Program Complexity in Hierarchical Module Checking*

Aniello Murano[1], Margherita Napoli[2], and Mimmo Parente[2]

[1] Università di Napoli "Federico II", Via Cintia, 80126 - Napoli, Italy
[2] Università di Salerno, Via Ponte don Melillo, 84084 - Fisciano (SA), Italy

Abstract. *Module checking* is a well investigated technique for verifying the correctness of open systems, which are systems characterized by an ongoing interaction with an external environment. In the classical module checking framework, in order to check whether an open system satisfies a required property, we first translate the entire system into an open model (*module*) that collects all possible behaviors of the environment and then check it with respect to a formal specification of the property. Recently, in the case of closed system, Alur and Yannakakis have considered hierarchical structure models in order to have models exponentially more succinct. A hierarchical model uses as nodes both ordinary nodes and supernodes, which are hierarchical models themselves. For *CTL* specifications, it has been shown that for the simple case of models having only single-exit supernodes, the hierarchical model checking problem is not harder than the classical one. On the contrary, for the more general multiple-exit case, the problem becomes PSPACE-complete.
In this paper, we investigate the program complexity of the CTL *hierarchical module checking problem*, that is, we consider the module checking problem for a fixed *CTL* formula and modules having also supernodes that are modules themselves. By exploiting an automata-theoretic approach through the introduction of hierarchical Büchi tree automata, we show that, in the single-exit case, the addressed problem remains in PTIME, while in the multiple-exit case, it becomes PSPACE-complete.

1 Introduction

Module checking is a useful technique that allows to verify the correctness of open systems [KVW01]. While the behavior of a closed system is fully characterized by internal states, an open system maintains an ongoing interaction with an external environment, and its behavior is fully affected by this interaction.

Classically, in order to check whether an open system satisfies a required property, we translate the entire system into a *module*, that is in a labeled state-transition graph whose set of states is partitioned into a set of system states (where the system makes a transition) and a set of environment states (where the environment makes a transition). Given a module $\mathcal{M}$, describing the system

* Work partially supported by MIUR PRIN Project no.2007-9E5KM8 and grant "Formal Methods for Closed and Open Systems" ex-60% 2006 Università di Salerno.

to be verified, and a temporal logic formula φ, specifying the desired behavior of the system, module checking asks whether, for all possible environments, $\mathcal{M}$ satisfies φ. Therefore, while in *model checking* it is sufficient to check whether the full computation tree obtained by unwinding $\mathcal{M}$ satisfies φ, in module checking it is necessary to verify that all trees obtained from the full computation tree by pruning some subtrees rooted in nodes corresponding to choices disabled by the environment (those trees represent the interactions of $\mathcal{M}$ with all the possible environments) satisfy φ. We collect all such trees in a set named $exec(\mathcal{M})$.

As a classic example of closed and open systems, we can think of two drink-dispensing machines. One machine, which is a closed system, repeatedly boils water, makes an internal nondeterministic choice, and serves either coffee or tea. The second machine, which is an open system, repeatedly boils water, asks the environment to choose between coffee and tea, and deterministically serves a drink according to the external choice. Both machines induce the same infinite tree of possible executions. Nevertheless, while the behavior of the first machine is determined by internal choices solely, the behavior of the second machine is determined also by external choices, made by its environment. Formally, in a closed system, the environment cannot modify any of the system variables. In contrast, in an open system, the environment can modify some of them. In [KVW01,AMV07], it has been shown that for systems modeled as single modules and specifications as branching time temporal logic formulas, module checking is exponentially harder than model checking.

In formal verification a very interesting question is how complex is to check for system correctness in the case we fix the specification. This question is usually addressed as the *program complexity* and in more details concerns the complexity of the verification question for the set $\{\mathcal{M} \mid \mathcal{M} \text{ satisfies } \varphi\}$, for a fixed formula φ[VW86]. Program complexity is receiving great attention in formal verification due to the fact that often the size of the system widely exceeds that of the formula, which is usually very small and therefore considered constant. This allows us to use in practice formal verification techniques whenever they result tractable with respect to the system. For example, we recall that for finite-state systems and specifications given as formulas of the branching-time temporal logic *CTL* ([CE81]), module checking is EXPTIME-complete, but the corresponding problem with a constant size formula is only PTIME-complete [KVW01].

Recently, in the case of complex closed systems, hierarchical structure models have been usefully considered, in order to have models exponentially more succinct. A hierarchical model uses as nodes both ordinary nodes or supernodes, which are models themselves [AY01,ABE+05,LNPP08]. The straightforward way to analyze a hierarchical closed machine is to flatten it (thus, incurring an exponential blow up) and apply a model checking tool on the resulting ordinary model. In [AY01], it has been shown that for linear-time specifications such as *LTL*, the cost of flattening can be avoided by showing that the *LTL* hierarchical model checking problem is not harder than the classical one. The same happens for *CTL* specifications, for models having one exit node. In the general case,

instead, *CTL* hierarchical model checking becomes exponentially harder and, in particular, the program complexity is PSPACE-complete.

In this paper, we investigate the program complexity of the module checking problem for *CTL* in the case of modules expressed by *hierarchical modules*, that is nodes of the module can be ordinary nodes or supernodes which are modules themselves. As a simple example, consider again the above drink-dispensing machine. Now suppose that both in the cases the environment makes a coffee or tea choice, the system allows the environment to have an extra choice between regular sugar or diet sugar. In both cases, we can remand the choice to another module. As for non-hierarchical open systems, in case we want to check whether it is possible for the designed hierarchical open machine to serve coffee with regular sugar, a straightforward way is to flatten it and then, by using the classical module checking technique, check whether the flatten module satisfies the *CTL* formula $AGEF\ coffe\_with\_regular\_sugar$. Unfortunately, by flattening the hierarchical module, we increase exponentially the size of the module and we immediately get that *CTL* hierarchical module checking is EXPTIME w.r.t. both the sizes of the hierarchical module and the formula.

In this paper, we show that for the addressed problem the cost of flattening can be avoided. In particular, we show that for single-exit hierarchical modules, the program complexity of the *CTL* hierarchical module checking problem is not harder than the classical one, while in the case of multiple-exit hierarchical modules, the addressed problem becomes PSPACE-complete.

