Communicating Hierarchical State Machines *

Rajeev Alur
University of Pennsylvania
alur @ cis.upenn.edu

Sampath Kannan
University of Pennsylvania
kannan @ cis.upenn.edu

Mihalis Yannakakis
Columbia University
mihalis@cs.columbia.edu

August 17, 2005

Abstract

Hierarchical state machines are finite state machines whose states themselves can be other machines. In spite of their popularity in many modeling tools for software design, very little is known concerning their complexity and expressiveness. In this paper, we study these questions for hierarchical state machines as well as for communicating hierarchical state machines, that is, finite state machines extended with both hierarchy and concurrency. We present a comprehensive set of results characterizing (1) the complexity of the reachability, emptiness and universality problems, (2) the complexity of the language inclusion and equivalence problems, and (3) the succinctness relationships between different types of machines.

1 Introduction

Finite state machines (FSMs) are widely used in the modeling of systems in various areas. 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 [CK96, CGP00, Hol97, VW94]. In the simplest setting, an FSM consists of a labeled graph whose vertices correspond to system states and edges correspond to system transitions. In practice, to describe complex systems using FSMs, several extensions are useful such as *communicating FSMs* in which the system is described by a collection of FSMs that operate concurrently and synchronize with one another periodically.

There is a rich body of theoretical results concerning complexity and expressiveness of state machines. By labeling the edges of an FSM with alphabet symbols and by introducing initial and final states, FSMs can be used to define regular languages. Analysis problems of interest include emptiness of the language, model checking with respect to temporal requirements, and inclusion and equivalence of the languages of two machines. For a single FSM, the complexity of some problems depends on whether the machine is deterministic or not. For instance, language equivalence is NL-complete for deterministic machines but PSPACE-complete for nondeterministic ones (cf. [HU79]). Introducing concurrency, that is, considering communicating FSMs, usually costs an extra exponential. For instance, simple reachability for communicating FSMs is PSPACE-complete, while equivalence equivalence is EXPSPACE-complete [HKV97]. Insight into the complexity of analysis problems is also provided by the succinctness afforded by various features. For instance, nondeterministic machines are exponentially more succinct than deterministic ones (cf. [HU79]), concurrent

*A preliminary version of this paper appears in *26th International Colloquium on Automata, Languages, and Programming* (ICALP), LNCS 1644, pp. 169–178, 1999. This research was supported in part by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

machines are exponentially more succinct than the sequential ones, and concurrent nondeterministic machines are doubly-exponentially more succinct than deterministic FSMs [DH94].

While the impact of features such as nondeterminism and concurrency on complexity and expressiveness of finite-state machines is well understood, there is almost no literature on understanding the impact of introducing *hierarchy* in state machines. In *hierarchical (nested) FSMs*, the states of an FSM can be ordinary states or *superstates* which are FSMs themselves. The notion of hierarchical FSMs was popularized by the introduction of STATECHARTS [Har87], and exists in various object-oriented software development methodologies such as ROOM [SGW94] and the Unified Modeling Language (UML [BJR97]). Hierarchical state machines have two descriptive advantages over ordinary FSMs. 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 FSMs, we need to specify components only once and then can reuse them in different contexts, leading to modularity and succinct system representations. In a recent paper, it is shown that the succinctness offered by sequential hierarchical state machines comes at no cost: the reachability problem for sequential hierarchical state machines can be solved without constructing the equivalent flattened ordinary FSM, and is P-complete [AY01]. In this paper, we investigate the reachability question for *communicating* hierarchical state machines, language equivalence problems for hierarchical and communicating hierarchical state machines, and succinctness issues.

Our first set of results concern algorithms for reachability in presence of both the concurrency and hierarchy constructs. While reachability of a single hierarchical state machine (HSM) can be solved in linear time [AY01], we show that the product of two HSMs is difficult to analyze and has a PSPACE-complete reachability problem. For *communicating hierarchical state machines* (CHM) that allow arbitrary nesting of the concurrency and hierarchy constructs, we show the reachability problem to be EXPSPACE-complete. This shows that hierarchy, in presence of concurrency, costs an extra exponential. Then, we proceed to identify a restriction on the use of the two constructs that avoids this extra cost. In *well-structured* CHMs, communication among hierarchical components is allowed only at the top level, and for such machines we show that one needs to pay only for concurrency: the reachability problem is PSPACE-complete as in the case of communicating FSMs.

Our second set of results concern checking equivalence of languages of two machines. For a sequential HSM, while language emptiness is P-complete, we show checking universality to be EXPSPACE-complete. It follows that checking language inclusion and equivalence are both EXPSPACE-complete problems. In the case of ordinary FSMs, *deterministic* machines have polynomial-time complexity for universality, inclusion, and equivalence, while the corresponding problems are PSPACE-complete for nondeterministic ones. This motivates us to consider these problems for deterministic HSMs: while universality becomes easy (P-complete), language inclusion is PSPACE-complete, and language equivalence can be solved in PSPACE (a matching lower bound remains an open question). When we consider communicating hierarchical machines, the costs due to concurrency and hierarchy add up: all of universality, language inclusion, and language equivalence are 2EXPSPACE-complete.

