Formal Analysis of Hierarchical State Machines
Dedicated to Zohar Manna on his 2^6th birthday

Rajeev Alur

Department of Computer and Information Science
University of Pennsylvania
3330 Walnut Street, Philadelphia, PA 19104, USA
Email: `alur@cis.upenn.edu`

1 Introduction

Finite state machines are widely used in the modeling of systems for various purposes. Descriptions using FSMs are useful to represent the flow of control (as opposed to data manipulation) and are amenable to formal analysis such as model checking [21, 23, 30, 42, 22]. In the simplest setting, a state machine consists of a labeled graph whose vertices correspond to system states and edges correspond to system transitions. In practice, to describe complex systems using state machines, several extensions are useful such as *communicating state machines* in which the system is described by a collection of state machines that operate concurrently and synchronize with one another, and *extended state machines* in which the edges are annotated with guards and updates that refer to a set of variables.

In *hierarchical state machines*, the states can be ordinary states or *superstates* which are state machines themselves. The notion of hierarchical state machines was popularized by the introduction of STATECHARTS [27], and exists in various object-oriented software development methodologies such as RSML [35], ROOM [40] and the Unified Modeling Language (UML [13]). Hierarchical state machines have two descriptive advantages over ordinary state machines. First, superstates offer a convenient structuring mechanism that allows us to specify systems by stepwise refinement, and to view it at different levels of granularity. Second, by allowing sharing of component machines, we need to specify components only once and then can reuse them in different contexts, leading to modularity and succinct system representations.

Features of concurrency and extension using variables are well studied from the perspective of formal analysis. Their impact on complexity of various decision problems is well understood (see, for instance, [25]), and modeling languages analyzed by typical model checkers such as SPIN [30] and SMV [38] support communicating extended state machines. On the other hand, hierarchy has not received much attention in the literature on formal analysis. This prompted us to initiate a systematic study of the analysis problems for hierarchical state machines. Analogous to the features of concurrency and extension with variables, hierarchy affords exponential succinctness (for example, an n-bit binary counter can be encoded using $O(n)$ description using any one of these three features).

However, unlike the other two features, for some analysis problems, hierarchy does not increase the complexity by an exponential factor. This paper surveys these results.

This paper is organized as follows. In Section 2, we define hierarchical state machines, and show that, even though the state machine underlying a hierarchical state machine can be exponentially larger, important analysis problems such as reachability and cycle detection can be solved in polynomial time. In Section 3, we study hierarchical state machines from an automata-theoretic perspective. We consider features of nondeterminism, hierarchy, and concurrency, in various combinations, and present results on relative succinctness, and the complexity of decision problems such as language emptiness, universality, inclusion, and equivalence. In Section 4, we address the problem of verifying hierarchical structures with respect to requirements specified in popular temporal logics such as LTL and CTL. Section 5 discusses modeling and symbolic analysis issues in presence of variables.

2 Hierarchical State Machines

2.1 Definition

A *hierarchical state machine* K is a tuple $\langle K_1, \dots K_n \rangle$ of modules, where each module K_i has the following components:

1. A finite set N_i of *nodes*.
2. A finite set B_i of *boxes*. The sets N_i and B_i are all pairwise disjoint.
3. A subset I_i of N_i, called *entry nodes*.
4. A subset O_i of N_i, called *exit nodes*.
5. An indexing function $Y_i : B_i \mapsto \{i+1 \dots n\}$ that maps each box of the i-th module to an index greater than i. If $Y_i(b) = j$, for a box b of module K_i, then b can be viewed as a reference to the definition of the module K_j. If b is a box of the module K_i with $j = Y_i(b)$, then pairs of the form (b, u) with $u \in I_j$ are the *calls* of K_i and pairs of the form (b, v) with $v \in O_j$ are the *returns* of K_i.
6. An edge relation E_i consisting of pairs (u, v) where the source u is either a node or a return of K_i, and the sink v is either a node or a call of K_i.

The module K_1 is called the *top-level* module of the HSM K. The edges connect nodes and boxes with one another. Note that the entry and exit nodes of a module serve as an interface for invoking the module: edges entering a box specify the entry nodes of the module associated with that box, while edges exiting a box specify the exit nodes among the possible returns. If every module has only a single entry node, we call the HSM *single entry*, and denote the unique entry node of each module K_i by in_i. Similarly, in a *single exit* HSM, each module has a single exit node, and the unique exit of the module K_i is denoted out_i.

With each hierarchical state machine, we can associate an ordinary state machine by recursively substituting each box by the module referenced by the box. Since different boxes can reference the same module, each node can appear in different contexts. In general, a state of the expanded state machine is a vector whose last component is a node, and the remaining components are boxes that specify the context. As a simple example of a hierarchical state machine, consider a specification of a digital clock. It contains three modules. The top-level module consists of a cycle though 24 boxes, one per hour of the day. Each such box is mapped to the second module consisting of a cycle containing 60 boxes counting minutes. Each of the boxes of this second module refers to the third module which is an ordinary state machine consisting of a cycle of 60 nodes counting seconds. In the expanded form of the hierarchical state machine, the state will have 3 components comprising of the box of the first module specifying the hour, the box of the second module specifying the minute, and the node of the third module specifying the second.

Formally, given a hierarchical state machine $K = \langle K_1, \dots K_n \rangle$, for each module K_i, we associate a state machine K_i^F, called the *expansion* of K_i, as follows:

1. The set $W_i \subseteq (\cup_{j \geq i} B_j)^* (\cup_{j \geq i} N_j)$ of *states* of K_i^F is defined by:
   – every node of K_i belongs to W_i.
   – if b is a box of K_i with $Y_i(b) = j$, and v is a state of K_j^F, then $b \cdot v$ belongs to W_i.
2. The initial states of K_i^F are I_i.
3. The set R_i of transitions of K_i^F is defined by:
   – for $(u, v) \in E_i$, $(u', v') \in R_i$, where if the source u is a node then $u' = u$ and if it is a return (b, w) then $u' = b \cdot w$, and if the sink v is a node then $v' = v$ and if it is a call (b, w) then $v' = b \cdot w$.
   – if b is a box of K_i with $Y_i(b) = j$, and (u, v) is a transition of K_j^F, then $(b \cdot u, b \cdot v)$ belongs to R_i.

The expansion K_1^F of the top-level module is also denoted K^F.

The *size* of the module K_i, denoted $|K_i|$, is the sum of $|N_i|$, $|B_i|$, and $|E_i|$. The size of K is the sum of the sizes of K_i. The *nesting depth* of K, denoted $nd(K)$, is the length j of the longest chain $i_1, i_2, \dots i_j$ of indices such that a box of K_{i_l} is mapped to i_{l+1}, for $1 \leq l < j$. Observe that each state of the expansion is a string of length at most the nesting depth, and the size of the expansion K^F can be exponential in the nesting depth, and is $O(|K|^{nd(K)})$. This bound is tight: for each n, we can construct a single-entry single-exit HSM of size $O(n)$ such that the expansion has 2^n states.

In many analysis problems, we will describe the algorithms for single entry HSMs. An HSM with multiple entry nodes can be translated to a single entry HSM by replacing a module with k entry nodes with k single-entry modules. Note that each box within a module also may need to be repeated k times, along with the edges out of the corresponding returns, and consequently, the blow-up in the total size due to the translation would be a factor of k^2 in the worst case.

We have required that the modules do not call each other in a recursive manner, that is, the call dependency among modules is acyclic. Removing this restriction leads to the definition of recursive state machines studied recently in

[1, 11]. Recursive state machines are analogous to the widely studied classical model of pushdown systems. Consequently, hierarchical state machines can be viewed as restricted forms of pushdown systems with stack size bounded a priori. Our definition allows a more direct, visual, state-based model of hierarchical control flow.

