Science of Computer Programming 83 (2014) 56–79

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Synthesis of hierarchical systems

Benjamin Aminof [a], Fabio Mogavero [b,*], Aniello Murano [b]

[a] Hebrew University, Jerusalem 91904, Israel
[b] Università degli Studi di Napoli "Federico II", 80126 Napoli, Italy

HIGHLIGHTS

- We describe how to synthesize a hierarchical system from a library of components.
- We propose a procedure that in rounds given a specification returns a correct system.
- We manage both branching-time (mu-Calculus) and linear-time specification languages.
- The resulting system is hierarchical and reuses components previously synthesized.
- The obtained complexity is not worst than that of classic flat synthesis procedure.

ARTICLE INFO

Article history:
Received 4 May 2012
Received in revised form 1 July 2013
Accepted 2 July 2013
Available online 23 July 2013

Keywords:
Hierarchical systems
mu-Calculus
Temporal logics
Parity games
Synthesis

ABSTRACT

In automated synthesis, given a specification, we automatically create a system that is guaranteed to satisfy the specification. In the classical temporal synthesis algorithms, one usually creates a "flat" system "from scratch". However, real-life software and hardware systems are usually created using preexisting libraries of reusable components, and are not "flat" since repeated sub-systems are described only once.

In this work we describe an algorithm for the synthesis of a hierarchical system from a library of hierarchical components, which follows the "bottom-up" approach to system design. Our algorithm works by synthesizing in many rounds, when at each round the system designer provides the specification of the currently desired module, which is then automatically synthesized using the initial library and the previously constructed modules. To ensure that the synthesized module actually takes advantage of the available high-level modules, we guide the algorithm by enforcing certain modularity criteria.

We show that the synthesis of a hierarchical system from a library of hierarchical components is EXPTIME-complete for μ-calculus, and 2EXPTIME-complete for LTL, both in the cases of complete and incomplete information. Thus, in all cases, it is not harder than the classical synthesis problem (of synthesizing flat systems "from scratch"), even though the synthesized hierarchical system may be exponentially smaller than a flat one.

1. Introduction

In formal verification and design, *synthesis* is the automated construction of a system from its specification. The basic idea is simple and appealing: instead of developing a system and then verifying that it is correct w.r.t. its specification, we use an automated procedure that, given a specification, builds a system that is correct by construction.

The first formulation of synthesis goes back to Church [20]. Later works on synthesis considered first *closed systems*, where the system is extracted from a constructive proof that the specification is satisfiable [28,41]. In the late 1980s, Pnueli

* Corresponding author.
E-mail address: fm@fabiomogavero.com (F. Mogavero).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.07.001

and Rosner [47] realized that such a synthesis paradigm is not of much interest when applied to *open systems* [31] (also called *reactive systems* [16,37]). Differently from closed systems, an open system interacts with an external environment and its correctness depends on whether it satisfies the specification with respect to all allowable environments. If we apply the techniques of [28,41] to open systems, we obtain a system that is correct only with respect to some specific environments. In [47], Pnueli and Rosner argued that the right way to approach synthesis of open systems is to consider the framework as a possibly infinite game between the environment and the system. A correct system can be then viewed as a winning strategy in this game, and synthesizing a system amounts to finding such a strategy.

The Pnueli and Rosner idea can be summarized as follows. Given sets Σ_I and Σ_O of inputs and outputs, respectively (usually, $\Sigma_I = 2^I$ and $\Sigma_O = 2^O$, where I is a set of input signals supplied by the environment and O is a set of output signals), one can view a system as a strategy $P : \Sigma_I^* \to \Sigma_O$ that maps a finite sequence of sets of input signals (i.e., the history of the actions of the environment so far) into a set of current output signals. When P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $\Sigma_I \cup \Sigma_O$. Though the system P is deterministic, it induces a *computation tree*. The branches of the tree correspond to external nondeterminism, caused by different possible inputs. Thus, the tree has a fixed branching degree $|\Sigma_I|$, and it embodies all the possible inputs (and hence also computations) of P. When we synthesize P from a linear temporal logic formula φ we require φ to hold in all the paths of P's computation tree. However, in order to impose possibility requirements on P we have to use a branching-time logic like μ-calculus. Given a branching specification φ over $\Sigma_I \cup \Sigma_O$, realizability of φ is the problem of determining whether there exists a system P whose computation tree satisfies φ. Correct synthesis of φ then amounts to constructing such a P. The above synthesis problem for linear-time temporal logic (LTL) specifications was addressed in [47], and for μ-calculus specifications in [26]. In both cases, the traditional algorithm for finding the desired P works by constructing an appropriate *computation tree-automaton* that accepts trees that satisfy the specification formula, and then looking for a finitely-representable witness to the non-emptiness of this automaton. Such a witness can be easily viewed as a finite-state system P realizing the specification.

In spite of the rich theory developed for system synthesis in the last two decades, little of this theory has been lifted to practice. In fact, apart from very recent results [14,24,51], the main classical approaches to tackle synthesis in practice are either to use heuristics (e.g., [30]) or to restrict to simple specifications (e.g., [44]). Some people argue that this is because the synthesis problem is very expensive compared to model-checking [36]. There is, however, something misleading in this perception: while the complexity of synthesis is given with respect to the specification only, the complexity of model-checking is given also with respect to a program, which can be very large. A common thread in almost all of the works concerning synthesis is the assumption that the system is to be built from "scratch". Obviously, real-world systems are rarely constructed this way, but rather by utilizing many preexisting reusable components, i.e., a *library*. Using standard preexisting components is sometimes unavoidable (for example, access to hardware resources is usually under the control of the operating system, which must be "reused"), and many times has other benefits (apart from saving time and effort, which may seem to be less of a problem in a setting of automatic – as opposed to manual – synthesis), such as maintaining a common code base, and abstracting away low level details that are already handled by the preexisting components. Another reason that may account for the limited use of synthesis in practice is that many designers find it extremely difficult and/or unnatural to write a complex specification in temporal logic. Indeed, a very common practice in the hardware industry is to consider a model of the desired hardware written in a high level programming language like ANSI-C to be a specification (a.k.a. "golden model") [45]. Moreover, even if a specification is written in temporal logic, the synthesized system is usually monolithic and looks very unnatural from the system designer's point of view. Indeed, in classical temporal synthesis algorithms one usually creates in one step a "flat" system, i.e., a system in which sub-systems may be repeated many times. On the contrary, real-life software and hardware systems are built step by step and are hierarchical (or even recursive) having repeated sub-systems (such as sub-routines) described only once. While hierarchical systems may be exponentially more succinct than flat ones, it has been shown that the cost of solving questions about them (like model-checking) are in many cases not exponentially higher [6,7,29]. Hierarchical systems can also be seen as a special case of recursive systems [3,4], where the nesting of calls to sub-systems is bounded. However, having no bound on the nesting of calls gives rise to infinite-state systems, and this may result in a higher complexity, especially when we have a bound that is small with respect to the rest of the system.

Consider for example the problem of synthesizing a 60-minutes chronograph displaying elapsed time in minutes and seconds. A naive solution, which is the one created by a traditional synthesis approach, is to create a transducer that contains at least 3600 explicit states, one for each successive clock signal. However, an alternative is to design a hierarchical transducer, composed of two different machines: one counts from 0 to 59 minutes (the *minutes-machine*), and the other counts from 0 to 59 seconds (the *seconds-machine*) – see Example 3.1 for a detailed description. In particular, by means of 60 special states, named *boxes* or *super-states*, the minutes-machine calls 60 times the seconds-machine. This hierarchical machine is arguably more natural, and it is definitely more succinct since it has an order of magnitude less states and boxes than the flat one with 3600 states. Once the 60-minutes chronograph design process has been completed the resulting transducer can be added to a library of components for future use, for example in a 24-hours chronograph. Then, it is enough to build a new machine (*hours-machine*) that counts from 0 to 24 hours using boxes to call the 60-minutes transducer found in the library.

In this work, we provide a uniform algorithm, for different temporal logics, for the synthesis of a hierarchical system from a library of hierarchical systems, which mimics the "bottom-up" approach to system design, where one builds a system

by iteratively constructing new modules based on previously constructed ones.[1] More specifically, we start the synthesis process by providing the algorithm with an initial library $\mathcal{L}_0$ of available hierarchical components (transducers), as well as atomic ones. We then proceed by synthesizing in rounds. At each round i, the system designer provides a specification formula φ_i of the currently desired hierarchical transducer, which is then automatically synthesized using the currently available transducers as possible sub-components. Once a new transducer is synthesized, it is added to the library to be used by subsequent iterations. The hierarchical transducer synthesized in the last round is the desired system.

Observe that it is easily conceivable that if the initial library $\mathcal{L}_0$ contains enough atomic components then the synthesis algorithm may use them exclusively, essentially producing a flat system. We thus have to direct the single-round synthesis algorithm in such a way that it produces modular, and not flat, results. The question of what makes a design more or less modular is very difficult to answer, and has received many (and often widely different) answers throughout the years (see [43] for a survey). We claim that some very natural modularity criteria are regular, and show how any criterion that can be checked by a parity tree automaton can be easily incorporated into our automata-based synthesis algorithm.

It is our belief that this approach carries with it many benefits. First, the resulting system can be quite succinct due to its hierarchical nature, as demonstrated by the chronograph example above. Second, we are certain that most designers will find it easier to write a series of relatively simple specification formulas than to write one monolithic formula describing, in one shot, the end result. Third, after each round the synthesized component can be tested and verified in isolation to gain confidence that there were no mistakes in the specification (or a bug in the synthesizer). Testing and verification of intermediate modules can be much easier due to their smaller size, and design errors can be discovered at an earlier stage. Also, the efforts spent on such intermediate modules is an one-time investment, as these modules can be reused in more than one project. Finally, the structure of the resulting system follows much more the high-level view that the designer had in mind, increasing his confidence in, and understanding of, the resulting system.

We show that while hierarchical systems may be exponentially smaller than flat ones, the problem of synthesizing a hierarchical system from a library of existing hierarchical systems is EXPTIME-complete for μ-calculus, and 2EXPTIME-complete for LTL. Thus, this problem is not harder than the classical synthesis problem of flat systems "from scratch". Furthermore, we show that this is true also in the case where the synthesized system has incomplete information about the environment's input.

The most technically challenging part of the hierarchical synthesis algorithm presented above is the algorithm for performing the synthesis step of a single round. As stated before, in the classical automata-theoretic approach to synthesis [47], synthesizing a system is reduced to the problem of finding a regular tree that is a witness to the non-emptiness of a suitable tree automaton. Here, we also reduce the synthesis problem to the non-emptiness problem of a tree automaton. However, unlike the classical approach, we build an automaton whose input is not a computation tree, but rather a system description in the form of a *connectivity tree* (inspired by the "control-flow" trees of [39]), which describes how to connect library components in a way that satisfies the specification formula. Essentially, every node in a connectivity tree is labeled by some transducer from the library, and the sons of each node correspond to the different exits this library transducer has. Thus, for example, if a node y labeled by $\mathcal{K}'$ has a son, that corresponds to exit e of $\mathcal{K}'$, labeled by $\mathcal{K}''$, then it means that the exit state e of $\mathcal{K}'$ should be connected to the (single) entry of $\mathcal{K}''$.

Given a library of hierarchical transducers, and a temporal logic specification φ, our single-round algorithm builds a tree automaton $\mathcal{A}_\varphi^{\mathcal{T}}$ such that $\mathcal{A}_\varphi^{\mathcal{T}}$ accepts a regular connectivity tree $\mathcal{T}$ iff it induces a hierarchical transducer $\mathcal{K}$ that satisfies φ. Given φ, it is well known how to build an automaton $\mathcal{A}_\varphi$ that accepts all trees that satisfy φ [27,36]. Hence, all we need to do is to have $\mathcal{A}_\varphi^{\mathcal{T}}$ simulate all possible runs of $\mathcal{A}_\varphi$ on the computation tree of $\mathcal{K}$. However, this is not easy since $\mathcal{A}_\varphi^{\mathcal{T}}$ has as its input not the computation tree of $\mathcal{K}$, but a completely different tree – the connectivity tree $\mathcal{T}$ describing how $\mathcal{K}$ is composed from library transducers.

The basic idea is that a copy of $\mathcal{A}_\varphi^{\mathcal{T}}$ that reads a node y of $\mathcal{T}$ labeled by a library transducer $\mathcal{K}'$ can simulate $\mathcal{A}_\varphi$ on the portion of the computation tree of $\mathcal{K}'$ until an exit of $\mathcal{K}'$ is reached. When such an exit is reached $\mathcal{T}$ is consulted to see to which library transducer the computation proceeds from that exit, and the simulation continues. Unfortunately, a direct implementation of this idea would result in an algorithm whose complexity is too high. Indeed, it is important to keep the size of $\mathcal{A}_\varphi^{\mathcal{T}}$ independent of the size of the transducers in the library and, on-the-fly simulation of the runs of $\mathcal{A}_\varphi$ on the computation trees of the library transducers would require an embedding of these transducers inside $\mathcal{A}_\varphi^{\mathcal{T}}$. The key observation at the heart of our solution to this problem is that while simulating $\mathcal{A}_\varphi$ on the computation tree of a library transducer $\mathcal{K}'$, no input is consumed by $\mathcal{A}_\varphi^{\mathcal{T}}$ until an exit of $\mathcal{K}'$ is encountered. Hence, we can perform these portions of the simulation off-line (thus circumventing the need to incorporate a copy of $\mathcal{K}'$ into $\mathcal{A}_\varphi^{\mathcal{T}}$) and incorporate a suitable summary of these simulations into the transition relation of $\mathcal{A}_\varphi^{\mathcal{T}}$. The difficult problem of summarizing the possibly infinitely many infinite runs of $\mathcal{A}_\varphi$ on the computation tree of every library transducer $\mathcal{K}'$, in a way which is independent of the size of this transducer, is made possible by a suitable adaptation of the *summary functions* used in [7] in order to summarize the possible moves in hierarchical sub-arenas of hierarchical parity games.

[1] While for systems built from scratch, a top-down approach may be argued to be more suitable, we find the bottom-up approach to be more natural when synthesizing from a library.

Related work. The issues of specification and correctness of modularly designed systems have received a fair attention in the formal verification literature. Examples of important works on this subject are [12,13,21,38,50]. Recently, synthesis of finite state systems has been gaining more and more attention as new achievements have significantly improved its practical applicability. Prominent results in this respect have been reported in [14,24,51]. In [24], a *Safraless synthesis approach* using BDDs as state space representation has been fruitfully investigated. This approach reduces significantly the system state representation and improves the performance of previous approaches to full LTL synthesis. In [51], an improved game theoretic approach to the synthesis has been introduced, by using and opportune decomposition of system properties. Specifically, the new algorithm consider *safety* and *persistence* system properties, which allow to build and solve the synthesis game incrementally. Indeed, each safety and persistence property is used to gradually construct a parity game and at each step an opportune refinement of the graph game is considered. At the end of the process a compact symbolic encoding of the parity game is produced, it is composed with the remaining LTL properties in one full game that is finally solved. The approach has been implemented in the tool *Vis* [18], as an extension of the technique described in [52], and efficiently compared with other synthesis tools. Finally, in [14] a tool for solving the LTL realizability and synthesis problems, namely *Acacia+*, is presented. The tool uses recent approaches that reduce the synthesis problem to safety games, which can be solved efficiently by symbolic incremental algorithms based on antichains (instead of BDDs). The reduction to safety games offers a useful compositional approach for large conjunctions of LTL formulas.

All the papers reported above have in common the fact that the synthesis approach always starts from scratch. Recently, a number of works on the automatic synthesis from reusable components has been proposed and showed to have practical applications in several scenarios, including robotics and multi-agent system behaviors [22,23]. Generally speaking, these works are interested in synthesizing a system controller from existing components in order to satisfy a desired target behavior, where a component can be any object such as a device, an agent, a software or hardware item, or a workflow. Notably, the idea of composing and reusing components has been strongly and fruitfully exploited in several computer science fields such as in *Service Oriented Computing*, under the name of "service composition" [1,19].

The work on automatic synthesis from reusable components that is closest to our approach is the one done by Lustig and Vardi on LTL synthesis from libraries of (flat) transducers [39]. The technically most difficult part of our work is an algorithm for performing the synthesis step of a single round of the multiple-rounds algorithm. To this end we use an automata-theoretic approach. However, as stated before, unlike the classical approach of [47], we build an automaton whose input is not a computation tree but rather a system description in the form of a connectivity tree. Taken by itself, our single-round algorithm extends the "control-flow" synthesis work from [39] in four directions. (i) We consider not only LTL specifications but also the modal μ-calculus. Hence, unlike [39], where universal co-Büchi tree automata were used, we have to use the more expressive alternating parity tree automata. Unfortunately, this is not simply a matter of changing the acceptance condition. Indeed, in order to obtain an optimal upper bound, a widely different approach, which makes use of the machinery developed in [7] is needed. (ii) We need to be able to handle libraries of hierarchical transducers, whereas in [39] only libraries of flat transducers are considered. (iii) A synthesized transducer has no top-level exits (since it must be able to run on all possible input words), and thus, its ability to serve as a sub-transducer of another transducer (in future iterations of the multiple-rounds algorithm) is severely limited – it is like a function that never returns to its caller. We therefore need to address the problem of synthesizing exits for such transducers. (iv) As discussed above, we incorporate into the algorithm the enforcing of modularity criteria.

Recently, an extension of [39] appeared in [40], where the problem of *Nested-Words Temporal Logic* (NWTL) synthesis from recursive component libraries has been investigated. NWTL extends LTL with special operators that allow one to handle "call and return" computations [2] and it is used in [40] to describe how the components have to be connected in the synthesis problem. We recall that in our framework the logic does not drive (at least not explicitly) the way the components have to be connected. Moreover, the approach used in [40] cannot be applied directly to the branching framework we consider in this paper, as we recall that already the satisfiability problem for μ-calculus with "call and return" is undecidable even for very restricted cases [5].

Structure of the article. The rest of this article is organized as follows. Section 2 contains the basic definitions used throughout the paper. In Section 3, we introduce hierarchical structures and transducers as well as their flat expansion. In Section 4.1, we give our hierarchical synthesis algorithm, and show how the single-round synthesis problem can be reduced to the automata setting with the help of connectivity trees. In Section 4.2, we discuss the problem of enforcing modularity. In Section 5, we use hierarchical two-player games to prove the correctness of our algorithm. In Section 6 we show how our algorithm can be adapted to address the problem of synthesizing hierarchical systems with incomplete information. Finally, in Section 7, we give a short summary.

2. Preliminaries

Trees. Let $\mathcal{D}$ be a set. A *$\mathcal{D}$-tree* is a prefix-closed subset $T \subseteq \mathcal{D}^*$ such that if $x \cdot c \in T$, where $x \in \mathcal{D}^*$ and $c \in \mathcal{D}$, then also $x \in T$. The *complete $\mathcal{D}$-tree* is the tree $\mathcal{D}^*$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. Given a word $x = y \cdot d$, with $y \in \mathcal{D}^*$ and $d \in \mathcal{D}$, we define $last(x)$ to be d. For $x \in T$, the nodes $x \cdot d \in T$, where $d \in \mathcal{D}$, are the *sons* of x. A *leaf* is a node with no sons. A *path* of 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 is a unique $d \in \mathcal{D}$ such that $x \cdot d \in \pi$. For an alphabet Σ, a *Σ-labeled $\mathcal{D}$-tree* is a pair $\langle T, V \rangle$ where $T \subseteq \mathcal{D}^*$ is a $\mathcal{D}$-tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ.

Asymmetric alternating tree automata. *Alternating tree automata* are a generalization of nondeterministic tree automata [42] (see [36], for more details). Intuitively, while a nondeterministic tree automaton that visits a node of the input tree sends exactly one copy of itself to each of the sons of the node, an alternating automaton can send several copies of itself (or none at all) to the same son.

An (asymmetric) *Alternating Parity Tree Automaton* (*APT*) is a tuple $\mathcal{A} = \langle \Sigma, \mathcal{D}, Q, q_0, \delta, F \rangle$, where Σ, $\mathcal{D}$, and Q are non-empty finite sets of *input letters*, *directions*, and *states*, respectively, $q_0 \in Q$ is an *initial state*, F is a *parity acceptance condition* to be defined later, and $\delta\colon\ Q \times \Sigma \mapsto \mathcal{B}^+(\mathcal{D} \times Q)$ is an *alternating transition function*, which maps a state and an input letter to a positive Boolean combination of elements in $\mathcal{D} \times Q$. Given a set $\mathcal{S} \subseteq \mathcal{D} \times Q$ and a formula $\theta \in \mathcal{B}^+(\mathcal{D} \times Q)$, we say that $\mathcal{S}$ satisfies θ (denoted by $\mathcal{S} \models \theta$) if assigning **true** to elements in $\mathcal{S}$ and **false** to elements in $(\mathcal{D} \times Q) \setminus \mathcal{S}$ makes θ true. A *run* of an APT $\mathcal{A}$ on a Σ-labeled $\mathcal{D}$-tree $\mathcal{T} = \langle T, V \rangle$ is a $(T \times Q)$-labeled $\mathbb{N}$-tree $\langle T_r, r \rangle$, where $\mathbb{N}$ is the set of non-negative integers, such that (i) $r(\varepsilon) = (\varepsilon, q_0)$ and (ii) for all $y \in T_r$, with $r(y) = (x, q)$, there exists a set $\mathcal{S} \subseteq \mathcal{D} \times Q$, such that $\mathcal{S} \models \delta(q, V(x))$, and there is a son y' of y, with $r(y') = (x \cdot d, q')$, for every $(d, q') \in \mathcal{S}$. Given a node y of a run $\langle T_r, r \rangle$, with $r(y) = (z, q) \in T \times Q$, we define $last(r(y)) = (last(z), q)$. An alternating parity automaton $\mathcal{A}$ is *nondeterministic* (denoted NPT), iff when its transition relation is rewritten in disjunctive normal form each disjunct contains at most one element of $\{d\} \times Q$, for every $d \in \mathcal{D}$. An automaton is *universal* (denoted UPT) if all the formulas that appear in its transition relation are conjunctions of atoms in $\mathcal{D} \times Q$.