Finally, we consider *succinctness* afforded by the hierarchical construct. Starting with deterministic FSMs, we know that nondeterminism can add exponential succinctness. We show that hierarchy can also add exponential succinctness. Furthermore, nondeterministic FSMs and deterministic HSMs are incomparable extensions: both can be exponentially more succinct with respect to each other. Allowing both nondeterminism and hierarchy gives double-exponential succinctness with respect to deterministic FSMs, and curiously enough, this double-exponential gap exists between nondeterministic HSMs and deterministic HSMs also. Allowing nondeterminism, hierarchy, as well as concurrency, gives triple-exponential succinctness with respect to deterministic FSMs (or even deterministic HSMs), and double-exponential succinctness with respect to nondeterministic FSMs (or even nondeterministic HSMs)!

2 Communicating Hierarchical State Machines

There are many variants of definitions of finite-state machines. We choose a definition in which edges are labeled with alphabet symbols. For simplicity, we restrict ourselves to a single initial and a single final state, but generalization to multiple initial and final states poses no technical problems. A *finite-state machine* (FSM) consists of

- a finite set Q of states,
- a finite alphabet Σ,
- an initial state $q^I \in Q$,
- a final state $q^F \in Q$,
- a set $\rightarrow \subseteq Q \times \Sigma \times Q$ of transitions.

Given a word $\rho = \sigma_0\sigma_1 \cdots \sigma_n$ over the alphabet Σ, an *accepting run* of the FSM 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 equals the initial state q^I, q_{n+1} equals the final state q^F, 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 $L(M)$.

Now we proceed to add two features: concurrency and hierarchy. For concurrency, a machine is composed of a set of component machines which synchronize on transitions labeled with common alphabet symbols. For hierarchy, the states of a machine can be other machines. Such states are popularly known as *superstates* [Har87]. The meaning of such a hierarchical definition is obtained by recursively substituting each superstate by the machine associated with it. Hierarchical definitions allow sharing of patterns. The following formal definition allows arbitrary nesting of the concurrency and hierarchy constructs.

A *communicating hierarchical state machine* (CHM) is one of the following three forms:

Base case: An FSM $(Q, \Sigma, q^I, q^F, \rightarrow)$ is a CHM.

Concurrency: If $M_1, M_2, \ldots M_k$ are CHMs then $M_1 \| M_2 \| \cdots \| M_k$ is a CHM.

Hierarchy: If $\mathcal{M}$ is a finite set of CHMs, $N = (Q, \Sigma, q^I, q^F, \rightarrow)$ is an FSM with states Q, and μ is a labeling function $\mu : Q \mapsto \mathcal{M}$ that associates each state $q \in Q$ with a CHM in $\mathcal{M}$, then the triple $(N, \mathcal{M}, \mu)$ is a CHM.

A CHM of the form $M_1 \| M_2 \| \cdots \| M_k$ is called a *product expression*, and each M_i is called a component of the product expression. A CHM of the form $(N, \mathcal{M}, \mu)$ is called a *hierarchical expression*, the FSM N is called the *top-level* of the hierarchical expression, and each CHM in $\mathcal{M}$ is called a component of the hierarchical expression.

As an illustration, see Figure 1 that shows a partial CHM. The machine M is product of two hierarchical expressions M_1 and M_2. The states of M_1 and M_2 are mapped to lower-level

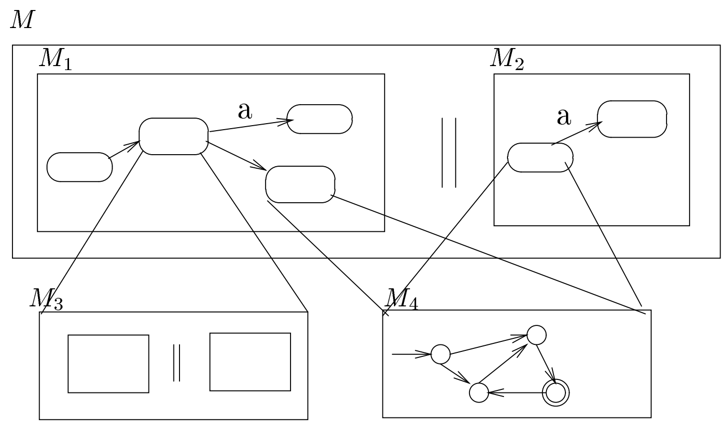

Figure 1: A schematic sample communicating hierarchical state machine

component machines. For instance, one state of M_1 is mapped to M_3, which in turn is a product of two machines, while another state of M_1 is mapped to the ordinary FSM M_4. Notice the sharing: machine M_4 is associated with states of both M_1 and M_2.

The semantics is defined by mapping each CHM M to an FSM $[\![M]\!]$:

Base case: If M is an FSM, then $[\![M]\!]$ equals M.

Concurrency: If M is the product expression $M_1 \| M_2 \| \cdots \| M_k$, then $[\![M]\!]$ is obtained by taking the product of the FSMs $[\![M_i]\!]$ corresponding to the components. Formally, suppose $[\![M_i]\!] = (Q_i, \Sigma_i, q_i^I, q_i^F, \rightarrow_i)$ for each i. Then,

- The state-space of $[\![M]\!]$ is $Q_1 \times \cdots \times Q_k$.
- The alphabet Σ of $[\![M]\!]$ is $\Sigma_1 \cup \cdots \cup \Sigma_k$.
- The initial state of $[\![M]\!]$ is $\langle q_1^I, \ldots q_k^I \rangle$.
- The final state of $[\![M]\!]$ is $\langle q_1^F, \ldots q_k^F \rangle$.
- For a symbol $\sigma \in \Sigma$, $[\![M]\!]$ has a σ-labeled transition from $\langle q_1, \ldots q_k \rangle$ to $\langle w_1, \ldots w_k \rangle$ iff
  - for every i such that $\sigma \in \Sigma_i$, the FSM $[\![M_i]\!]$ has a transition (q_i, σ, w_i), and
  - for every i such that $\sigma \notin \Sigma_i$, $w_i = q_i$.