For the upper bounds, we use an automata-theoretic approach via tree automata. In particular, we introduce *hierarchical nondeterministic Büchi tree automata* (*HNBT*) and use a reduction to the emptiness problem for this automata. In more details, given a hierarchical module $\mathcal{M}$ and a *CTL* formula φ, we first construct in polynomial time an *HNBT* $\mathcal{A}_\mathcal{M}$ accepting $exec(\mathcal{M})$. The construction of $\mathcal{A}_\mathcal{M}$ we propose here extends that used in [KVW01] by also taking into account that $\mathcal{M}$ is in a hierarchical shape. Thus, $\mathcal{A}_\mathcal{M}$ will have, for each supernode in the hierarchical module (which is a hierarchical module itself) a corresponding supernode (which is a hierarchical automaton itself) with the same number of exit nodes. From the formula side, accordingly to [KVW00], we construct in exponential time a nondeterministic Büchi tree automaton (*NBT*) $\mathcal{A}_{\neg\varphi}$ accepting all models that do not satisfy φ, with the intent to check that none of them are in $exec(\mathcal{M})$. Thus, we check that $\mathcal{M}$ models φ for every possible choice of the environment by checking whether $\mathcal{L}(\mathcal{A}_\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$ is empty. To obtain the result, we first show that the product of the *HNBT* $\mathcal{A}_\mathcal{M}$ with the *NBT* $\mathcal{A}_{\neg\varphi}$ can be performed in polynomial time, turning into an *HNBT* having a number of exit nodes that depends on the number of states of $\mathcal{A}_{\neg\varphi}$, which in turn depends on the size of φ. Since we are interested on the program complexity of the hierarchical module checking problem, we assume the formula to be fixed. Therefore, the obtained *HNBT* will have multiple exits if $\mathcal{A}_\mathcal{M}$ does, and a constant number of exit-nodes otherwise. Then, we show that the emptiness problem for an *HNBT* can be solved in PTIME if it only admits constant (and in particular single-) exits, while it is PSPACE-complete in the case of multiple ex-

its. Thus, we get the desired upper bounds. To show matching lower bounds, for the single-exit case, we recall that the program complexity of the classical module checking problem is PTIME-hard and that, for multiple exits, the program complexity of *CTL* hierarchical model checking closed system is PSPACE-hard.

The paper is self-contained and is organized as follows. In the next section, we give basic definitions, introduce open hierarchical state machines, and define the hierarchical module checking problem for *CTL*. In Section 3, we briefly recall *NBT* and introduce *HNBT*. Then, we solve the emptiness problem for *HNBT*. Finally, in Section 4, we solve the hierarchical module checking problem for *CTL*.

2 Preliminary

In this section, we introduce the *hierarchical module checking problem* for *CTL*.

Let $\mathbb{N}$ be the set of positive integers. A *tree* T is a prefix closed subset of $\mathbb{N}^*$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For $x \in T$, the set of *children* of x (in T) is $children(T, x) = \{x \cdot i \in T \mid i \in \mathbb{N}\}$. For $k \geq 1$, the complete k-ary tree is the tree $\{1, \ldots, k\}^*$. For $x \in T$, a path π of T from x is a set $\pi \subseteq T$ such that $x \in \pi$ and for each $y \in \pi$ such that $children(T, y) \neq \emptyset$, there is exactly one node in $children(T, y)$ belonging to π. In the following, for *a path of* T, we mean a path of T from the root ε. For an alphabet Σ, a Σ-labeled tree is a pair (T, V), where T is a tree and $V : T \to \Sigma$ maps each node of T to a symbol in Σ.

In this paper, we consider open systems, i.e. systems that interact with their environment and whose behavior depends on this interaction. The global behavior of such a system is described by a finite state machine (also called *module* [KVW01]) $\mathcal{M} = (AP, S, E, R, in, L)$, where AP is a finite set of atomic propositions, $S \cup E$ is a finite set of states partitioned into a set S of *system* states and a set E of *environment* states (we use W to denote $S \cup E$), $R \subseteq W \times W$ is a total transition relation, $in \in W$ is an initial state, and $L : W \to 2^{AP}$ maps each state w to the set of atomic propositions that hold in w. For $(w, w') \in R$, we say that w' is a successor of w. For each state $w \in W$, we denote by $succ(w)$ the ordered tuple of w's successors. When the module $\mathcal{M}$ is in a system state w_s, then all the states in $succ(w_s)$ are possible next states. On the other hand, when $\mathcal{M}$ is in an environment state w_e, then the possible next states (that are in $succ(w_e)$) depend on the current environment. Since the behavior of the environment is not predictable, we have to consider all the possible sub-tuples of $succ(w_e)$. The only constraint, since we consider environments that cannot block the system, is that at least one transition from w_e exists leading into a next state in $succ$ (not all these transitions may be disabled by the environment).

The set of all the maximal computations of $\mathcal{M}$ starting from the initial state in is described by a W-labeled tree $(T_{\mathcal{M}}, V_{\mathcal{M}})$, called *computation tree*, which is obtained by unwinding $\mathcal{M}$ in the usual way. The problem of deciding, for a given branching-time formula ψ over AP, whether $(T_{\mathcal{M}}, L \circ V_{\mathcal{M}})$ satisfies ψ, denoted $\mathcal{M} \models \psi$, is the usual *model-checking problem* [CE81,QS81]. On the other hand, for an open system, $(T_{\mathcal{M}}, V_{\mathcal{M}})$ corresponds to a very specific environment, i.e. a

maximal environment that never restricts the set of its next states. Therefore, when we examine a branching-time specification ψ w.r.t. a module $\mathcal{M}$, ψ should hold not only in $(T_{\mathcal{M}}, V_{\mathcal{M}})$, but also in all the trees obtained by pruning from $(T_{\mathcal{M}}, V_{\mathcal{M}})$ subtrees whose root is a child (successor) of a node corresponding to an environment state. The set of these labeled trees is denoted by $exec(\mathcal{M})$, and is formally defined as follows. $(T, V) \in exec(\mathcal{M})$ iff $T \subseteq T_{\mathcal{M}}$, V is the restriction of $V_{\mathcal{M}}$ to the tree T, and for all $x \in T$ the following holds:

- if $V_{\mathcal{M}}(x) = w \in S$ and $succ(w) = \langle w_1, \ldots, w_n \rangle$, then $children(T, x) = \{x \cdot 1, \ldots, x \cdot n\}$ (note that for $1 \leq i \leq n$, $V(x \cdot i) = V_{\mathcal{M}}(x \cdot i) = w_i$);
- if $V_{\mathcal{M}}(x) = w \in E$ and $succ(w) = \langle w_1, \ldots, w_n \rangle$, then there is a subtuple $\langle w_{i_1}, \ldots, w_{i_p} \rangle$ of $succ(w)$, with $p \geq 1$, such that $children(T, x) = \{x \cdot i_1, \ldots, x \cdot i_p\}$ (note that for $1 \leq j \leq p$, $V(x \cdot i_j) = V_{\mathcal{M}}(x \cdot i_j) = w_{i_j}$).

Intuitively, each labeled tree (T, V) in $exec(\mathcal{M})$ corresponds to a different behavior of the environment. In the following, we consider the trees in $exec(\mathcal{M})$ as 2^{AP}-labeled trees, i.e. taking the label of a node x to be $L(V(x))$.

In this paper, we consider the branching-time temporal logic *CTL* as system specification. *CTL* was introduced by Emerson and Clarke in 1981 [CE81] as a tool for specifying and verifying concurrent programs. *CTL* formulas are built from a set AP of *atomic propositions* using boolean operators, the linear-temporal operators X ("next time") and U ("until"), coupled with the path quantifiers A ("for all paths") or E ("for some path"). For a formal definition of *CTL* see [CGP99]. The *closure* $cl(\varphi)$ of a *CTL* formula φ is the set of all subformulas of φ, including φ. The size $|\varphi|$ of φ is defined as the number of elements in $cl(\varphi)$. Given a *CTL* formula φ, we say that (T, V) satisfies φ if $((T, V), \varepsilon) \models \varphi$.