Symmetric alternating tree automata. A *symmetric* alternating parity tree automaton with ε-moves (*SAPT*) [32] does not distinguish between the different sons of a node, and can send copies of itself only in a universal or an existential manner. Formally, an SAPT is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ is a finite input alphabet, Q is a finite set of states, partitioned into four sets, universal ($Q^\wedge$), existential ($Q^\vee$), ε-and ($Q^{(\varepsilon,\wedge)}$), and ε-or ($Q^{(\varepsilon,\vee)}$) states (we also write $Q^{\vee,\wedge} = Q^\vee \cup Q^\wedge$, and $Q^\varepsilon = Q^{(\varepsilon,\vee)} \cup Q^{(\varepsilon,\wedge)}$), $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to (Q \cup 2^Q)$ is a transition function such that for all $\sigma \in \Sigma$, we have that $\delta(q, \sigma) \in Q$ for $q \in Q^{\vee,\wedge}$, and $\delta(q, \sigma) \in 2^Q$ for $q \in Q^\varepsilon$, and F is a parity acceptance condition, to be defined later. In other words, the transition relation returns a single state, if the automaton is in an $Q^{\vee,\wedge}$ state, and a set of states if it is instead in a Q^ε state. We assume that Q contains in addition two special states `ff` and `tt`, called *rejecting sink* and *accepting sink*, respectively, such that $\forall a \in \Sigma : \delta(\mathtt{tt}, a) = \mathtt{tt}, \delta(\mathtt{ff}, a) = \mathtt{ff}$. The classification of `ff` and `tt` as universal or existential states is arbitrary. Transitions from states in Q^ε launch copies of $\mathcal{A}$ that stay on the same input node as before the transition, while transitions from states in $Q^{\vee,\wedge}$ launch copies that advance to sons of the current node. Note that for an SAPT the set $\mathcal{D}$ of directions of the input trees plays no role in the definition of a run. When a symmetric alternating tree automaton $\mathcal{A}$ runs on an input tree it starts with a copy in state q_0 at the root of the tree. It then follows δ in order to send further copies. For example, if a copy of $\mathcal{A}$ is in state $q \in Q^{(\varepsilon,\vee)}$, reads a node x labeled σ, and $\delta(q, \sigma) = \{q_1, q_2\}$, then this copy proceeds either to state q_1 or to state q_2, and keeps reading x. As another example, if $q \in Q^\wedge$ and $\delta(q, \sigma) = q_1$, then $\mathcal{A}$ sends a copy in state q_1 to every son of x. Note that different copies of $\mathcal{A}$ may read the same node of the input tree. Formally, a *run* of $\mathcal{A}$ on a Σ-labeled $\mathcal{D}$-tree $\langle T, V \rangle$ is a $(T \times Q)$-labeled $\mathbb{N}$-tree $\langle T_r, r \rangle$. A node in T_r labeled by (x, q) describes a copy of $\mathcal{A}$ in state q that reads the node x of T. A run has to satisfy $r(\varepsilon) = (\varepsilon, q_0)$ and, for all $y \in T_r$ with $r(y) = (x, q)$, the following hold:

- If $q \in Q^\wedge$ (resp. $q \in Q^\vee$) and $\delta(q, V(x)) = p$, then for each son (resp. for exactly one son) $x \cdot d$ of x, there is a node $y \cdot i \in T_r$ with $r(y \cdot i) = (x \cdot d, p)$;
- If $q \in Q^{(\varepsilon,\wedge)}$ (resp. $q \in Q^{(\varepsilon,\vee)}$) and $\delta(q, V(x)) = \{p_0, \ldots, p_k\}$, then for all $i \in \{0..k\}$ (resp. for one $i \in \{0..k\}$) the node $y \cdot i \in T_r$, and $r(y \cdot i) = (x, p_i)$.

Parity acceptance condition. A *parity condition* is given by means of a coloring function on the set of states. Formally, a *parity condition* is a function $F : Q \to C$, where $C = \{C_{\min}, \ldots, C_{\max}\} \subset \mathbb{N}$ is a set of colors. The size $|C|$ of C is called the *index* of the automaton. For an SAPT, we also assume that the special state `tt` is given an even color, and `ff` is given an odd color. For an infinite path $\pi \subseteq T_r$ of a run $\langle T_r, r \rangle$, let $maxC(\pi)$ be the maximal color that appears infinitely often along π. Similarly, for a finite path π, we define $maxC(\pi)$ to be the maximal color that appears at least once in π. An infinite path $\pi \subseteq T_r$ satisfies the acceptance condition F iff $maxC(\pi)$ is even. A run $\langle T_r, r \rangle$ is *accepting* iff all its infinite paths satisfy F. The automaton $\mathcal{A}$ accepts an input tree $\langle T, V \rangle$ if there is an accepting run of $\mathcal{A}$ on $\langle T, V \rangle$. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of Σ-labeled $\mathcal{D}$-trees accepted by $\mathcal{A}$. We say that an automaton $\mathcal{A}$ is *non-empty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

A wide range of temporal logics can be translated to alternating tree automata (details can be found in [36] and in Appendix A). In particular:

Theorem 2.1. *(See [27,36].) Given a temporal-logic formula φ, it is possible to construct an SAPT $\mathcal{A}_\varphi$ such that $\mathcal{L}(\mathcal{A}_\varphi)$ is exactly the set of trees satisfying φ. In particular, we have that*

- *if φ is a μ-calculus formula, then $\mathcal{A}_\varphi$ is an alternating parity automaton with $O(|\varphi|)$ states and index $O(|\varphi|)$;*
- *if φ is an* LTL *formula, then $\mathcal{A}_\varphi$ is a universal parity automaton with $2^{O(|\varphi|)}$ states, and index 2.*

For technical convenience we sometimes refer to functions (like transitions and labeling functions) as relations, and in particular, we consider $\emptyset$ to be a function with an empty domain.

3. Hierarchical systems

3.1. Structures

Hierarchical structures [6] are a generalization of Kripke structures in which repeated sub-structures are specified only once. Technically, some of the states in a hierarchical structure are *boxes* (alternatively, *super-states*), in which inner hierarchical structures are nested. Formally, a hierarchical structure is a tuple $\mathcal{S} = \langle \Sigma_O, \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle \rangle$, where Σ_O is a non-empty set of output letters and, for every $1 \leqslant i \leqslant n$, we have that the substructure $\mathcal{S}_i = \langle W_i, \mathcal{B}_i, in_i, Exit_i, \tau_i, \mathcal{R}_i, \Lambda_i \rangle$ has the following elements.

- W_i is a finite set of *states*. $in_i \in W_i$ is an *initial state*,[2] and $Exit_i \subseteq W_i$ is a set of *exit-states*. States in $W_i \setminus Exit_i$ are called *internal states*.
- A finite set $\mathcal{B}_i$ of *boxes*. We assume that $W_1, \ldots, W_n$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$ are pairwise disjoint.
- An *indexing function* $\tau_i : \mathcal{B}_i \to \{i+1, \ldots, n\}$ that maps each box of the i-th sub-structure to a sub-structure with an index greater than i. If $\tau_i(b) = j$ we say that b *refers* to $\mathcal{S}_j$.
- A *nondeterministic transition relation* $\mathcal{R}_i \subseteq (\bigcup_{b \in \mathcal{B}_i} (\{b\} \times Exit_{\tau_i(b)}) \cup W_i) \times (W_i \cup \mathcal{B}_i)$. Thus, when the transducer is at a state $u \in W_i$, or at an exit e of a box b, it moves either to a state $s \in W_i$, or to a box $b' \in \mathcal{B}_i$. A move to a box b' implicitly leads to the unique initial state of the sub-structure that b' refers to.
- A *labeling function* $\Lambda_i : W_i \to \Sigma_O$ that maps states to output letters.

The sub-structure $\mathcal{S}_1$ is called the *top-level* sub-structure of $\mathcal{S}$. Thus, for example, the top-level boxes of $\mathcal{S}$ are the elements of $\mathcal{B}_1$, etc. We also call in_1 the initial state of $\mathcal{S}$, and $Exit_1$ the *exits* of $\mathcal{S}$. Note that the fact that boxes can refer only to sub-structures of a greater index implies that the nesting depth of structures is finite. In contrast, in the *recursive* setting such a restriction does not exist. Also note that moves from an exit $e \in Exit_i$ of a sub-structure $\mathcal{S}_i$ are not specified by the transition relation $\mathcal{R}_i$ of $\mathcal{S}_i$, but rather by the transition relation of the sub-structures that contain boxes that refer to $\mathcal{S}_i$. The exits of $\mathcal{S}$ allow us to use it as a sub-structure of another hierarchical structure. When we say that a hierarchical structure $\mathcal{S} = \langle \Sigma_O, \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle \rangle$ is a sub-structure of another hierarchical structure $\mathcal{S}' = \langle \Sigma_O, \langle \mathcal{S}'_1, \ldots, \mathcal{S}'_{n'} \rangle \rangle$, we mean that $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\} \subseteq \{\mathcal{S}'_2, \ldots, \mathcal{S}'_{n'}\}$. The size $|\mathcal{S}_i|$ of a sub-structure $\mathcal{S}_i$ is the sum $|W_i| + |\mathcal{B}_i| + |\mathcal{R}_i|$. The size $|\mathcal{S}|$ of $\mathcal{S}$ is the sum of the sizes of its sub-structures. We sometimes abuse notation and refer to the *hierarchical structure* $\mathcal{S}_i$ which is formally the hierarchical structure $\langle \Sigma_O, \langle \mathcal{S}_i, \mathcal{S}_{i+1}, \ldots, \mathcal{S}_n \rangle \rangle$ obtained by taking $\mathcal{S}_i$ to be the top-level sub-structure. The special case of a hierarchical structure with a single sub-structure with no boxes and no exits is simply the classical *Kripke structure*, and we denote it by $\mathcal{S} = \langle \Sigma_O, W, in, \mathcal{R}, \Lambda \rangle$.

3.2. Transducers

A *hierarchical transducer* (alternatively, *hierarchical Moore machine*) can be viewed as a hierarchical structure with the addition of an input alphabet that determines which transition has to be taken from each state and box exit. Unlike hierarchical structures which are nondeterministic, a hierarchical transducer has a deterministic transition function. For ease of exposition, we also forbid internal moves from exit nodes.

Formally, a hierarchical transducer is a tuple $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$ with $\mathcal{K}_i = \langle W_i, \mathcal{B}_i, in_i, Exit_i, \tau_i, \delta_i, \Lambda_i \rangle$, where Σ_O and the elements $W_i, \mathcal{B}_i, in_i, Exit_i, \tau_i, \Lambda_i$ of each *sub-transducer* $\mathcal{K}_i$ are as in a hierarchical structure, Σ_I is a non-empty set of input letters, and for every $1 \leqslant i \leqslant n$ the element $\delta_i : (\bigcup_{b \in \mathcal{B}_i} (\{b\} \times Exit_{\tau_i(b)}) \cup (W_i \setminus Exit_i)) \times \Sigma_I \to W_i \cup \mathcal{B}_i$ is a *transition function*. Thus, when the transducer is at an internal state $u \in (W_i \setminus Exit_i)$, or at an exit e of a box b, and it reads an input letter $\sigma \in \Sigma_I$, it moves either to a state $s \in W_i$, or to a box $b' \in \mathcal{B}_i$. As for hierarchical structures, a move to a box b' implicitly leads to the unique initial state of the sub-transducer that b' refers to. The size $|\mathcal{K}_i|$ of a sub-transducer $\mathcal{K}_i$ is the sum $|W_i| + |\mathcal{B}_i| + |\delta_i|$. The special case of a hierarchical transducer with a single sub-transducer with no boxes and no exits is simply the classical *Moore machine*, and we denote it by $\mathcal{K} = \langle \Sigma_I, \Sigma_O, W, in, \delta, \Lambda \rangle$.

Observe that in the definitions above of hierarchical structures and transducers we do not allow boxes as initial states. An alternative definition allows a box b to serve as an initial state of a sub-transducer $\mathcal{K}_i$, in which case the entry point to that transducer is the initial state of the sub-transducer $\mathcal{K}_j$ that b refers to. Note that this process may have to be repeated if $\mathcal{K}_j$ itself designates a box as its entry point, but due to the hierarchical nesting of transducers this process would terminate yielding a (simple) state in at most n steps. Also note that our definition of the transition function of a hierarchical transducer does not allow an exit $e \in Exit_i$ to have internal outgoing edges inside the transducer $\mathcal{K}_i$. Indeed, if b is a box of $\mathcal{K}_j$ that refers to $\mathcal{K}_i$ then the transition function of $\mathcal{K}_j$ specifies where the computation should continue when it reaches the exit e in the context of the box b. An alternative definition is to allow an exit $e \in Exit_i$ to have internal transitions inside $\mathcal{K}_i$ on some letters in Σ_I, and behave as an exit only with respect to the remaining letters in Σ_I. More formally, one can define the transition function δ_i to be a partial function $\delta_i : (\bigcup_{b \in \mathcal{B}_i} (\{b\} \times Exit_{\tau_i(b)}) \cup W_i) \times \Sigma_I \to W_i \cup \mathcal{B}_i$, such that for every exit $e \in Exit_i$, there is at least one letter $\sigma \in \Sigma_I$ for which $\delta_i(e, \sigma)$ is not defined (i.e., an exit cannot have

[2] We assume a single entry for each sub-structure. Multiple entries can be handled by duplicating sub-structures.

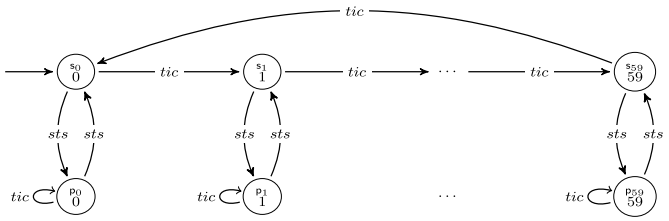


Fig. 1. The seconds-counter transducer $\mathcal{K}_{sc}$.

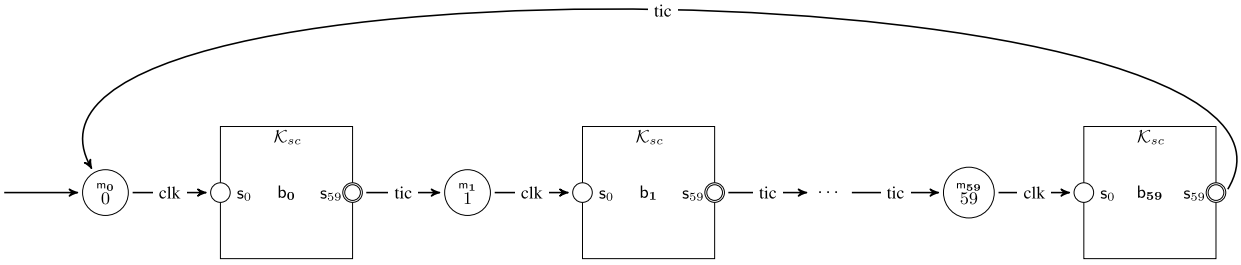


Fig. 2. The minutes-counter transducer $\mathcal{K}_{mc}$.

all its transitions remain inside $\mathcal{K}_i$), and for every $j < i$, every box $b \in \mathcal{B}_j$ that refers to $\mathcal{K}_i$, and every $\sigma \in \Sigma_I$, we have that $\delta_j((b, e), \sigma)$ is defined iff $\delta_i(e, \sigma)$ is not defined. Obviously, for every state $s \in W_i \setminus Exit_i$ and every $\sigma \in \Sigma_I$ we require that $\delta_i(s, \sigma)$ be defined. When constructing actual transducers it is easier to use the less restrictive definitions above, and we do so along the examples. However, except for these examples, in the rest of the paper we use the more restrictive definitions given before. The reason is that we believe the reader will benefit much more from having the constructions and proofs not burdened by the extra technicalities that the more permissive definitions entail, as they are easy to add once the core idea is grasped.

Example 3.1 *(Chronograph).* We now give an example of a two-level hierarchical transducer modeling a chronograph with a display of 60 minutes and 60 seconds and the capability to be paused.[3] The chronograph input signals are $\{tic, sts, clk\}$. The *tic* signal is given once a second by an external oscillator, and is used to drive the counting; the *sts* signal is a "start and stop" signal and it switches the chronograph back and forth from processing to ignoring the *tic* signals; and the *clk* signal is simply the system clock signal. For simplicity, we make the (very reasonable) assumptions that the hardware is such that the system clock is orders of magnitude faster than the *tic* oscillator (which is derived from it), and that *sts* signals from the user are not generated at least for one clock cycle after a *tic*. We start by describing the low level component of the chronograph, which is a transducer counting from 0 to 59 seconds.

The *seconds-counter transducer* $\mathcal{K}_{sc}$ given in Fig. 1 is simply a Moore machine with 120 states that counts the number of *tic* signals received so far. The basic counting is handled by 60 states numbered from s_0 (which is also the initial state of the transducer) to s_{59}, each of which is labeled by an output signal in $0, \ldots, 59$ encoding (in a way suitable for the seconds' display module) the number of passed seconds. For every $i \in \{0, \ldots, 59\}$, the *sts* signal pauses/un-pauses the counter by forcing a move from a state s_i to its paused counterpart p_i and vice versa. This transducer makes no special use of the system clock, and thus every state has a *clk* self loop (which we do not draw in Fig. 1). Formally, the seconds-counter transducer is the Moore machine $\mathcal{K}_{sc} = \langle \Sigma_I, \Sigma_O, W, in, \delta, \Lambda \rangle$, where $\Sigma_I = \{tic, sts, clk\}$, $\Sigma_O = \{0, \ldots, 59\}$, $W = \{s_i, p_i \mid 0 \leqslant i < 60\}$, $in = \mathsf{s}_0$, and both the transition function δ and the labeling function Λ are given in Fig. 1. Note that the seconds-counter transducer has no exit states since the definition of a transducer we use is geared towards describing autonomous fully-independent components which have an internal transition from every state on every possible input (which is necessarily not the case with exit states).

The *minutes-counter transducer* $\mathcal{K}_{mc}$, given in Fig. 2 is a hierarchical transducer containing 60 states and 60 boxes $\mathsf{b}_0, \ldots, \mathsf{b}_{59}$, all referring to the same sub-transducer $\mathcal{K}_{sc}$, which is the seconds-counter from Fig. 1 with the state s_{59} serving as an exit with respect to the *tic* signal.[4] Thus, the transition inside $\mathcal{K}_{sc}$ whose source is state s_{59} and is labeled by *tic* is removed. This is done because we need to connect one structure to the following one without introducing nondeterminism.

[3] A natural way to build a chronograph using flip–flops and combinatorial logic is to have one counter counting from 0 to 59 seconds using a clock that ticks once a second, and another counter that counts from 0 to 59 minutes using as its clock the carry (or overflow) flag of the first counter. Thus, the input signals to the minutes counter are derived from the output signals of the seconds counter. Unfortunately, this kind of synthesis (called *data flow synthesis*) is known to be undecidable already for the very restricted case of LTL specifications and systems that are merely pipelines [39].

[4] The synthesis algorithm we present in this article automatically takes care of designating certain states as exits when it makes use of a library transducer with no exits (like the seconds-counter) while constructing another transducer (like the minutes-counter). Note that for this example we use a

The updating of the minutes' display is handled by 60 states numbered from m_0 (which is also the initial state of the transducer) to m_{59}, each of which is labeled by a set of output signals $\Sigma_O = 0, \ldots, 59$ encoding the number of passed minutes. When the computation is in the state s_{59} of a box b_i and it receives a *tic* signal, it exits the box and enters the state m_{i+1}, which increments the minutes display. At the next system clock signal *clk* the computation enters box b_{i+1} that resets the seconds display and starts counting the 59 seconds. Note that, by our assumptions on the hardware, we can safely assume that while in state m_{i+1} there is no need to process *tic* and *sts* signals. However, for completeness, one can assume they are handled by a self loop (which we do not draw).

It is not hard to see how one can use the minutes-counter transducer as a sub-transducer of a more elaborate chronograph capable of counting up to 24 hours, and then use that as a sub-transducer for a chronograph that also counts days, etc.

3.3. Flat expansions

A sub-transducer without boxes is *flat*. A hierarchical transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle W, \emptyset, in, Exit, \emptyset, \delta, \Lambda \rangle \rangle$ with a single (hence flat) sub-transducer is flat, and we denote it using the shorter notation $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle W, in, Exit, \delta, \Lambda \rangle \rangle$. Each hierarchical transducer $\mathcal{K}$ can be transformed into an equivalent flat transducer $\mathcal{K}^f = \langle \Sigma_I, \Sigma_O, \langle W^f, in_1, Exit_1, \delta^f, \Lambda^f \rangle \rangle$ (called its *flat expansion*) by recursively substituting each box by a copy of the sub-transducer it refers to. Since different boxes can refer to the same sub-transducer, states may appear in different contexts. In order to obtain unique names for states in the flat expansion, we prefix each copy of a sub-transducer's state by the sequence of boxes through which it is reached. Thus, a state $(b_0, \ldots, b_k, w)$ of $\mathcal{K}^f$ is a vector whose last component w is a state in $\bigcup_{i=1}^{n} W_i$, and the remaining components $(b_0, \ldots, b_k)$ are boxes that describe its context. The labeling of a state $(b_0, \ldots, b_k, w)$ is determined by its last component w. For simplicity, we refer to vectors of length one as elements (that is, w, rather than (w)).[5] Formally, given a hierarchical transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$, for each sub-transducer $\mathcal{K}_i = \langle W_i, \mathcal{B}_i, in_i, Exit_i, \tau_i, \delta_i, \Lambda_i \rangle$ we inductively define its flat expansion $\mathcal{K}_i^f = \langle W_i^f, in_i, Exit_i, \delta_i^f, \Lambda_i^f \rangle$ as follows.