Hierarchy: If $M = (N, \mathcal{M}, \mu)$ with top-level FSM $N = (Q, \Sigma, q^I, q^F, \rightarrow)$ then

- A state of $[\![M]\!]$ is of the form (q, w) where $q \in Q$ and w is a state of the FSM $[\![\mu(q)]\!]$ associated with q.
- A symbol σ belongs to the alphabet of $[\![M]\!]$ if σ belongs to the alphabet Σ of the top-level FSM N, or if σ belongs to the alphabet of $[\![\mu(q)]\!]$ for some $q \in Q$.
- The initial state of $[\![\mu(q^I)]\!]$ is the initial state of $[\![M]\!]$.
- The final state of $[\![\mu(q^F)]\!]$ is the final state of $[\![M]\!]$.
- $[\![M]\!]$ has two types of transitions
  - For a transition (q, σ, q') of the top-lavel FSM N, $[\![M]\!]$ has a σ-labeled transition from the final state of $[\![\mu(q)]\!]$ to the initial state of $[\![\mu(q')]\!]$.

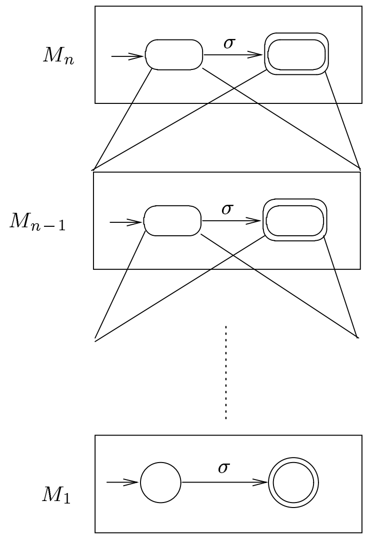

Figure 2: Counting with hierarchical state machines

– For $q \in Q$, if (w, σ, w') is a transition of $[\![\mu(q)]\!]$ then $((q, w), \sigma, (q, w'))$ is a transition of $[\![M]\!]$.

Now we can associate a language with each CHM: the language $L(M)$ of a CHM M is the same as the language $L([\![M]\!])$ of the FSM associated with M.

A CHM can be represented by a DAG. A terminal node corresponds to the base case, and has an associated FSM. An internal node may correspond to a product expression or a hierarchical expression. A node for a product expression is labeled with the operator $\|$, and its children are the components of the product. Note that the concurrency operator is associative. Consequently, we assume that the immediate children of a product expression are not themselves product expressions. A node for a hierarchical expression has the top-level FSM associated with it, and the edges connect the states of the top-level FSM with other nodes. When counting the size of the description of a CHM, we consider the size of this DAG representation. For instance, in Figure 1, the size of the FSM M_4 is counted only once, even though it appears multiple times. Two important parameters of this DAG representation are *width* and *depth*: width of a CHM M is the maximum number of components in the product nodes, and depth is the length of the longest path in the DAG.

A CHM M is called a *hierarchical state machine* (HSM) if it does not contain any product expression. Thus, HSMs do not involve any concurrency. To illustrate the power of HSMs, we show that HSMs can count with exponential succinctness (see Figure 2). The alphabet in this example consists of a single symbol σ. It is easy to construct a sequence of HSMs $M_1, M_2, \ldots M_n$ as follows. M_1 is an FSM with $L(M_1) = \{\sigma\}$, and for $i > 1$, M_i is a hierarchical expression with two superstates, each mapped to M_{i-1} so that $L(M_i) = L(M_{i-1}) \cdot \sigma \cdot L(M_{i-1})$. Thus, the language $L(M_i)$ of the HSM M_i contains precisely the string of length $2^i - 1$.

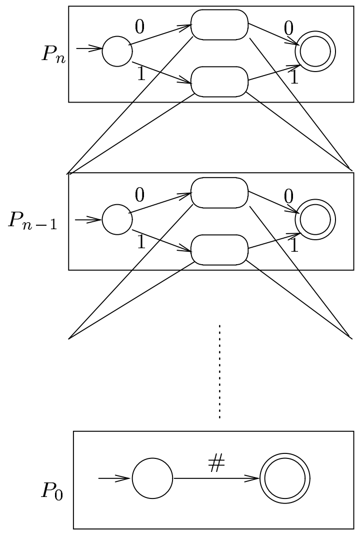

Figure 3: Hierarchical state machine accepting strings of the form $w \# w^R$

3 Reachability

The reachability problem for a CHM M is to determine if the final state of $[\![M]\!]$ is reachable from the initial state of $[\![M]\!]$. Alternatively, this can be viewed as checking if the language $L(M)$ is empty. The reachability problem for ordinary FSMs is in NL. Introducing hierarchy in FSMs comes at a minimal cost: the reachability problem for HSMs can be solved in linear time, and is P-complete [AY01]. On the other hand, introducing concurrency in FSMs is expensive: the reachability problem for product of FSMs is PSPACE-complete. Now we want to study the impact of the combination of the two constructs. We begin by considering product of hierarchical state machines.