For a module $\mathcal{M}$ and a *CTL* formula ψ, we say that $\mathcal{M}$ satisfies ψ, denoted $\mathcal{M} \models_r \psi$, if all the trees in $exec(M)$ satisfy ψ. The problem of deciding whether $\mathcal{M}$ satisfies ψ is called *module checking* [KVW01]. Note that $\mathcal{M} \models_r \psi$ implies $\mathcal{M} \models \psi$ (since $(T_{\mathcal{M}}, V_{\mathcal{M}}) \in exec(\mathcal{M})$), but the converse in general does not hold. Also, note that $\mathcal{M} \not\models_r \psi$ is *not* equivalent to $\mathcal{M} \models_r \neg\psi$. Indeed, $\mathcal{M} \not\models_r \psi$ just states that there is some tree $(T, V) \in exec(\mathcal{M})$ satisfying $\neg\psi$.

Open Hierarchical State Machines. An *open hierarchical state machine*, or *hierarchical module* $\mathcal{M}$ over a set AP of atomic propositions is a tuple $(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ of *components*, where each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$, $1 \leq i \leq n$, has the following elements:

- A finite set S_i of *system nodes*;
- A finite set E_i of *environment nodes*. We assume $S_i \cap E_i = \emptyset$, and $W_i = S_i \cup E_i$;
- A finite set Box of *boxes* (or supernodes). We assume $W_i \cap Box_i = \emptyset$;
- An initial node in_i of W_i;
- A subset O_i of W_i, called exit-nodes.
- A labeling function $L_i : W_i \to 2^{AP}$ labeling each node with a subset of AP.
- An indexing function $Y_i : Box_i \to \{i + 1, \ldots, n\}$ that maps each box of the i-th component to an index greater than i. That is, if $Y_i(b) = j$, for a box b of $\mathcal{M}_i$, then b can be viewed as a reference to the component $\mathcal{M}_j$.

– An edge relation R_i. Each edge in R_i is a pair (u, v) with source u and sink v: source u either is a node of $\mathcal{M}_i$, or is a pair (u_1, u_2), where u_1 is a box of $\mathcal{M}_i$ with $Y_i(u_1) = j$ and u_2 is an exit-node of $\mathcal{M}_j$, and the sink v is either a node or a box of $\mathcal{M}_i$.

The edges connect nodes and boxes with one another. Edges entering a box implicitly connect to the unique initial node of the component associated with that box. On the other hand, edges exiting a box explicitly specify an exit-node among the possible exit-nodes of the component associated with that box. A hierarchical module is closed (called *hierarchical model* in [AY01]) if for all components $\mathcal{M}_i$, we have $E_i = \emptyset$.

By extending an idea used for closed hierarchical models, we can associate to a hierarchical module an ordinary flat module, by recursively substituting each box with the component indexed by the box. Since different boxes can be associated with the same component, each node can appear in different contexts. The expanded flat module will be denoted $\mathcal{M}^f$. Its states are tuples $\langle u_1, \ldots, u_h \rangle$, $h \geq 1$, whose last component u_h is a node, while all the other are boxes. Moreover, each u_j belongs to the $\mathcal{M}_i$ which the box u_{j-1} refers to. A state in the flat module is either a system or environment state depending on whether u_h is a system or an environment node, and also the propositional labeling of the state is determined by the labeling of u_h.

Now we proceed to a formal definition of expansion of a hierarchical module $\mathcal{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$. For each component $\mathcal{M}_i$, we define the module $\mathcal{M}_i^f = (AP, S_i^f, E_i^f, R_i^f, in_i^f, L_i^f)$ as the expanded structure of $\mathcal{M}_i$ obtained as follows:

– $in_i^f = \langle in_i \rangle$;
– The set S_i^f (resp., E_i^f) of system (resp., environment) nodes of $\mathcal{M}_i^f$ is defined inductively:
  • if u is a system (resp., environment) node of $\mathcal{M}_i$ then $\langle u \rangle$ belongs to S_i^f (resp., E_i^f);
  • if u is a box of $\mathcal{M}_i$ with $Y_i(u) = j$, and $\langle u_1, \ldots, u_h \rangle$ is a system (resp., environment) state of $\mathcal{M}_j^f$, where $h \geq 1$, then $\langle u, u_1, \ldots, u_h \rangle$ belongs to S_i^f (resp., E_i^f).
– The transition relation R_i^f of $\mathcal{M}_i^f$ is defined inductively as follows:
  • for $(u, v) \in R_i$, if the sink v is a node then $(\langle u \rangle, \langle v \rangle) \in R_i^f$, and if v is a box with $Y_i(v) = j$ then $(\langle u \rangle, \langle v, in_j \rangle) \in R_i^f$;
  • if w is a box of $\mathcal{M}_i$ with $Y_i(w) = j$, and $(\langle u_1, \ldots, u_h \rangle, \langle v_1, \ldots, v_{h'} \rangle)$ is a transition of $\mathcal{M}_j^f$, for $h, h' \geq 1$, then $(\langle w, u_1, \ldots, u_h \rangle, \langle w, v_1, \ldots, v_{h'} \rangle)$ belongs to R_i^f.
– The labeling function $L_i^f : W_i^f \to 2^{AP}$ of $\mathcal{M}_i^f$ (where $W_i^f = S_i^f \cup E_i^f$) is defined inductively as follows:
  • if w is a node of $\mathcal{M}_i$, then $L_i^f(\langle w \rangle) = L_i(w)$;
  • if $w = \langle u, u_1, \ldots, u_h \rangle$, where $h \geq 1$, and u is a box of $\mathcal{M}_i$ with $Y_i(u) = j$, then $L_i^f(w) = L_j^f(\langle u_1, \ldots, u_h \rangle)$.

The module $\mathcal{M}_1^f$ is the expanded structure of $\mathcal{M}$ and therefore we just indicate it as $\mathcal{M}^f$ in the following.

The size $|\mathcal{M}_i|$ of $\mathcal{M}_i$ is the sum of $|W_i|$, $|Box_i|$, and $|R_i|$. The size of the hierarchical module $\mathcal{M}$ is the sum of the sizes of all $\mathcal{M}_i$. The nesting depth of $\mathcal{M}$, denoted $nd(\mathcal{M})$, is the length of the longest chain $i_1, i_2, \ldots, i_j$ of indices such that a box of $\mathcal{M}_{i_l}$ is mapped to i_{l+1}. Observe that each state of the expanded structure is a vector of length at most the nesting depth, and the size of the expanded module $\mathcal{M}^f$ can be exponential in the nesting depth, and is $O(|\mathcal{M}|^{nd(\mathcal{M})})$.

The *hierarchical module checking problem* for *CTL* is to decide for a given hierarchical module $\mathcal{M}$ and a *CTL* formula φ, whether $\mathcal{M}^f \models_r \varphi$.