2.2 Reachability

The core computational problem in many verification problems can be formulated as a reachability question. In the current context, the input to the reachability problem consists of a hierarchical state machine K, and a subset $T \subseteq \cup_i N_i$ of nodes, called the *target region*. Given (K, T), the reachability problem is to determine whether or not some state whose last component is in the target region T is reachable from some entry node of the top-level module K_1. The target region is usually specified implicitly, but we assume, that given a node u, the test $u \in T$ can be performed in $O(1)$ time.

Even though the expansion of a hierarchical state machine K can have exponential number of states, the reachability problem can be solved in time polynomial in the size of K. Let us assume that the HSM K is a single entry HSM. The algorithm simply needs to make sure that every module is searched at most once even if it appears in multiple contexts. We will briefly review the algorithm of [7] that is a modification of the classical depth-first search algorithm. The algorithm performs a depth-first search starting at the unique entry node of K_1 using the global data structure *visited* to store the nodes and boxes. While processing a node u, the algorithm first checks if $u \in T$ holds, and if so, the algorithm terminates. Otherwise, it examines the edges out of u, and continues to process nodes and boxes that are not yet visited. While processing a box b with $Y(b) = i$, the algorithm checks if the entry node in_i of the i-th module was visited before. If it was not visited before, it searches the structure K_i by initiating a depth-first search starting at the unique entry node in_i. At the end of this search, the set of exit nodes of K_i that are reachable from in_i will be stored in the data structure *visited*. The search can, then, continue by exploring edges from the returns (b, v) such that v is known to be reachable in K_i. If the algorithm subsequently visits some other box c that is mapped to K_i, it does not search K_i again, but simply uses the information stored in *visited* to continue the search from the appropriate returns of the box c. This is an on-the-fly algorithm that needs to compute the edges out of a node or a return only on demand, and terminates as soon as a target node is encountered.

When the modules have multiple entry nodes, we can use the translation into single-entry HSMs that costs a factor of k^2, where k is the maximum number of entry nodes per module. In terms of complexity, for ordinary state machines, deciding reachability between two states is in NLOGSPACE, while for hierarchical state machines, the reachability problem becomes PTIME-hard even if we require a single exit for every module [7]. The hardness proof is by reducing *alternating reachability* (that is, deciding the existence of winning strategies in two-player game graphs) to reachability in hierarchical state machines.

Proposition 1. *The reachability problem (K,T) for hierarchical state machines is* PTIME*-complete, and can be solved in time $O(|K| \cdot k^2)$, where each module of K has at most k entry nodes.*

Note that if a module has multiple entry nodes, it is searched multiple times: for every entry node u, the algorithm needs to compute which nodes are reachable from u. To process a box mapped to a module K_j, the relevant information is which exits are reachable from which entries within K_j. If a module has multiple entry nodes, but a single exit node, then it is possible to compute this information by searching the module just once, starting the search at the unique exit node, and traversing the edges in the reverse direction. This idea can be generalized: the modules with fewer entries than exits are searched forwards starting at the entry nodes, and the others are searched backwards starting at the exit nodes. For each module K_i, let k_i be the minimum of $|I_i|$ and $|O_i|$. Then, reachability problem can be solved in time $O(|K| \cdot k^2)$, where k is the maximum of these k_i values [1].

2.3 Cycle Detection

The basic problem encountered during verification of liveness requirements is to check whether a specified state can be reached repeatedly [42, 29]. As in the reachability problem, the input to the *cycle detection problem* consists of a hierarchical state machine K, and a target region T of nodes. Given (K,T), the cycle detection problem is to determine whether there exists a state u whose last component is in the target region T such that (1) u is reachable from an entry node of the top-level module, and (2) u is reachable from itself.

Let us assume that every module has a unique entry node. Let us call a path/cycle to be a T-path/T-cycle if it contains a state whose last component is in T. For each module K_i, we want compute the following information:

Q1 Is there a T-cycle reachable from the entry node in_i within K_i^F?
Q2 Partition the exit nodes O_i into three sets: the exits that are reachable from in_i along some T-path within K_i^F, the exits that are reachable from in_i but not along any T-path within K_i^F, and exits that are not reachable from in_i within K_i^F.

Basically, this information is all that is needed to process boxes that reference K_i. For the module K_n that does not contain any boxes, Q1 and Q2 can be computed easily using standard search algorithms. Now consider a module K_i such that Q1 and Q2 have been solved for all modules K_j with $j > i$. Consider a box b such that $Y_i(b) = j$. Then, we can replace b by introducing a node b_e corresponding to the invocation of K_j and a node b_x for every exit x of K_j. Edges entering b are connected to b_e, and edges from the returns (b,x) connect from the respective nodes b_x. If there is a T-cycle reachable from in_j within K_j^F, then we add a self-loop on b_e, and add b_e to the target set T. For an exit x of K_j, if x is reachable from in_j within K_j^F then we add an edge from b_e to b_x, and in addition, if x is reachable from in_j along a T-path, we add b_x to the target set.

After transforming each box in this manner, we can process K_i as an ordinary state machine, and solve Q1 and Q2 for K_i. Each module is processed precisely once, and this can be done in linear time.

The strategy described above is a bottom-up approach that may explore modules unnecessarily. An on-the-fly algorithm for cycle detection involving two interleaved depth-first searches is presented in [7] by adapting the nested depth first search of [24]. The complexity bound for cycle detection is the same as that for reachability.

Proposition 2. *The cycle detection problem* (K, T) *for hierarchical state machines is* PTIME*-complete, and can be solved in time* $O(|K| \cdot k^2)$*, where each module of* K *has at most* k *entry nodes.*

3 Hierarchical Automata

In this section, we consider hierarchical structures from an automata theoretic perspective, as generators of regular languages.

3.1 Hierarchical Generators

A *(nondeterministic) finite automaton* (NFA) consists of a finite set Q of states, a finite alphabet Σ, a set $I \subseteq Q$ of initial states, a set $T \subseteq Q$ of final states, and a set $E \subseteq Q \times \Sigma \times Q$ of transitions. Given a word $\rho = \sigma_0\sigma_1 \cdots \sigma_n$ over the alphabet Σ, an *accepting run* of M over ρ is a sequence $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} q_{n+1}$ such that q_0 is an initial state in I, q_{n+1} belongs to the final states T, and for $0 \leq i \leq n$, (q_i, σ_i, q_{i+1}) is a transition of M. The set of words $\rho \in \Sigma^*$ over which M has an accepting run is called *the language of* M, denoted $L(M)$. The NFA M is *deterministic* (DFA) if (1) there is only one initial state, and (2) for every state q and every symbol σ, there is at most one σ-labeled transition with source q.

A *(nondeterministic) hierarchical automaton* (NHA) M is a hierarchical state machine $K = \langle K_1, \ldots K_n \rangle$ whose edges are annotated with the symbols in Σ (that is, the edge relation E_i of every module K_i consists of triples (u, σ, v) where $\sigma \in \Sigma$, the source u is either a node or a return of K_i, and the sink v is either a node or a call of K_i) together with a set $T \subseteq \cup_i N_i$ of final nodes. The expansion M^F of an NHA is obtained by expanding the hierarchical state machine K, and declaring each state with its last component in T to be a final state. Every edge in the expansion uniquely corresponds to an edge in one of the edge relations E_i, and inherits the symbol labeling that edge. The language of an NHA M is the language of its expansion. The NHA M is said to be *deterministic* (DHA) if the expansion M^F is deterministic. Note that this is the same as requiring that (1) the top-level module has a single entry node, (2) in every module K_i, for every node or return u, for every symbol σ, there is at most one sink v with $(u, \sigma, v) \in E_i$, and (3) if a return (b, v) of a module K_i has a σ-labeled outgoing edge for a box b mapped to K_j, then the exit node v does not have any outgoing

σ-labeled edges within K_j. The last condition ensures determinism concerning returning from module calls.

