2\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_1, 2\rangle, 1) = \langle s_0, 1\rangle$.
   - $\delta^{\mathcal{CL}}(\langle t_1, 2\rangle, 2) = \langle t_0, 2\rangle$.
6. The output function is:
   - $L^{\mathcal{CL}}(\langle s_0, 1\rangle) = a$.
   - $L^{\mathcal{CL}}(\langle s_1, 1\rangle) = c$.
   - $L^{\mathcal{CL}}(\langle t_0, 2\rangle) = b$.
   - $L^{\mathcal{CL}}(\langle t_0, 2\rangle) = c$.

It is not hard to verify that the composed system $\mathcal{C}^{\mathcal{L}}$ realizes $\mathbf{G}\,[(0 \to \mathbf{X}\,c) \wedge (1 \to \mathbf{X}\,a) \wedge (2 \to \mathbf{X}\,b)]$.

4.3 Control-flow synthesis

Theorem 8. *The control-flow library LTL synthesis problem 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 [21]. 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. Intuitively, an infinite tree composition represents possible flow of control in a composition. The root is labeled by some $i \leq |\mathcal{L}|$ where the run begins when $C_i \in \mathcal{L}$ 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 $C_i \in \mathcal{L}$. Every finite composition $\mathcal{C} = \langle [n_F], \mathcal{L}, [m], i_0, \rho, \emptyset, L \rangle$ can be unfolded to an infinite composition tree in the following way: We first inductively define an $[m]$-labeled tree $\langle [n_F]^*, \tau_{[m]} \rangle$ in which $\tau_{[m]}(\varepsilon) = i_0$ and for every $v \in [n_f]^*$ and $j \in [n_f]$ we have $\tau_{[m]}(v \cdot j) = \rho(\tau_{[m]}(v), j)$. We now define the infinite composition tree $\langle [n_f]^*, \tau \rangle$ by setting for every $v \in [n_f]^*$ the label $\tau(v) = L_{\mathcal{C}}(\tau_{[m]}(v))$.

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 [22] and dualize it [10]. The effect of giving the control to a component C_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 C_i, what possible states $\mathcal{A}_\varphi$ might be in when C_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 $C = \{\Sigma_I, \Sigma_O, Q_C, q^C_0, \delta_C, F_C, L_C\}$, 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 $e_C(q, s) = \{\langle q', b\rangle \mid \exists w \in \Sigma_I$ s.t. $s = \delta^*_C(w)$ and $q' \in \delta^*_\varphi(q, C, w)$ 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 $w \in \Sigma^\omega_I$ 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 $\mathcal{A}_\varphi^q$ 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 $e_C(q, s_j^i)$. Formally,
     $\Delta(\langle q, b\rangle, C) = \bigwedge_{i\in[n_F]} \bigwedge_{\langle q_i, b_i\rangle \in e_C(q, s_j^i)} (i, \langle q_i, b_i\rangle)$.
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 9 $L(\mathcal{A})$ is empty iff φ is not realizable from $\mathcal{L}$ □

Proof: Assume first that φ is realizable from $\mathcal{L}$. Then, there exists a composition $\mathcal{C}$ realizing φ. The composition $\mathcal{C}$ induces an unfolded infinite tree composition $\langle [n_F]^*, \tau\rangle$ as described above. We claim that $\langle [n_F]^*, \tau\rangle \in \mathcal{L}$.

Assume, towards contradiction, otherwise. Then, there exists a rejecting run $\langle T, r\rangle$ of $\mathcal{A}$ on $\langle T, \tau\rangle$, and in it a rejecting path labeled by $\pi = \pi_1\pi_2 \ldots \in [n_F]^\omega$. We denote $\pi[0] = \varepsilon$ and for every $i > 0$ we denote by $\pi[i]$ the node $\pi_1 \ldots \pi_i \in [n_F]^*$. Similarly, we denote by $\langle q_i, b_i\rangle$ the value of $r(\pi[i])$.

By the construction of $\mathcal{A}$, for each $i \geq 0$ there exits a word $w_i \in \Sigma_I^*$ such that:

1. $\tau(\pi[i])$ run on w_i ends in the π_{i+1}-th final state, and
2. There exists a finite run r_i^φ of $\mathcal{A}_\varphi^{q_i}$ on $\Sigma(w_i, \tau(i))$ that ends in q_{i+1}. Furthermore, r_i^φ traverses a state in α iff $b_i = 1$.

We denote by w the word $w_0 \cdot w_1 \cdot \ldots \in \Sigma_I^\omega$, and by w_Σ the word $\Sigma(w_0, \tau(\pi[0])) \cdot \Sigma(w_1, \tau(\pi[1])) \ldots \in \Sigma^\omega$.

Since r is rejecting on π, for infinitely many $i > 0$ we have $b_i = 1$. Therefore, the run of $\mathcal{A}_\varphi$ on w_Σ passes infinitely often through α and is therefore rejecting. This contradicts the assumption that the composition $\mathcal{C}$ satisfies φ. We reached contradiction implying that if φ is satisfied by a composition $\mathcal{C}$ then the unfolding of $\mathcal{C}$ is accepted by $\mathcal{A}$.

If, on the other hand, $L(\mathcal{A})$ is not empty, then by theorem 4.9 in [10] there exists in $\mathcal{L}(\mathcal{A})$ a tree T induced by a transducer T_c with at most $2^{O(|\varphi|\log(|\varphi|))}$ states. This transducer induce a composition $\mathcal{C}$ with the same structure.