As noted above, the last component of every state is a node (all the others being boxes), and the system or environment nature of the last component as well as its propositional labeling determines the nature and the propositional labeling of the entire state, respectively.

In the following sections we will consider the cases of hierarchical module *single-exit* (all the O_i contain just element), or *multiple-exit* and the special case of hierarchical module with a constant number of exit-nodes (*constant-exit*).

3 Tree automata

In order to solve the program complexity of the hierarchical module checking problem for *CTL*, we use an automata theoretic approach; in particular, we exploit the formalisms of *Nondeterministic Büchi Tree Automata* (*NBT*) [Rab70,VW86] and introduce *Hierarchical Nondeterministic Büchi Tree Automata* (*HNBT*), that is *NBT* where states can be either ordinary node states or box states, which are tree automata themselves. Analogously to hierarchical modules, we consider both the cases single- or multiple-exit *HNBT*. *HNBT* extend to infinite trees the notion of hierarchical automata introduced in [ABE$^+$05] on infinite words.

Nondeterministic Büchi Tree Automata (*NBT*). Here, we briefly recall the definition of *NBT* over complete k-ary trees, for a given $k \geq 1$. Formally, a *NBT* is a tuple $\mathcal{A} = (\Sigma, Q, in, \delta, \mathcal{F})$, where Σ is a finite input alphabet, Q and in are as in modules and they represent a finite set of states, and an initial state, respectively; $\delta : Q \times \Sigma \to 2^{Q^k}$ is a transition function, and $\mathcal{F} \subseteq Q$ is a Büchi acceptance condition.

Intuitively, when the automaton is in state q, reading an input node x labeled by $\sigma \in \Sigma$, then the automaton chooses a tuple $(q_1, \ldots, q_k) \in \delta(q, \sigma)$ and splits in k copies such that for each $1 \leq i \leq k$, a copy in state q_i is sent to the node $x \cdot i$ in the input tree.

A run of $\mathcal{A}$ on a Σ-labeled k-ary tree (T, V) (where $T = \{1, \ldots, k\}^*$) is a Q-labeled tree (T, r) such that $r(\varepsilon) = in$ and for each $x \in T$, we have that $(r(x \cdot 1), \ldots, r(x \cdot k)) \in \delta(r(x), V(x))$. For a path $\pi \subseteq T$, let $inf_r(\pi) \subseteq Q$ be the set of states that appear as the labels of infinitely many nodes in π. For a Büchi condition $\mathcal{F} \subseteq Q$, π is *accepting* if $inf_r(\pi) \cap \mathcal{F} \neq \emptyset$. A run (T, r) is *accepting* if all its paths are accepting. The automaton $\mathcal{A}$ accepts an input tree (T, V) iff there

is an accepting run of $\mathcal{A}$ over (T, V). The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of Σ-labeled (complete) k-ary trees accepted by $\mathcal{A}$. The emptiness problem for $\mathcal{A}$ is to check whether $\mathcal{L}(\mathcal{A}) = \emptyset$. The *size* $|\mathcal{A}|$ of an *NBT* $\mathcal{A}$ is $|Q| + |\delta|$, note that $|\delta|$ is at most $|\Sigma| \cdot |Q|^{k+1}$.

It is well-known that formulas of *CTL* can be translated into equivalent tree automata (accepting the models of the given formula). In particular, given a *CTL* formula φ one can construct an *NBT* over k-ary trees, for some degree $k \geq 1$, having as number of states (independent from k) $2^{O(|\varphi| \log(|\varphi|))}$, as stated in the following lemma.

Lemma 1 ([KVW00,Var98]).
Given a CTL *formula φ over AP and $k \geq 1$, one can construct a* NBT $\mathcal{A}_\varphi$ *with number of states $2^{O(|\varphi| \log |\varphi|)}$ (independent from k) that accepts exactly the set of 2^{AP}-labeled complete k-ary trees that satisfy φ.*

Hierarchical Nondeterministic Büchi Tree Automata(*HNBT*). We now introduce *HNBT* over complete k-ary trees (for a given $k \geq 1$), as a hierarchical extension of *NBT*. An *HNBT* $\mathcal{A}$ over an alphabet Σ is a tuple $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, with each $\mathcal{A}_i$ such that $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$ where W_i, Box_i, in_i, O_i, and Y_i are as in the components of hierarchical modules, $\mathcal{F}_i \subseteq W_i$ is a set of accepting states, and $\delta_i : (W_i \cup (Box_i \times \bigcup_{j>i} O_j)) \times \Sigma \to 2^{(W_i \cup Box_i)^k}$. A tuple $(q_1, \ldots, q_k)$ is in $\delta_i(q, \sigma)$ only if either $q \in W_i$ or q is a pair (b, s), where $b \in Box_i$ with $Y_i(b) = j$ and s is an exit-node of $\mathcal{A}_j$. Moreover, each q_i can be either a node state or a box state of $\mathcal{A}_i$.

The *size* $|\mathcal{A}_i|$ is $|W_i| + |Box_i| + |\delta_i|$, note that $|\delta_i|$ is at most $|\Sigma| \cdot (|W_i| + |Box_i|)^{k+2}$. The size of $\mathcal{A}$ is the sum of the sizes of all $\mathcal{A}_i$. Similarly to hierarchical modules, we can flat a *HNBT* $\mathcal{A}$ into an *NBT* $\mathcal{A}^f$ by defining the NBT's $\mathcal{A}_i^f$ similarly to what has been done for $\mathcal{M}_i^f$. Thus a state of $\mathcal{A}_i^f$ is a tuple consisting of all box states and having necessarily as last component a node state. A state of $\mathcal{A}_i^f$ is final if its last component is in $\mathcal{F}_j$, with $j \geq i$. A tree (T, V) is accepted by a *HNBT* $\mathcal{A}$ if there is an accepting run of $\mathcal{A}^f$ on (T, V). The language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of the accepted trees.

To exploit the automata theoretic approach for the module checking problem for hierarchical module, we solve the emptiness problem for *HNBT*.

Lemma 2. *The emptiness problem for a single-exit* HNBT *is in* PTIME. *The emptiness problem for a multiple-exit* HNBT *is in* PSPACE.

Proof (sketch). For an *NBT* $\mathcal{A}$, one can check in polynomial time its emptiness by simply checking whether there exists in $\mathcal{A}$ a set G of "good" final states which is reachable by itself (i.e., for each state w of G there is a run that contains a subtree starting from w and whose frontier is contained in G) and that from the initial state of $\mathcal{A}$ it is possible to reach G [Rab70,VW86].

We now prove that also in the case of single-exit *HNBT* we can check emptiness in polynomial time, by opportunely embedding a component-wise exploration of the hierarchical automaton into the above *NBT*'s emptiness algorithm.