Since the expansion of an NHA is a finite automaton, NHAs can generate only regular languages, but they can be much more succinct than ordinary automata. To compare succinctness, we will look at families of languages $\{L_n | n = 1, 2, \ldots, \}$ and consider the number of states needed by an automaton that accepts L_n in the formalisms to be compared. By classical results on succinctness of nondeterministic automata, NFAs are exponentially more succinct than DFAs (cf. [31]).

Consider the family of languages L_n given by $\{w \# w^R \mid w \in \{0,1\}^n\}$. Consider the sequence $M_0, M_1, \ldots M_n$ of single-entry single-exit DHAs over the alphabet $\{0, 1, \#\}$. M_0 is an ordinary automaton with $L(M_0) = \{\#\}$, and for $i > 0$, M_i is a hierarchical automaton with two boxes, each mapped to M_{i-1}, so that $L(M_i)$ is the union of $0 \cdot L(M_{i-1}) \cdot 0$ and $1 \cdot L(M_{i-1}) \cdot 1$. Thus, there is a DHA of size $O(n)$ that recognizes L_n [1]. On the other hand we can show that any NFA to recognize L_n must use at least 2^n states.

Proposition 3. *Deterministic hierarchical automata are exponentially more succinct than nondeterministic finite automata.*

The gap between NHAs and NFAs is singly exponential since there is an exponential expansion. It is interesting to note that nondeterminism and hierarchy are incomparable extensions. Consider the language L_n consisting of strings w such that for some i, $w_i = w_{i+n}$. It is easy to see that there is an $O(n)$ NFA to accept L_n. A deterministic automaton, on the other hand, needs to remember the first n symbols, and thus, has to have at least 2^n states. Even deterministic hierarchical automaton must have 2^n states.

Proposition 4. *Nondeterministic finite automata are exponentially more succinct than deterministic hierarchical automata.*

The next result establishes that, while nondeterminism introduces exponential succinctness for ordinary automata, it introduces doubly-exponential succinctness in presence of hierarchy. Observe that an NHA can be converted into an equivalent DHA with a doubly-exponential blow-up via expansion followed by determinization. Consider the language L_n consisting of strings w such that for some i, $w_i = w_{i+2^n}$. There is a $O(n)$ NHA for L_n where the automaton simply guesses the position i, remembers the symbol w_i, counts 2^n symbols succinctly using hierarchy, and checks that the next symbol is in fact equal to w_i. On the other hand, we can show that any DHA for this language must have states doubly exponential in n.

Proposition 5. *Nondeterministic hierarchical automata are doubly exponentially more succinct than deterministic hierarchical automata.*

[1] It is worth noting that the language $\{ w \# w \mid w \in \{0,1\}^n \}$ cannot be defined succinctly by hierarchical automata. In fact, we can prove that every NHA accepting this language must be of exponential size.

Finally, consider the succinctness due to intersection. Consider the language L_n consisting of strings of the form $w \# w^R \# w$ such that $w \in \{0,1\}^n$. This is intersection of the languages of two DHAs of $O(n)$ size. However, it can be shown that any hierarchical acceptor (even if it is non-deterministic) requires exponential-size.

Proposition 6. *Products of deterministic hierarchical automata are exponentially more succinct than nondeterministic hierarchical automata.*

3.2 Decision Problems

The emptiness question for a hierarchical automaton $M = (K, T)$ is to check whether the language $L(M)$ is empty. This is same as the reachability question for hierarchical state machines: whether the target set T is reachable from a top-level entry node in the expansion, and is Ptime-complete as discussed earlier.

We now consider the problem of testing emptiness of the intersection of the languages of two automata. For NFAs, the product of two automata has a quadratic blow-up. However, to analyze product of two hierarchical automata, there is no way around expansion, thus, requiring the exponential price. Recall that for pushdown automata, emptiness of a single automaton can be solved in polynomial-time, but emptiness of the intersection of two pushdown automata is undecidable [31]. In the case of hierarchical automata, emptiness of a single automaton is Ptime, but emptiness of the intersection of two automata is Pspace-complete. The upper bound follows from the fact that emptiness of the intersection can be tested by searching the product of the two expansions, and the lower bound is by reduction from the halting problem for linear bounded automata [4]. Let us briefly describe the main idea in the encoding. Let A be a linear-bounded automaton. A configuration of A can be described by a string w of length n. A halting run of A can be described by a sequence $w_0, w_1, \ldots w_k$ such that each w_i describes a configuration of A, w_0 is the initial configuration, w_k is the halting configuration, and each w_{i+1} is obtained from w_i by a legal move of the machine. Consider the language L_A consisting of strings $w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_k$ corresponding to halting runs $w_0, w_1, \ldots w_k$ of A. We can construct two hierarchical automata M_1 and M_2 such that $L(M_1) \cap L(M_2)$ equals L_A. The automaton M_1 ensures that for each $i \geq 0$, the configuration $2i+1$ is a successor of the configuration $2i$ in the reversed order. Similarly, the automaton M_2 ensures that for each $i \geq 0$, the configuration $2i+2$ is a successor of the configuration $2i+1$ in the reversed order. In addition, one of them has to ensure that w_0 is the initial configuration and w_k is the halting configuration (this can be easily done by even an ordinary automaton).

Proposition 7. *The problem of testing emptiness of the intersection of languages of two hierarchical automata is* Pspace*-complete.*

The universality problem for hierarchical automata is to decide whether the complement of the language of an NHA is empty. The language equivalence problem is to decide, given two NHAs M_1 and M_2, whether $L(M_1)$ equals $L(M_2)$. Recall that for NFAs both universality and language equivalence problems are PSPACE-complete. These problems are EXPSPACE-complete for NHAs.

The lower bound for the universality problem is established by a reduction from the halting problem of EXPSPACE Turing machines. We briefly sketch the basic idea. Let A be a deterministic exponential space Turing Machine. We can construct an NHA that accepts strings in the complement of the strings encoding legal halting computations of A on a given input string. The main obstacle is to detect existence of a configuration i, and a cell position j in the i-th configuration which does not follow from the previous configuration. For this, the hierarchical automaton non-deterministically guesses the pair (i, j), "remembers" the contents of the cells $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$ from the $(i-1)$-th configuration, uses hierarchy to count succinctly to skip as many symbols as the exponential length of a single configuration, and then checks that cell (i, j) does not follow from the transition rules of A.

For the upper bound, given M_1 and M_2, it suffices to check the emptiness of the product of M_1 with the complement of M_2, and the emptiness of the product of M_2 with the complement of M_1. This can be done by expansion and complementation. For instance, the product of M_1^F and the complement of M_2^F has doubly-exponentially many states, and can be searched in EXPSPACE.

Proposition 8. *The universality and language equivalence problems for nondeterministic hierarchical automata are* EXPSPACE-*complete.*

For ordinary automata, problems such as universality, inclusion, and equivalence, are much easier if we consider deterministic automata. The results for hierarchical automata show somewhat subtle distinctions between these problems. Observe that if M is a DHA, then complementing it, that is, constructing a DHA accepting the complement of $L(M)$, is easy. Consequently, checking universality has the same complexity as checking emptiness, namely, PTIME. Given DHAs M_1 and M_2, testing language inclusion $L(M_1) \subseteq L(M_2)$ reduces to checking emptiness of the product of M_1 and the complement of M_2. This can be done in PSPACE. The problem is PSPACE-hard as in case of testing the emptiness of the product of two hierarchical automata.