Next, we show that $\mathcal{C}$ satisfies φ. Given an input word word $w \in \Sigma_I^\omega$, during the run of $\mathcal{C}$ on w the control might be held by different components at different cycles. At the beginning of $\mathcal{C}$'s run, the control is held by $L_\mathcal{C}(i_0)$. As the run progresses the first component might relinquish control. We denote by C_1 the first component that retains control, namely $L_\mathcal{C}(i_0)$, by w_1 the subword on which C_1 retains control, and by j_1 the index of the final state s_{j_1} into which C_1 enters at the end of w_1 (for the case in which w_1 is finite). If C_1 relinquish control, then the control is passed to some other component C_2 and so forth. We construct inductively the sequences $\{C_i\}$, $\{w_i\}$, and $\{j_i\}$. Note that the sequences might be finite or infinite. For simplicity sake, assume first that no component retain control over an infinite suffix of the run and therefore the sequences are infinite.

We denote by $T = \langle [n_F]^*, \tau\rangle$ the labeled tree induced by T_C. Recall that T_C was chosen by the fact that T is in the language of $\mathcal{A}$. The sequence $\mathbf{j} = j_1 j_2 \ldots$ is a path in T, that is labeled by $C_1C_2 \ldots$. Since $\mathcal{A}$ accepts T, the path $\mathbf{j}$ is traversed and accepted by $\mathcal{A}$. Next, we show the latter implies that $\mathcal{A}_\varphi$ accepts w.

Note that $w = w_1 w_2 \ldots$. We also denote $w_\Sigma = \Sigma(w_1, C_1)$ $\Sigma(w_2, C_2) \ldots$. For $i \geq 0$ let $q_0^i, q_1^i, \ldots$ be a run of $\mathcal{A}_\varphi$ on w_i. Note that by definition of control-flow composition, the state $q_{|w_i|}^i$ can be identified with the state q_0^{i+1}. For every $i > 0$ we denote by r^{i-1} the state q_0^i.

By the construction of $\mathcal{A}$, there exists a sequence of bits $b_0, b_1 \ldots$, such that the run of $\mathcal{A}$ on $\mathbf{j}$ contains a path labeled by $\langle r_0, b_0\rangle\langle r_1, b_1\rangle \ldots$. Furthermore, if the run of $\mathcal{A}_\varphi$ on w_i traverses through a α then b_i is 1. Since $\mathcal{A}$'s run is accepting there must be a suffix of the run in which all the b_i's are 0. Therefore, in the run of $\mathcal{A}_\varphi$ on w there must be suffix of the run in which α is not traversed. Thus, $\mathcal{A}_\varphi$ accepts w.

Finally, we have to consider the case in which some component C_i never relinquish control and the sequences are finite. Since $\mathcal{A}$ accepts T the component C_i cannot be a dead end for r^{i-1}. Therefore the run of $\mathcal{A}_\varphi^q$ on the suffix w_i must be accepting. Thus, for every word w the induced word w_Σ is accepted by $\mathcal{A}_\varphi$ implying that $\mathcal{C}$ realize φ. □

Note that while Claim 9 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 8 and all that is left is the complexity analysis. The input to the problem is a library $\mathcal{L} = \{C_1, \ldots, C_{|\mathcal{L}|}\}$ 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 $C_i \in \mathcal{L}$, an $\mathcal{A}_\varphi$-state $q \in Q_\varphi$, and a final state $s_j^i \in F_i$ the value of $e_C(q, s_j^i)$ can be computed by deciding emptiness of small variants of the product of $\mathcal{A}_\varphi$ and C_i. Thus, computing $e_C(q, s^i_j)$ is nondeterministic logspace in $2^{O(|\varphi|)} \cdot |C_i|$. The complexity of computing $\mathcal{A}$ is nondeterministic logspace in $2^{O(|\varphi|)} \cdot n_F \cdot (\sum_{i=1}^{|\mathcal{L}|} |C_i|)$. 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 [23]). Thus, the overall complexity of the problem is doubly exponential in $|\varphi|$ and polynomially dependent on the size of the library.

$\square$

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 8, 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 8. 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 (1963) 23–35
2. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages. (1989) 179–190
3. Sifakis, J.: A framework for component-based construction extended abstract. In: Proc. 3rd Int. Conf. on Software Engineering and Formal Methods (SEFM 2005), IEEE Computer Society (2005) 293–300
4. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer (2004)
5. Berardi, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: ICSOC. (2003) 43–58
6. Sardiña, S., Patrizi, F., Giacomo, G.D.: Automatic synthesis of a global behavior from multiple distributed behaviors. In: AAAI. (2007) 1063–1069
7. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS **138** (1969) 295–311
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. (1990) 746–757
10. Kupferman, O., Vardi, M.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science. (2005) 531–540
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 195, Springer (2005) 83–104
13. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. on Foundations of Computer Science. (1977) 46–57
14. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Proc. 7th ACM Symp. on Principles of Programming Languages. (1980) 163–173
15. Muller, D., Schupp, P.: Alternating automata on infinite trees. Theoretical Computer Science **54** (1987) 267–276
16. Muller, D., Saoudi, A., Schupp, P.: Alternating automata, the weak monadic theory of the tree and its complexity. In: Proc. 13th Int. Colloq. on Automata, Languages, and Programming. Volume 226 of Lecture Notes in Computer Science., Springer (1986) 275 – 283
17. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
18. Nain, S., Vardi, M.Y.: Branching vs. linear time: Semantical perspective. In: 5th Int. Symp. on Automated Technology for Verification and Analysis. Volume 4762 of Lecture Notes in Computer Science., Springer (2007) 19–34

19. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
20. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Infornation Processing Letters **22**(6) (1986) 307–309
21. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science (1992)
22. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation **115**(1) (1994) 1–37
23. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games: A Guide to Current Research. Volume 2500 of Lecture Notes in Computer Science. Springer (2002)