In fact, for a single-exit *HNBT* $\mathcal{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ instead of checking emptiness for the flat $\mathcal{A}_1^f$ we will use a simple property (checkable in polynomial time) on suitable NBTs $\widehat{\mathcal{A}}_i$ and sets $\widehat{Box_i}$, constructed from $\mathcal{A}_i$. Let us for the moment give a non-constructive definition of these latter sets: for each i, let $\widehat{Box_i} = \{b \in Box_i \mid Y_i(b) = j,\ \mathcal{L}(\mathcal{A}_j^f) \neq \emptyset\}$. Moreover, let each $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, ini_i, O_i, Y_i, \mathcal{F}_i)$, the *NBT* $\widehat{\mathcal{A}}_i$ is a tuple $(\Sigma, \widehat{Q}_i, \widehat{in}_i, \widehat{\delta}_i, \widehat{\mathcal{F}}_i)$, where

- $\widehat{Q}_i \subseteq W_i \cup Box_i$ and $W_i \subseteq \widehat{Q}_i$;
- $\widehat{\mathcal{F}}_i \subseteq \mathcal{F}_i \cup Box_i$ and $\mathcal{F}_i \subseteq \widehat{\mathcal{F}}_i$;

To each $\widehat{\mathcal{A}}_i$ can be associated a set of non complete k-ary trees, possibly having some finite paths, which can be seen, roughly speaking, as accepted by a Tree Automaton (not a NBT) with the following acceptance conditions: on the finite paths the acceptance is obtained considering the states of $\widehat{Box_i}$ as final states, while the infinite paths are accepted with the usual Büchi condition $\widehat{\mathcal{F}}_i$. To check the emptiness for the given *HNBT*, we will check whether such set of trees for $\widehat{\mathcal{A}}_1$ is empty. For each $\widehat{\mathcal{A}}_i$ we distinguish three different kinds of paths π of its runs, all starting from $\widehat{in}_i$:

A. π either is infinite and goes through a state of $\widehat{\mathcal{F}}_i$ infinitely often or is finite and its last state belongs to $\widehat{Box_i}$.
B. π is a finite path whose last state is in O_i (recall that $O_i \subseteq \widehat{Q}_i$, since $W_i \subseteq \widehat{Q}_i$) and π does not contain any state of $\widehat{\mathcal{F}}_i$.
C. π is a finite path whose last state is in O_i and it does contain at least one state of $\widehat{\mathcal{F}}_i$.

Let us now define inductively and bottom-up all $\widehat{\mathcal{A}}_i$ and the $\widehat{Box_i}$. For the base, let $\widehat{\mathcal{A}}_n = \mathcal{A}_n$ and $\widehat{Box_n} = \emptyset$. Suppose now that for $1 \leq i \leq n$ we have already defined all $\widehat{\mathcal{A}}_j$, for $j > i$. The sets $\widehat{Q}_i$ and $\widehat{\mathcal{F}}_i$ contain W_i and $\mathcal{F}_i$, respectively and, given a box $b \in Box_i$ such that $Y_i(b) = j$, we have that:

- if there exists a run of $\widehat{\mathcal{A}}_j$ containing at least a type-**B** path and all the others are either type-**A** paths or type-**C** paths, then $b \in \widehat{Q}_i$.
- if there exists a run of $\widehat{\mathcal{A}}_j$ containing at least a type-**C** path and all the remaining are of type-**A** paths, then $b \in \widehat{Q}_i$ and $b \in \widehat{\mathcal{F}}_i$.
- if there exists a run of $\widehat{\mathcal{A}}_j$ whose all paths are of type-**A** paths, then $b \in \widehat{Box_i}$ and $b \in \widehat{Q}_i$.

Moreover $\widehat{in}_i = in_i$ and

- for $q \in W_i$, $\widehat{\delta}_i(q, \sigma) = \delta_i(q, \sigma)$ and
- for $q \in Box_i \cap \widehat{Q}_i$, $\widehat{\delta}_i(q, \sigma) = \delta_i((q, s), \sigma)$ with $s \in O_i$.

Observe that the set $\widehat{Box_i}$ contains a box $b \in Box_i$ if it appears in an accepting run of $\mathcal{A}_i^f$. Moreover if there exists a run of $\widehat{\mathcal{A}}_j$ containing a type-**C** path π, then this run can be taken infinitely often in an accepting run (T, r) of

$\mathcal{A}_i^f$: in this way, in fact, the final state of $\widehat{\mathcal{A}}_j$ occurring in π, appears infinitely often in some paths of (T, r). Thus we consider b as a final state of $\widehat{\mathcal{A}}_i$.

Now it is easy to see that a tree (T, V) is accepted by $\mathcal{A}_i^f$ if and only if there exists a run of $\widehat{\mathcal{A}}_i$ whose paths are all of type-**A** . Observe that the set $\widehat{Box_i}$ is now defined constructively and is consistent with the previous definition.

Now since $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if there exists a tree (T, V) accepted by $\mathcal{A}_1^f$, we have that an algorithm to solve the emptiness problem can be easily given, based on the construction of $\widehat{\mathcal{A}}_i$'s. Actually it remains to be convinced that the existence of the runs required in those constructions can be done in polynomial time. This can be accomplished by using a fixed-point algorithm which resembles the one given by Rabin in [Rab70] for *NBT*. In an extended version of the paper we will give the full details of such algorithm.

Consider now a multiple-exit *HNBT*A. We now sketch a nondeterministic algorithm running in polynomial space, and from Savitch's theorem, we get our result. As in the single-exit case, we will construct some NBT $\widehat{\mathcal{A}}_i$'s, but now the construction is accomplished nondeterministically and in a top-down way, starting from $i = 1$. To obtain each $\widehat{\mathcal{A}}_i$, the algorithm, for each box $b \in Box_i$ with $Y_i(b) = j$, either guesses that $\mathcal{L}(\mathcal{A}_j^f) \neq \emptyset$ or chooses two (possibly empty) sets $X \subseteq O_j$ and $Y \subseteq O_j$ and guesses, for some $s \in O_j$, that there exists a run in $\widehat{\mathcal{A}}_j$ having type-**B** paths from $\widehat{in}_j$ to $s \in X$ and type-**C** paths from $\widehat{in}_j$ to $s \in Y$. According to these choices, the NBT $\widehat{\mathcal{A}}_i$ is constructed, similarly as in the case of single-exit. Then the algorithm proceeds by checking the guessed property of $\widehat{\mathcal{A}}_{Y_i(b)}$, for each $b \in Box_i$. In this step $\widehat{\mathcal{A}}_{Y_i(b)}$ is constructed, and this obviously, implies other guesses for the boxes belonging to $\mathcal{A}_{Y_i(b)}$. This chain of guesses naturally ends when $\widehat{\mathcal{A}}_n$ has to be constructed, since Box_n is empty. Observe that the overall space necessary during the execution of the algorithm does not exceed the size of $\mathcal{A}$ and this concludes the proof. □

We now conclude this section by showing that the above results are also tight. For the single-exit case, notice that *NBT* are a special case of *HNBT* and for *NBT* the emptiness problem is already known to be PTIME-hard.