Proposition 9. *For deterministic hierarchical automata, the universality problem is* PTIME-*complete, and the language inclusion problem is* PSPACE-*complete.*

For DHAs, language equivalence can be solved in PSPACE, but its complexity is sandwiched somewhere between that of the PTIME universality problem and the PSPACE language inclusion problem. It remains open to determine its exact complexity.

3.3 Communicating Hierarchical Automata

Now we proceed to consider models with hierarchy as well as concurrency. For concurrency, an automaton is composed of a set of component automata which synchronize on transitions labeled with common alphabet symbols. For hierarchy, the states of an automaton can be other automata. The following formal definition allows arbitrary nesting of the concurrency and hierarchy constructs.

A *communicating hierarchical automaton* (CHA) is a tuple $\langle K_1, \ldots K_n \rangle$ such that each module K_i is one of the following two forms:

1. It is a tuple of indices $(j_1, j_2, \ldots j_l)$ with $i < j_p \leq n$. Such a module is called a product module, and represents the product of the CHAs K_{j_p}.
2. It is a hierarchical module $(N_i, B_i, I_i, O_i, Y_i, E_i)$ with nodes N_i, boxes B_i, an indexing function Y_i mapping the boxes to modules indexed higher than i, entry nodes I_i, exit nodes O_i, and Σ-labeled edges E_i connecting nodes, calls, and returns.

Note that the module K_n must be an ordinary NFA. The hierarchical modules are as before, except that the boxes can be mapped to modules that are defined using both product and hierarchy constructs. The expansion K_i^F of each module of a CHA is defined in the natural way. For a product module K_i, the expansion K_i^F is simply the product of the expansions $K_{j_p}^F$ of its components. For a hierarchical module K_i, the states and the transitions of the expansion are defined as in case of HSMs in Section 2.1. The language of a CHA M is the language of its expansion as usual.

A CHA can be represented by a directed acyclic graph. A terminal vertex corresponds to the base case, and has an associated NFA. An internal vertex may correspond to a product module or a hierarchical module. The children of a vertex for a product module are the components of the product. We can assume that the immediate children of a product module are not themselves product modules. A vertex for a hierarchical module has the top-level module associated with it, and the children are the modules referenced by its boxes. Two important parameters of this DAG representation are its *width* and *depth*: width of a CHA M is the maximum number of components in the product module, and depth is the length of the longest path in the DAG.

In CHAs, the concurrency and hierarchy operators are arbitrarily nested, and the product components can synchronize with each other at different levels of hierarchy. These features contribute significantly to the complexity of the reachability problem. First, we establish that for a CHA M, the number of states of M^F is $O(k^{d^m})$, where each module has at most k nodes/boxes, M has width $d > 1$ and depth m. The proof is by induction on the depth m. If M is an ordinary automaton, M^F equals M, and has size $O(k)$. Now, consider a CHA M with depth $m > 0$. The root is either a hierarchical module or a product module. If it is a hierarchical module, then its top-level state machine has at most k boxes, each of which is mapped to a CHA of depth $m - 1$ or less. Consequently, for each box b, by induction, the module it is mapped to has size $O(k^{d^{m-1}})$, and hence, M^F is of size $k \cdot O(k^{d^{m-1}})$. If the root is a product module, then it has at most d components, each of which has depth $m - 1$ or less, and by induction, has size $O(k^{d^{m-1}})$ for the expansion. Since M^F is the product of the expansions of the components, its size is $O(k^{d^{m-1}})^d$, which is $O(k^{d^m})$.

This implies that reachability problem for a CHM can be solved in time $O(k^{d^m})$, that is, doubly-exponential in the depth in the worst case. Since the expansion M^F need not be constructed explicitly, and can be searched in space that is required to represent a single state, the upper bound on reachability is EXPSPACE.

For the lower bound, consider the sequence of CHAs $N_0, N_1, \ldots N_n$ as follows. The automaton N_0 accepts strings of the form $\sigma \Sigma^{2^n} \sigma \Sigma^*$. The automaton N_0 remembers the first symbol, then skips over the next 2^n symbols, and checks if the following symbol is same as the remembered one. Having defined the automaton N_i, the automaton N_{i+1} is a product of two automata. The first one is N_i. The second one skips over the first 2^i symbols, and then starts a copy of N_i. The second component can be defined as a hierarchical two-state automaton, whose first box is mapped to an automaton that counts 2^i, and the second box is mapped to N_i. Observe that the size of description of N_n is only $O(n)$. A string of the form $w\,w'$, where w and w' are strings of length 2^n, is accepted by N_n precisely when $w = w'$. Intuitively, N_n has exponentially many copies in parallel each checking a different position of w. This shows that CHA can ensure "copying" of exponential-sized strings. This can be used to construct a CHA that accepts only legal encodings of computations of EXPSPACE Turing machines, and lower bound follows.

Proposition 10. *Reachability problem for communicating hierarchical automata is* EXPSPACE*-complete, and can be solved in time $O(k^{d^m})$ for a CHA of depth m, width $d > 1$, and with each hierarchical module having at most k nodes/boxes.*

Since reachability problem for CHAs is EXPSPACE-hard, [4] identifies a subclass of *well-structured* automata with PSPACE-complete reachability problem (note that reachability problem for communicating automata is PSPACE-hard even in absence of hierarchy). The restriction ensures that if two (or more) hierarchical automata are composed together by the product module, then they can synchronize only at the top level. This is enforced by syntactic restrictions on which alphabet symbols are allowed to appear as labels on edges.

Recall that for NFAs, problems such as universality and language inclusion are exponentially harder than reachability. The same holds for CHAs. Showing 2EXPSPACE-hardness requires some tricky encoding. Consider a 2EXPSPACE Turing machine A. A configuration of A can be represented by a string w of length 2^{2^n} over an appropriate alphabet Σ (assume Σ does not contain the symbols 0 or 1). We introduce an additional binary counter of exponential size to encode the configurations of double-exponential length. For an integer j, let $bin(j)$ be the binary string over $\{0,1\}$ encoding j. Then, for each configuration $w = w_0 \ldots w_{2^{2^n}-1}$, define

$$enc(w) \;=\; bin(0)w_0 \,\%\, bin(1)w_1 \,\%\, \ldots \,\%\, bin(2^{2^n}-1)w_{2^{2^n}-1} \,\%$$

A halting run of A can be described by a sequence $w_0, w_1, \ldots w_k$ of configurations w_i of the usual form. Consider the language L_A

$$\{\; enc(w_0) \,\#\, enc(w_1) \,\#\, \cdots \,\#\, enc(w_k) \;\mid\; w_0, w_1, \ldots w_k \text{ is a halting run of A } \}.$$

We construct a CHA M that accepts the complement of L_A, which would imply the lower bound for checking universality. The automaton M needs to check that the string does not encode a halting run. Errors concerning initial and halting configurations can be detected easily. The automaton M needs to detect errors in encoding using the binary counter. In L_A, each cell σ is encoded by a block of length 2^n+2 of the form $\{0,1\}^{2^n}\sigma\%$. Counting with exponential succinctness allows to detect a violation of this. Furthermore, an error occurs if the binary counter following # is not 0^* or preceding # is not 1^*. For each cell, the counter of the successive cells must be incremented. A violation of this can be detected by an NHA of size $O(n)$ due to the properties of a binary counter.