3.1 Product of HSMs

For ordinary FSMs, the product of two FSMs has a quadratic blow-up. In [AY01], it is shown that the product of an HSM with an FSM can be constructed to yield an HSM with quadratic number of states. As the next result shows, the product of two HSMs cannot be constructed efficiently.

Before we present the result, consider the sequence $P_0, P_1, \ldots P_n$ of HSMs over the alphabet $\{0, 1, \#\}$ (see Figure 3). P_0 is an FSM with $L(P_0) = \{\#\}$, and for $i > 0$, P_i is a hierarchical expression with two superstates, each mapped to P_{i-1}, so that $L(P_i)$ is the union of $0 \cdot L(P_{i-1}) \cdot 0$ and $1 \cdot L(P_{i-1}) \cdot 1$. Thus, the language accepted by P_n is $\{ \ w \# w^R \ \mid \ w \in \{0,1\}^n \ \}$. It is worth noting that the language $\{ \ w \# w \ \mid \ w \in \{0,1\}^n \ \}$ cannot be defined succinctly by HSMs (in fact, we can prove that every HSM accepting this language must be of exponential size). Intuitively, an HSM can be viewed as a pushdown automaton with bounded stack size.

Recall that for pushdown automata, emptiness of single automaton can be solved in polynomial-time, but emptiness of intersection of two pushdown automata is undecidable. In the case of HSMs, emptiness of a single HSM is linear-time, but emptiness of the intersection of two HSMs is PSPACE-complete.

Proposition 1 *Reachability problem for $M_1 \| M_2$, where M_1 and M_2 are HSMs, is* PSPACE*-complete.*

Proof: To check reachability of $M_1 \| M_2$, we can first flatten the two HSMs, construct ordinary FSMs $[\![M_1]\!]$ and $[\![M_2]\!]$, and search their product. If n is a bound on the description of the two HSMs, and d is a bound on the nesting depths of the two, then the product $[\![M_1]\!] \| [\![M_2]\!]$ has at most n^{2d} states, and can be searched in PSPACE, giving the upper bound.

For the lower bound, we reduce the halting problem for linear-bounded automata, a PSPACE-complete problem, to checking emptiness of the intersection of two HSMs. Let T be a linear-bounded automaton. A configuration of T can be described by a string w of length n. A halting run of T can be described by a sequence $w_0, w_1, \ldots w_k$ such that each w_i describes a configuration of T, 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_T:

$$\{\ w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_k \ \mid\ w_0, w_1, \ldots w_k \text{ is a halting run of T } \}.$$

We will construct two HSMs M_1 and M_2 such that $L(M_1) \cap L(M_2)$ equals L_T. The HSM 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 HSM 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 FSM).

Each M_i is similar to the HSM P_n of Figure 3 that accepts strings of the form $w \# w^R$. Two changes are needed. First, the machine M_i needs to check repeatedly, which amounts to adding a #-labeled transition from the final state to the initial state. Secondly, the string following # is not identical to the reverse of the string preceding it, rather a legal successor configuration. This can be achieved easily by remembering at each level information about the adjacent levels. ■

3.2 Complexity of CHMs

In CHMs, the concurrency and hierarchy operators are arbitrarily nested, and the product components can synchronize with each other at different levels of hierarchy. These features make the reachability problem significantly difficult to solve. In fact, to solve the reachability problem for a CHM M, one cannot do better than the obvious solution of constructing the flattened FSM $[\![M]\!]$ (locally), and applying the standard reachability algorithm to it.

Proposition 2 *For a CHM M, the number of states of $[\![M]\!]$ is $O(n^{d^m})$, where each FSM in M has at most n states, M has width $d > 1$ and depth m.*

Proof: The proof is by induction on the depth m of the CHM. First let us consider the case when $m = 0$. Then, M is an ordinary FSM, and since $[\![M]\!]$ equals M, its size is $O(n)$.

Now, consider an FSM M with depth $m > 0$. The root is either a hierarchical expression or a product expression. If it is a hierarchical expression, then its top-level FSM has at most n states, each of which is mapped to a CHM of depth $m-1$ or less. Consequently, for each state q, by induction, the FSM $[\![\mu(q)]\!]$ is of size $O(n^{d^{m-1}})$, and $[\![M]\!]$ is of size $n \cdot O(n^{d^{m-1}})$.

If the root is a product expression, then it has at most d components $M_1, \ldots M_d$. Each M_i has depth $m-1$ or less, and by induction, $[\![M_i]\!]$ is of size $O(n^{d^{m-1}})$. Since $[\![M]\!]$ is the product of $[\![M_i]\!]$, its size is $O(n^{d^{m-1}})^d$, which is $O(n^{d^m})$. ■

This implies that reachability problem for a CHM can be solved in time $O(n^{d^m})$, that is, doubly-exponential in the depth in the worst case. The precise complexity class is EXPSPACE.

Theorem 3 *Reachability problem for CHMs is* EXPSPACE*-complete.*

Proof: Given a CHM M, the flattened machine $[\![M]\!]$ need not be constructed explicitly, and can be searched in space that is required to represent a single state of $[\![M]\!]$. Since $[\![M]\!]$ has doubly-exponentially many states, a state can be represented in exponential space, giving EXPSPACE upper bound.