- The set of states $W_i^f \subseteq W_i \cup (\mathcal{B}_i \times (\bigcup_{j=i+1}^{n} W_j^f))$ is defined as follows.
  - For all states w of W_i, put w into W_i^f.
  - For all boxes b of $\mathcal{K}_i$, such that $\tau_i(b) = j$ and the tuple $(u_1, \ldots, u_h)$ is a state in W_j^f, the tuple $(b, u_1, \ldots, u_h)$ is a state in W_i^f.
- The transition function δ_i^f is defined as follows.
  - For all $u \in W_i$, or $u = (b, e)$ with $b \in \mathcal{B}_i$ and $e \in Exit_{\tau_i(b)}$, let $v = \delta_i(u, \sigma)$. If v is a state, set $\delta_i^f(u, \sigma) = v$, otherwise, if v is a box, set $\delta_i^f(u, \sigma) = (v, in_{\tau_i(v)})$. Note that $(v, in_{\tau_i(v)})$ is indeed a state of W_i^f by the second item in the definition of states above.
  - For all boxes b of $\mathcal{K}_i$, where $\delta^f_{\tau_i(b)}((u_1, \ldots, u_h), \sigma) = (v_1, \ldots, v_{h'})$ is a transition of $\mathcal{K}^f_{\tau_i(b)}$, set the transition of $\mathcal{K}_i^f$ to be $\delta_i^f((b, u_1, \ldots, u_h), \sigma) = (b, v_1, \ldots, v_{h'})$.
- Finally, the labeling Λ_i^f is defined as follows.
  - For all $u \in W_i$, set $\Lambda_i^f(u) = \Lambda_i(u)$.
  - For all $u \in W_i^f$ such that $u = (b, u_1, \ldots, u_h)$ with $b \in \mathcal{B}_i$, set $\Lambda_i(u) = \Lambda^f_{\tau_i(b)}(u_1, \ldots, u_h)$.

The transducer $\langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1^f \rangle \rangle$ is the required flat expansion $\mathcal{K}^f$ of $\mathcal{K}$. An *atomic transducer* is a flat transducer made up of a single node that serves as both an entry and an exit. For each letter $\varsigma \in \Sigma_O$ there is an atomic transducer $K_\varsigma = \langle \{p\}, p, \{p\}, \emptyset, \{(p, \varsigma)\} \rangle$ whose single state p is labeled by ς.

The definition of a flat expansion $\mathcal{S}^f = \langle \Sigma_O, W^f, in, \mathcal{R}^f, \Lambda^f \rangle$ of a hierarchical structure $\mathcal{S}$, can be obtained by the natural modifications to the definition of the flat expansion of a transducer (see also [7]). Observe that the flat expansion $\mathcal{S}^f$ of a hierarchical structure $\mathcal{S}$ is a Kripke structure, which can be unwound into a tree $\mathcal{T}_\mathcal{S} = \langle T_\mathcal{S}, V_\mathcal{S} \rangle$. We call $\mathcal{T}_\mathcal{S}$ the *unwinding* of $\mathcal{S}$. Formally, $\mathcal{T}_\mathcal{S}$ is a Σ_O-labeled W^f-tree, where a node y in the tree has a son $y \cdot d'$ for every d' for which there is a transition $(last(y), d') \in \mathcal{R}^f$. The label of a node $y \neq \varepsilon$ is $V_\mathcal{S}(y) = \Lambda^f(last(y))$, and $V_\mathcal{S}(\varepsilon) = \Lambda^f(in)$.

3.4. Run of a transducer

Consider a hierarchical transducer $\mathcal{K}$ with $Exit_1 = \emptyset$ that interacts with its environment. At point j in time, the environment provides $\mathcal{K}$ with an input $\sigma_j \in \Sigma_I$, and in response $\mathcal{K}$ moves to a new state, according to its transition relation, and

definition of hierarchical transducers that allows states to maintain their internal transitions on some input signals, and act as exits only with respect to the remaining signals.

[5] A helpful way to think about this is using a stack – the boxes $b_0, \ldots, b_k$ are pushed into the stack whenever a sub-transducer is called, and are popped in the corresponding exit.

outputs the label of that state. The result of this infinite interaction is a computation of $\mathcal{K}$, called the *trace* of the run of $\mathcal{K}$ on the word $\sigma_1 \cdot \sigma_2 \cdots$. In the case that $Exit_1 \neq \emptyset$, the interaction comes to a halt whenever $\mathcal{K}$ reaches an exit $e \in Exit_1$, since top-level exits have no outgoing transitions. Formally, a *run* of a hierarchical transducer $\mathcal{K}$ is defined by means of its flat expansion $\mathcal{K}^{\mathrm{f}}$. Given a finite input word $v = \sigma_1 \cdots \sigma_m \in \Sigma_I^*$, a *run* (*computation*) of $\mathcal{K}$ on v is a sequence of states $r = r_0 \cdots r_m \in (W^{\mathrm{f}})^*$ such that $r_0 = in_1$, and $r_j = \delta^{\mathrm{f}}(r_{j-1}, \sigma_j)$, for all $0 < j \leqslant m$. Note that since $\mathcal{K}$ is deterministic it has at most one run on every word, and that if $Exit_1 \neq \emptyset$ then $\mathcal{K}$ may not have a run on some words. The *trace* of the run of $\mathcal{K}$ on v is the word of inputs and outputs $\mathrm{trc}(\mathcal{K}, v) = (\Lambda^{\mathrm{f}}(r_1), \sigma_1) \cdots (\Lambda^{\mathrm{f}}(r_m), \sigma_m) \in (\Sigma_O \times \Sigma_I)^*$. The notions of traces and runs are extended to infinite words in the natural way.

The computations of $\mathcal{K}$ can be described by a *computation tree* whose branches correspond to the runs of $\mathcal{K}$ on all possible inputs, and whose labeling gives the traces of these runs. Note that the root of the tree corresponds to the empty word ε, and its labeling is not part of any trace. However, if we look at the computation tree of $\mathcal{K}$ as a sub-tree of a computation tree of a transducer $\mathcal{K}'$ of which $\mathcal{K}$ is a sub-transducer, then the labeling of the root of the computation tree of $\mathcal{K}$ is meaningful, and it corresponds to the last element in the trace of the run of $\mathcal{K}'$ leading to the initial state of $\mathcal{K}$. Formally, given $\sigma \in \Sigma_I$, the computation tree $\mathcal{T}_{\mathcal{K},\sigma} = \langle T_{\mathcal{K},\sigma}, V_{\mathcal{K},\sigma} \rangle$, is a $(\Sigma_O \times \Sigma_I)$-labeled $(W^{\mathrm{f}} \times \Sigma_I)$-tree, where: (i) the root ε is labeled by $(\Lambda^{\mathrm{f}}(in_1), \sigma)$; (ii) a node $y = (r_1, \sigma_1) \cdots (r_m, \sigma_m) \in (W^{\mathrm{f}} \times \Sigma_I)^+$ is in $T_{\mathcal{K},\sigma}$ iff $in_1 \cdot r_1 \cdots r_m$ is the run of $\mathcal{K}$ on $v = \sigma_1 \cdots \sigma_m$, and its label is $V_{\mathcal{K},\sigma}(y) = (\Lambda^{\mathrm{f}}(r_m), \sigma_m)$. Thus, for a node y, the labels of the nodes on the path from the root (excluding the root) to y are exactly $\mathrm{trc}(\mathcal{K}, v)$. Observe that the leaves of $\mathcal{T}_{\mathcal{K},\sigma}$ correspond to pairs (e, σ'), where $e \in Exit_1$ and $\sigma' \in \Sigma_I$. However, if $Exit_1 = \emptyset$, then the tree has no leaves, and it represents the runs of $\mathcal{K}$ over all words in Σ_I^*. We sometimes consider a leaner computation tree $\mathcal{T}_{\mathcal{K}} = \langle T_{\mathcal{K}}, V_{\mathcal{K}} \rangle$ that is a Σ_O-labeled Σ_I-tree, where a node $y \in \Sigma_I^+$ is in $T_{\mathcal{K}}$ iff there is a run r of $\mathcal{K}$ on y. The label of such a node is $V_{\mathcal{K}}(y) = \Lambda^{\mathrm{f}}(last(r))$ and the label of the root is $\Lambda^{\mathrm{f}}(in_1)$. Observe that for every $\sigma \in \Sigma_I$, the tree $\mathcal{T}_{\mathcal{K}}$ can be obtained from $\mathcal{T}_{\mathcal{K},\sigma}$ by simply deleting the first component of the directions of $\mathcal{T}_{\mathcal{K},\sigma}$, and the second component of the labels of $\mathcal{T}_{\mathcal{K},\sigma}$.

Recall that the labeling of the root of a computation tree of $\mathcal{K}$ is not part of any trace (when it is not a sub-tree of another tree). Hence, in the definition below, we arbitrarily fix some letter $\varrho \in \Sigma_I$. Given a temporal logic formula φ, over the atomic propositions AP where $2^{AP} = \Sigma_O \times \Sigma_I$, we have the following:

Definition 3.1. A hierarchical transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$, with $Exit_1 = \emptyset$, satisfies a formula φ (written $\mathcal{K} \models \varphi$) iff the tree $\mathcal{T}_{\mathcal{K},\varrho}$ satisfies φ, for an arbitrary fixed letter $\varrho \in \Sigma_I$.

Observe that given φ, finding a flat transducer $\mathcal{K}$ such that $\mathcal{K} \models \varphi$ is the classic synthesis problem studied (for LTL formulas) in [47].

4. Hierarchical synthesis

In this section we describe our algorithm for bottom-up synthesis of a hierarchical transducer from a library of hierarchical transducers. For our purpose, a *library* $\mathcal{L}$ is simply a finite set of hierarchical transducers with the same input and output alphabets. Formally, $\mathcal{L} = \{\mathcal{K}^1, \ldots, \mathcal{K}^\lambda\}$, and for every $1 \leqslant i \leqslant \lambda$, we have that $\mathcal{K}^i = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}^i_1, \ldots, \mathcal{K}^i_{n_i} \rangle \rangle$. Note that a transducer in the library can be a sub-transducer of another one, or share common sub-transducers with it. The set of transducers in $\mathcal{L}$ that have no top-level exits is denoted by $\mathcal{L}^{=\emptyset} = \{\mathcal{K}^i \in \mathcal{L}\colon Exit^i_1 = \emptyset\}$, and its complement is $\mathcal{L}^{\neq\emptyset} = \mathcal{L} \setminus \mathcal{L}^{=\emptyset}$.

The synthesis algorithm is provided with an initial library $\mathcal{L}_0$ of hierarchical transducers. A good starting point is to include in $\mathcal{L}_0$ all the atomic transducers, as well as any other relevant hierarchical transducers, for example from a standard library. Obviously, the choice of the initial library is entirely in the hands of the designer. We then proceed by synthesizing in rounds. At each round $i > 0$, the system designer provides a specification formula φ_i of the currently desired hierarchical transducer $\mathcal{K}^i$, which is then automatically synthesized using the transducers in $\mathcal{L}_{i-1}$ as possible sub-transducers. Once a new transducer is synthesized it is added to the library, to be used in subsequent rounds. Technically, the hierarchical transducer synthesized in the last round is the output of the algorithm.

Input: An initial library $\mathcal{L}_0$, and a list of specification formulas $\varphi_1, \ldots, \varphi_m$
Output: A hierarchical transducer satisfying φ_m
for $i = 1$ **to** m **do**
  synthesize $\mathcal{K}^i$ satisfying φ_i using the transducers in $\mathcal{L}_{i-1}$ as sub-transducers
  $\mathcal{L}_i \leftarrow \mathcal{L}_{i-1} \cup \{\mathcal{K}^i\}$
end
return $\mathcal{K}^m$

Algorithm 1: Hierarchical synthesis algorithm.

The main challenge in implementing the above bottom-up hierarchical synthesis algorithm is of course coming up with an algorithm for performing the synthesis step of a single round. As noted in Section 1, a transducer that was synthesized in a previous round has no top-level exits, which severely limits its ability to serve as a sub-transducer of another transducer. Our single-round algorithm must therefore address the problem of synthesizing exits for such transducers. In Section 4.1

we give our algorithm for single-round synthesis of a hierarchical transducer from a library of hierarchical transducers, and present the core proof of its correctness; the remaining details of this proof, which are based on a game-theoretic approach, are given in Section 5. In Section 4.2 we address the problem of enforcing modularity, and add some more information regarding the synthesis of exits. Finally, in Section 6, we address the problem of hierarchical synthesis with imperfect information.

4.1. Single-round synthesis algorithm

We now formally present the problem of hierarchical synthesis from a library (that may have transducers without top-level exits) of a single temporal logic formula. Given a transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle \in \mathcal{L}^{=\emptyset}$, where $\mathcal{K}_1 = \langle W_1, \mathcal{B}_1, in_1, \emptyset, \tau_1, \delta_1, \Lambda_1 \rangle$, and a set $E \subseteq W_1$, the transducer $\mathcal{K}^E$ is obtained from $\mathcal{K}$ by setting E to be the set of top-level exits, and removing all the outgoing edges from states in E. Formally, $\mathcal{K}^E = \langle \Sigma_I, \Sigma_O, \langle \langle W_1, \mathcal{B}_1, in_1, E, \tau_1, \delta_1', \Lambda_1 \rangle, \mathcal{K}_2, \ldots, \mathcal{K}_n \rangle \rangle$, where the transition relation δ_1' is the restriction of δ_1 to sources in $W_1 \setminus E$. For convenience, given a transducer $\mathcal{K} \in \mathcal{L}^{\neq\emptyset}$ we sometimes refer to it as $\mathcal{K}^{Exit_1}$. For every $\mathcal{K} \in \mathcal{L}$, we assume some fixed ordering on the top-level states of $\mathcal{K}$, and given a set $E \subseteq W_1$, and a state $e \in E$, we denote by $idx(e, E)$ the relative position of e in E, according to this ordering. Given a library $\mathcal{L}$, and an upper bound $el \in \mathbb{N}$ on the number of allowed top-level exits, we let $\mathcal{L}^{el} = \mathcal{L}^{\neq\emptyset} \cup \{\mathcal{K}^E\colon\ \mathcal{K} \in \mathcal{L}^{=\emptyset} \wedge |E| \leqslant el\}$. The higher the number el, the more exits the synthesis algorithm is allowed to synthesize, and the longer it may take to run. As we show later, el should be at most polynomial[6] in the size of φ. In general, we assume that el is never smaller than the number of exits in any sub-transducer of any hierarchical transducer in $\mathcal{L}$. Hence, for every $\mathcal{K}^E \in \mathcal{L}^{el}$ and every $e \in E$, we have that $1 \leqslant idx(e, E) \leqslant el$.

Definition 4.1. Given a library $\mathcal{L}$ and a bound $el \in \mathbb{N}$, we say that:

- A hierarchical transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots \mathcal{K}_n \rangle \rangle$ is $\langle \mathcal{L}, el \rangle$-*composed* if (i) for every $2 \leqslant i \leqslant n$, we have that $\mathcal{K}_i \in \mathcal{L}^{el}$; (ii) if $w \in W_1$ is a top-level state, then the atomic transducer $K_{\Lambda_1(w)}$ is in $\mathcal{L}$.
- A formula φ is $\langle \mathcal{L}, el \rangle$-*realizable* iff there is an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer $\mathcal{K}$ that satisfies φ. The $\langle \mathcal{L}, el \rangle$-synthesis problem is to find such a $\mathcal{K}$.

Intuitively, an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer $\mathcal{K}$ is built by synthesizing its top-level sub-transducer $\mathcal{K}_1$, which specifies how to connect boxes that refer to transducers from $\mathcal{L}^{el}$. To eliminate an unnecessary level of indirection, boxes that refer to atomic transducers are replaced by regular states.[7]

Note that for each transducer $\mathcal{K}' \in \mathcal{L}^{=\emptyset}$ we can have as many as $\Omega(|\mathcal{K}'|)^{el}$ copies of $\mathcal{K}'$ in $\mathcal{L}^{el}$, each with a different set of exit states. In Section 4.2 we show how, when we synthesize $\mathcal{K}$, we can limit the number of such copies that $\mathcal{K}$ uses to any desired value (usually one per $\mathcal{K}'$).

4.1.1. Connectivity trees

In the classical automata-theoretic approach to synthesis [47], synthesizing a system is reduced to the problem of finding a regular tree that is a witness to the non-emptiness of a tree automaton running on computation trees. At first glance, it looks like extending this approach to solve our hierarchical synthesis problem may not be too hard: if we can build an automaton that only accepts a computation tree if it corresponds to an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer then, by taking the product of this automaton with an automaton that only accepts models of the specification formula, our synthesis problem reduces to finding a (regular) witness to the non-emptiness of the product automaton. Unfortunately, this natural extension fails since it is impossible to construct an automaton that can detect if the computation tree specifies the moves from exits of one transducer to the entry of another transducer in a consistent way. The difficulty is that the destination to move to from an exit depends only on the exit and the next input letter, and is independent of *how* that exit was reached (inside the current transducer). Thus, all paths in the computation tree that reach the same exit of the current transducer must agree on the next transducer to move to (on the same input letter). Observe that such paths may have nodes that are arbitrarily far apart since they correspond to moves of the transducer on different input words. Indeed, it is not hard to come up with an example where two disjoint infinite paths must agree on infinitely many moves from exits to entrances. This infinite amount of synchronization provides very strong evidence that trying to solve the problem by adding extra annotations to the computation tree, and/or by having copies of the automaton that read different paths coordinate by carrying with them identical guesses, would also fail.

We thus solve the (single-round) hierarchical synthesis problem by reducing it to the non-emptiness problem of a tree automaton whose input is not a computation tree, but rather a system description in the form of a *connectivity tree* (inspired

[6] In practical terms, the exits of a sub-module represent its set of possible return values. Since finite state modules are usually not expected to have return values over large domains (such as the set of integers), we believe that our polynomial bound for el is not too restrictive.

[7] When using our strict definition of a hierarchical transducer where the initial state is indeed a state and not a box, one has to assume that the library has at least one atomic transducer for use by the initial state. Obviously, if one chooses to allow boxes as initial states (as done in the chronograph example), this requirement becomes superfluous.

by the "control-flow" trees of [39]), which describes how to connect library components in a way that satisfies the specification formula. Specifically, given a library $\mathcal{L} = \{\mathcal{K}^1, \ldots, \mathcal{K}^\lambda\}$ and a bound $el \in \mathbb{N}$, connectivity trees represent hierarchical transducers that are $\langle \mathcal{L}, el \rangle$-composed, in the sense that every $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer induces a regular connectivity tree, and vice versa.

Formally, a *connectivity tree* $\mathcal{T} = \langle T, V \rangle$ for $\mathcal{L}$ and el, is an $\mathcal{L}^{el}$-labeled complete $(\{1, \ldots, el\} \times \Sigma_I)$-tree, where the root is labeled by an atomic transducer. Intuitively, a node x with $V(x) = \mathcal{K}^E$ represents a top-level state q if $\mathcal{K}^E$ is an atomic transducer, and otherwise it represents a top-level box b that refers to $\mathcal{K}^E$. The label of a son $x \cdot (idx(e, E), \sigma)$ specifies the destination of the transition from the exit e of b (or from a state q, if $\mathcal{K}^E$ is atomic – in which case it has a single exit) when reading σ. Sons $x \cdot (i, \sigma)$, for which $i > |E|$, are ignored. Thus, a path $\pi = (i_0, \sigma_0) \cdot (i_1, \sigma_1) \cdots$ in a connectivity tree $\mathcal{T}$ is called *meaningful*, iff for every $j > 0$, we have that i_j is not larger than the number of top-level exits of $V(i_{j-1}, \sigma_{j-1})$. A connectivity tree $\mathcal{T} = \langle T, V \rangle$ is *regular* if there is a flat transducer $\mathcal{M} = \langle \{1, \ldots, el\} \times \Sigma_I, \mathcal{L}^{el}, \langle M, m_0, \emptyset, \delta^{\mathcal{T}}, \Lambda^{\mathcal{T}} \rangle \rangle$, such that $\mathcal{T}$ is equal to the (lean) computation tree $\mathcal{T}_{\mathcal{M}}$.

Lemma 4.1. *Every $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer induces a regular connectivity tree, and every regular connectivity tree for $\mathcal{L}$ and el induces an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer.*

Proof. For the first direction, let $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$, where $\mathcal{K}_1 = \langle W_1, \mathcal{B}_1, in_1, \tau_1, \delta_1, \Lambda_1 \rangle$, be an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer. We construct a flat transducer $\mathcal{M}$ whose computation tree $\mathcal{T}_{\mathcal{M}}$ is the required connectivity tree. The elements of $\mathcal{M}$ are as follows:

- $M = W_1 \cup \mathcal{B}_1$, and $m_0 = in_1$.
- If $w \in W_1$, then for every $\sigma \in \Sigma_I$, we have that $\delta^{\mathcal{T}}(w, (1, \sigma)) = \delta_1(w, \sigma)$, and for every $1 < i \leqslant el$ we (arbitrarily) let $\delta^{\mathcal{T}}(w, (i, \sigma)) = m_0$.
- For $b \in \mathcal{B}_1$, let $\mathcal{K}^E \in \mathcal{L}^{el}$ be the sub-transducer that b refers to. For every $\sigma \in \Sigma_I$, if $1 \leqslant i \leqslant |E|$ then $\delta^{\mathcal{T}}(b, (i, \sigma)) = \delta_1((b, e), \sigma)$, where $e \in E$ is such that $idx(e, E) = i$; and if $|E| < i \leqslant el$ then we (arbitrarily) let $\delta^{\mathcal{T}}(b, (i, \sigma)) = m_0$.
- For $w \in W_1$ we have that $\Lambda^{\mathcal{T}}(w) = K_\varsigma$, where $\Lambda_1(w) = \varsigma$.
- For $b \in \mathcal{B}_1$ we have that $\Lambda^{\mathcal{T}}(b) = \mathcal{K}_{\tau_1(b)}$.