The tricky part is to detect an error if adjacent configurations are not related by a legal move of A. For illustration of the basic idea, let us consider it to be an error if there exists a cell position j such that some two adjacent configurations differ in the contents of the j-th cell. even though we do not know how to count with double-exponential succinctness, we can use the fact that in our encoding, matching cells of the two configurations must have identical counters. We will construct a machine to accept the language

$$L' \;=\; \{\; b\,\sigma\,\%\,[\{0,1\}^{2^n}\Sigma\%]^*\#[\{0,1\}^{2^n}\Sigma\%]^*\; b\,\sigma'\% \;\;|\; \sigma \neq \sigma' \text{ and } b \in \{0,1\}^{2^n} \;\}.$$

Note that $(\Sigma \cup \{0,1,\%,\#\})^* \cdot L' \cdot (\Sigma \cup \{0,1,\%,\#\})^*$ is the language of all the strings containing the desired error. Accepting L' requires checking that the counters in the first and the last blocks are identical. Since b is of exponential length, we will use an $O(n)$ CHA that spawns 2^n parallel modules, each checking one bit of b. This is done along the same lines as the proof of Expspace-hardness of reachability of CHAs. In particular, let N_1 be the automaton that remembers the first bit, skips over a string whose length is a multiple of 2^n+2 (block length) and which contains precisely one # symbol. Then, N_1 checks that the following symbol matches the remembered bit, and halts after reading the following % symbol. Notice that if N_1 is started at the p-th bit of the counter b, it will ensure matching of the p-th bit of the counter in the last block before it halts. The machine N_{i+1} equals $N_i \| \{0,1\}^{2^i} \cdot N_i$. The claim is that the automaton N_n, in conjunction with a machine that enforces the $\sigma \neq \sigma'$ requirement, accepts the language L'. The key to the construction is that even though the exponentially many parallel copies of N_1 guess independently the number of blocks to skip over, they halt on the same % symbol following the guess, which enforces synchronization.

Proposition 11. *For communicating hierarchical automata, the universality, language inclusion, and language equivalence problems are 2Expspace-complete.*

The results concerning hierarchical automata are summarized in Figure 1.

The ability of CHAs to copy exponential-sized strings also yields significant succinctness. Consider the languages L_n consisting of $w_0\#w_1\#\cdots\#w_k$ such that each w_i is of size 2^n, and $w_i = w_j$ for some i,j. A nondeterministic CHA of size $O(n)$ to accept L_n can be constructed. An NFA can guess the position i, but needs to remember the exponentially long string w_i, and hence, is at least

	Emptiness	Intersection	Universality	Inclusion	Equivalence
NFA	Nlogspace	Nlogspace	Pspace	Pspace	Pspace
DFA	Nlogspace	Nlogspace	Nlogspace	Nlogspace	Nlogspace
NHA	Ptime	Pspace	Expspace	Expspace	Expspace
DHA	Ptime	Pspace	P	Pspace	$\in$ Pspace
CHA	Expspace	Expspace	2Expspace	2Expspace	2Expspace

Fig. 1. Summary of complexity results for decision problems

$$\begin{tikzcd}
\text{CHA} \arrow[rr, "\text{2exp}"] \arrow[dr, "\text{3exp}"'] & & \text{NHA} \arrow[dl, "\text{2exp}"'] \arrow[dr, "\text{1exp}"] & \\
& \text{DHA} \arrow[rr, bend left, "\text{1exp}"] \arrow[dr, "\text{1exp}"'] & & \text{NFA} \arrow[ll, bend left, "\text{1exp}"] \arrow[dl, "\text{1exp}"] \\
& & \text{DFA} &
\end{tikzcd}$$

Fig. 2. Summary of succinctness results

of double-exponential size. A DFA needs to remember all the words w_i read so far, and is at least of triple-exponential size.

Proposition 12. *Communicating hierarchical automata are doubly exponentially more succinct than NHAs (and NFAs), and triply-exponentially more succinct than DHAs (and DFAs).*

The various succinctness claims are summarized in Figure 2.

4 Temporal Logic Model Checking

In this section, we consider the problem of verifying that a model given as a hierarchical state machine satisfies a temporal requirement expressed using automata over infinite words or using temporal logics. Kripke structures are state-transition graphs whose states are labeled with atomic propositions, and are used as models for temporal logics. Analogously, to interpret temporal logic formulas over hierarchical state machines, we label the nodes with atomic propositions. A *hierarchical Kripke structure* K over a set P of atomic propositions is a hierarchical state machine $\langle K_1, \dots K_n \rangle$ where each module K_i has an additional component: a labeling function $X_i : N_i \mapsto 2^P$ that labels each node with a subset of P. The expansions K_i^F are defined as before. Recall that the last component of every state in the expansion is a node, and the propositional labeling of the last component determines the propositional labeling of the entire state. This implies that the state assertions cannot refer to the context, and this choice is important for the algorithms and the complexity bounds. A more permissive definition that allows the specifications to refer to the context is considered in [34].

4.1 Automata Emptiness

Given a Kripke structure M whose states are labeled with atomic propositions P, we can associate an ω-language $\mathcal{L}(M)$ of infinite words over the alphabet 2^P as follows. An execution of M is an infinite path in the state-transition graph of M starting at an initial state. A *trace* corresponding to an execution is obtained by replacing each state in the execution by the corresponding label. The ω-language $\mathcal{L}(M)$ is the set of all traces of M. The ω-language $\mathcal{L}(K)$ of a hierarchical Kripke structure K is the ω-language of its expansion $\mathcal{L}(K^F)$.

A *Büchi automaton* A over P consists of a Kripke structure M over P and a set T of accepting states. An execution $w_0 w_1 \dots$ of M is called *accepting* if $w_i \in T$ for infinitely many indices i. The ω-language $\mathcal{L}(A)$ is the set of all traces corresponding to the accepting executions.

The input to the *automata emptiness* problem consists of a hierarchical structure K over P and a Büchi automaton A over P. Given (K, A), the automata emptiness problem is to determine whether the intersection $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ is empty. This is the automata-theoretic approach to verification: if the automaton A accepts undesirable or bad behaviors, checking emptiness of $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ corresponds to ensuring that the model has no bad behaviors (see [42], [33], [41], and [29] for more details on ω-automata and their role in formal verification).

We solve the automata emptiness problem (K, A) by reduction to a cycle detection problem for the hierarchical structure obtained by constructing the product of K with A. The basic property of the product construction is that some trace of K^F is accepted by A iff there is a reachable cycle in $K \otimes A$ containing a node of the form (u, w) for an accepting state w of A. The product has size $O(|K| \cdot |A|)$, and the number of entry nodes per module gets multiplied by the number of states of A. This leads to the following bound:

Proposition 13. *The automata emptiness question (K, A) can be solved by reduction to the cycle detection problem in time $O(a^2 \cdot k^2 \cdot |A| \cdot |K|)$, where a is the number of states in A, and each module of K has at most k entry nodes.*

4.2 Linear Temporal Logic

Requirements of trace-sets can be specified using the temporal logic LTL [39, 37]. A formula φ of LTL over propositions P is interpreted over an infinite sequence over 2^P. A hierarchical Kripke structure K satisfies a formula φ iff every trace in $\mathcal{L}(K)$ satisfies the formula φ. The LTL model-checking problem is to determine whether or not the input hierarchical structure K satisfies the input LTL formula φ.

To solve the model-checking problem, we construct a Büchi automaton $A_{\neg\varphi}$ such that the language $\mathcal{L}(A_{\neg\varphi})$ consists of precisely those traces that do not satisfy φ. This can be done using one of the known translations from LTL to Büchi automata [36, 42]. The number of states of $A_{\neg\varphi}$ is $O(2^{|\varphi|})$. Then, the hierarchical structure K satisfies φ iff $\mathcal{L}(K) \cap \mathcal{L}(A_{\neg\varphi})$ is empty. Thus, the model-checking problem reduces to the automata-emptiness problem. The complexity of solving the automata-emptiness problem, together with the cost of translating an LTL formula to a Büchi automaton, yields $O(k^2 \cdot |K| \cdot 8^{|\varphi|})$ bound on the model checking.