For the multiple-exit case, we use a polynomial reduction from the model-checking problem for multiple-exit hierarchical state machines w.r.t constant *CTL* formulas, which is known to be PSPACE-hard [AY01]. In order to apply this reduction, we have first to define a *cross product* between an *HNBT* $\mathcal{A}'$ and an *NBT* $\mathcal{A}''$, say it $\mathcal{A}' \otimes \mathcal{A}''$, that allows to construct in polynomial time an *HNBT* whose flattening is equivalent to the Cartesian product of the *NBT*s $\mathcal{A}'^f$ and $\mathcal{A}''$. We now formally show how to construct $\mathcal{A}' \otimes \mathcal{A}''$.

Let $\mathcal{A}' = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be an *HNBT*, with $\mathcal{A}_i = (\Sigma, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$, and $\mathcal{A}'' = (\Sigma, Q, in, \delta, \mathcal{F}'')$ with $Q = \{q_1, \ldots, q_m\}$ and $in = q_1$. The product $\mathcal{A}' \otimes \mathcal{A}''$ is the *HNBT* $\mathcal{A} = \langle \mathcal{A}_{11}, \ldots, \mathcal{A}_{1m}, \ldots, \mathcal{A}_{n1}, \ldots, \mathcal{A}_{nm} \rangle$, where each component $\mathcal{A}_{ij} = (\Sigma, W_i \times Q, \delta_{ij}, Box_i \times Q, (in_i, q_j), O_i \times Q, Y_{ij}, \mathcal{F}_{ij})$, $1 \leq i \leq m$ and $1 \leq j \leq n$, is such that

- $\mathcal{F}_{ij} = \mathcal{F}_i \times \mathcal{F}''$
- $Y_{ij}(b, q) = m(i' - 1) + j'$ if $Y_i(b) = i'$ and $q = q_{j'}$,

– if $(q_1'', \ldots, q_k'') \in \delta(q'', \sigma)$ and $(q_1', \ldots, q_k') \in \delta_i(q', \sigma)$ then
  - if $q' \in W_i$ then $((q_1', q_1''), \ldots, (q_k', q_k'')) \in \delta_{ij}((q', q''), \sigma)$
  - if $q' = (b, s)$, with $b \in Box_i$ and $s \in O_{Y_i(b)}$, then $((q_1', q_1''), \ldots, (q_k', q_k'')) \in \delta_{ij}((b', s'), \sigma)$, where $b' = (b, q)$, for some $q \in Q$, and $s' = (s, q'')$)

Lemma 3. *Given an* HNBT $\mathcal{A}'$ *and an* NBT $\mathcal{A}''$*, the* HNBT $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$ *accepts the language* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}'')$ *and has size* $O(|Q|^2 \cdot |\mathcal{A}'| \cdot |\mathcal{A}''|)$.

Proof. Let $\mathcal{A}'$ be an *HNBT*, $\mathcal{A}''$ be an *NBT* and $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$, as described above, with components $\mathcal{A}_{ij}$. Since $\mathcal{A}^f$ is not isomorphic to $\mathcal{A}'^f \times \mathcal{A}''$, we prove the lemma by showing an isomorphism among the run of $\mathcal{A}^f$ and those of $\mathcal{A}'^f \times \mathcal{A}''$.

Given a run (T, r) of $\mathcal{A}^f$, we can define a run (T, r') of $\mathcal{A}'^f$ and a run (T, r'') of $\mathcal{A}''$ as follows. The run (T, r') is obtained by projecting for each state $\langle (q_1', q_1''), \ldots, (q_h', q_h'') \rangle$ in (T, r) the first components, thus getting the state $\langle q_1', \ldots, q_h' \rangle$ of $\mathcal{A}'^f$. The run (T, r'') of $\mathcal{A}''$ is obtained by projecting the second component of the node in each state (recall that only (q_h', q_h'') is a node, while all the other are boxes). Symmetrically, given the two runs, we can define (T, r) of $\mathcal{A}^f$. Now it immediately follows that (T, r) is accepting if and only if (T, r') and (T, r'') are both accepting runs. Thus the accepted languages is $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}'')$.

Consider now the size of each component $\mathcal{A}_{ij}$: from the definition of the cross product $\otimes$, each component $\mathcal{A}_{ij}$ is obtained by pairing the initial node in_i of $\mathcal{A}_i$ with q_j. The number of the states is $|W_i| \cdot |Q|$ and the number of superstates is $|Box_i| \cdot |Q|$. The size $|\delta_{ij}|$ of the transition function is bounded by $|\delta_i| \cdot |\delta| \cdot |Q|$, since when the transition is defined on a pair (b, s) then b can be paired with any state $q \in Q$. Thus the overall size of $\mathcal{A} = \mathcal{A}' \otimes \mathcal{A}''$ is $O(|Q|^2 \cdot |\mathcal{A}'| \cdot |\mathcal{A}''|)$. □

Let us now turn back to the desired reduction. Let φ be a fixed *CTL* formula and $\mathcal{M}$ be a hierarchical closed state machine with multiple exits. Let $\mathcal{A}_\mathcal{M}$ be an *HNBT* obtained from $\mathcal{M}$ by considering all its states as accepting and collecting all its relations in "tree-like" transitions (observe that $\mathcal{A}_\mathcal{M}$ suffices to be deterministic). More formally, for each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$, in $\mathcal{M}$ we add to $\mathcal{A}_\mathcal{M}$ the component $\mathcal{A}_i = (2^{AP}, W_i, \delta_i, Box_i, in_i, O_i, Y_i, \mathcal{F}_i)$, where $W_i = S_i \cup E_i$, $\mathcal{F}_i = W_i$, and $\delta_i(w, L_i(w)) = \{(w_i, \ldots, w_d)\}$ iff, for each $1 \leq j \leq d$, $(w, w_d) \in R_i$. By Lemma 1, we can construct an *NBT* $\mathcal{A}_\varphi$ accepting all models of φ. Note that $\mathcal{A}_\varphi$ has a fixed size, since φ is also fixed. By Lemma 3, we can construct in polynomial time an *HNBT* $\mathcal{A}_{\mathcal{M} \otimes \varphi}$ accepting the intersection[1] of $\mathcal{A}_K$ and $\mathcal{A}_\varphi$. Clearly, K satisfies φ iff $\mathcal{L}(\mathcal{A}_{K \otimes \varphi}) \neq \emptyset$. This, together with Lemma 2 leads to the desired result.

Theorem 1. *(*i*) The emptiness problem for a single-exit* HNBT *is* Ptime*-complete. (*ii*) The emptiness problem for a multiple-exit* HNBT *is* Pspace*-complete.*

[1] One can observe that $\mathcal{A}_\mathcal{M}$ may not be complete and that $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_\varphi$ may disagree on the number of node successors. It is not hard to see that, by duplicating successor states, we can adapt the previous constructions in order to obtain $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_\varphi$ as k-ary complete automata.