Recall that a son $y \cdot (i, \sigma)$, of a node y in a connectivity tree $\mathcal{T} = \langle T, V \rangle$, is meaningless if i is larger than the number of exits of the transducer $V(y)$. Hence, our choice to direct the corresponding transitions of M to the node m_0 is arbitrary and was done only for technical completeness.

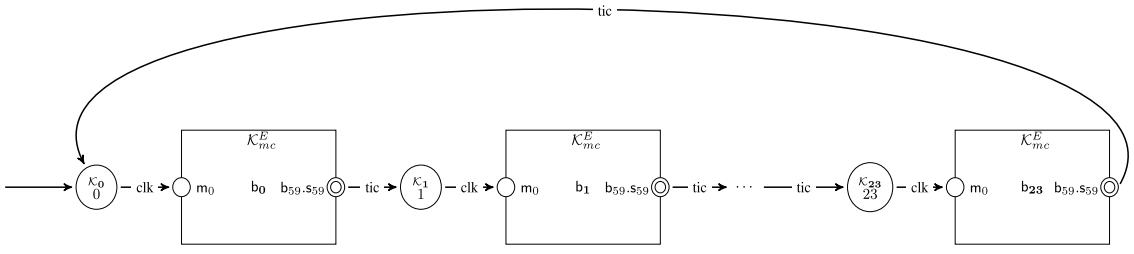
For the other direction, given a regular connectivity tree $\mathcal{T} = \langle T, V \rangle$ generated by the transducer $\mathcal{M} = \langle \{1, \ldots, el\} \times \Sigma_I, \mathcal{L}^{el}, \langle M, m_0, \emptyset, \delta^{\mathcal{T}}, \Lambda^{\mathcal{T}} \rangle \rangle$, it is not hard to see that it induces an $\langle \mathcal{L}, el \rangle$-composed hierarchical transducer $\mathcal{K}$, whose top-level sub-transducer $\mathcal{K}_1$ is basically a replica of $\mathcal{M}$. Every node $m \in M$ becomes a state of $\mathcal{K}_1$ if $\Lambda^{\mathcal{T}}(m)$ is an atomic transducer and, otherwise, it becomes a box of $\mathcal{K}_1$ which refers to the top-level sub-transducer of $\Lambda^{\mathcal{T}}(m)$. The destination of a transition from an exit e of a box m, with $\Lambda^{\mathcal{T}}(m) = \mathcal{K}^E$, when reading a letter $\sigma \in \Sigma_I$, is given by $\delta^{\mathcal{T}}(m, (idx(e, E), \sigma))$. If m is a state, then $\Lambda^{\mathcal{T}}(m)$ is an atomic transducer with a single exit and thus, the destination of a transition from m when reading a letter $\sigma \in \Sigma_I$, is given by $\delta^{\mathcal{T}}(m, (1, \sigma))$. For a box b of $\mathcal{K}_1$, let $\Lambda^{\mathcal{T}}(b) = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_{(b,1)}, \ldots, \mathcal{K}_{(b,n_b)} \rangle \rangle$, and denote by $sub(b) = \{\mathcal{K}_{(b,1)}, \ldots, \mathcal{K}_{(b,n_b)}\}$ the set of sub-transducers of $\Lambda^{\mathcal{T}}(b)$, and by $E(b)$ the set of top-level exits of $\Lambda^{\mathcal{T}}(b)$.

Formally, $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$, where $\mathcal{K}_1 = \langle W_1, \mathcal{B}_1, m_0, \tau_1, \delta_1, \Lambda_1 \rangle$, and:

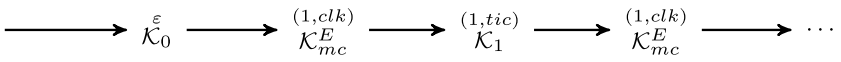
- $W_1 = \{w \in M\colon\ \exists \varsigma \in \Sigma_O$ s.t. $\Lambda^{\mathcal{T}}(w) = K_\varsigma\}$. Note that since the root of a connectivity tree is labeled by an atomic transducer then $m_0 \in W_1$.
- $\mathcal{B}_1 = M \setminus W_1$.
- The sub-transducers $\{\mathcal{K}_2, \ldots, \mathcal{K}_n\} = \bigcup_{\{b \in B_1\}} sub(b)$.
- For $b \in \mathcal{B}_1$, we have that $\tau_1(b) = i$, where i is such that $\mathcal{K}_i = \mathcal{K}_{(b,1)}$.
- For $w \in W_1$, and $\sigma \in \Sigma_I$, we have that $\delta_1(w, \sigma) = \delta^{\mathcal{T}}(w, (1, \sigma))$.
- For $b \in \mathcal{B}_1$, we have that $\delta_1((b, e), \sigma) = \delta^{\mathcal{T}}(b, (idx(e, E(b)), \sigma))$, for every $e \in E(b)$ and $\sigma \in \Sigma_I$.
- Finally, for $w \in W_1$ we have that $\Lambda_1(w) = \varsigma$, where ς is such that $\Lambda^{\mathcal{T}}(w) = K_\varsigma$. □

We now come back to the chronograph example and give a description of the connectivity tree of an extra component, the counter of hours, which is synthesized from the already described hierarchical minutes-counter transducer.

Example 4.1 *(Hours-counter transducer).* Suppose we want to synthesize an hours-counter from a library of simpler transducers, such as the minutes-counter and some atomic transducers (to be used as internal states of the desired machine). For instance, we can consider the following library $\mathcal{L} = \{\mathcal{K}_{mc}, \mathcal{K}_0, \ldots, \mathcal{K}_{23}\}$, where $\mathcal{K}_{ms}$ is the minutes-counter transducer presented in Example 3.1 and $\mathcal{K}_0, \ldots, \mathcal{K}_{23}$ represent 24 "states" for the output of hour signals (for the display module) from 0 to 23, respectively. Since $\mathcal{K}_{mc}$ can have as an exit the last state of its internal box b_{59}, the synthesis algorithm can succeed already with the bound el on the number of exits set to 1, using $\mathcal{K}^E_{mc} \in \mathcal{L}^1$, where $E = \{\mathsf{b}_{59}.\mathsf{s}_{59}\}$ with respect to



(a) The counter.



(b) Part of the connectivity tree.

Fig. 3. The hours-counter transducer $\mathcal{K}_{hc}$ and its connectivity tree.

the *tic* signal.[8] Now, once a reasonable specification of the right behavior of the hours-counter is given by a temporal logic formula, we can run our synthesis algorithm on $\mathcal{L}^1$, from which we may derive, as a possible outcome, the hierarchical transducer $\mathcal{K}_{hc}$ depicted in Part (a) of Fig. 3. Observe that all atomic transducers are depicted as classic internal states. The *hours-counter* transducer $\mathcal{K}_{hc}$ has a structure that is very similar to that of the minutes-counter $\mathcal{K}_{mc}$. It is a hierarchical machine having 24 boxes, from b_0 to b_{23}, all referring to the same sub-transducer $\mathcal{K}^E_{mc}$, plus 24 states. Each box b_i describes the elapsing of minutes within the $(i+1)$-th hour. Note that, as we did with the minutes-counter, we can safely assume that the states $\mathcal{K}_0, \ldots, \mathcal{K}_{23}$ do not have to process *tic* and *sts* signals. For the sake of completeness, one can assume self loops (which we do not draw in the figure) on these states for these signals. Part (b) of Fig. 3 shows one path of the computation tree of the hours-counter transducer (for readability, we do not show the whole tree). Each node in this path is written by giving the labeling of the node, and above it the last letter of the node's name (i.e., an input signal together with the number of the exit – which is always 1 since $el = 1$). For example, the root of the tree, representing the first component $\mathcal{K}_0$ in the hierarchical transducer $\mathcal{K}_{hc}$, has ε as its name (since at the beginning there is no input) and its labeling is $\mathcal{K}_0$. This node has a son, called $(1, clk)$, which represents the move from $\mathcal{K}_0$ when a *clk* signal is received. The node $(1, clk)$ is labeled by $\mathcal{K}^E_{mc}$ since it represents the box b_0. The edge from the node $(1, clk)$ to the son $(1, clk) \cdot (1, tic)$, which is labeled by $\mathcal{K}_1$, implies that when a computation reaches the exit $\mathsf{b}_{59}.\mathsf{s}_{59}$ of $\mathcal{K}^E_{mc}$ and a *tic* signal is received, the computation moves to the state $\mathcal{K}_1$.

4.1.2. From synthesis to automata emptiness

Given a library $\mathcal{L} = \{\mathcal{K}^1, \ldots, \mathcal{K}^\lambda\}$, a bound $el \in \mathbb{N}$, and a temporal logic formula φ, our aim is to build an APT $\mathcal{A}^{\mathcal{T}}_{\varphi}$ such that it accepts a regular connectivity tree $\mathcal{T} = \langle T, V \rangle$ iff it induces a hierarchical transducer $\mathcal{K}$ such that $\mathcal{K} \models \varphi$. Recall that by Definition 3.1 and Theorem 2.1, $\mathcal{K} \models \varphi$ iff $\mathcal{T}_{\mathcal{K},\mathcal{Q}}$ is accepted by the SAPT $\mathcal{A}_\varphi$. The basic idea is thus to have $\mathcal{A}^{\mathcal{T}}_{\varphi}$ simulate all possible runs of $\mathcal{A}_\varphi$ on $\mathcal{T}_{\mathcal{K},\mathcal{Q}}$. Unfortunately, since $\mathcal{A}^{\mathcal{T}}_{\varphi}$ has as its input not $\mathcal{T}_{\mathcal{K},\mathcal{Q}}$, but the connectivity tree $\mathcal{T}$, this is not a trivial task. In order to see how we can solve this problem, we first have to make the following observation.

Let $\mathcal{T} = \langle T, V \rangle$ be a regular connectivity tree, and let $\mathcal{K}$ be the hierarchical transducer that it induces. Consider a node u in the computation tree $T_{\mathcal{K},\mathcal{Q}}$ which corresponds to a point along a computation where $\mathcal{K}$ just enters a top level box b (or state[9]). That is, $last(u) = ((b, in_{\tau_1(b)}), \sigma)$. Observe that the root of $T_{\mathcal{K},\mathcal{Q}}$ is such a node. Let $\mathcal{K}^E$ be the library sub-transducer that b refers to, and note that the sub-tree $\mathcal{T}^u$, rooted at u, represents the traces of computations of $\mathcal{K}$ that start from the initial state of $\mathcal{K}^E$, in the context of the box b. The sub-tree $prune(\mathcal{T}^u)$, obtained by pruning every path in $\mathcal{T}^u$ at the first node $\hat{u}$, with $last(\hat{u}) = ((b, e), \hat{\sigma})$ for some $e \in E$ and $\hat{\sigma} \in \Sigma_I$ (i.e., at the first point the computation reaches an exit of $\mathcal{K}^E$), represents the portions of these traces that stay inside $\mathcal{K}^E$. Note that $prune(\mathcal{T}^u)$ is essentially independent of the context b in which $\mathcal{K}^E$ appears, and is isomorphic to the computation tree $\mathcal{T}_{\mathcal{K}^E,\sigma}$ of $\mathcal{K}^E$ (the isomorphism being to simply drop the component b from every letter in the name of every node in $prune(\mathcal{T}^u)$). Moreover, every son v (in $T_{\mathcal{K},\mathcal{Q}}$), of such a leaf $\hat{u}$ of $prune(\mathcal{T}^u)$, is of the same form as u. I.e., $last(v) = ((b', in_{\tau_1(b')}), \sigma')$, where $b' = \delta_1((b, e), \sigma')$ is a top-level box (or state) of $\mathcal{K}$. Indeed, once an exit of a transducer referred to by a top level box of $\mathcal{K}$ is reached, a computation of $\mathcal{K}$ must proceed, according to the transition relation δ_1 of it's top level sub-transducer $\mathcal{K}_1$, either to a top level state or to the entrance of another top level box. It follows that $T_{\mathcal{K},\mathcal{Q}}$ is isomorphic to a concatenation of sub-trees of the form $\mathcal{T}_{\mathcal{K}^E,\sigma}$, where the transition from a leaf of one such sub-tree to the root of another is specified by the transition relation δ_1, and is thus given explicitly by the connectivity tree $\mathcal{T}$.

The last observation is the key to how $\mathcal{A}^{\mathcal{T}}_{\varphi}$ can simulate, while reading $\mathcal{T}$, all the possible runs of $\mathcal{A}_\varphi$ on $\mathcal{T}_{\mathcal{K},\mathcal{Q}}$. The general idea is as follows. Consider a node u of $T_{\mathcal{K},\mathcal{Q}}$ such that $prune(\mathcal{T}^u)$ is isomorphic to $\mathcal{T}_{\mathcal{K}^E,\sigma}$. A copy of $\mathcal{A}^{\mathcal{T}}_{\varphi}$ that reads

[8] Note that, like in the previous example, we consider a setting where boxes may also serve as exits, and only with respect to some signals.

[9] Here we think of top-level states of $\mathcal{K}$ as boxes that refer to atomic transducers.

a node y of $\mathcal{T}$ labeled by $\mathcal{K}^E$ can easily simulate, without consuming any input, all the portions of the runs of any copy of $\mathcal{A}_\varphi$ that start by reading u and remain inside $prune(\mathcal{T}^u)$. This simulation can be done by simply constructing $\mathcal{T}_{\mathcal{K}^E,\sigma}$ on the fly and running $\mathcal{A}_\varphi$ on it. For every simulated copy of $\mathcal{A}_\varphi$ that reaches a leaf $\hat{u}$ of $prune(\mathcal{T}^u)$, the automaton $\mathcal{A}_\varphi^{\mathcal{T}}$ sends copies of itself to the sons of y in the connectivity tree in order to continue the simulation of $\mathcal{A}_\varphi$ on the different sub-trees of $T_{\mathcal{K},\varrho}$ rooted at sons of $\hat{u}$. Recall that $last(\hat{u})$ is of the form $((b,e),\hat{\sigma})$, that is, $\hat{u}$ represents a point in a computation of $\mathcal{K}$ where an exit e of a top level box b is reached. Observe that for every input letter $\sigma' \in \Sigma_I$, the node $z = y \cdot (idx(e,E),\sigma')$ in the connectivity tree represents the box b' to which $\mathcal{K}$ should proceed from exit e of box b when reading σ', and the label of z is the library sub-transducer to which b' refers. Thus, the simulation of a copy of $\mathcal{A}_\varphi$ that proceeds to a son $v = \hat{u} \cdot ((b', in_{\tau_1(b')}),\sigma')$ is handled by a copy of $\mathcal{A}_\varphi^{\mathcal{T}}$ that is sent to the son $z = y \cdot (idx(e,E),\sigma')$.

Our construction of $\mathcal{A}_\varphi^{\mathcal{T}}$ implements the above idea, with one important modification. In order to obtain optimal complexity in successive rounds of Algorithm 1, it is important to keep the size of $\mathcal{A}_\varphi^{\mathcal{T}}$ independent of the size of the transducers in the library. Unfortunately, simulating the runs of $\mathcal{A}_\varphi$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$ on the fly would require an embedding of $\mathcal{K}^E$ inside $\mathcal{A}_\varphi^{\mathcal{T}}$. Recall, however, that no input is consumed by $\mathcal{A}_\varphi^{\mathcal{T}}$ while running such a simulation. Hence, we can perform these simulations off-line instead, in the process of building the transition relation of $\mathcal{A}_\varphi^{\mathcal{T}}$. Obviously, this requires a way of summarizing the possibly infinite number of runs of $\mathcal{A}_\varphi$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$, which we do by employing the concept of *summary functions* from [8]. Let $\mathcal{A}_\varphi = \langle \Sigma_O \times \Sigma_I, Q_\varphi, q_\varphi^0, \delta_\varphi, F_\varphi\rangle$, let $\mathcal{A}_\varphi^q$ be the automaton $\mathcal{A}_\varphi$ using $q \in Q$ as an initial state, and let C be the set of colors used in the acceptance condition F_φ. Following the above observations, we next turn our attention to the problem of how to effectively summarize the run $\langle T_r, r\rangle$ of $\mathcal{A}_\varphi^q$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$.

First, we define a total ordering $\succcurlyeq$ on the set of colors C by letting $c \succcurlyeq c'$ when c is better, from the point of view of the parity acceptance condition of $\mathcal{A}_\varphi$, than c'. Thus, any even color is better than all the odd colors, the larger the even color the better, and if one has to choose between two odd colors it is best to "minimize the damage" by taking the smaller odd number. Formally, $c \succcurlyeq c'$ if the following holds: if c' is even then c is even and $c \geqslant c'$; and if c' is odd then either c is even, or c is also odd and $c \leqslant c'$. For example: $4 \succcurlyeq 2 \succcurlyeq 0 \succcurlyeq 1 \succcurlyeq 3$. We denote by $\min^{\succcurlyeq}$ the operation of taking the minimal color, according to $\succcurlyeq$, of a finite set of colors.

Consider now the run tree $\langle T_r, r\rangle$ of $\mathcal{A}_\varphi^q$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$. Note that if $z \in T_r$ is a leaf, then $r(z)$ is of the form $(a \cdot (e,\sigma'), p)$ for some string a, with $p \in Q_\varphi^{\vee,\wedge}$ (i.e., p is not an ε-state), and $e \in E$. Indeed, if p is an ε-state then $\mathcal{A}_\varphi^q$ can proceed without consuming any input, and hence the run can be extended beyond z; similarly, if e is not an exit of $\mathcal{K}^E$ then it has successors inside $\mathcal{K}^E$, and again the run can be extended beyond z. Every such leaf z represents a copy of $\mathcal{A}_\varphi$ that is in state p and is reading a node of the computation tree of $\mathcal{K}^E$ whose last component is (e,σ'). It turns out (and is proved in [8]) that it is not important to remember all the colors this copy of $\mathcal{A}_\varphi$ encountered along the way to this node, but only the maximal color according to $\succcurlyeq$ (this ultimately hinges upon the fact that parity games are memoryless – see [8]). It is also not important to differentiate between two copies of $\mathcal{A}_\varphi$ that have reached two different nodes y, y' of the computation tree of $\mathcal{K}^E$ if $last(y) = last(y') = (e,\sigma')$ (thus both copies are going to read the same future input sub-tree) and both copies are in the same state p and have encountered the same maximal color. Moreover, if there are two copies of $\mathcal{A}_\varphi$ that have reached, with the same state p, two (possibly the same) nodes y, y' with $last(y) = last(y') = (e,\sigma')$, but have encountered different maximal colors c, c' where $c \succcurlyeq c'$, it is enough to remember the information of the copy that is "more behind" in its attempt to satisfy the acceptance condition. I.e., the copy that encountered c'. The intuitive reason is that since both copies are going to read the same input sub-tree from the same state, if the copy that has encountered a less favorable maximal color in the past is going to accept then the copy that encountered a more favorable color is bound to accept too.

To capture the above intuition, we define a function $g_r : E \times \Sigma_I \times Q_\varphi^{\vee,\wedge} \to C \cup \{\dashv\}$, called the *summary function* of $\langle T_r, r\rangle$, which summarizes this run. Given $h = (e,\sigma',p) \in E \times \Sigma_I \times Q_\varphi^{\vee,\wedge}$, if there is no leaf $z \in T_r$, such that $r(z)$ is of the form $(a \cdot (e,\sigma'), p)$, then $g_r(h) = \dashv$; otherwise, $g_r(h) = c$, where c is the maximal color encountered by the copy of $\mathcal{A}_\varphi$ which made the least progress towards satisfying the acceptance condition, among all copies that reach a leaf $z \in T_r$ of the form $(a \cdot (e,\sigma'), p)$. Formally, given $h = (e,\sigma',p) \in E \times \Sigma_I \times Q_\varphi^{\vee,\wedge}$, let $paths(r,h)$ be the set of all the paths in $\langle T_r, r\rangle$ that end in a leaf $z \in T_r$ with $r(z) = (a \cdot (e,\sigma'), p)$, for some a. Then, $g_r(h) = \dashv$ if $paths(r,h) = \emptyset$ and otherwise, $g_r(h) = \min^{\succcurlyeq}\{\max C(\pi) \colon \pi \in paths(r,h)\}$.