An alternative approach to solve the LTL model-checking problem (K, φ) is to search for an accepting cycle in the product of the expansion K^F with the automaton $A_{\neg\varphi}$. This product has $|K|^{nd(K)} \cdot 2^{|\varphi|}$ states, and each state of this product can be represented in space $O(|K| \cdot |\varphi|)$. Transitions of the product can be computed efficiently using standard techniques, and consequently the search can be performed in space $O(|K| \cdot |\varphi|)$. This gives a PSPACE upper bound on the LTL model-checking problem. It is known that the LTL model-checking problem for ordinary Kripke structures is PSPACE-hard. These results are summarized in the following:

Proposition 14. *The LTL model-checking problem (K, φ) can be solved in time $O(k^2 \cdot |K| \cdot 8^{|\varphi|})$, where each module of K has at most k entry nodes, and is* PSPACE-*complete.*

4.3 Computation Tree Logic

Now we turn our attention to verifying requirements specified in the branching-time temporal logic CTL [21]. Branching-time logics provide quantification over computations of the system allowing specification of requirements such as "along some computation, eventually p" and "along all computations, eventually p." We refer the reader to [21] to syntax and semantics of CTL. The *CTL model-checking problem* is to decide, given a hierarchical Kripke structure K and a CTL formula φ, whether all the top-level entry nodes satisfy φ.

The classical CTL model checking algorithm for Kripke structures processes all the subformulas of the input formula φ in an increasing order of complexity, and for each state u, determines whether or not the subformula holds at u. The model checking for hierarchical Kripke structures works on a similar principle. The algorithm considers subformulas of φ starting with the innermost subformulas, and extends the label $X(u)$, at each node u, with the subformulas that are satisfied at u. At the beginning, for each node u, the labeling set $X(u)$ is initialized to contain the atomic propositions that are true at the node u. The subformulas whose top-level connective is a logical operator are processed in the natural way. For the subformulas whose top-level connective is a temporal operator, whether a node satisfies such a subformula can depend on the context. Consequently, the algorithm repeatedly transforms the hierarchical structure K, and is designed to maintain the following property: after the algorithm processes a subformula ψ, if a node u is labeled with ψ then in the expansion of the structure K^F, *every* state with the last component u satisfies ψ, and if a node u is not labeled with ψ then in K^F, *no* state with the last component u satisfies ψ.

Let us consider a formula $\psi = \exists\bigcirc\,\phi$. The truth of ϕ at every node is already known. Consider a node u of a module K_i. Multiple boxes of other modules may be mapped to K_i, and hence, u may appear in multiple contexts in the expansion. If u is not an exit node, then the successors of u do not depend on the context. Hence, the truth of ψ is identical in all states corresponding to u, and can be determined from the successors of u within K_i (if u has an edge into a box that is mapped to K_j, then the truth of ϕ at the corresponding entry node of K_j must be examined). If u is an exit node of K_i, then the truth of ψ may depend on the context. Let us assume that u is the only exit node of K_i. In this case, we need to create two copies of K_i: K_i^0 and K_i^1. The superscript indicates whether or not the exit-node of K_i has some successor that satisfies ϕ and is outside K_i. The exit node of K_i^1 is labeled with ψ. The exit node of K_i^0 is labeled with ψ only if it has a successor *within* K_i that satisfies ϕ. The mapping of boxes, originally mapped to K_i, must be consistent with the intended meaning: a box which has a successor satisfying ϕ is mapped to K_i^1 and to K_i^0 otherwise. If K_i has 2 exit-nodes u and v, then for different boxes mapped to K_i, whether the exit node u satisfies ψ can vary, and similarly, whether the exit node v satisfies ψ can vary. Consequently, we need to split K_i into four copies, depending whether both, only u, only v, or none, have a ψ-successor outside K_i. In general, if there are d exit-nodes, processing of a single temporal subformula can generate 2^d copies of each structure in the worst case.

Now let us consider an until-formula $\psi = \psi_1\,\exists\mathcal{U}\psi_2$. Whether a node u of a structure K_i satisfies ψ may depend on what happens after exiting K_i, and thus, different occurrences may assign different truth values to $\psi_1\,\exists\mathcal{U}\psi_2$, requiring splitting of each structure into two. Again, let us assume single-entry single-exit modules. The computation proceeds in two phases. In the first phase, we partition the index-set $\{1, \dots n\}$ into three sets, YES, NO, and MAYBE, with the following interpretation. An index i belongs to YES when the unique entry node in_i satisfies the until-formula ψ within the expansion K_i^F. Then, in K^F, every occurrence of the entry node in_i satisfies ψ. Now consider an index i that does not belong to YES. It belongs to MAYBE if within K_i^F there is a path from in_i to the unique exit node out_i that contains only states labeled with ψ_1. In this case, it is possible that for some occurrences of K_i^F in K^F, the entry node satisfies ψ depending on whether or not the corresponding exit-node satisfies ψ. In the last case, the index i belongs to NO, and in every occurrence of K_i^F, the entry node does not satisfy the formula ψ. This happens when upon entering K_i^F, a path is guaranteed to encounter a node violating $\neg\psi_1$ before visiting a node satisfying ψ_2 or the exit node of K_i.

To express the computation of the first phase succinctly, let us interpret CTL formulas over the module K_i by considering it to be an ordinary Kripke structure over vertices $N_i \cup B_i$. In this interpretation, a box is considered like an ordinary vertex, which is labeled with YES, NO, or MAYBE, according to the characterization of the index that it is mapped to. Observe that a box labeled with YES is like a node labeled with ψ_2, and a box labeled with MAYBE is like a node labeled with ψ_1. Now, for i going down from n to 1, if in_i satisfies $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})$, then i is added to YES, otherwise, if in_i satisfies $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(out_i \wedge \psi_1)$, then i is added to MAYBE, else i is added to NO. The computation for each index i can be performed by a simple depth-first search over the nodes and boxes of K_i starting at the node in_i in time $|K_i|$.

In the second phase, the new hierarchical structure K' along with the labeling of ψ is constructed. To obtain K', each structure K_i is split into two: K_i^0 and K_i^1. A box b that is previously mapped to K_i will be mapped to K_i^1 if there is a path starting at b that satisfies ψ, and otherwise to K_i^0. Consequently, nodes within K_i^1 can satisfy ψ along a path that exits K_i, while nodes within K_i^0 can satisfy ψ only if they satisfy it within K_i. Consider a box b of K_i that is originally mapped to K_j. If b has a successor satisfying $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})$, then in K_i^0, b is mapped to K_j^1, else it is mapped to K_j^0. Similarly, if b has a successor satisfying $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))$ then in K_i^1, b is mapped to K_j^1, else it is mapped to K_j^0. A node u of K_i inherits the labels in both copies. If u satisfies $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})$ then ψ is added to its labels in K_i^0, and if it satisfies $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))$, then ψ is added to its labels in K_i^1.

The construction of K' is immediate if we have computed, for each K_i, the set of nodes and boxes satisfying $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})$ and satisfying $(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))$ are computed. Since the nodes of K_i are already labeled with ψ_1 and ψ_2, and the boxes are labeled with YES, MAYBE, or NO, these two sets can be computed in time $O(|K|)$, as in the standard labeling algorithm of CTL [21].