For the lower bound, consider an EXPSPACE Turing machine T. A configuration of T can be represented by a string w of length 2^n over an appropriate alphabet Σ. A halting run of T can be described by a sequence $w_0, w_1, \ldots w_k$ such that each w_i describes a configuration of T, 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_T

$$\{\ w_0 \# w_1 \# w_2 \# w_3 \# \cdots \# w_k \ \mid\ w_0, w_1, \ldots w_k \text{ is a halting run of T } \}.$$

We will construct a CHM M_T whose language is L_T, and that will imply the EXPSPACE-hardness. The CHM M_T is the product of three CHMs M_1, M_2, and M_3. The machine M_1 is an HSM that ensures that w_0 is the initial configuration, w_k is the halting configuration, each configuration is of length 2^n, and successive configurations are separated by the $\#$ symbol. The size of M_1 is only $O(n)$, since we know how to count with exponential succinctness. Machine M_2 ensures that, for every $i \geq 0$, the configuration $2i+1$ is a legal successor of the configuration $2i$. Machine M_3 ensures that, for every $i \geq 0$, the configuration $2i+2$ is a legal successor of the configuration $2i+1$. It suffices to describe the construction of M_2.

Before we describe M_2, let us construct a sequence of machines $N_0, N_1, \ldots N_n$ as follows. The machine N_0 accepts strings of the form $\sigma\Sigma^{2^n}\sigma\Sigma^*$. The machine 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 machine N_i, the machine N_{i+1} is

$$N_i \parallel \Sigma^{2^i} \cdot N_i.$$

That is, N_{i+1} is a product of two machines, the first one being 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 machine, whose first state is mapped to a machine that counts 2^i, and second state is mapped to N_i. Observe that since CHMs are represented as DAGs, the size of description of N_n is only $O(n)$. Second, a string of the form $w \# w'$, where w and w' are strings of length 2^n, is accepted by N_n precisely when $w = w'$.

The above construction shows how to construct a succinct CHM that ensures copying of an exponential-length string. The desired machine M_2 requires two changes to the above construction. First, we want to ensure that w' is a legal successor configuration of w, rather than equality of the two. This can be achieved by modifying the construction so that a symbol adjoining the read-head of T is treated differently. Second, we would like to do the check repeatedly, which amounts to restarting N_n after it reaches final state. ■

3.3 Well-structured CHMs

Since reachability problem for CHMs is EXPSPACE-hard, we wish to identify subclasses with lower complexity. We proceed to define well-structured CHMs where arbitrary levels of hierarchy of product components cannot synchronize with each other. A CHM M is said to be *well structured* if for every product expression M' in M, if a proper component M'' of M' is a hierarchical expression, then for every proper component N of M'' and every component N' of M', the alphabet of $[\![N']\!]$ is disjoint from the alphabet of $[\![N]\!]$. Note that the alphabet plays a crucial role in the definition of the concurrency operator since components synchronize on common symbols. The restriction to well structured machines ensures that if two (or more) hierarchical machines are composed together, then they can synchronize only at the top level. For example, in Figure 1, a is a common symbol to the two components M_1 and M_2. Well-structuredness requires that a is used to label transitions of only the top-level FSMs of M_1 and M_2, and cannot appear, for instance, in M_3 or M_4. The reachability problem for well structured CHMs is PSPACE-complete, where the exponential cost is only due to concurrency, but not hierarchy.

Theorem 4 *The reachability problem for a well structured CHM M can be solved in time $O(k \cdot n^d)$, where k is the number of operators in M, each FSM in M has at most n states, and d is its width.*

Proof: We sketch an algorithm to check emptiness of the language of a well-structured CHM. Given a DAG representation of a well-structured CHM M, the algorithm proceeds in a bottom-up manner, processing nodes of smallest depth first. For each node processed, the algorithm determines if the language of the corresponding machine is empty or not.

To process a node corresponding to an ordinary FSM N, the algorithm simply determines if $L(N)$ is empty or not. This can be done using a standard search algorithm with time complexity linear in the size of N.

Now consider a node corresponding to a hierarchical expression N, whose top-level FSM is N'. For every state q of N', we already know if the language of the machine $[\![\mu(q)]\!]$ is empty or nonempty. If the language of machine associated with q is empty, we simply delete q from N'. Then, the language $L(N)$ is nonempty iff in the top-level FSM N', after deletion as above, there is a path from the initial state to the final state.

Finally, consider the case of a node which corresponds to a product expression N with d components $N_1, \ldots N_d$. Each component N_i is either an ordinary FSM or a hierarchical expression. If N_i is a hierarchical expression with top-level FSM N_i', we delete every state q in N_i' such that the language of the machine associated with q is empty (this information is already known due to bottom-up processing). Then, we can treat N_i as an ordinary FSM represented by N_i' after deletions. This is because, due to well-structuredness requirement, we know that components do not synchronize below the level of N_i'. Then, we can simply take the product of all ordinary FSMs N_i, and check if the product has empty language or not. This costs $O(n^d)$ since each FSM has size at most n.

Note that the bottom-up approach ensures that each node is processed just once. This leads to the time complexity of $O(k \cdot n^d)$. ∎

The results of this section are summarized in Figure 4 which lists the complexity of reachability for state machines as we add the features of concurrency and hierarchy.