Let $Sf(\mathcal{K}^E,\sigma,q)$ be the set of summary functions of the runs of $\mathcal{A}_\varphi^q$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$. If $\mathcal{T}_{\mathcal{K}^E,\sigma}$ has no leaves, then $Sf(\mathcal{K}^E,\sigma,q)$ contains only the empty summary function $\emptyset$. For $g \in Sf(\mathcal{K}^E,\sigma,q)$, let $g^{\neq\dashv} = \{h \in E \times \Sigma_I \times Q_\varphi^{\vee,\wedge} \colon\ g(h) \neq \dashv\}$. Based on the ordering $\succcurlyeq$ we defined for colors, we can define a partial order $\succcurlyeq$ on $Sf(\mathcal{K}^E,\sigma,q)$, by letting $g \succcurlyeq g'$ if for every $h \in (E \times \Sigma_I \times Q_\varphi^{\vee,\wedge})$ the following holds: $g(h) = \dashv$, or $g(h) \neq \dashv \neq g'(h)$ and $g(h) \succcurlyeq g'(h)$. Observe that if r and r' are two non-rejecting runs, and $g_r \succ g_{r'}$, then extending r to an accepting run on a tree that extends $\mathcal{T}_{\mathcal{K}^E,\sigma}$ is always not harder than extending r' – either because $\mathcal{A}_\varphi$ has less copies at the leaves of r, or because these copies encountered better maximal colors. Given a summary function g, we say that a run $\langle T_r, r\rangle$ *achieves* g if $g_r \succcurlyeq g$; we say that g is *feasible* if there is a run $\langle T_r, r\rangle$ that achieves it; and we say that g is *relevant* if it can be achieved by a memoryless[10] run that is

[10] A run of an automaton $\mathcal{A}$ is memoryless if two copies of $\mathcal{A}$ that are in the same state, and read the same input node, behave in the same way on the rest of the input.

not rejecting (i.e., by a run that has no infinite path that does not satisfy the acceptance condition of $\mathcal{A}_\varphi$). We denote by $Rel(\mathcal{K}^E, \sigma, q) \subseteq Sf(\mathcal{K}^E, \sigma, q)$ the set of relevant summary functions.

We are now ready to give a formal definition of the automaton $\mathcal{A}^{\mathcal{T}}_\varphi$. Given a library $\mathcal{L} = \{\mathcal{K}^1, \ldots, \mathcal{K}^\lambda\}$, a bound $el \in \mathbb{N}$, and a temporal-logic formula φ, let $\mathcal{A}_\varphi = \langle \Sigma_O \times \Sigma_I, Q_\varphi, q^0_\varphi, \delta_\varphi, F_\varphi \rangle$, let $C = \{C_{\min}, \ldots, C_{\max}\}$ be the colors in the acceptance condition of $\mathcal{A}_\varphi$, and for $\mathcal{K}^E \in \mathcal{L}^{el}$, let Λ^E be the labeling function of the top-level sub-transducer of $\mathcal{K}^E$. The automaton $\mathcal{A}^{\mathcal{T}}_\varphi = \langle \mathcal{L}^{el}, (\{1, \ldots, el\} \times \Sigma_I), (\Sigma_I \times Q^{\vee,\wedge}_\varphi \times C) \cup \{q_0\}, q_0, \delta, \alpha \rangle$ has the following elements.

- For every $\mathcal{K}^E \in \mathcal{L}^{el}$ we have that $\delta(q_0, \mathcal{K}^E) = \delta((\varrho, q^0_\varphi, C_{\min}), \mathcal{K}^E)$ if $\mathcal{K}^E$ is an atomic transducer and, otherwise, $\delta(q_0, \mathcal{K}^E) = \mathbf{false}$.
- For every $(\sigma, q, c) \in \Sigma_I \times Q^{\vee,\wedge}_\varphi \times C$, and every $\mathcal{K}^E \in \mathcal{L}^{el}$, we have $\delta((\sigma, q, c), \mathcal{K}^E) = \bigvee_{g \in Rel(\mathcal{K}^E, \sigma, q)} \bigwedge_{(e, \hat{\sigma}, \hat{q}) \in g^{\neq \dashv}} \bigoplus_{\sigma' \in \Sigma_I} ((idx(e, E), \sigma'), (\sigma', \delta_\varphi(\hat{q}, (\Lambda^E(e), \hat{\sigma})), g(e, \hat{\sigma}, \hat{q})))$, where $\bigoplus = \bigwedge$ if $\hat{q} \in Q^\wedge_\varphi$, and $\bigoplus = \bigvee$ if $\hat{q} \in Q^\vee_\varphi$.
- $\alpha(q_0) = C_{\min}$; and $\alpha((\sigma, q, c)) = c$, for every $(\sigma, q, c) \in \Sigma_I \times Q^{\vee,\wedge}_\varphi \times C$.

Intuitively, $\mathcal{A}^{\mathcal{T}}_\varphi$ first checks that the root of its input tree $\mathcal{T}$ is labeled by an atomic proposition and then proceeds to simulate all the runs of $\mathcal{A}_\varphi$ on $\mathcal{T}_{\mathcal{K},\varrho}$. A copy of $\mathcal{A}^{\mathcal{T}}_\varphi$ at a state (σ, q, c), that reads a node y of $\mathcal{T}$ labeled by $\mathcal{K}^E$, considers all the non-rejecting runs of $\mathcal{A}^q_\varphi$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$, by looking at the set $Rel(\mathcal{K}^E, \sigma, q)$ of summary functions for these runs. It then sends copies of $\mathcal{A}^{\mathcal{T}}_\varphi$ to the sons of y to continue the simulation of copies of $\mathcal{A}_\varphi$ that reach the leaves of $\mathcal{T}_{\mathcal{K}^E,\sigma}$.

The logic behind the definition of $\delta((\sigma, q, c), \mathcal{K}^E)$ is as follows. Since every summary function $g \in Rel(\mathcal{K}^E, \sigma, q)$ summarizes at least one non-rejecting run, and it is enough that one such run can be extended to an accepting run of $\mathcal{A}_\varphi$ on the remainder of $\mathcal{T}_{\mathcal{K},\varrho}$, we have a disjunction on all $g \in Rel(\mathcal{K}^E, \sigma, q)$. Every $(e, \hat{\sigma}, \hat{q}) \in g^{\neq \dashv}$ represents one or more copies of $\mathcal{A}_\varphi$ at state $\hat{q}$ that are reading a leaf $\hat{u}$ of $\mathcal{T}_{\mathcal{K}^E,\sigma}$ with $last(\hat{u}) = (e, \hat{\sigma})$, and all these copies must accept their remainders of $\mathcal{T}_{\mathcal{K},\varrho}$. Hence, we have a conjunction over all $(e, \hat{\sigma}, \hat{q}) \in g^{\neq \dashv}$.

A copy of $\mathcal{A}_\varphi$ that starts at the root of $\mathcal{T}_{\mathcal{K}^E,\sigma}$ may give rise to many copies that reach a leaf $\hat{u}$ of $\mathcal{T}_{\mathcal{K}^E,\sigma}$ with $last(\hat{u}) = (e, \hat{\sigma})$, but we only need to consider the copy which made the least progress towards satisfying the acceptance condition, as captured by $g(e, \hat{\sigma}, \hat{q})$. To continue the simulation of such a copy on its remainder of $\mathcal{T}_{\mathcal{K},\varrho}$, we send a copy of $\mathcal{A}^{\mathcal{T}}_\varphi$ to a son $y \cdot (idx(e, E), \sigma')$ of y in the connectivity tree, whose label specifies where $\mathcal{K}$ should go to from the exit e when reading σ', as follows. Recall that the leaf $\hat{u}$ corresponds to a node u of $\mathcal{T}_{\mathcal{K},\varrho}$ such that $last(u) = ((b, e), \hat{\sigma})$ and b is a top-level box of $\mathcal{K}$ that refers to $\mathcal{K}^E$. Also recall that every node in $\mathcal{T}_{\mathcal{K},\varrho}$ has one son for every letter $\sigma' \in \Sigma_I$. Hence, a copy of $\mathcal{A}_\varphi$ that is at state $\hat{q}$ and is reading u, sends one copy in state $q' = \delta_\varphi(\hat{q}, (\Lambda^E(e), \hat{\sigma}))$ to each son of u, if $\hat{q} \in Q^\wedge_\varphi$; and only one such copy, to one of the sons of u, if $\hat{q} \in Q^\vee_\varphi$. This explains why $\bigoplus$ is a conjunction in the first case, and is a disjunction in the second. Finally, a copy of $\mathcal{A}^{\mathcal{T}}_\varphi$ that is sent to direction $(idx(e, E), \sigma')$ carries with it the color $g(e, \hat{\sigma}, \hat{q})$. The color assigned to q_0 is of course arbitrary.

The construction above implies the following lemma:

Lemma 4.2. *$\mathcal{A}^{\mathcal{T}}_\varphi$ accepts a regular connectivity tree $\mathcal{T} = \langle T, V \rangle$ iff $\mathcal{T}$ induces a hierarchical transducer $\mathcal{K}$, such that $\mathcal{T}_{\mathcal{K},\varrho}$ is accepted by $\mathcal{A}_\varphi$.*

Proof. The core of the proof is game-theoretic. Recall that the game-based approach to model checking a flat system $\mathcal{S}$ with respect to a branching-time temporal logic specification φ, reduces the model-checking problem to solving a game (called the *membership game* of $\mathcal{S}$ and A_φ) obtained by taking the product of $\mathcal{S}$ with the alternating tree automaton $\mathcal{A}_\varphi$ [36]. In [8], this approach was extended to hierarchical structures, and it was shown there that given a hierarchical structure $\mathcal{S}$ and an SAPT $\mathcal{A}$, one can construct a hierarchical membership game $\mathcal{G}_{\mathcal{S},\mathcal{A}}$ such that Player 0 wins $\mathcal{G}_{\mathcal{S},\mathcal{A}}$ iff the tree obtained by unwinding $\mathcal{S}$ is accepted by $\mathcal{A}$. In particular, when $\mathcal{A}$ accepts exactly all the tree models of a branching-time formula φ, the above holds iff $\mathcal{S}$ satisfies φ. Furthermore, it is shown in [8] that one can *simplify* the hierarchical membership game $\mathcal{G}_{\mathcal{S},\mathcal{A}}$, by replacing boxes of the top-level arena with gadgets that are built using Player 0 summary functions, and obtain an equivalent flat game $\mathcal{G}^{\mathrm{s}}_{\mathcal{S},\mathcal{A}}$.

Given a regular connectivity tree $\mathcal{T} = \langle T, V \rangle$ that induces a hierarchical transducer $\mathcal{K}$, we prove Lemma 4.2 by showing that the flat membership game $\mathcal{G}^{\mathrm{s}}_{\mathcal{S},\mathcal{A}_\varphi}$, where $\mathcal{S}$ is a hierarchical structure whose unwinding is the computation tree $\mathcal{T}_{\mathcal{K},\varrho}$, is equivalent to the flat membership game $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}^{\mathcal{T}}_\varphi}$, of $\mathcal{A}^{\mathcal{T}}_\varphi$ and a Kripke structure $K^{\mathcal{T}}$ whose unwinding is $\mathcal{T}$. Thus, $\mathcal{A}_\varphi$ accepts $\mathcal{T}_{\mathcal{K},\varrho}$ iff $\mathcal{A}^{\mathcal{T}}_\varphi$ accepts $\mathcal{T}$. The equivalence of these two games follows from the fact that they have isomorphic arenas and winning conditions. Consequently, our proof of Lemma 4.2 is mainly syntactic in nature, and basically amounts to constructing the structures $\mathcal{S}$ and $K^{\mathcal{T}}$, constructing the game $\mathcal{G}_{\mathcal{S},\mathcal{A}_\varphi}$, simplifying it to get $\mathcal{G}^{\mathrm{s}}_{\mathcal{S},\mathcal{A}_\varphi}$, and constructing the membership game $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}^{\mathcal{T}}_\varphi}$. The remaining details can be found in Section 5. □

We now state our main theorem.

Theorem 4.1. *The $\langle \mathcal{L}, el\rangle$-synthesis problem is* EXPTIME*-complete for a μ-calculus formula φ, and is 2*EXPTIME*-complete for an* LTL *formula (for el that is at most polynomial in $|\varphi|$ for μ-calculus, or at most exponential in $|\varphi|$ for* LTL*).*

Proof. The lower bounds follow from the same bounds for the classical synthesis problem of flat systems [34,49], and the fact that it is immediately reducible to our problem if $\mathcal{L}$ contains all the atomic transducers. For the upper bounds, since an APT accepts some tree iff it accepts some regular tree (and $\mathcal{A}_\varphi^{\mathcal{T}}$ obviously only accepts trees which are connectivity trees), by Lemma 4.2 and Theorem 2.1, we get that an LTL or a μ-calculus formula φ is $\langle \mathcal{L}, el\rangle$-realizable iff $L(\mathcal{A}_\varphi^{\mathcal{T}}) \neq \emptyset$. Checking the emptiness of $\mathcal{A}_\varphi^{\mathcal{T}}$ can be done either directly, or by first translating it to an equivalent NPT $\mathcal{A}'^{\mathcal{T}}_\varphi$. For reasons that will become apparent in Section 4.2 we choose the latter. Note that the known algorithms for checking the emptiness of an NPT are such that if $L(\mathcal{A}_\varphi^{\mathcal{T}}) \neq \emptyset$, then one can extract a regular tree in $L(\mathcal{A}_\varphi^{\mathcal{T}})$ from the emptiness checking algorithm [48]. The upper bounds follow from the analysis given below of the time required to construct $\mathcal{A}_\varphi^{\mathcal{T}}$ and check for its non-emptiness.

By Theorem 2.1, the number of states $|Q_\varphi|$ and the index k of $\mathcal{A}_\varphi$ is $|Q_\varphi| = 2^{O(|\varphi|)}$, $k = 2$ for LTL, and $|Q_\varphi| = O(|\varphi|)$, $k = O(|\varphi|)$ for μ-calculus. The most time consuming part in the construction of $\mathcal{A}_\varphi^{\mathcal{T}}$ is calculating for every $(\mathcal{K}^E, \sigma, q) \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)$, the set $Rel(\mathcal{K}^E, \sigma, q)$. Calculating $Rel(\mathcal{K}^E, \sigma, q)$ can be done by checking for every summary function $g \in Sf(\mathcal{K}^E, \sigma, q)$ if it is relevant. Our proof of Lemma 4.2 also yields that, by [8], the latter can be done in time $O((|K| \cdot |Q_\varphi|)^k \cdot (k+1)^{|E|\cdot|Q_\varphi|\cdot k})$. Observe that the set $Sf(\mathcal{K}^E, \sigma, q)$ is of size $(k+1)^{|E|}$, and that the number of transducers in $\mathcal{L}^{el}$ is $O(\lambda \cdot m^{el})$, where m is the maximal size of any $\mathcal{K} \in \mathcal{L}$. It follows that for an LTL (resp. μ-calculus) formula φ, the automaton $\mathcal{A}_\varphi^{\mathcal{T}}$ can be built in time at most polynomial in the size of the library, exponential in *el*, and double exponential (resp. exponential) in $|\varphi|$.

We now analyze the time it takes to check for the non-emptiness of $\mathcal{A}_\varphi^{\mathcal{T}}$. Recall that for every $\eta \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)$, the set $Rel(\eta)$ is of size at most $(k+1)^{el}$, and thus, the size of the transition relation of $\mathcal{A}_\varphi^{\mathcal{T}}$ is polynomial in $|\mathcal{L}|$ and $|\varphi|$, and exponential in *el*. Checking the emptiness of $\mathcal{A}_\varphi^{\mathcal{T}}$ is done by first translating it to an equivalent NPT $\mathcal{A}'^{\mathcal{T}}_\varphi$. By [42], given an APT with $|Q|$ states and index k, running on Σ-labeled $\mathcal{D}^*$-trees, one can build (in time polynomial in the descriptions of its input and output automata) an equivalent NPT with $(|Q| \cdot k)^{O(|Q|\cdot k)}$ states, an index $O(|Q| \cdot k)$, and a transition relation of size $|\Sigma| \cdot (|Q| \cdot k)^{O(|D|\cdot|Q|\cdot k)}$. It is worth noting that this blow-up in the size of the automaton is independent from the size of the transition relation of $\mathcal{A}_\varphi^{\mathcal{T}}$. By [36,54], the emptiness of $\mathcal{A}'^{\mathcal{T}}_\varphi$ can be checked in time $|\Sigma| \cdot (|Q| \cdot k)^{O(|D|\cdot|Q|^2\cdot k^2)}$ (and if it is not empty, a witness is returned). Recall that $|\Sigma| = |\mathcal{L}^{el}| = O(\lambda \cdot m^{el})$, and that $|D| = el \cdot |\Sigma_I|$. By substituting the values calculated above for $|Q|$ and k, the theorem follows. □

Note that when using the single-round $\langle \mathcal{L}, el\rangle$-synthesis algorithm as a sub-routine of the multiple-rounds Algorithm 1, it is conceivable that the transducer $\mathcal{K}^i$ synthesized at iteration i will be exponential (or even double-exponential for LTL) in the size of the specification formula φ_i. At this point it is probably best to stop the process, refine the specifications (i.e., break step i into multiple sub-steps), and try again. However, it is important to note that even if the process is continued, and $\mathcal{K}^i$ is added to the library, the time complexity of the succeeding iterations does not deteriorate since the single-round $\langle \mathcal{L}, el\rangle$-synthesis algorithm is only polynomial in the maximal size m of any transducer in the library.

4.2. Enforcing modularity

In this section, we address two main issues that may hinder the efforts of our single-round $\langle \mathcal{L}, el\rangle$-synthesis algorithm to synthesize a succinct hierarchical transducer $\mathcal{K}$.

The first issue is that of ensuring that, when possible, $\mathcal{K}$ indeed makes use of the more complex transducers in the library (especially transducers synthesized in previous rounds) and does not rely too heavily on the less complex, or atomic, transducers. An obvious and most effective solution to this problem is to simply not have some (or all) of the atomic transducers present in the library. The second issue is making sure that $\mathcal{K}$ does not have too many sub-transducers, which can happen if it uses too many copies of the same transducer $\mathcal{K}' \in \mathcal{L}^{=\emptyset}$, each with a different set of exits. We also discuss some other points of interest regarding the synthesis of exits.

We address the above issues by constructing, for each constraint we want to enforce on the synthesized transducer $\mathcal{K}$, an APT $\mathcal{A}$, called the constraint monitor, such that $\mathcal{A}$ accepts only connectivity trees that satisfy the constraint.

We then synthesize $\mathcal{K}$ by checking the non-emptiness of the product of $\mathcal{A}_\varphi^{\mathcal{T}}$ with all the constraints monitors, instead of the single $\mathcal{A}_\varphi^{\mathcal{T}}$. Note that a nondeterministic monitor (i.e., an NPT) of exponential size can also be used, without adversely affecting the time-complexity, if the product with it is taken *after* we translate the product of $\mathcal{A}_\varphi^{\mathcal{T}}$ and the other (polynomial) APT monitors, to an equivalent NPT.

A simple and effective way to enforce modularity in Algorithm 1 is that once a transducer $\mathcal{K}^i$ is synthesized in round i, one incorporates in subsequent rounds a monitor that rejects any connectivity tree containing a node labeled by some key sub-transducers of $\mathcal{K}^i$. This effectively enforces any transducer synthesized using a formula that refers to atomic propositions present only in $\mathcal{K}^i$ (and its disallowed sub-transducers) to use $\mathcal{K}^i$, and not try to build its functionality from scratch. As to other ways to enforce modularity, the question of whether one system is more modular than another, or how to construct a modular system, has received many, and often widely different, answers. Here we only discuss how certain

simple modularity criteria can be easily implemented on top of our algorithm. For example, some people would argue that a function that has more than, say, 10 consecutive lines of code in which no other function is called, is not modular enough. A monitor that checks that in no path in a connectivity tree there are more than 10 consecutive nodes labeled with an atomic transducer can easily enforce such a criterion. We can even divide the transducers in the library into groups, based on how "high level" they are, and enforce lower counts on lower level groups. Essentially, every modularity criterion that can be checked by a polynomial APT, or an exponential NPT, can be used. Enforcing one context-free property can also be done, albeit with an increase in the time complexity. Other non-regular criteria may be enforced by directly modifying the non-emptiness checking algorithm instead of by using a monitor, and we reserve this for future work.

As for the issue of synthesized exits, recall that for each transducer $\mathcal{K}' \in \mathcal{L}^{=\emptyset}$ we can have as many as $\Omega(|\mathcal{K}'|)^{el}$ copies of $\mathcal{K}'$ in $\mathcal{L}^{el}$, each with a different set of exit states. Obviously, we would not like the synthesized transducer $\mathcal{K}$ to use so many copies as sub-transducers. It is not hard to see that one can, for example, build an NPT of size $O(|\mathcal{L}^{el}|)$ that guesses for every $\mathcal{K}' \in \mathcal{L}^{=\emptyset}$ a single set of exits E, and accepts a connectivity tree iff the labels of all the nodes in the tree agree with the guessed exits. Note that after the end of the current round of synthesis, we may choose to add $\mathcal{K}'^{E}$ to the library (in addition, or instead of $\mathcal{K}'$).

Another point to note about the synthesis of exits is that while a transducer $\mathcal{K}$ surely satisfies the formula φ_i it was synthesized for, $\mathcal{K}^E$ may not. Consider for example a transducer $\mathcal{K}$ which is simply a single state, labeled with p, with a self loop. If we remove the loop and turn this state into an exit, it will no longer satisfy $\varphi_i = p \wedge Xp$ or $\varphi_i = Gp$. Now, depending on one's point of view, this may be either an advantage (more flexibility) or a disadvantage (loss of original intent). We believe that this is mostly an advantage, however, in case it is considered a disadvantage, a few possible solutions come to mind. First, for example if $\varphi_i = Gp$, one may wish for $\mathcal{K}$ to remain without exits and enforce $E = \emptyset$. Another option, for example if $\varphi_i = p \wedge Xp$, is to synthesize in round i a modified formula like $\varphi_i' = p \wedge \neg exit \wedge X(p \wedge exit)$, with the thought of exits in mind. Yet another option is to add, at iterations after i, a monitor that checks that if K^E is the label of a node in the connectivity tree then φ_i is satisfied. The monitor can check that φ_i is satisfied inside K^E, in which case the monitor is a single state automaton, that only accepts if E is such that $K^E \models \varphi_i$ (possibly using semantics over truncated paths [25]); alternatively, the monitor can check that φ_i is satisfied in the currently synthesized connectivity tree, starting from the node labeled by K^E, in which case the monitor is based on $\mathcal{A}^{\mathcal{T}}_{\varphi_i}$.