The processing of formulas of the form $\exists\Box\phi$ requires similar splitting and two-phase algorithm. This leads to the following complexity:

Proposition 15. *The CTL model-checking problem* (K, φ) *can be solved in time* $O(k^2 \cdot |K| \cdot 2^{|\varphi| d})$*, where each module of* K *has at most* d *exit nodes and* k *entry nodes.*

The labeling algorithm described above is not optimal in terms of space requirements. It is known that deciding whether an ordinary Kripke structure M satisfies a CTL formula φ can be solved in space $O(|\varphi| \cdot \mathit{log}\ |M|)$ [32]. For a hierarchical structure K, the size of the expansion K^F is $O(|K|^{nd(K)})$. The expansion need not be constructed explicitly. Each state of the expansion can be represented in space $O(nd(K) \cdot |K|)$. The number of successors of a state of K^F is only polynomial in the size of K: the number of successors of an expanded state whose last component is the node u of the module K_i equals the number of successors of u if u is not an exit node, and is bounded by the product of the number of edges in K and the number of boxes mapped to the index i, if u is an exit-node. The successors of any state of the expansion can be computed in space polynomial in the size of K. It follows that the space-efficient algorithm of [32] requires space polynomial in the size of K, and consequently, CTL model

	Ordinary Structure M	Hierarchical Structure K
Reachability	$O(\|M\|)$	$O(\|K\|)$
Automata-emptiness	$O(\|M\| \cdot \|A\|)$	$O(k^2 \cdot a^2 \cdot \|K\| \cdot \|A\|)$
LTL model checking	$O(\|M\| \cdot 2^{\|\varphi\|})$	$O(k^2 \cdot \|K\| \cdot 8^{\|\varphi\|})$
CTL model checking	$O(\|M\| \cdot \|\varphi\|)$	$O(k^2 \cdot \|K\| \cdot 2^{\|\varphi\|d})$

Table 1. Summary of model checking results.

checking for hierarchical structures is in PSPACE. Hardness holds even when all modules have only one exit node, and this is established by reduction from the satisfiability of quantified Boolean formulas [7]. Hardness can also be established for fixed CTL formulas if modules have multiple exit nodes, and thus, the parametric complexity in the size of the structure is also PSPACE.

Proposition 16. *Model checking of CTL formulas for single exit hierarchical structures is* PSPACE*-complete, and the structure-complexity of CTL model checking for hierarchical structures is* PSPACE*-complete.*

The model checking results are summarized in Table 1. For hierarchical structures, model checking of branching-time formulas seems more expensive than model checking of linear-time formulas. This difference is intuitively due to the different complexities of the *local* and *global* model-checking problems. In local model checking, we want to check whether some (or all) paths starting at a specified state (e.g., an initial state) satisfy a linear-time property, while in the global model checking, we want to compute all (reachable) states such that some (or all) paths starting at that state satisfy a linear-time property. For ordinary state machines, both local and global variants can be solved in time linear in the size of the structure (even though the local, or on-the-fly, algorithms for checking linear-time properties are preferred in practice). For hierarchical structures, the local variant can be solved efficiently by our algorithm that avoids repeated analysis of a shared substructure. The global variant is more expensive, as it requires splitting of a substructure because the satisfaction of formulas can vary from context to context. CTL model checking requires solving the global problem repeatedly, due to the nesting of path quantifiers in the formula.

5 Extended Hierarchical State Machines

State-machine based specification languages are typically extended by introducing variables ranging over finite domains to allow description of complex systems concisely. This has motivated us to define the language of *hierarchic reactive modules* that allows hierarchical descriptions with variables [2,3]. In this section, we briefly review the relevant features of the language and techniques for BDD-based symbolic reachability analysis.

Before we describe the modeling language, let us examine the cost of adding variables. Suppose we have an extended hierarchical state machine K with

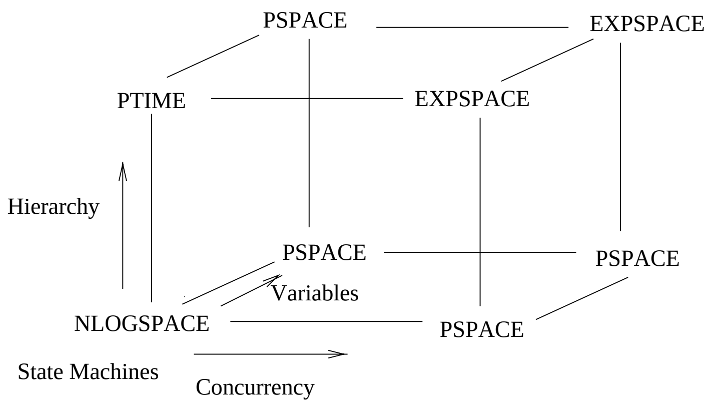

Fig. 3. Impact on complexity of reachability due to various features

m global boolean variables, and the edges have guards and assignments that read/write these variables. The expansion would suffer a blow-up due to the nesting as well as due to recording of the values of the variables. The number of states of the expansion will be $O(|K|^{nd(K)} \cdot 2^m)$, and reachability can be solved in PSPACE [2]. Note that for communicating extended state machines, reachability problem is in PSPACE, as the state can be completely described by the control locations of all the components and the values of all the Boolean variables, and transitions can be computed in polynomial space. When all the three features are present, that is, if we consider extended communicating hierarchical automata, again by augmenting each node with the values of the variables, and applying the expansion for CHA, we get an ordinary state machine of doubly exponential size, which can be searched in EXPSPACE. These results are summarized in Figure 3.

5.1 Modeling Language

The central component of the description language of hierarchical reactive modules is a *mode*. A mode is an HSM with a few extensions. In addition to entry, exit, and internal nodes, boxes (which are called submodes), and transitions connecting them, a mode also allows declaration of typed variables. Variables are partitioned into *local* and *global*. The scoping rules are as in standard structured programming languages: whenever a box inside a mode M refers to another mode N, a global variable of N must either be a global or local variable of M. The local variables of N are not accessible to M, while the global variables of N can be used to share information between the two at the time of invocation of N as well as upon its return. Each edge of a mode M, besides specifying the source

[2] To make this claim precise, we need to specify the syntax for writing guards and assignments, but the bound should hold for typical choices such as propositional formulas. Also, these bounds assume that all variables are global.

and the sink, has an associated guard which is a Boolean predicate over the variables of M, and an assignment to some of the variables of M. Global variables can further classified into *read* and *write* variables, and this limits the allowed access on transitions. The precise choice of allowed types of the variables and the syntax for writing guards and assignments is guided by the tradeoff between desired expressiveness and efficient representation and analysis using symbolic techniques.

Another feature that is useful in a modeling language based on hierarchical state machines is the use of *group transitions.* For this purpose, each mode has a special, default exit node dx. A transition starting at dx is called a *group transition* of the corresponding mode. It may be taken whenever the control is inside the mode. That is, a group transition of a mode is enabled at all the nodes and boxes inside a mode. Allowing such transitions has two benefits. First, it allows succinct textual or visual description. For example, if every node has a transition to a node u, then we can replace all these edges by a single edge from the default exit node dx to u. Second, since dx is an exit node, edges can connect the corresponding return in the higher level modes, and this can be used to model preemption. For example, suppose a box b of a mode M is mapped to the mode N. Then an edge of M connecting the return (b, dx) to a node (or a call) of M can allow the transfer of control from inside N to a node of M. This models interrupting the execution within N when the guard labeling this group transition becomes enabled. Furthermore, such a transition can test variables of M that are not visible to N. To define the group transitions precisely, we need to formalize their priority with respect to the internal transitions. Consider a node u within a mode M, an edge e with source u and an edge e' starting at the default exit dx or the corresponding return from a box mapped to M. In our modeling language, the edge e has a priority over e' (that is, the guard of e' has an implicit conjunct corresponding to the negation of the guard of e). This corresponds to *weak* preemption. However, other choices are possible: the group transition e' can have a higher priority than the local transition e, or both can have the same priorities (meaning if both are enabled, the choice is nondeterministic).

In the language of hierarchical reactive modules, concurrency is allowed only at the highest level: a system is a collection of top-level (hierarchical) modes that communicate via global shared variables. STATECHARTS, on the other hand, are similar to communicating hierarchical automata, with a richer event-based synchronization mechanism and nested hierarchy and concurrency constructs.