4 Language Equivalence

In this section, we consider equivalence problem for hierarchical state machines and communicating hierarchical state machines. Each HSM M defines the language $L(M)$. The emptiness problem for HSMs is to determine if $L(M)$ is empty, and can be solved in P using the reachability algorithm. The universality problem for HSMs is to determine if the complement of $L(M)$ is empty. This problem turns out to be much harder:

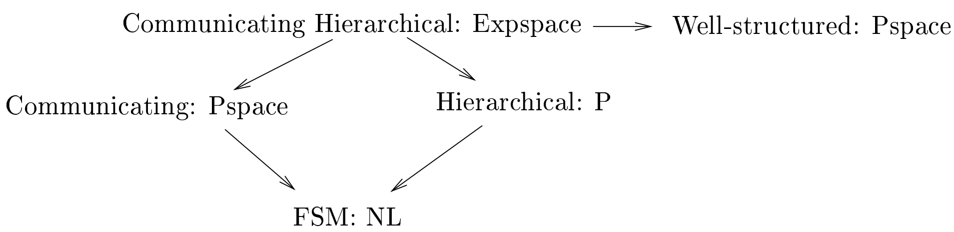

Figure 4: Summary of reachability results

Theorem 5 *The universality problem for HSMs is* EXPSPACE*-complete.*

Proof: We will show that an HSM can recognize invalid computations of an exponential space turing machine. This will show that determining universality of the language of an HSM is EXPSPACE-hard.

Recall that the HSM M_n can count to 2^n symbols succinctly (see Figure 2). Let T be an exponential space Turing Machine with transition function δ_T and let x be an input to T. Note that since we are talking about EXPSPACE, we can assume without loss of generality that T is deterministic.

The HSM for recognizing invalid computations will guess one of four types of errors — the tableau is ill-formed, the instantaneous description (ID) in the first row does not correspond to the input x, the ID in the last row is not accepting, or there exists an ID i, and a cell position j in the i^{th} ID which does not follow from the $(i-1)^{th}$ ID. The first three types of errors are easy to check. For the last case, the HSM non-deterministically guesses the pair (i, j), "remembers" cell positions $(i-1, j-1), (i-1, j)$, and $(i-1, j+1)$, uses M_n to count upto the width of the tableau (which is some fixed exponential function) and then checks that cell (i, j) does not follow by the rules of δ_T.

To show that the problem is in EXPSPACE, note that the flattened version of the HSM has at most doubly exponentially many states and the universality problem can be viewed as a reachability problem in the complement of this flattened machine. Since reachability is in NL, this can be done in log of doubly exponential, i.e., singly exponential space. Somewhat more precisely, we don't actually construct the doubly exponential sized flattened machine but explore it locally to stay within the prescribed space bound. ∎

Two HSMs M_1 and M_2 are *trace equivalent* if their languages are identical. This problem, and also the language inclusion problem, have the same complexity as the universality problem:

Theorem 6 *The trace equivalence problem for HSMs is* EXPSPACE*-complete.*

Proof: Since we can test universality of an HSM M by testing its trace equivalence to a simple machine that accepts Σ^*, it is clear that trace equivalence is EXPSPACE-hard.

For the upper bound, given M_1 and M_2, it suffices to check emptiness of the product of M_1 with complement of M_2, and the emptiness of the product of M_2 with complement of M_1. This can be done by flattening and complementing. For instance, the product of $[\![M_1]\!]$ and the complement of $[\![M_2]\!]$ has doubly-exponentially many states, and can be searched in EXPSPACE. ∎

For ordinary FSMs, problems such as universality, inclusion, and equivalence, are much easier if we consider deterministic variants. The results for HSMs show somewhat subtle distinctions between these problems. Recall that an FSM is deterministic if for every state q and every symbol

σ, there is at most one σ-labeled transition with source q. An HSM M is deterministic if $[\![M]\!]$ is deterministic. This entails two requirements: each base FSM and each top-level FSM of a hierarchical expression is deterministic, and furthermore, if a state q of a top-level FSM of a hierarchical expression has a σ-labeled outgoing transition, then the final state of the FSM associated with q has no σ-labeled outgoing transition. The latter condition ensures determinism concerning exiting lower-level FSMs.

Theorem 7 *For deterministic HSMs, the universality problem is P-complete, and the language inclusion problem is* PSPACE*-complete.*

Proof: The universality problem is the reachability problem in the complement HSM and is P-complete.

Now we show that language inclusion is PSPACE-complete. Let M_1 and M_2 be the two HSMs with associated languages L_1 and L_2 where the problem is to determine if $L_1 \subseteq L_2$. This is equivalent to testing reachability in the product machine formed by M_1 and the complement of M_2. This is in PSPACE.