5. Hierarchical games and the proof of Lemma 4.2

We now give the details of the proof of Lemma 4.2 which makes heavy use of hierarchical two-player parity games. We start by providing some necessary definitions and constructs. Additional information regarding these constructs can be found in [8].

5.1. The product of a transducer and a Kripke structure

Given a hierarchical transducer $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$, whose input alphabet Σ_I is the output alphabet of a Kripke structure $\mathcal{S} = \langle \Sigma_I, W, in, \mathcal{R}, \Lambda \rangle$, we can build a hierarchical structure $\mathcal{K} \otimes \mathcal{S}$ by taking the product of $\mathcal{K}$ and $\mathcal{S}$. The hierarchical structure $\mathcal{K} \otimes \mathcal{S}$ has a sub-structure $\mathcal{K}_{i,q}$ for every $2 \leqslant i \leqslant n$ and every state $q \in W$, which is essentially the product of the sub-transducer $\mathcal{K}_i$ with $\mathcal{S}$, where the initial state of $\mathcal{K}_i$ is paired with the state q of $\mathcal{S}$. For $i = 1$, we need only the sub-structure $\mathcal{K}_{1,in}$. The hierarchical order of the sub-structures is consistent with the one in $\mathcal{K}$. Thus, the sub-structure $\mathcal{K}_{i,q}$ can be referred to by boxes of a sub-structure $\mathcal{K}_{j,p}$ only if $i > j$. Let $\mathcal{K}_i = \langle W_i, \mathcal{B}_i, in_i, Exit_i, \tau_i, \delta_i, \Lambda_i \rangle$, then $\mathcal{K}_{i,q} = \langle W_i \times W, \mathcal{B}_i \times W, (in_i, q), Exit_i \times W, \tau_{i,q}, \mathcal{R}_{i,q}, \Lambda_{i,q} \rangle$ is such that:

- For $(b, w) \in \mathcal{B}_i \times W$, we have that $\tau_{i,q}(b, w) = (\tau_i(b), w)$.
- For $(u, w) \in W_i \times W$, we have that $((u, w), (v, w')) \in \mathcal{R}_{i,q}$ iff $(w, w') \in \mathcal{R}$ and $\delta_i(u, \Lambda(w')) = v$.
- For $(b, w) \in \mathcal{B}_i \times W$, and an exit $(e, w') \in Exit_{\tau_i(b)} \times W$ of it, we have that $\langle ((b, w), (e, w')), (v, w'') \rangle \in \mathcal{R}_{i,q}$ iff $(w', w'') \in \mathcal{R}$ and $\delta_i((b, e), \Lambda(w'')) = v$.
- For $(u, w) \in W_i \times W$, we have that $\Lambda_{i,q}((u, w)) = (\Lambda_i(u), \Lambda(w))$.

Given $\sigma \in \Sigma_I$, consider the Kripke structure $\mathcal{S}^\sigma = \langle \Sigma_I, \Sigma_I, \sigma, \Sigma_I \times \Sigma_I, \Sigma_I \times \Sigma_I \rangle$, that has one state for every letter in Σ_I (labeled by that letter), its initial state is σ, and it has a transition from every letter to every letter. Then, it is easy to see that the following holds.

Lemma 5.1. *Given a hierarchical transducer $\mathcal{K}$, and a letter $\sigma \in \Sigma_I$, the computation tree $\mathcal{T}_{\mathcal{K},\sigma}$ can be obtained by unwinding the hierarchical structure $\mathcal{K} \otimes \mathcal{S}^\sigma$.*

5.2. Hierarchical membership games

A *hierarchical two-player game* [8] is a game played between two players, referred to as Player 0 and Player 1. The game is defined by means of a hierarchical arena and a winning condition. The players move a token along the hierarchical

arena, and the winning condition specifies the objectives of the players as to the sequence of states traversed by the token. A *hierarchical arena* is a hierarchical structure with an empty output alphabet, in which the state space of each of the underlying sub-structures is partitioned into states belonging to Player 0 and states belonging to Player 1. When the token is in a state belonging to one of the players, it chooses a successor to which the token is moved. We refer to the underlying substructures as *sub-arenas*. Formally, a hierarchical two-player game is a pair $\mathcal{G} = (\mathcal{V}, \Gamma)$, where $\mathcal{V} = \langle \mathcal{V}_1, \ldots, \mathcal{V}_n \rangle$ is a hierarchical arena, and Γ is a winning condition. For every $1 \leqslant i \leqslant n$, the sub-arena $\mathcal{V}_i = \langle W_i^0, W_i^1, \mathcal{B}_i, in_i, Exit_i, \tau_i, \mathcal{R}_i \rangle$, is simply a hierarchical structure $\langle \emptyset, \langle W_i^0 \cup W_i^1, \mathcal{B}_i, in_i, Exit_i, \tau_i, \mathcal{R}_i, \emptyset \rangle \rangle$ whose set of states $W_i = W_i^0 \cup W_i^1$ is partitioned to Player 0 states W_i^0, and Player 1 states W_i^1. The parity winning condition $\Gamma : \bigcup_i W_i \to C$ maps all states (of all sub-arenas) to a finite set of colors $C = \{C_{\min}, \ldots, C_{\max}\} \subset \mathbb{N}$.

Given a hierarchical structure $\mathcal{S} = \langle \Sigma, \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle \rangle$ and an SAPT $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, the hierarchical two-player game $\mathcal{G}_{\mathcal{S},\mathcal{A}} = (\mathcal{V}, \Gamma)$ for $\mathcal{S}$ and $\mathcal{A}$ is defined as follows. The hierarchical arena $\mathcal{V}$ has a sub-arena $\mathcal{V}_{i,q}$ for every $2 \leqslant i \leqslant n$ and state $q \in Q$, which is essentially the product of the structure $\mathcal{S}_i$ with $\mathcal{A}$, where the initial state of $\mathcal{S}_i$ is paired with the state q of $\mathcal{A}$. For $i = 1$, we need only the sub-arena V_{1,q_0}. The hierarchical order of the sub-arenas is consistent with the one in $\mathcal{S}$. Thus, the sub-arena $\mathcal{V}_{i,q}$ can be referred to by boxes of sub-arena $\mathcal{V}_{j,p}$ only if $i > j$. Let $\mathcal{S}_i = \langle W_i', \mathcal{B}_i', in_i', Exit_i', \tau_i', \mathcal{R}_i', \Lambda_i' \rangle$. Then, the sub-arena $\mathcal{V}_{i,q} = \langle W_{i,q}^0, W_{i,q}^1, \mathcal{B}_{i,q} in_{i,q}, Exit_{i,q}, \tau_{i,q}, \mathcal{R}_{i,q} \rangle$ is defined as follows.

- $W_{i,q}^0 = W_i' \times (Q^{\vee} \cup Q^{(\varepsilon,\vee)})$, $W_{i,q}^1 = W_i' \times (Q^{\wedge} \cup Q^{(\varepsilon,\wedge)})$, $in_{i,q} = (in_i', q)$, and $Exit_{i,q} = Exit_i' \times Q^{\vee,\wedge}$.
- $\mathcal{B}_{i,q} = \mathcal{B}_i' \times Q$, and $\tau_{i,q}(b, q) = (\tau_i'(b), q)$.
- For a state $u = (w, \hat{q}) \in W_i' \times Q$, if $\hat{q} \in Q^{\varepsilon}$ and $\delta(\hat{q}, \Lambda_i'(w)) = \{p_0, \ldots, p_k\}$, then $(u, v) \in \mathcal{R}_{i,q}$ iff $v \in \{(w, p_0), \ldots, (w, p_k)\}$; and if $\hat{q} \in Q^{\vee,\wedge}$, then $(u, v) \in \mathcal{R}_{i,q}$ iff $v = (w', \delta(\hat{q}, \Lambda_i'(w)))$ and $(w, w') \in \mathcal{R}_i'$.
- For $(b, p) \in \mathcal{B}_i' \times Q$, and an exit $(e, \hat{q}) \in Exit'_{\tau_i'(b)} \times Q^{\vee,\wedge}$ of this box, we have that $(((b, p), (e, \hat{q})), v) \in \mathcal{R}_{i,q}$ iff $v = (w', \delta(\hat{q}, \Lambda_i'(e)))$ and $((b, e), w') \in \mathcal{R}_i'$.

The winning condition of the game $\mathcal{G}_{\mathcal{S},\mathcal{A}}$ is induced by the acceptance condition of $\mathcal{A}$. Thus, for each state (w, q) of a sub-arena $\mathcal{V}_{i,q}$, we have that $\Gamma(w, q) = F(q)$. For the formal definition of plays, strategies, etc., please see [8]. It is important to note that, as is the case for flat membership games, a Player 0 strategy for $\mathcal{G}_{\mathcal{S},\mathcal{A}}$ corresponds to a run of $\mathcal{A}$ on the unwinding of $\mathcal{S}$, a memoryless Player 0 strategy corresponds to a memoryless run, and a winning Player 0 strategy corresponds to an accepting run. Furthermore, a Player 0 strategy for a sub-arena $\mathcal{V}_{i,q}$ corresponds to a run of A, starting in state q, on the unwinding of the sub-structure $\mathcal{S}_i$.

Theorem 5.1. *(See [8].) Given a hierarchical structure $\mathcal{S}$ and an SAPT $\mathcal{A}$, we have that $\mathcal{A}$ accepts the unwinding $\mathcal{T}_{\mathcal{S}}$ of $\mathcal{S}$, iff Player* 0 *has a winning strategy in the hierarchical game $\mathcal{G}_{\mathcal{S},\mathcal{A}}$.*

We now have all the definitions necessary to construct the membership games $\mathcal{G}^{\mathrm{s}}_{\mathcal{K}\otimes\mathcal{S}^{\varrho},\mathcal{A}_{\varphi}}$, and $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}_{\varphi}^{\mathcal{T}}}$.

5.3. The membership game $\mathcal{G}^{\mathrm{s}}_{\mathcal{K}\otimes\mathcal{S}^{\varrho},\mathcal{A}_{\varphi}}$

Given a library $\mathcal{L}$ of hierarchical transducers with input and output alphabets Σ_I and Σ_O, and a bound $el \in \mathbb{N}$, let $\mathcal{T} = \langle T, V \rangle$ be a regular connectivity tree, let $\mathcal{M} = \langle \{1, \ldots, el\} \times \Sigma_I, \mathcal{L}^{el}, \langle M, m_0, \emptyset, \delta^{\mathcal{T}}, \Lambda^{\mathcal{T}} \rangle \rangle$ be a flat transducer such that $\mathcal{T}$ is equal to the (lean) computation tree $\mathcal{T}_{\mathcal{M}}$, and let $\mathcal{K} = \langle \Sigma_I, \Sigma_O, \langle \mathcal{K}_1, \ldots, \mathcal{K}_n \rangle \rangle$ be the hierarchical transducer induced by it. Recall that for every $b \in M$, we denote by $E(b)$ the set of top-level exits of $\Lambda^{\mathcal{T}}(b)$. For the purpose of this proof, it is much more convenient to consider a slightly different version of the induced hierarchical transducer $\mathcal{K}$, where the top level sub-transducer $\mathcal{K}_1$ contains only boxes and no states. That is, we replace every top-level state w, with a box that refers to the atomic transducer K_{ς}, where ς is such that $\Lambda^{\mathcal{T}}(w) = \varsigma$. We also relax the definition of hierarchical transducers (as well as hierarchical structures and arenas) to allow the top-level initial state to be not a state but a box. Formally, $\mathcal{K}_1 = \langle \emptyset, M, m_0, \tau_1, \delta_1, \emptyset \rangle$, where:

- For $b \in M$, we have that $\tau_1(b) = i$, where i is such that $\mathcal{K}_i$ is the top-level sub-transducer of $\Lambda^{\mathcal{T}}(b)$.
- For $b \in M$, we have for every $e \in E(b)$, and every $\sigma \in \Sigma_I$, that $\delta_1((b, e), \sigma) = \delta^{\mathcal{T}}(b, (idx(e, E(b)), \sigma))$.

It is easy to see that the difference between the version of $\mathcal{K}$ with top-level states, and the modified version without them, is mainly syntactic. Thus, for example, the two versions have isomorphic flat expansions and computation trees, and Lemma 5.1 also holds if $\mathcal{K}$ has no top-level states. Also, note that since the set of directions of the input trees of an SAPT plays no role in the definition of its run, the computation trees of these two versions of $\mathcal{K}$ are indistinguishable by any SAPT. Finally, one can easily verify that Theorem 5.1 remains valid also if the hierarchical structure $\mathcal{S}$ has no top-level states.

By Lemma 5.1, the computation tree $\mathcal{T}_{\mathcal{K},\varrho}$ can be obtained by unwinding the hierarchical structure $\mathcal{K}\otimes\mathcal{S}^{\varrho}$. By definition, $\mathcal{K}\otimes\mathcal{S}^{\varrho}$ has a sub-structure $\mathcal{K}_{i,\sigma}$, for every $2 \leqslant i \leqslant n$ and every $\sigma \in \Sigma_I$, which is the product of $\mathcal{K}_i$ with $\mathcal{S}^{\sigma}$, plus a top-level sub-structure $\mathcal{K}_{1,\varrho} = \langle \emptyset, M \times \Sigma_I, (m_0, \varrho), \emptyset, \tau_{1,\varrho}, \mathcal{R}_{1,\varrho}, \Lambda_{1,\varrho} \rangle$, where:

- For $(b,\sigma) \in M \times \Sigma_I$, we have that $\tau_{1,\varrho}(b,\sigma) = (i,\sigma)$, where i is such that $\mathcal{K}_i$ is the top-level sub-transducer of $\Lambda^{\mathcal{T}}(b)$.
- For $(b,\sigma) \in M \times \Sigma_I$, and an exit $(e,\hat{\sigma}) \in Exit_{\tau_1(b)} \times \Sigma_I$ of this box, we have that $\langle((b,\sigma),(e,\hat{\sigma})),(b',\sigma')\rangle \in \mathcal{R}_{1,\varrho}$ iff $\delta^{\mathcal{T}}(b,(idx(e,E(b)),\sigma')) = b'$.

Given a temporal logic formula φ, by Definition 3.1 and Theorem 2.1, $\mathcal{K} \models \varphi$ iff $\mathcal{T}_{\mathcal{K},\varrho}$ is accepted by the SAPT $\mathcal{A}_\varphi$. Hence, by Theorem 5.1, $\mathcal{K} \models \varphi$ iff Player 0 has a winning strategy in the hierarchical membership game $\mathcal{G}_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$ of $\mathcal{K}\otimes\mathcal{S}^\varrho$ and $\mathcal{A}_\varphi$. Let $\mathcal{A}_\varphi = \langle \Sigma_I \cup \Sigma_O, Q_\varphi, q^0_\varphi, \delta_\varphi, F_\varphi\rangle$ be an SAPT with Q_φ partitioned to $Q_\varphi^{(\varepsilon,\wedge)}$, $Q_\varphi^{(\varepsilon,\vee)}$, $Q_\varphi^{\wedge}$, and $Q_\varphi^{\vee}$, and let $\mathcal{K}\otimes\mathcal{S}^\varrho$ be as above. By definition, $\mathcal{G}_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$ has a sub-arena $\mathcal{V}_{i,\sigma,q}$ for every $(i,\sigma,q) \in \{1,\ldots,n\} \times \Sigma_I \times Q_\varphi$, which is the product of the sub-structure $\mathcal{K}_i \otimes \mathcal{S}^\sigma$ with $\mathcal{A}^q_\varphi$ (recall that $\mathcal{A}^q_\varphi$ is $\mathcal{A}_\varphi$ with initial state q), plus a top-level sub-arena $V_{1,\varrho,q^0_\varphi} = \langle \emptyset,\emptyset, M\times\Sigma_I\times Q_\varphi, (m_0,\varrho,q^0_\varphi),\emptyset,\check{\tau},\check{\mathcal{R}}\rangle$, where:

- For $(b,\sigma,q) \in M\times\Sigma_I\times Q_\varphi$, we have that $\check{\tau}(b,\sigma,q) = (i,\sigma,q)$, where i is such that $\mathcal{K}_i$ is the top-level sub-transducer of $\Lambda^{\mathcal{T}}(b)$.
- For a box $(b,\sigma,q) \in M\times\Sigma_I\times Q_\varphi$, let $\Lambda^{\mathcal{T}}(b) = \mathcal{K}^E$, and let Λ^E be the labeling function of the top-level sub-transducer of $\mathcal{K}^E$. Given an exit $(e,\hat{\sigma},\hat{q}) \in E\times\Sigma_I\times Q_\varphi^{(\varepsilon,\vee)}$ of this box, we have that $\langle((b,\sigma,q),(e,\hat{\sigma},\hat{q})),(b',\sigma',q')\rangle \in \check{\mathcal{R}}$ iff $q' = \delta_\varphi(\hat{q},(\Lambda^E(e),\hat{\sigma}))$ and $\delta^{\mathcal{T}}(b,(idx(e,E),\sigma')) = b'$.

In [8], in order to solve a hierarchical game, one simplifies it, turning it into an equivalent flat game, by replacing every box of the top-level sub-arena with a gadget that is a 3-level DAG. We briefly recall below the structure of these gadgets, and describe the result of the simplification of the membership game $\mathcal{G}_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$. Let $\beta = (b,\sigma,q)$ be a box of $V_{1,\varrho,q^0_\varphi}$, let $\mathcal{V}_{i,\sigma,q}$ be the sub-arena that it refers to, let $\Lambda^{\mathcal{T}}(b) = \mathcal{K}^E$, and let $Rel(\mathcal{K}^E,\sigma,q)$ be the set of all relevant summary functions[11] of the runs of $\mathcal{A}^q_\varphi$ on $\mathcal{T}_{\mathcal{K}^E,\sigma}$. A gadget $H_{(\mathcal{K}^E,\sigma,q)}$ contains all the nodes reachable from the root p of the following 3-level DAG structure:

- The set of nodes of $H_{(\mathcal{K}^E,\sigma,q)}$ is $\{p\} \cup Rel(\mathcal{K}^E,\sigma,q) \cup (E\times\Sigma_I\times Q_\varphi^{\vee,\wedge}\times C)$. The Player 0 nodes are $\{p\}\cup(E\times\Sigma_I\times Q_\varphi^{\vee}\times C)$, and the Player 1 nodes are $Rel(\mathcal{K}^E,\sigma,q)\cup(E\times\Sigma_I\times Q_\varphi^{\wedge}\times C)$.
- The set of edges is $\bigcup_{g\in Rel(\mathcal{K}^E,\sigma,q)}(\{(p,g)\}\cup\{(g,(h,g(h))):\ h\in(E\times\Sigma_I\times Q_\varphi^{\vee,\wedge})\wedge g(h)\neq\dashv\})$.
- A node $(e,\sigma,q',c) \in (E\times\Sigma_I\times Q_\varphi^{\vee,\wedge}\times C)$ is colored by c. These are the only colored nodes.

The simplification of $V_{1,\varrho,q^0_\varphi}$ is performed by replacing every box $\beta = (b,\sigma,q)$ with a copy of the gadget $H_{(\Lambda^{\mathcal{T}}(b),\sigma,q)}$. To prevent name clashes between copies of the same gadget, we let H^β be a copy of $H_{(\Lambda^{\mathcal{T}}(b),\sigma,q)}$ with all nodes renamed by superscripting them with β. A box β is substituted with H^β by replacing every transition that enters β with a transition that enters the root p^β of H^β, and replacing every transition that exits β through an exit (e,σ,q') with one transition, going out of every leaf of the form (e,σ,q',c) that is present in H^β. Applying this simplification to $V_{1,\varrho,q^0_\varphi}$, we obtain the flat game $\mathcal{G}^s_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi} = (V^s,\Gamma^s)$, where $\mathcal{V}^s = \langle W^{0s}, W^{1s}, \emptyset, in^s, \emptyset, \emptyset, \mathcal{R}^s\rangle$, and Γ^s are as follows:

- $W^s = \bigcup_{\beta\in(M\times\Sigma_I\times Q_\varphi)}(H^\beta)$, where:
  - $W^{0s} = \bigcup_{\beta=(b,\sigma,q)\in M\times\Sigma_I\times Q_\varphi}(\{p^\beta\}\cup\{(e,\hat{\sigma},\hat{q},c)^\beta:\ (e,\hat{\sigma},\hat{q},c)\in(E(b)\times\Sigma_I\times Q_\varphi^{\vee}\times C)\})\cap W^s$.
  - $W^{1s} = \bigcup_{\beta=(b,\sigma,q)\in M\times\Sigma_I\times Q_\varphi}(\{g^\beta:\ g\in Rel(\Lambda^{\mathcal{T}}(b),\sigma,q)\}\cup\{(e,\hat{\sigma},\hat{q},c)^\beta : (e,\hat{\sigma},\hat{q},c)\in(E(b)\times\Sigma_I\times Q_\varphi^{\wedge}\times C)\})\cap W^s$.
- $in^s = p^{(m_0,\varrho,q^0_\varphi)}$.
- For every $\beta = (b,\sigma,q) \in (M\times\Sigma_I\times Q_\varphi)$, with $\Lambda^{\mathcal{T}}(b) = \mathcal{K}^E$, the following transitions are in $\mathcal{R}^s$:
  - $\bigcup_{g\in Rel(\mathcal{K}^E,\sigma,q)}\{(p^\beta,g^\beta)\}$.
  - $\bigcup_{g\in Rel(\mathcal{K}^E,\sigma,q)}\{(g^\beta,(h,g^\beta(h))):\ h\in(E\times\Sigma_I\times Q_\varphi^{\vee,\wedge})\wedge g^\beta(h)\neq\dashv\}$.
  - Let Λ^E be the labeling function of the top-level sub-transducer of $\mathcal{K}^E$. For every node $(e,\hat{\sigma},\hat{q},c)^\beta$ of H^β, and every $\beta' = (b',\sigma',q') \in (M\times\Sigma_I\times Q_\varphi)$, we have that the transition $((e,\hat{\sigma},\hat{q},c)^\beta, p^{\beta'})$ is in $\mathcal{R}^s$ iff $q' = \delta_\varphi(\hat{q},(\Lambda^E(e),\hat{\sigma}))$ and $b' = \delta^{\mathcal{T}}(b,(idx(e,E),\sigma'))$.
- For every $\beta \in M\times\Sigma_I\times Q_\varphi$, and every node $w = (e,\sigma',q',c)^\beta$ of H^β, we have that $\Gamma^s(w) = c$. All the other nodes[12] are colored by $C_{\min}$.