As the modeling language becomes rich with features such as hierarchy, concurrency, variables, group transitions, the operational semantics needs to be defined carefully. This is, again, done by defining the expansion that captures the underlying state machine. In this case, the node and the sequence of boxes giving the context, is called the *control* state, and this needs to be augmented with the *data* state that gives the values of all the variables of the active modes. Another important issue concerns *compositional observational* semantics. This means that two modes with identical interfaces in terms of entry and exit nodes and global variables, and identical observable behaviors should be interchangeable in any context. To formalize this, the notion of *observable behavior* must be made precise. For the hierarchical reactive modules language, this is defined using traces as follows. The execution of a mode can be best understood as a game, that is, as an alternation of moves, between the mode and its environment. In a *mode move*, the mode gets the state from the environment along its entry nodes. It then keeps executing until it gives the state back to the environment along one of its exit nodes. In an *environment move*, the environment gets the state along one of the mode's exit nodes. Then it may update any variable except the mode's local ones. Finally, it gives the state back to the mode along one of its entry modes. An *execution* of a mode M is a sequence of transitions of the mode. Given such an execution, the corresponding *trace* is obtained by projecting the states in the execution to the set of global variables. The *observational semantics* of a mode M consists of its entry and exit nodes, global variables, and the set of its traces. The key compositionality result is that the observational semantics of a mode can be constructed from its top-level state machine and the observational semantics of the modes it invokes. This also forms the basis for defining modular reasoning principles which allow deduction of properties of a mode from the properties of the modes it invokes [2].

5.2 Symbolic Search

Models described using boolean (or enumerated and other finite types) can be searched using symbolic model checking techniques that employ representations based on *ordered binary decision diagrams* (BDDs) [16, 17, 38]. Instead of obtaining a symbolic representation of the expanded mode, and then applying these classical techniques, our reachability analyzer attempts to exploit the hierarchical structure for representation as well as for search.

To begin with, the pool of variables is not global. For instance, the state can consist of variables x and y in one mode, and x and z in another. Instead of obtaining a symbolic representation of the transition relation of the expansion of a mode, we maintain the hierarchical control structure, and simply compile the guards and assignments annotating edges into a BDD-based symbolic form. This is basically a more general form of partitioned representation of the transition relation than the usually employed conjunctive partitioning [15]. Consequently, even if a mode appears in multiple contexts, its representation appears only once. This representation can also exploit the scoping rules. For example, if a mode does not have write access to a variable x, then the symbolic representation of the transitions within the mode do not have to include the constraint that x stays unchanged.

In symbolic fixpoint computation, the set of reachable states of a model is computed iteratively starting from the set of initial states by repeatedly applying the *image* computation. Instead of a single global BDD, we maintain the reachable states as a forest of BDDs indexed by the nodes. This allows us to *partition* the state space in regions, each containing all states with the same control state. During the image computation, the scoping rules are exploited to employ early quantification. For example, to process an edge from a node u to a node v, we need to conjunct the set of reachable states at node u with the transition relation associated with this edge. If v is an exit node, then we know that the local variables are no longer relevant, and can be existentially quantified. It is also worth noting that the search has a mixture of enumerative and symbolic techniques: it is enumerative over the nodes (that is, the control state), and symbolic over variables (that is, the data state).

Finally, if a mode M is instantiated many times or contains many local variables, we can do some preprocessing to improve the performance of reachability computation as follows. Note that an execution of M starts at an entry node, follows a sequence of transitions (which may involve invoking submodes), and finishes at an exit node. For the calling mode, this can be viewed as a macro-transition that connects entry nodes to exit nodes based on the values of the global variables, possibly updating them. We have experimented with computing the BDDs for such macro-transitions. Computation of macro-transitions requires computing the transitive closure of the transitions within M. This can be expensive, but once these are computed, the boxes referencing M can simply be replaced by its macro transitions, and the local variables of M are no longer needed.

These optimizations and their impact on the computational requirements of the symbolic search are described in [3, 6]. The speed-up due to the technique of computing the macro-transitions is also described in the context of analysis of Boolean recursive programs [8]. Heuristics for choosing variable ordering and for exploiting locality of variables to compile hierarchical specifications into the modeling language of SMV are discussed in [20]. A backwards reachability algorithm that exploits the hierarchical structure for early termination is discussed in [10].

6 Conclusions

We have discussed results on model checking and succinctness of hierarchical state machines. These results indicate when hierarchical descriptions can be analyzed efficiently without constructing the underlying expanded description. These results also suggest the restrictions that a modeling language should impose to lower the complexity. In particular, in presence of concurrency and hierarchy, unrestricted synchronization across levels is very expensive, and also poses obstacles for compositional semantics. In terms of developing heuristics that improve the performance of analysis tools in presence of all the features, some progress has been reported, but we believe that it merits further investigation. In particular, in recent years SAT-based techniques have been shown to be very effective in analyzing models with large number of variables [12], and it may be possible to optimize the SAT solvers by exploiting the hierarchical structure.

If the modules are allowed to call each other recursively, the model is equivalent to pushdown systems, for which model checking problems are known to be decidable, both for linear-time and branching-time requirements [18, 14], and has been a subject of renewed interest in the context of software analysis using abstraction [1, 11, 9, 26, 28]. Another topic of current interest involves two-player games on hierarchical or recursive structures [43, 19, 5].

Acknowledgements. The bulk of the results in this survey are originally reported in publications [7, 4, 3], and we thank our collaborators Kousha Etessami, Radu Grosu, Sampath Kannan, Salvatore La Torre, P. Madhusudan, Michael McDougall, Zijiang Yang, and Mihalis Yannakakis. This research was supported by the National Science Foundation awards CCR97-34115, CCR99-70925, and ITR/SY 0121431, by the Semiconductor Research Corporation award 99-TJ-688, and by the U.S. Army Research Office URI award DAAD19-01-1-0473.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of the 13th International Conference on Computer Aided Verification*, LNCS 2102, pages 207–220. Springer, 2001.
2. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 390–402, 2000.
3. R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 280–295. Springer, 2000.
4. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Automata, Languages and Programming, 26th International Colloquium*, pages 169–178. Springer, 1999.
5. R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS'03: Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2619, pages 363–378, 2003.
6. R. Alur, M. McDougall, and Z. Yang. Exploiting behavioral hierarchy for efficient model checking. In *Computer Aided Verification, 14th International Conference*, LNCS 2404, pages 338–342. Springer, 2002.
7. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):1–31, 2001.
8. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.
9. T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.
10. G. Behrmann, K. Larsen, H. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *TACAS '99: Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 1579, pages 163–177. Springer, 1999.
11. M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming, 28th International Colloquium*, volume LNCS 2076, pages 652–666. Springer, 2001.

12. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.
13. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.
14. A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*, LNCS 1243, pages 135–150. Springer, 1997.
15. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.
16. R. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
17. J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
18. O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92: Concurrency Theory, Third International Conference*, LNCS 630, pages 123–137. Springer, 1992.
19. A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, LNCS 2404, pages 428–441. Springer, 2002.
20. W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–519, 1998.
21. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
22. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2000.
23. E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.
24. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
25. D. Drusinsky and D. Harel. On the power of bounded concurrency I: finite automata. *Journal of the ACM*, 41(3):517–539, 1994.
26. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 232–247. Springer, 2000.
27. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
28. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.
29. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
30. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
31. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

32. O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.
33. R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
34. S. La Torre, M. Napoli, M. Parento, and G. Parlato. Hierarchical and recursive state machines with context dependent properties. In *Automata, Languages and Programming, 30th International Colloquium*, LNCS 2719, pages 776–789, 2003.
35. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.
36. O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.
37. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-verlag, 1991.
38. K. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
39. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
40. B. Selic, G. Gullekson, and P. Ward. *Real-time object oriented modeling and design*. J. Wiley, 1994.
41. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.
42. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
43. I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, 2001.