To see PSPACE-hardness, observe that proof of Proposition 1 establishing PSPACE-hardness of checking emptiness of the intersection of languages of two HSMs, applies to the case of deterministic HSMs also. We will show that we can use language inclusion to determine if a tableau represents a valid computation of a PSPACE Turing Machine. Let $ID_1, ID_2, \ldots, ID_k$ be the sequence of instantaneous descriptions of a (deterministic) PSPACE machine. Note that each of these IDs is polynomially long, and without loss of generality, we can take this length to be some fixed polynomial. Let us in fact consider the sequence where every even ID is reversed. A product HSM can be formed from two HSMs N_1 and N_2, where N_1 checks that every even ID follows from the previous one in this modified sequence and N_2 checks that every odd ID follows from the previous one. This product HSM accepts a string if and only if it represents an accepting tableau of the PSPACE machine. To reduce to language inclusion we ask if the language of N_1 is contained in the complement of the language of N_2. This is exactly the condition under which the intersection is empty and there are no accepting tableaux. Thus, this problem is PSPACE-complete. ∎

For deterministic HSMs, trace equivalence can be solved in PSPACE, but its complexity is sandwiched somewhere between that of the P-complete universality problem and the PSPACE-complete language inclusion problem. It remains open to determine its exact complexity. Recall that an HSM is a pushdown automaton with an a priori bound on the depth of the stack. For deterministic pushdown automata, universality is P-complete, while language inclusion is decidable. The language equivalence problem for deterministic pushdown automata was open for several decades, and was recently proved to be decidable [Sen01].

Finally, we establish that the universality, language inclusion, and language equivalence problems for communicating hierarchical machines are complete for double-exponential space.

Theorem 8 *For CHMs, the universality, language inclusion, and language equivalence problems are 2*EXPSPACE*-complete.*

Proof: We will only consider universality. Given a CHM M, the size of $[\![M]\!]$ is at most double-exponential, and hence, there exists a triple-exponential deterministic FSM that accepts $L(M)$. Universality can be checked by searching this deterministic FSM in space logarithmic in its size, that is, doubly-exponential in the size of M. This implies 2EXPSPACE upper bound for checking universality of M.

Showing 2EXPSPACE-hardness is perhaps the most challenging proof in this paper. Consider a 2EXPSPACE Turing machine T. A configuration of T 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 T 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_T

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

We will construct a CHM M that accepts the complement of L_T, which would imply the lower bound for checking universality. The machine M needs to check that the string does not encode a halting run. Errors concerning initial and halting configurations can be detected easily.

The machine M needs to detect errors in encoding using the binary counter. In L_T, 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 a nondeterministic HSM of size $O(n)$ due to the properties of a binary counter (left as an exercise).

The tricky part is to detect an error if adjacent configurations are not related by a legal move of the machine. 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

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

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 $O(n)$ CHM that spawns 2^n parallel machines, each checking one bit of b. This is done along the same lines as the proof of EXPSPACE-hardness of reachability of CHMs. In particular, let N_1 be the machine 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 $N_i \parallel \{0,1\}^{2^i} \cdot N_i$. The claim is that the machine 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 (this is like the barrier synchronization). ■

Another popular equivalence for FSMs is bisimulation. To check if two HSMs are bisimilar, we can run the standard bisimulation algorithm on their expanded versions. This establishes that

	Emptiness	Intersection	Universality	Inclusion	Equivalence
FSM	NL	NL	PSPACE	PSPACE	PSPACE
Det FSM	NL	NL	NL	NL	NL
HSM	P	PSPACE	EXPSPACE	EXPSPACE	EXPSPACE
Det HSM	P	PSPACE	P	PSPACE	$\in$ PSPACE
CHM	EXPSPACE	EXPSPACE	2EXPSPACE	2EXPSPACE	2EXPSPACE

Figure 5: Summary of complexity results

bisimulation of HSMs can be checked in exponential time. There is no matching lower bound. The results concerning HSMs and CHMs are summarized in Figure 5. The table lists the known complexity bounds for ordinary FSMs also for comparison.

5 Succinctness

In this section we are concerned with the expressive power of the hierarchical construct, both in presence and absence of the concurrency construct. To discuss expressive power, we will look at families of languages $\mathcal{L} = \{L_n | n = 1, 2, \ldots\}$ and consider the number of states $\#(\mathcal{L}, n)$ needed by a machine that accepts L_n in some particular formalism. Where necessary we adopt the baroque practice of denoting the formalism by a superscript for $\#(\mathcal{L}, n)$. Thus $\#^{FSM}(\mathcal{L}, n)$ denotes the number of states necessary and sufficient for an FSM to recognize L_n. We will say that a formalism F_1 can be exponentially more succinct than a formalism F_2 if there is a family of languages $\mathcal{L}$ for which $\#^{F1}(\mathcal{L}, n) = O(\log[\#^{F2}(\mathcal{L}, n)])$.

While hierarchical state machines define regular languages, they can be much more succinct than ordinary FSMs. Note that there is an exponential translation from HSMs to FSMs, and the translation preserves determinism. Recall that nondeterministic FSMs can be exponentially more succinct than deterministic ones. The following proposition is established by showing that the language $L_n = \{w \# w^R \mid w \in \{0,1\}^n\}$ can be recognized by a deterministic HSM of linear size, while any nondeterministic FSM to recognize L_n must be at least exponential in n.

Proposition 9 *Deterministic HSMs can be exponentially more succinct than nondeterministic FSMs.*

Proof: Consider the language family $\mathcal{L}$ where $L_n = \{w \# w^R \mid |w| = n, w \in \{0,1\}^*\}$. There is an HSM of linear size that recognizes this family (see Figure 3). In other words, $\#^{HSM}(\mathcal{L}, n) = O(n)$. Note that the HSM of Figure 3 is deterministic.