Theorem 5.2. *(See [8].) Player 0 wins the hierarchical game $\mathcal{G}_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$ iff it wins the simplified game $\mathcal{G}^s_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$.*

[11] In [8], summary functions were defined in terms of Player 0 strategies. For the special case of the membership game we consider, Player 0 strategies correspond to runs of $\mathcal{A}_\varphi$, and the definition of summary functions given in [8] coincides with the one given in Section 4.1.2.

[12] In [8], these nodes were left uncolored. However, since there is no cycle that goes only through uncolored nodes, coloring such nodes with $C_{\min}$ does not change anything.

Applying Theorems 5.1 and 5.2 to the constructions above we get that:

Corollary 5.1. *$\mathcal{A}_\varphi$ accepts $\mathcal{T}_{\mathcal{K},\varrho}$ iff Player 0 wins the game $\mathcal{G}^s_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$.*

5.4. The membership game $\mathcal{G}_{K^\mathcal{T},\mathcal{A}^\mathcal{T}_\varphi}$

We now turn our attention to constructing the membership game $\mathcal{G}_{K^\mathcal{T},\mathcal{A}^\mathcal{T}_\varphi}$. As before, let $\mathcal{M} = \langle\{1,\ldots,el\}\times\Sigma_I, \mathcal{L}^{el}, \langle M, m_0, \emptyset, \delta^\mathcal{T}, \Lambda^\mathcal{T}\rangle\rangle$ be a flat transducer such that $\mathcal{T}$ is equal to the (lean) computation tree $\mathcal{T}_\mathcal{M}$. It is not hard to see that $\mathcal{T}$ can be obtained by unwinding the following Kripke structure[13] $K^\mathcal{T} = \langle\mathcal{L}^{el}, W, in, \mathcal{R}, \Lambda\rangle$, where:

- $W = M\times\{1,\ldots,el\}\times\Sigma_I$, and $in = (m_0, 1, \varrho)$.
- For $(b,i,\sigma),(b',i',\sigma')\in W$, we have that $((b,i,\sigma),(b',i',\sigma'))\in\mathcal{R}$ iff $\delta^\mathcal{T}(b,(i',\sigma')) = b'$.
- For every $(b,i,\sigma)\in W$, we have that $\Lambda(b,i,\sigma) = \Lambda^\mathcal{T}(b)$.

Observe that since the transitions of $\mathcal{A}^\mathcal{T}_\varphi$ mix conjunctions and disjunctions, one cannot construct the arena of the membership game of $K^\mathcal{T}$ and $\mathcal{A}^\mathcal{T}_\varphi$ by directly taking their product. Indeed, doing so would result with nodes of the arena that cannot be assigned to any single player. Hence, we first unfold the transition relation of $\mathcal{A}^\mathcal{T}_\varphi$, and obtain an equivalent automaton $\tilde{A}^\mathcal{T}_\varphi = \langle\mathcal{L}^{el}, \mathcal{Q}, q_0, \tilde{\delta}, \tilde{F}\rangle$, where the moves from every state are either pure conjunctions or pure disjunctions. Note that this requires that we allow $\tilde{A}^\mathcal{T}_\varphi$ to have ε-moves. Thus, like an SAPT, the states of $\tilde{A}^\mathcal{T}_\varphi$ are divided into four sets $\mathcal{Q}^{(\varepsilon,\vee)}$, $\mathcal{Q}^{(\varepsilon,\wedge)}$, $\mathcal{Q}^\vee$ and $\mathcal{Q}^\wedge$. However, unlike an SAPT, we allow states in $\mathcal{Q}^{\vee,\wedge}$ to send copies in different states to different directions. That is, for $s\in\mathcal{Q}^{\vee,\wedge}$, and $\mathcal{K}^E\in\mathcal{L}^{el}$, we have that $\tilde{\delta}(s,\mathcal{K}^E)\subseteq(\{1,\ldots,el\}\times\Sigma_I)\times\mathcal{Q}$.

To construct $\tilde{A}^\mathcal{T}_\varphi$, we simply have to unfold the transition relation of $\mathcal{A}^\mathcal{T}_\varphi$. That is, for every state $s = (\sigma,q,c)\in\Sigma_I\times Q^{\vee,\wedge}_\varphi\times C$ of $\mathcal{A}^\mathcal{T}_\varphi$, and every $\mathcal{K}^E\in\mathcal{L}^{el}$, we direct the transition from s, when reading $\mathcal{K}^E$, to the entry node of a 3-level DAG gadget $H_{(\mathcal{K}^E,\sigma,q)}$ that unfolds $\delta((\sigma,q,c),\mathcal{K}^E)$. Since $\delta((\sigma,q,c),\mathcal{K}^E)$ is not dependent on the color c, the gadget only depends on σ, q and K^E. We use the same notation for naming these gadgets, as for the gadgets used in the simplification of the game $\mathcal{G}_{\mathcal{K}\otimes\mathcal{S}^\varrho,\mathcal{A}_\varphi}$, for the simple reason that they are exactly the same gadgets. In fact, the transition relation of $\mathcal{A}^\mathcal{T}_\varphi$ is defined the way it is precisely because unfolding it gives these gadgets. Note that resolving the outermost disjunction and conjunction of $\delta((\sigma,q,c),\mathcal{K}^E)$ amounts to choosing some $g\in Rel(\mathcal{K}^E,\sigma,q)$, and some $(e,\sigma',q')\in(E\times\Sigma_I\times Q^{\vee,\wedge}_\varphi)$, and that once g is chosen, the only leaf of the form $(e,\hat{\sigma},\hat{q},c)$ that is reachable from g is such that $c = g(e,\hat{\sigma},\hat{q})$. Hence, $H_{(\mathcal{K}^E,\sigma,q)}$ faithfully represents the unfolding of $\delta((\sigma,q,c),\mathcal{K}^E)$ even though its leaves include the extra component c. Also, note that since the gadgets are used to simply unfold the transition relation, only the moves going out of their leaves are not ε-moves and correspond to real moves on the input tree. Before we formally describe $\tilde{A}^\mathcal{T}_\varphi$, observe that even though every gadget $H_{(\mathcal{K}^E,\sigma,q)}$ is used only once in $\tilde{A}^\mathcal{T}_\varphi$, the names of nodes are not unique across different gadgets. Hence, to get unique names, for every $\eta\in\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi$ we subscript the names of nodes in H_η with η. Formally, $\tilde{A}^\mathcal{T}_\varphi = \langle\mathcal{L}^{el}, \mathcal{Q}, q_0, \tilde{\delta}, \tilde{F}\rangle$, where:

- $\mathcal{Q} = (\Sigma_I\times Q^{\vee,\wedge}_\varphi\times C)\cup\{q_0\}\cup\bigcup_{\eta\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)}(H_\eta)$, where:
  - $\mathcal{Q}^{(\varepsilon,\vee)} = (\Sigma_I\times Q^{\vee,\wedge}_\varphi\times C)\cup\{q_0\}\cup\bigcup_{\eta\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)}\{p_\eta\}$.
  - $\mathcal{Q}^{(\varepsilon,\wedge)} = \bigcup_{\eta\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)}\{g_\eta : g\in Rel(\eta)\}$.
  - $\mathcal{Q}^\vee = \bigcup_{\eta=(\mathcal{K}^E,\sigma,q)\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)}\{(e,\hat{\sigma},\hat{q},c)_\eta\in H_\eta\colon\ (e,\hat{\sigma},\hat{q},c)\in(E\times\Sigma_I\times Q^\vee_\varphi\times C)\}\cap\mathcal{Q}$.
  - $\mathcal{Q}^\wedge = \bigcup_{\eta=(\mathcal{K}^E,\sigma,q)\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)}\{(e,\hat{\sigma},\hat{q},c)_\eta\in H_\eta\colon\ (e,\hat{\sigma},\hat{q},c)\in(E\times\Sigma_I\times Q^\wedge_\varphi\times C)\}\cap\mathcal{Q}$.
- Since certain states are only reachable via ε moves, where the input does not change, $\tilde{\delta}$ is a partial function which is defined only for $s\in\mathcal{Q}$, and $\mathcal{K}^E\in\mathcal{L}^{el}$, for which it is possible to reach s when the current input is $\mathcal{K}^E$. Thus:
  - $\tilde{\delta}(q_0,\mathcal{K}^E) = \tilde{\delta}((\varrho,q^0_\varphi,C_{\min}),\mathcal{K}^E)$.
  - If $s = (\sigma,q,c)\in(\Sigma_I\times Q^{\vee,\wedge}_\varphi\times C)$, then $\tilde{\delta}(s,\mathcal{K}^E) = \{p_{(\mathcal{K}^E,\sigma,q)}\}$.
  - If $s = p_\eta$, where $\eta\in(\mathcal{L}^{el}\times\Sigma_I\times Q_\varphi)$, then $\tilde{\delta}(s,\mathcal{K}^E) = \{g_\eta\colon\ g\in Rel(\eta)\}$.
  - If $s = g_\eta$, where g is a summary function in $Rel(\mathcal{K}^E,\sigma,q)$, then $\tilde{\delta}(s,\mathcal{K}^E) = \{(h,g(h))\colon\ h\in(E\times\Sigma_I\times Q^{\vee,\wedge}_\varphi)\wedge g(h)\neq\dashv\}$.
  - If $s = (e,\hat{\sigma},\hat{q},c)_\eta$, then $\tilde{\delta}(s,\mathcal{K}^E) = \bigcup_{\sigma'\in\Sigma_I}\{((idx(e,E),\sigma'),(\sigma',\delta_\varphi(\hat{q},(\Lambda^E(e),\hat{\sigma})),c))\}$, where Λ^E is the labeling function of the top-level sub-transducer of $\mathcal{K}^E$.
- Finally, for every $s\in\mathcal{Q}$, if $s = (e,\hat{\sigma},\hat{q},c)_\eta\in\mathcal{Q}^{\vee,\wedge}$, or $s = (\sigma,q,c)\in(\Sigma_I\times Q^{\vee,\wedge}_\varphi\times C)$, then $\tilde{F}(s) = c$; otherwise, $\tilde{F}(s) = C_{\min}$.

[13] Technically, the unwinding of $K^\mathcal{T}$ has the set of directions $M\times\{1,\ldots,el\}\times\Sigma_I$, whereas the set of directions of $\mathcal{T}$ is $\{1,\ldots,el\}\times\Sigma_I$. However, by simply ignoring the M component, we get $\mathcal{T}$.

Note that since states in $(\Sigma_I \times Q_\varphi^{\vee,\wedge} \times C) \cup \{q_0\}$ (i.e., the original states of $\mathcal{A}_\varphi^{\mathcal{T}}$) have a single successor per input, their classification as $(\varepsilon, \vee)$ states, and not as $(\varepsilon, \wedge)$ states, is arbitrary. Also, note that since Lemma 4.2 which we are trying to prove only concerns connectivity trees, we do not care how $\tilde{A}_\varphi^{\mathcal{T}}$ behaves on trees where the root is not labeled by an atomic transducer. Going over the definitions of $\mathcal{A}_\varphi^{\mathcal{T}}$ and $\tilde{A}_\varphi^{\mathcal{T}}$, one can easily see that given a regular connectivity tree $\mathcal{T}$, then $\mathcal{A}_\varphi^{\mathcal{T}}$ accepts the unwinding of $K^{\mathcal{T}}$ iff $\tilde{A}_\varphi^{\mathcal{T}}$ does.

Our last step is to construct the arena of the membership game $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}_\varphi^{\mathcal{T}}}$. Recall that, intuitively, this arena is the product of $K^{\mathcal{T}}$ and $\tilde{A}_\varphi^{\mathcal{T}}$, and that the successors of every node (w, s), where w is a state of $K^{\mathcal{T}}$ and s is a state of $\tilde{A}_\varphi^{\mathcal{T}}$, are pairs (w', s'), where s' is a successor of s that is sent to the successor w' of w when reading the label of w. Note, however, that we can eliminate some redundancies, as follows. By the last item in the definition of the transition relation $\tilde{\delta}$, for every node $w = (b, i, \sigma)$ of $K^{\mathcal{T}}$ and every state of the form $s = (\sigma', q, c) \in \Sigma_I \times Q_\varphi^{\vee,\wedge} \times C$ of $\tilde{A}_\varphi^{\mathcal{T}}$, the node (w, s) of the product arena is reachable only if $\sigma = \sigma'$; and by the second item in the definition of $\tilde{\delta}$, every such node (w, s) has only a single successor, namely, $(w, p_{(\Lambda^{\mathcal{T}}(b),\sigma,q)})$. Thus, we can simply eliminate the node (w, s) and direct every incoming transition directly to its single successor. Note that since all of the predecessors of (w, s) have the same color c as (w, s), skipping it does not affect the winning condition. For similar reasons, the initial node $((m_0, 1, \varrho), q_0)$ of the arena can be replaced by its successor $((m_0, 1, \varrho), p_{(\Lambda^{\mathcal{T}}(m_0),\varrho,q_\varphi^0)})$. Formally, after removing the above redundancies, we have that $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}_\varphi^{\mathcal{T}}} = (\tilde{V}, \tilde{\Gamma})$, with $\tilde{\mathcal{V}} = \langle \tilde{W}^0, \tilde{W}^1, \emptyset, \tilde{in}, \emptyset, \emptyset, \tilde{\mathcal{R}} \rangle$, where:

- $\tilde{W} = (M \times \{1, \ldots, el\} \times \Sigma_I) \times \bigcup_{\eta \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)} (H_\eta)$, where:
  - $\tilde{W}^0 = (M \times \{1, \ldots, el\} \times \Sigma_I) \times (\bigcup_{\eta=(\mathcal{K}^E,\sigma,q) \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)} (\{p_\eta\} \cup \{(e, \hat{\sigma}, \hat{q}, c)_\eta : (e, \hat{\sigma}, \hat{q}, c) \in (E \times \Sigma_I \times Q_\varphi^{\vee} \times C)\})) \cap \tilde{W}$.
  - $\tilde{W}^1 = (M \times \{1, \ldots, el\} \times \Sigma_I) \times (\bigcup_{\eta=(\mathcal{K}^E,\sigma,q) \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)} (\{g_\eta :\ g \in Rel(\eta)\} \cup \{(e, \hat{\sigma}, \hat{q}, c)_\eta : (e, \hat{\sigma}, \hat{q}, c) \in (E \times \Sigma_I \times Q_\varphi^{\wedge} \times C)\})) \cap \tilde{W}$.
- $\tilde{in} = ((m_0, 1, \varrho), p_{(\Lambda^{\mathcal{T}}(m_0),\varrho,q_\varphi^0)})$.
- For every $w = (b, i, \grave{\sigma}) \in M \times \{1, \ldots, el\} \times \Sigma_I$, with $\Lambda^{\mathcal{T}}(b) = \mathcal{K}^E$, and every $\eta = (\mathcal{K}^E, \sigma, q) \in \{\mathcal{K}^E\} \times \Sigma_I \times Q_\varphi$, the following transitions are in $\tilde{\mathcal{R}}$:
  - $\bigcup_{g \in Rel(\mathcal{K}^E,\sigma,q)} \{\langle (w, p_\eta), (w, g_\eta) \rangle\}$.
  - $\bigcup_{g \in Rel(\mathcal{K}^E,\sigma,q)} \{\langle (w, g_\eta), (w, (h, g(h))) \rangle :\ h \in (E \times \Sigma_I \times Q_\varphi^{\vee,\wedge}) \wedge g(h) \neq \dashv\}$
  - Let Λ^E be the labeling function of the top-level sub-transducer of $\mathcal{K}^E$. For every node $(e, \hat{\sigma}, \hat{q}, c)_\eta$ of H_η, and every $\sigma' \in \Sigma_I$, we have that the transition $\langle (w, (e, \hat{\sigma}, \hat{q}, c)_\eta), ((b', i', \sigma'), p_{\eta'}) \rangle$ is in $\tilde{\mathcal{R}}$, iff $b' = \delta^{\mathcal{T}}(b, (idx(e, E), \sigma'))$, and $i' = idx(e, E)$, and $\eta' = (\Lambda^{\mathcal{T}}(b'), \sigma', \delta_\varphi(\hat{q}, (\Lambda^E(e), \hat{\sigma})))$.
- Finally, for every $w \in M \times \{1, \ldots, el\} \times \Sigma_I$, for every $\eta \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)$, and every $s = (e, \sigma', q', c)_\eta$ of H_η, we have that $\tilde{\Gamma}(w, s) = c$. All the other nodes are colored by $C_{\min}$.

Observe that for every $w = (b, i, \grave{\sigma}) \in M \times \{1, \ldots, el\} \times \Sigma_I$, and every $\eta = (\mathcal{K}^E, \sigma, q) \in (\mathcal{L}^{el} \times \Sigma_I \times Q_\varphi)$, by the third item in the definition of $\tilde{\mathcal{R}}$, the only nodes of the form (w, p_η) that have incoming edges are such that $\mathcal{K}^E = \Lambda^{\mathcal{T}}(b)$, and $\grave{\sigma} = \sigma$. By the first and second items in the definition of $\tilde{\mathcal{R}}$ this property propagates, and we get that for every $s_\eta \in H_\eta$ (be it the root, a summary function node, or a leaf) the node (w, s_η) has incoming edges only if $\mathcal{K}^E = \Lambda^{\mathcal{T}}(b)$ and $\grave{\sigma} = \sigma$. We can thus delete all the nodes (w, s_η) in $\tilde{\mathcal{V}}$ for which the above connections between w and η do not exist, since they are not reachable. Also, note that the transitions going out of a node (w, s_η) of $\tilde{\mathcal{V}}$, where $w = (b, i, \sigma) \in M \times \{1, \ldots, el\} \times \Sigma_I$, are completely independent of i, and that the classification of (w, s_η) as a Player 0 or a Player 1 node is also independent of i. Thus, we can safely merge all nodes that differ only in their $\{1, \ldots, el\}$ component into a single node by dropping the $\{1, \ldots, el\}$ component.

After deleting the unreachable nodes mentioned above, and dropping the $\{1, \ldots, el\}$ component, we get that for every $\beta = (b, \sigma, q) \in M \times \Sigma_I \times Q_\varphi$, and every state $s_\eta \in H_{(\Lambda^{\mathcal{T}}(b),\sigma,q)}$, there is exactly one node left in $\tilde{\mathcal{V}}$ that corresponds to β and s_η, that is, the node $((b, \sigma), s_\eta)$. By mapping every such node $((b, \sigma), s_\eta)$ of $\tilde{\mathcal{V}}$, to the node s^β of the arena $\mathcal{V}^s$ of the game $\mathcal{G}^s_{\mathcal{K} \otimes \S^\varrho, \mathcal{A}_\varphi}$, we get an isomorphism between (what remained of) $\tilde{\mathcal{V}}$ and $\mathcal{V}^s$. That is, the two arenas (including the coloring) are identical up to the naming of their nodes. Hence, the games $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}_\varphi^{\mathcal{T}}}$ and $\mathcal{G}^s_{\mathcal{K} \otimes \mathcal{S}^\varrho, \mathcal{A}_\varphi}$ are equivalent. Recall that by the construction above Player 0 wins $\mathcal{G}_{K^{\mathcal{T}},\mathcal{A}_\varphi^{\mathcal{T}}}$ iff $\mathcal{A}_\varphi^{\mathcal{T}}$ accepts $\mathcal{T}$, and that by Corollary 5.1 Player 0 wins $\mathcal{G}^s_{\mathcal{K} \otimes \mathcal{S}^\varrho, \mathcal{A}_\varphi}$ iff $\mathcal{A}_\varphi$ accepts $\mathcal{T}_{\mathcal{K},\varrho}$, which completes the proof.

6. Incomplete information

A natural setting that was considered in the synthesis literature is that of incomplete information [34,35]. For example, in a distributed system, it is common that one processor cannot see the local variables of another processor, and only has access to the shared global variables. In the incomplete information setting, in addition to the set of input signals I that

the system can read, the environment also has internal signals H that the system cannot read, and one should synthesize a system whose behavior depends only on the readable signals, but satisfies a specification which refers also to the unreadable signals. It is important to note that all signals, both I and H, are visible to the synthesis algorithm, but that the signals in H are not visible to the synthesized program when it runs. More formally, the specification temporal logic formula is given with respect to the alphabet $\Sigma_I = 2^{I \cup H}$ (instead of just 2^I as in the complete information setting), and the synthesized system can be viewed as a strategy $P : (2^I)^* \to \Sigma_O$ that maps a finite sequence of sets of the visible input signals (i.e., the visible part of the history of the actions of the environment so far) into a set of current output signals. In other words, the behavior of the synthesized system must be the same given two histories that differ only in their H components because they are indistinguishable by the system due to its incomplete information. As for the complete information setting, when P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $\Sigma_I \cup \Sigma_O$, and it induces a *computation tree*. This tree has a fixed branching degree $|\Sigma_I|$, and it embodies all the possible inputs (and hence also computations) of P. Realizability of a temporal specification φ over Σ_I is the problem of determining whether there exists a system P whose computation tree satisfies φ, and synthesis amounts to finding such a P.