4 Deciding hierarchical Module Checking

In this section, we solve the program complexity of the *CTL* hierarchical module checking problem. In particular, we show that this problem is in PTIME for single-exit modules and in PSPACE in the multiple-exit case. By recalling that the program complexity for the classical *CTL* module checking is PTIME-hard and the program complexity for the *CTL* hierarchical model checking is PSPACE-hard, we get that our results are also tight.

Our solution to both problems is based on an automata–theoretic approach, by extending an idea of [KVW01]. In practice, we take into account that the input module is hierarchical. Therefore, we extend [KVW01]'s idea to each component of the module and use the automata product $\otimes$ introduced in the previous section, instead of a classical Cartesian product. In more details, let $\mathcal{M}$ be a hierarchial module and φ a fixed *CTL* formula. We decide the module checking problem for $\mathcal{M}$ against φ by building an *HNBT* $\mathcal{A}_{\mathcal{M}\otimes\neg\varphi}$ as $\mathcal{A}_{\mathcal{M}} \otimes \mathcal{A}_{\neg\varphi}$. Essentially, the first automaton, $\mathcal{A}_{\mathcal{M}}$, is an *HNBT* that accepts trees of $exec(\mathcal{M}^f)$, and the second automaton is an *NBT* $\mathcal{A}_{\neg\varphi}$ that accepts all trees that do not satisfy φ. Thus, $\mathcal{M} \models_r \varphi$ iff $\mathcal{L}(\mathcal{A}_{\mathcal{M}\otimes\neg\varphi})$ is empty. Now, recall from Lemma 3 that the cross product between $\mathcal{A}_{\mathcal{M}}$ and $\mathcal{A}_{\neg\varphi}$ corresponds to an *HNBT* (which can be constructed in polynomial time) whose flattening is equivalent to the Cartesian product of the *NBT*'s $\mathcal{A}^f_{\mathcal{M}}$ and $\mathcal{A}_{\neg\varphi}$. The component automata of the obtained *HNBT* will have a number of exit nodes that depends on the number of states of $\mathcal{A}_{\neg\varphi}$, which in turn depends, by Lemma 1, on the size of the formula φ. Since here we are interested on the program complexity of the hierarchical module checking problem, we assume the formula to be fixed. Therefore, $\mathcal{A}_{\mathcal{M}\otimes\neg\varphi}$ will have multiple exits iff $\mathcal{A}_{\mathcal{M}}$ does, and a constant number of exit nodes, otherwise.

Let us now discuss about the emptiness problem for *HNBT*'s with a constant number of exits. That is, we are interested in determining the complexity of the emptiness problem for the set $\{\mathcal{A} \mid \mathcal{A}$ is an *HNBT* with at most d-exit nodes$\}$, for a fixed natural number d. First observe that in the algorithm we have proposed in Lemma 2 for checking the emptiness of *HNBT*'s with single exits, each box either contributes to check the existence of an accepting run or not at all. On the opposite, in the multiple-exit case, we have to remember for each box which exit node ensures acceptance and which does not. Therefore, for each box some splitting may be required. For instance consider a component $\mathcal{A}_i$ with two exit nodes w_1 and w_2. It may be that a run exits in w_1 trough paths all visiting at least a final state and in w_2 trough a path that does not. Thus, we need to split $\mathcal{A}_i$ into four copies, depending whether both, only w_1, only w_2, or none can be considered in the extended set of final states. In general, if we start in Lemma 2 with an *HNBT* having at most d exit nodes, we need to generate 2^d copies of each component automaton, in the worst case. Since d is a fixed parameter, it turns out that the emptiness problem also for this automata remains in PTIME as reported in the following proposition.

Proposition 1. *The emptiness problem for a constant-exit* HNBT *is in* PTIME.

To conclude with our idea of solving the program complexity for *CTL* hierarchical module checking let us give some details on how to construct $\mathcal{A}_{\mathcal{M}}$ for a hierarchical module $\mathcal{M} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$, with each $\mathcal{M}_i = (AP, S_i, E_i, R_i, Box_i, O_i, in_i, L_i, Y_i)$. First, we recall that each component automaton of $\mathcal{A}_{\mathcal{M}}$ can only work on complete k-ary trees, while trees in $exec(M^f)$ may be not. To overcome this problem, we consider an equivalent representation of $exec(M^f)$ in which all nodes have degree $k = max\{bd(w) \mid w \in \bigcup_i S_i \cup E_i\}$, where $bd(w)$ denotes the branching degree of w (i.e. the number of its successors). Now, recall that each tree in $exec(M^f)$ is a 2^{AP}-labeled tree that is obtained from (T_{M^f}, V_{M^f}) by suitably pruning some of its subtrees. We can encode the tree (T_{M^f}, V_{M^f}) as a $2^{AP} \cup \{\bot\}$-labeled complete k-ary tree (where $\bot$ is a fresh atomic proposition not belonging to AP) in the following way: for each node $x \in T_M$ with d children $(x \cdot 1, \ldots, x \cdot d)$ (note that $1 \leq d \leq k$ as R_i is total), we add the children $(x \cdot (d+1), \ldots, x \cdot k)$ and label these new nodes with $\bot$; finally, for each node x labeled by $\bot$ we add recursively k-children labeled by $\bot$. Let $(\{1, \ldots, k\}^*, V')$ be the tree thus obtained. Then, we can encode a tree $(T, V) \in exec(M^f)$ as the $2^{AP} \cup \{\bot\}$-labeled complete k-ary tree obtained from $(\{1, \ldots, k\}^*, V')$ preserving all the labels of nodes of $(\{1, \ldots, k\}^*, V')$ that either are labeled by $\bot$ or belong to T, and replacing all the labels of nodes (together with the labels of the corresponding subtrees) pruned in (T, V) with the label $\bot$. In this way, all the trees in $exec(M^f)$ have the same structure (they all coincide with $\{1, \ldots, k\}^*$), and they differ only in their labeling. Thus, the proposition $\bot$ is used to denote both "disabled" nodes and "completion" nodes[2]. Moreover, since we consider environments that do not block the system, for each node associated with an enabled environment node, at least one successor is not labeled by $\bot$. Let us denote by $\widehat{exec}(\mathcal{M})$ the set of all $2^{AP} \cup \{\bot\}$-labeled k-ary trees obtained from $(\{1, \ldots, k\}^*, V')$ in the above described manner. We now show an *HNBT* $\mathcal{A}_{\mathcal{M}}$ accepting $\widehat{exec}(\mathcal{M})$. $\mathcal{A}_{\mathcal{M}}$ is the tuple $\langle \mathcal{A}_{(1,\top)}, \mathcal{A}_{(2,\top)}, \mathcal{A}_{(2,\bot)}, \mathcal{A}_{(2,\vdash)}, \ldots, \mathcal{A}_{(n,\top)}, \mathcal{A}_{(n,\bot)}, \mathcal{A}_{(n,\vdash)} \rangle$, where for $1 \leq i \leq n$ and $x \in \{\bot, \top, \vdash\}$, each $\mathcal{A}_{(i,x)} = \langle \Sigma, W_i', \delta_i, Box_i', (in_i, x), O_i', Y_i', W_i \rangle$ is defined as follows (recall $W_i = S_i \cup E_i$):