On the other hand we can show that any nondeterministic FSM to recognize L_n must use at least exponential in n states. To see this, consider the set of all 2^n strings that are in the language L_n. Let M be a nondeterministic machine for L_n and consider the accepting paths in M for each of these 2^n accepted strings. If two of these accepting paths are in the same state after reading the $\#$ symbol, say the paths that start with $x\#$ and $y\#$ for $x \neq y$, then we can see that the machine will also accept $x\#y^R$ which it should not. Thus there must be 2^n distinct states. ∎

The next result establishes that, while nondeterminism introduces exponential succinctness for ordinary FSM, it introduces doubly-exponential succinctness in presence of hierarchy. The proof uses the language $L_n = \{w \mid w_i = w_{i+2^n} \text{ for some } i\}$ to establish the gap:

Proposition 10 *Nondeterministic HSMs are doubly-exponentially more succinct than deterministic HSMs.*

Proof: Let $\mathcal{L}$ be the language family where

$$L_n \;=\; \{w \mid |w| = 2 * 2^n \text{ and exists } i \text{ such that } w_i = w_{i+2^n}\}.$$

Clearly there is a linear-sized nondeterministic HSM for L_n where the machine simply guesses the position i, remembers the symbol w_i, counts 2^n symbols, and checks that the next symbol is in fact equal to w_i.

On the other hand, we can show that any family of deterministic HSMs for this language family must have doubly exponential size. ■

Note that the gap between nondeterministic HSMs and nondeterministic FSMs is singly exponential. It is interesting to note that nondeterminism and hierarchy are incomparable extensions, which is established using the language $L_n \;=\; \{w \mid w_i = w_{i+n} \text{ for some } i\}$.

Proposition 11 *Nondeterministic FSMs can be exponentially more succinct than deterministic HSMs.*

Proof: Consider the language family where

$$L_n \;=\; \{w \mid |w| = 2n \text{ and exists } i \text{ such that } w_i = w_{i+n}\}.$$

It is easy to see that there is an $O(n)$ nondeterministic machine to accept L_n. A deterministic FSM, on the other hand, needs to remember the first n symbols, and thus, has to have at least 2^n states. Even deterministic HSMs must have 2^n states, and this proof is along the lines of the proof of the previous proposition. ■

Finally, we consider the expressive power added by concurrency and show that this is significant.

Proposition 12 *A product of two HSMs can be exponentially more succinct than nondeterministic HSMs.*

Proof: Consider the language family where

$$L_n \;=\; \{w \# w^R \# w \mid w \in \{0,1\}^n\}.$$

There is intersection of the languages of two deterministic HSMs. However, it can be shown that any HSM for it (even if it is non-deterministic) requires exponential-size. ■

When unrestricted use of concurrency and hierarchy is allowed, we get double-exponential succinctness compared to both FSMs and HSMs, and triple-exponential succinctness compared to the deterministic variants of FSMs and HSMs.

Proposition 13 *Communicating hierarchical machines can be doubly-exponentially more succinct than nondeterministic HSMs (and nondeterministic FSMs), and triply-exponentially more succinct than deterministic HSMs (and deterministic FSMs).*

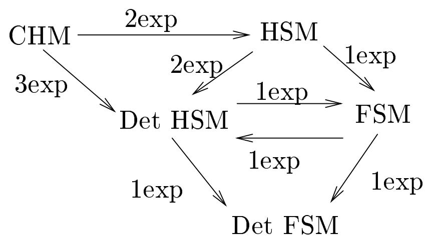

Figure 6: Summary of succinctness results

Proof: Consider the language family where

$$L_n \;=\; \{w_0\#w_1\#\cdots\#w_k \;\mid\; |w_i| = 2^n \text{ for each } i \text{ and } w_i = w_j \text{ for some } i,j\}.$$

A nondeterministic CHM of size $O(n)$ to accept L_n can be constructed along the lines of the proof of 2EXPSPACE-hardness of CHM universality. A nondeterministic machine can guess the position i, but needs to remember the exponentially long string w_i, and hence, is at least of double-exponential size. A deterministic machine needs to remember all the words w_i read so far, and is at least of triple-exponential size. ∎

The various succinctness claims are summarized in Figure 6.

6 Conclusions

In this paper, we have answered several questions concerning complexity and succinctness of hierarchical state machines. As the summary figures at the end of each section indicate, the complexity bounds do not change in a uniform way with many peculiarities. In terms of practice, the EXPSPACE-hardness of the reachability problem in presence of unrestricted use of hierarchy and concurrency constructs suggests that modeling languages should enforce well-structured use of the constructs to lower the complexity of the analysis problems. The results concerning the reachability problem should carry over to the more general question of model checking of linear-time requirements. We have presented a comprehensive picture that characterizes the power and complexity of sequential and communicating hierarchical state machines. A few questions concerning subclasses of communicating hierarchical machines, such as the precise complexity of equivalence of deterministic CHMs, remain to be resolved.

References

[AY01] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):1–31, 2001.

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison Wesley, 1997.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* MIT Press, 2000.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I: finite automata. *Journal of the ACM*, 41(3):517–539, 1994.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HKV97] D. Harel, O. Kupferman, and M.Y. Vardi. On the complexity of verifying concurrent transition systems. In *Proc. 8th Conferance on Concurrency Theory*, LNCS 1243, pages 258–272. Springer-Verlag, 1997.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Sen01] G. Senizergues. $L(A) = L(B)$? Decidability results from complete formal systems. *Theoretical Computer Science*, 251:1–166, 2001.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-time object oriented modeling and design.* J. Wiley, 1994.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.