Given this high similarity between the complete and incomplete information settings, it may seem that adapting a synthesis algorithm designed for the complete information setting to the incomplete setting should be easy. Unfortunately, this has not been the case for the traditional synthesis algorithms for branching time logics. To appreciate the difficulty, recall that the traditional synthesis algorithms work by constructing an appropriate *computation tree-automaton* that accepts computation trees that satisfy the specification formula. A finitely-representable witness to the non-emptiness of this automaton is the desired system. Observe that a computation tree has a fixed branching degree $|\Sigma_I|$, and it is labeled by letters in $\Sigma_I \times \Sigma_O$. In the complete information setting, there are no restrictions placed on the Σ_O components of the labeling. A node y labeled by (σ_i, σ_o), is taken to mean that given the history of input signals represented by y, the system represented by this computation tree would output σ_o. However, in the incomplete information setting not every labeling of a computation tree represents a legal system – we must add the restriction that two nodes y, y' in the tree that differ only in their hidden, i.e. H, signals (recall that now $\Sigma_I = 2^{I \cup H}$) must have the same labeling! Indeed, since as far as the system can see these two nodes are indistinguishable it must produce the same output in both cases. Thus, computation trees that do not satisfy this additional restriction do not represent a legal system, and should be rejected by the computation tree automaton. The problem is that this restriction is not regular in the sense that a finite tree automaton cannot check that a computation tree satisfies it. Hence, the computation tree automaton developed for the complete information setting cannot be easily adapted to reject trees that violate the restriction imposed by the incomplete information setting. See [9–11] for a related discussion on recursive (formally, pushdown) systems with incomplete information.

As we noted before, the hierarchical synthesis problem studied in this article presents difficulties that prevent us from using the computation tree automaton approach. Recall that the automaton at the heart of our single-round synthesis algorithm does not run on computation trees, but rather on connectivity trees. Therefore, in our case the similarities between the complete and incomplete information settings can be used to their fullest, as we can easily move from complete to incomplete informations with almost no work. In other words, our machinery turns out to be so powerful that can handle the incomplete information with slightly changes in the overall algorithm. Indeed, the simple required modifications are the following:

- In the definition of the problem, let $\Sigma_I = 2^{I \cup H}$ (instead of $\Sigma_I = 2^I$).
- Define the connectivity trees to be $\mathcal{L}^{el}$-labeled complete $(\{1, \ldots, el\} \times 2^I)$-trees (instead of $(\{1, \ldots, el\} \times \Sigma_I)$-trees). This ensures that the synthesized transducer behaves in the same way on input letters that differ only in their hidden components.
- As a result of the above change in the definition of connectivity trees, the expression $\bigoplus_{\sigma' \in \Sigma_I}$ in the transition function of $\mathcal{A}^{\mathcal{T}}_{\varphi}$ should be changed to $\bigoplus_{\sigma' \in 2^I}$.

All the proofs remain valid with the natural changes resulting from the above. Thus, our algorithm solves, with the same complexity, also the hierarchical synthesis problem with incomplete information, and we have:

Theorem 6.1. *The $\langle \mathcal{L}, el \rangle$-synthesis problem with incomplete information is* EXPTIME*-complete for a μ-calculus formula φ, and is* 2EXPTIME*-complete for an* LTL *formula (for el that is at most polynomial in $|\varphi|$ for μ-calculus, or at most exponential in $|\varphi|$ for* LTL*).*

7. Conclusion

We presented an algorithm for the synthesis of hierarchical systems which takes as input a library of hierarchical transducers and a sequence of specification formulas. Each formula drives the synthesis of a new hierarchical transducer based on the current library, which contains all the transducers synthesized in previous iterations together with the starting library. The main challenge in this approach is to come up with a single-round synthesis algorithm that is able to efficiently synthesize the required transducer at each round. We have provided such an algorithm that works efficiently, i.e., with the same complexity as the corresponding one for flat systems; and uniformly, i.e., it can handle different temporal logic

specifications, including the modal μ-calculus, as well as imperfect information. In order to ensure that the single-round algorithm makes real use of previously synthesized transducers we have suggested the use of auxiliary automata to enforce modularity criteria. We believe that by decoupling the process of enforcing modularity from the core algorithm for single-round synthesis we gain flexibility that allows one to apply different approaches to enforcing modularity, as well as future optimizations to the core synthesis algorithm.

Appendix A. The propositional μ-calculus and LTL

The propositional μ-calculus. The μ-calculus is a propositional modal logic augmented with least and greatest fixpoint operators. We consider a μ-calculus where formulas are constructed from Boolean propositions with Boolean connectives, the temporal operators *EX* ("exists next") and *AX* ("for all next"), as well as least (μ) and greatest (ν) fixpoint operators. We assume that μ-calculus formulas are written in positive normal form (negation only applied to atomic propositions).

Formally, let Σ_O and *Var* be finite and pairwise disjoint sets of *atomic propositions* and *propositional variables*. The set of *μ-calculus* formulas over Σ_O and *Var* is the smallest set such that

- true and false are formulas;
- p and $\neg p$, for $p \in \Sigma_O$, are formulas;
- $x \in \mathsf{Var}$ is a formula;
- $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are formulas if φ_1 and φ_2 are formulas;
- $AX\varphi$ and $EX\varphi$ are formulas if φ is a formula;
- $\mu y.\varphi(y)$ and $\nu y.\varphi(y)$ are formulas if y is a propositional variable and $\varphi(y)$ is a formula containing y as a free variable.

Observe that we use positive normal form, i.e., negation is applied only to atomic propositions.

We call μ and ν *fixpoint operators* and use λ to denote a fixpoint operator μ or ν. A propositional variable y occurs *free* in a formula if it is not in the scope of a fixpoint operator λy, and *bounded* otherwise. Note that y may occur both bounded and free in a formula. A *sentence* is a formula that contains no free variables. For a formula $\lambda y.\varphi(y)$, we write $\varphi(\lambda y.\varphi(y))$ to denote the formula that is obtained by one-step unfolding, i.e., replacing each free occurrence of y in φ with $\lambda y.\varphi(y)$.

The closure $cl(\varphi)$ of a μ-calculus sentence φ is the smallest set of μ-calculus formulas that contains φ and is closed under sub-formulas (that is, if ψ is in the closure, then so do all its sub-formulas that are sentences) and fixpoint applications (that is, if $\lambda y.\varphi(y)$ is in the closure, then so is $\varphi(\lambda y.\varphi(y))$). For every μ-calculus formula φ, the number of elements in $cl(\varphi)$ is linear in the length of φ. Accordingly, we define the size $|\varphi|$ of φ to be the number of elements in $cl(\varphi)$. A μ-calculus formula is *guarded* if for every variable y, all the occurrences of y that are in a scope of a fixpoint modality λ are also in a scope of a modality *AX* or *EX* that is itself in the scope of λ. Thus, a μ-calculus sentence is guarded if for all $y \in Var$, all the occurrences of y are in the scope of a next modality. Given a μ-calculus formula, it is always possible to construct in linear time an equivalent guarded formula (cf. [15,36]).

The semantics of the μ-calculus is defined with respect to a *Kripke structure* $\mathcal{S} = \langle \Sigma_O, W, in, R, \Lambda \rangle$, where Σ_O represents a set of atomic propositions, W is a finite set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $(w, w') \in R$), in_0 is an initial state, and $\Lambda : W \to 2^{\Sigma_O}$ maps each state to the set of atomic propositions true in that state. If $(w, w') \in R$, we say that w' is a *successor* of w. A *path* in S is an infinite sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geqslant 0$, $(w_i, w_{i+1}) \in R$. We denote the suffix $w_i, w_{i+1}, \ldots$ of π by π^i. We define the size $|S|$ of S as $|W| + |R|$.

Informally, a formula $EX\varphi$ holds at a state w of a Kripke structure S if φ holds at least in one successor of w. Dually, the formula $AX\varphi$ holds in a state w of a Kripke structure S if φ holds in all successors of w. Readers not familiar with fixpoints might want to look at [15,17,33,53] for instructive examples and explanations of the semantics of the μ-calculus. To formalize the semantics, we introduce valuations that allow to associate sets of points to variables. Given a Kripke structure $\mathcal{S} = \langle \Sigma_O, W, in, \mathcal{R}, \Lambda \rangle$ and a set $\{y_1, \ldots, y_n\}$ of propositional variables in Var, a *valuation* $\mathcal{V} : \{y_1, \ldots, y_n\} \to 2^W$ is an assignment of subsets of W to the variables $y_1, \ldots, y_n$. For a valuation $\mathcal{V}$, a variable y, and a set $W' \subseteq W$, we denote by $\mathcal{V}[y \leftarrow W']$ the valuation obtained from $\mathcal{V}$ by assigning W' to y. A formula φ with free variables among $y_1, \ldots, y_n$ is interpreted over the structure S as a mapping φ^S from valuations to 2^W, i.e., $\varphi^S(\mathcal{V})$ denotes the set of states that satisfy φ under valuation $\mathcal{V}$. The mapping φ^S is defined inductively as follows:

- $\mathsf{true}^S(\mathcal{V}) = W$ and $\mathsf{false}^S(\mathcal{V}) = \emptyset$;
- for $p \in \Sigma_O$, we have $p^S(\mathcal{V}) = W'$ such that $p \in \Lambda(w)$ for each $w \in W$ and $(\neg p)^S(\mathcal{V}) = W \setminus p^S(\mathcal{V})$;
- for $y \in \mathsf{Var}$, we have $y^S(\mathcal{V}) = \mathcal{V}(y)$;
- $(\varphi_1 \wedge \varphi_2)^S(\mathcal{V}) = \varphi_1^S(\mathcal{V}) \cap \varphi_2^S(\mathcal{V})$
- $(\varphi_1 \vee \varphi_2)^S(\mathcal{V}) = \varphi_1^S(\mathcal{V}) \cup \varphi_2^S(\mathcal{V})$;
- $(EX\varphi)^S(\mathcal{V}) = \{w \in W\colon\ \exists w'.(w, w') \in R \text{ and } w' \in \varphi^S(\mathcal{V})\}$;
- $(AX\varphi)^S(\mathcal{V}) = \{w \in W\colon\ \forall w'.\mathit{if}\,(w, w') \in R \text{ then } w' \in \varphi^S(\mathcal{V})\}$;
- $(\mu y.\varphi(y))^S(\mathcal{V}) = \bigcap\{W' \subseteq W\colon\ \varphi^S(\mathcal{V}[y \leftarrow W']) \subseteq W'\}$;
- $(\nu y.\varphi(y))^S(\mathcal{V}) = \bigcup\{W' \subseteq W\colon\ W' \subseteq \varphi^S(\mathcal{V}[y \leftarrow W'])\}$.

Note that no valuation is required for a sentence.

Let $\mathcal{S} = \langle \Sigma_O, W, in, R, \Lambda \rangle$ be a Kripke structure and φ a sentence. For a state $w \in W$, we say that φ *holds* at w in S, denoted $S, w \models \varphi$, if $w \in \varphi^S(\emptyset)$. S is a *model* of φ if there is a $w \in W$ such that $S, w \models \varphi$. Finally, φ is *satisfiable* if it has a model.

Linear temporal logic. *Linear Temporal Logic* (LTL) was introduced by Pnueli to specify and verify properties of reactive systems [46]. Given a set of atomic propositions Σ_O, an LTL formula is composed of atomic propositions, the Boolean connectives *conjunction* ($\wedge$) and *negation* ($\neg$), and the temporal operators *Next* (X) and *Until* ($\mathcal{U}$). LTL formulas are built up in the usual way from the above operators and connectives, according to the following grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid \varphi U \varphi$$

where p is an atomic proposition. The semantics of LTL formulas is given with respect to an infinite word $w = \sigma_0\sigma_1 \ldots \sigma_n \ldots$ over 2^{Σ_O}, which can be seen as a labeling of a path from a Kripke structure. The satisfaction relation $w \models \varphi$ is defined in the standard way:

- if φ is an atomic proposition, then $w \models \varphi$ if and only if $\varphi \in \sigma_0$;
- $w \models \neg\varphi$ if and only if $w \models \varphi$ does not hold;
- $w \models \varphi_1 \wedge \varphi_2$ if and only if $w \models \varphi_1$ and $w \models \varphi_2$;
- $w \models X\varphi$ if and only if $w_{\geqslant 1} \models \varphi$;
- $w \models \varphi_1 U \varphi_2$ if and only if there exists $i \geqslant 0$ such that $w_{\geqslant i} \models \varphi_2$ and $w_{\geqslant j} \models \varphi_1$ for all j such that $0 \leqslant j < i$.

References

[1] G. Alonso, F. Casati, H.A. Kuno, V. Machiraju, Web Services – Concepts, Architectures and Applications. Data-Centric Systems and Applications, Springer-Verlag, 2004.
[2] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, L. Libkin, First-order and temporal logics for nested words, Log. Methods Comput. Sci. 4 (4) (2008), Article 11, 44 pp.
[3] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T.W. Reps, M. Yannakakis, Analysis of recursive state machines, ACM Trans. Program. Lang. Syst. 27 (4) (2005) 786–818.
[4] R. Alur, S. Chaudhuri, K. Etessami, P. Madhusudan, On-the-fly reachability and cycle detection for recursive state machines, in: TACAS'05, in: LNCS, vol. 3440, Springer-Verlag, 2005, pp. 61–76.
[5] R. Alur, S. Chaudhuri, P. Madhusudan, A fixpoint calculus for local and global program flows, in: POPL'06, ACM, 2006, pp. 153–165.
[6] R. Alur, M. Yannakakis, Model checking of hierarchical state machines, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 273–303.
[7] B. Aminof, O. Kupferman, A. Murano, Improved model checking of hierarchical systems, in: VMCAI'10, in: LNCS, vol. 5944, Springer-Verlag, 2010, pp. 61–77.
[8] B. Aminof, O. Kupferman, A. Murano, Improved model checking of hierarchical systems, Inf. Comput. 210 (2012) 68–86.
[9] B. Aminof, A. Legay, A. Murano, O. Serre, μ-calculus pushdown module checking with imperfect state information, in: IFIP TCS'08, in: IFIP, vol. 273, Springer-Verlag, 2008, pp. 333–348.
[10] B. Aminof, A. Legay, A. Murano, O. Serre, M.Y. Vardi, Pushdown module checking with imperfect information, Inf. Comput. 213 (2013) 1–17.
[11] B. Aminof, A. Murano, M.Y. Vardi, Pushdown module checking with imperfect information, in: CONCUR'07, in: LNCS, vol. 4703, Springer-Verlag, 2007, pp. 460–475.
[12] S. Bliudze, J. Sifakis, A notion of glue expressiveness for component-based systems, in: CONCUR'08, in: LNCS, vol. 5201, Springer-Verlag, 2008, pp. 508–522.
[13] S. Bliudze, J. Sifakis, Synthesizing glue operators from glue constraints for the construction of component-based systems, in: Software Composition 2011, in: LNCS, vol. 6708, Springer-Verlag, 2011, pp. 51–67.
[14] A. Bohy, V. Bruyère, E. Filiot, N. Jin, J.-F. Raskin, Acacia+, a tool for ltl synthesis, in: CAV'12, in: LNCS, vol. 7358, Springer-Verlag, 2012, pp. 652–657.
[15] P.A. Bonatti, C. Lutz, A. Murano, M.Y. Vardi, The complexity of enriched mu-calculi, Log. Methods Comput. Sci. 4 (3) (2008), Article 11, 27 pp.
[16] L. Bozzelli, A. Murano, A. Peron, Pushdown module checking, Form. Methods Syst. Des. 36 (1) (2010) 65–95.
[17] J. Bradfield, C. Stirling, Modal μ-calculi, in: Handbook of Modal Logic, 2006, pp. 722–756.
[18] R.K. Brayton, G.D. Hachtel, A.L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S.A. Edwards, S.P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, T. Villa, Vis: A system for verification and synthesis, in: CAV'96, in: LNCS, vol. 1102, Springer-Verlag, 1996, pp. 428–432.
[19] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, F. Patrizi, Automatic service composition and synthesis: the roman model, IEEE Data Eng. Bull. 31 (3) (2008) 18–22.
[20] A. Church, Logic, arithmetics, and automata, in: Proc. International Congress of Mathematicians, 1962, Institut Mittag-Leffler, 1963, pp. 23–35.
[21] L. de Alfaro, T.A. Henzinger, Interface-based design, in: Engineering Theories of Software-Intensive Systems, in: NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, Springer-Verlag, 2005, pp. 83–104.
[22] G. De Giacomo, F. Patrizi, S. Sardiña, Automatic behavior composition synthesis, Artif. Intell. 196 (2013) 106–142.
[23] G. De Giacomo, S. Sardiña, Automatic synthesis of new behaviors from a library of available behaviors, in: IJCAI'07, 2007, pp. 1866–1871.
[24] R. Ehlers, Unbeast: Symbolic bounded synthesis, in: TACAS'11, in: LNCS, vol. 6605, Springer-Verlag, 2011, pp. 272–275.
[25] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D.V. Campenhout, Reasoning with temporal logic on truncated paths, in: CAV'03, in: LNCS, vol. 2725, Springer-Verlag, 2003, pp. 27–39.
[26] E. Emerson, Temporal and modal logic, in: J.V. Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, MIT Press, 1990, pp. 997–1072, Ch. 16.
[27] E. Emerson, C. Jutla, Tree automata, μ-calculus and determinacy, in: FOCS'91, IEEE, 1991, pp. 368–377.
[28] E.A. Emerson, E.M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, Sci. Comput. Program. 2 (3) (1982) 241–266.
[29] S. Göller, M. Lohrey, Fixpoint logics on hierarchical structures, in: FSTTCS'05, in: LNCS, vol. 3821, Springer-Verlag, 2005, pp. 483–494.
[30] D.P. Guelev, M.D. Ryan, P.Y. Schobbens, Synthesising features by games, Electron. Notes Theor. Comput. Sci. 145 (2006) 79–93.

[31] D. Harel, A. Pnueli, On the development of reactive systems, in: Logics and Models of Concurrent Systems, in: NATO Advanced Summer Institutes, vol. F-13, Springer-Verlag, New York, USA, 1985, pp. 477–498.
[32] D. Janin, I. Walukiewicz, Automata for the modal μ-calculus and related results, in: MFCS'95, in: LNCS, vol. 969, Springer-Verlag, 1995, pp. 552–562.
[33] D. Kozen, Results on the propositional μ-calculus, J. Theor. Comput. Sci. 27 (1983) 333–354.
[34] O. Kupferman, M. Vardi, μ-calculus synthesis, in: MFCS'00, in: LNCS, vol. 1893, Springer-Verlag, 2000, pp. 497–507.
[35] O. Kupferman, M. Vardi, Synthesis with incomplete information, Adv. Temp. Log. (2000) 109–127.
[36] O. Kupferman, M. Vardi, P. Wolper, An automata-theoretic approach to branching-time model checking, J. ACM 47 (2) (2000) 312–360.
[37] O. Kupferman, M.Y. Vardi, P. Wolper, Module checking, Inf. Comput. 164 (2) (2001) 322–344.
[38] R. Lanotte, A. Maggiolo-Schettini, A. Peron, Structural model checking for communicating hierarchical machines, in: MFCS'04, in: LNCS, vol. 3153, Springer-Verlag, 2004, pp. 525–536.
[39] Y. Lustig, M.Y. Vardi, Synthesis from component libraries, in: FOSSACS'09, in: LNCS, vol. 5504, Springer-Verlag, 2009, pp. 395–409.
[40] Y. Lustig, M.Y. Vardi, Synthesis from recursive-components libraries, in: GandALF 2011, in: EPTCS, vol. 54, 2011, pp. 1–16.
[41] Z. Manna, R.J. Waldinger, A deductive approach to program synthesis, ACM Trans. Program. Lang. Syst. 2 (1) (1980) 90–121.
[42] D. Muller, P. Schupp, Alternating automata on infinite trees, J. Theor. Comput. Sci. 54 (1987) 267–276.
[43] P. Müller, Modular Specification and Verification of Object-Oriented Programs, Springer-Verlag, 2002.
[44] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive designs, in: VMCAI'06, in: LNCS, vol. 3855, Springer-Verlag, 2006, pp. 364–380.
[45] C. Pixley, Formal verification of commercial integrated circuits, IEEE Des. Test Comput. 18 (4) (2001).
[46] A. Pnueli, The temporal logic of programs, in: FOCS'77, IEEE, 1977, pp. 46–57.
[47] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: POPL'89, ACM Press, 1989, pp. 179–190.
[48] M. Rabin, Weakly definable relations and special automata, in: Proc. Symp. Math. Logic and Foundations of Set Theory, North Holland, 1970, pp. 1–23.
[49] R. Rosner, Modular synthesis of reactive systems, PhD thesis, Weizmann Institute of Science, 1992.
[50] J. Sifakis, A framework for component-based construction extended abstract, in: SEFM'05, IEEE Computer Society, 2005, pp. 293–300.
[51] S. Sohail, F. Somenzi, Safety first: a two-stage algorithm for the synthesis of reactive systems, Int. J. Softw. Tools Technol. Transf. STTT (2012) 1–22.
[52] S. Sohail, F. Somenzi, K. Ravi, A hybrid algorithm for ltl games, in: VMCAI'08, in: LNCS, vol. 4905, Springer-Verlag, 2008, pp. 309–323.
[53] R.S. Streett, E.A. Emerson, An automata theoretic decision procedure for the propositional μ-calculus, Inf. Comput. 81 (3) (1989) 249–264.
[54] T. Wilke, Alternating tree automata, parity games, and modal μ-calculus, Bull. Soc. Math. Belg. 8 (2) (2001).