- $\Sigma = 2^{AP} \cup \{\bot\}$;
- $W_i' = W_i \times \{\bot, \top, \vdash\}$. The automaton $\mathcal{A}_{(i,x)}$ starts from (in_i, x). For example, the computation starts from $(in_i, \bot)$ whenever a box corresponding to $\mathcal{A}_i$ has been disabled. From states of the form $(w, \bot)$, $\mathcal{A}_{(i,x)}$ can read only the letter $\bot$, from states of the form $(w, \top)$, it can read only letters in 2^{AP}. Finally, when $\mathcal{A}_{(i,x)}$ is in state $(w, \vdash)$, then it can read both letters in 2^{AP} and the letter $\bot$. In this last case, it is left to the environment to decide whether the transition to a state of the form $(w, \vdash)$ is enabled. The three types of states are used to ensure that the environment enables all transitions from enabled system nodes, enables at least one transition from each enabled environment node, and disables transitions from disabled nodes.
- $O_i' = O_i \times \{\bot, \top, \vdash\}$. Clearly, we can have three types of exit nodes.

[2] As stated in [KVW01], the use of the atomic proposition $\bot$ must be taken into account while building $\mathcal{A}_{\neg\varphi}$. This can be easily handled by opportunely modifying the formula φ by exploiting an argument similar to that used in [KVW01].

- $Box'_i = Box_i \times \{\bot, \top, \vdash\}$. As for states, we can have three types of boxes, which are used to ensure that, regarding the initial node w of the component automaton corresponding to a box, the environment can enable all transitions from w whenever w is an enabled system node, enable at least one transition from w whenever w is an enabled environment node, and disables transitions from w whenever w is a disabled node.
- Y'_i is such that $Y'_i(b,x) = Y_i(b), x$. That is, from a box (b,x), $\mathcal{A}_i$ calls the automaton $\mathcal{A}_{(Y_i(b),x)}$. Then the computation continues from $(in_{Y_i(b)}, x)$ as described above.
- The transition function $\delta_i : (W'_i \cup (Box'_i \times \bigcup_{j>i} O'_j)) \times \Sigma \to 2^{(W'_i \cup Box'_i)^k}$ is defined as follows. Let $z = w \in W_i$ or $w \in O_j$ such that $z = (b,w) \in (Box_i \times \bigcup_{j>i} O_j)$ and $Y_i(b) = j$. Let $succ(z) = (z_1, \ldots, z_d)$, with $1 \leq d \leq k$, and for $m, m' \in \{\top, \vdash, \bot\}$, let z' be either (w,m), if $z = w$, or $z' = ((b,m'),(w,m))$, if $z = (b,w)$. Then, δ_i is as follows:
  - For $w \in W_i \cup O_j$ and $m \in \{\vdash, \bot\}$, we have

$$\delta_i(z', \bot) = \{(\underbrace{(z_1,\bot),\ldots,(z_d,\bot),(z,\bot),\ldots,(z,\bot)}_{k \text{ pairs}})\}$$

    That is, $\delta_i(z', \bot)$ contains exactly one k-tuple. In this case all the successors of the current node are disabled.
  - For $w \in S_i \cup (S_j \cap O_j)$ and $m \in \{\top, \vdash\}$ we have

$$\delta_i(z', L_i(w)) = \{(\underbrace{(z_1,\top),\ldots,(z_d,\top),(z,\bot),\ldots,(z,\bot)}_{k \text{ pairs}})\}$$

  - For $m \in \{\top, \vdash\}$ and either $w \in E_i$ and $g = w$ or $w \in (E_j \cap O_j)$ and $g = b$ we have $\delta_i(z', L_i(w)) =$

$$\begin{array}{c} \{((z_1,\top),(z_2,\vdash),\ldots,(z_d,\vdash),\ (g,\bot),\ldots,(g,\bot)), \\ ((z_1,\vdash),(z_2,\top),\ldots,(z_d,\vdash),\ (g,\bot),\ldots,(g,\bot)), \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ ((z_1,\vdash),(z_2,\vdash),\ldots,(z_d,\top),\ (g,\bot),\ldots,(g,\bot)\ )\}. \end{array}$$

    That is, $\delta_i(z', L_i(w))$ contains d k-tuples. When the automaton proceeds according to the i-th tuple, the environment can disable the transitions to all successors of the current state, except the transition associated with z_i, which must be enabled.

One can be convinced on the fact that $\mathcal{A}$ is polynomial in the size of $\mathcal{M}$, by noting that each $\mathcal{A}_{(i,x)}$ has $3 \cdot |\mathcal{M}_i|$ states, and $|\delta_i|$ is bounded by $O(k \cdot |R_i|)$. Thus, by summing up our idea trough the above polynomial construction of $\mathcal{A}_{\mathcal{M}}$, the exponential-constant time construction given by Lemma 1 for $\mathcal{A}_{\neg\varphi}$, Lemma 3, Proposition 1, and Lemma 2 the following result holds.

Theorem 2. *The program complexity for the* CTL *hierarchical module checking problem is Ptime-complete in the case of single-exit modules and Pspace-complete in in the case of multiple-exit modules.*

5 Conclusions

In this paper, we have introduced and solved the program complexity for the hierarchical module checking problem for *CTL*, both in the case of single-exit and multiple-exit modules. An immediate exponential solution can be obtained by flattening the hierarchical module and then apply the classical algorithm. By avoiding the flattening, we have shown algorithms having a better performance and, in particular, working not harder than those used in the closed hierarchical system case. As future directions, it would be worth to consider more involved scenarios both on the module checking side (e.g., pushdown module checking [BMP05], systems with incomplete information [AMV07]) and on the hierarchical side (e.g., recursive state machine [ABE$^+$05], with both nodes and boxes labeled with atomic propositions [LNPP08]).

References

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[AMV07] B. Aminof, A. Murano, and M.Y. Vardi. Pushdown module checking with imperfect information. In *CONCUR '07*, LNCS 4703, pages 461–476. Springer-Verlag, 2007.

[AY01] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[BMP05] Laura Bozzelli, Aniello Murano, and Adriano Peron. Pushdown module checking. In *LPAR'05*, LNCS 3835, pages 504–518. Springer-Verlag, 2005.

[CE81] E.M. Clarke and E.A. Emerson. Design and verification of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *J. of the ACM*, 47(2):312–360, 2000.

[KVW01] O. Kupferman, M.Y. Vardi, and P. Wolper. Module Checking. *Information and Computation*, 164(2):322–344, 2001.

[LNPP08] S. LaTorre, M. Napoli, M. Parente, and G. Parlato. Verification of scope-dependent hierarchical state machines. *Information and Computation.* 206(9,10): 1161-1177, 2008.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in Cesar. In *Proceedings of the Fifth International Symposium on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1981.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. *Mathematical Logic and Foundations of Set theory*, 1970.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *ICALP'98*, LNCS 1443, pages 628–641. Springer-Verlag, 1998.

[VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. of Computer and System Sciences*, 32(2):182–221, 1986.