Hierarchical State Machines

Mihalis Yannakakis

Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract. Hierarchical state machines are finite state machines whose states themselves can be other state machines. Hierarchy is a useful construct in many modeling formalisms and tools for software design, requirements and testing. We summarize recent work on hierarchical state machines with or without concurrency. We discuss issues of expressiveness and succinctness, the complexity of basic operations (such as emptiness, equivalence etc.), and algorithmic problems in the analysis of hierarchical machines to check for their properties.

1 Introduction

Finite state machines constitute one of the most fundamental modeling mechanisms in Computer Science. They have been widely used to model systems in a wide variety of areas, including sequential circuits, event-driven software, communication protocols and many others. In its most basic form, a finite state machine consists of a directed graph whose nodes represent system states and edges correspond to system transitions. When equipped with an input (and/or output) alphabet, FSM's become language recognizers defining the regular languages (or transducers). A rich theory of FSM's had been developed since the 50's regarding their expressive power, their operations and the analysis of their properties.

In practice, several extensions are useful to enhance the expressive power and to describe more complex systems more efficiently. Two popular extensions that are fairly well understood are the addition of variables to form so-called *extended finite state machines*, and the addition of concurrency (communication) to form *communicating finite state machines*. A third useful extension that has been less well studied is the addition of a hierachical (nesting) capability, to form *hierarchical state machines*, i.e., machines whose nodes can be ordinary states or superstates which are FSM's themselves. This is a very useful construct to structure and represent large systems.

The notion of hierarchical FSMs was popularized by the introduction of STATECHARTS [Har87], and exists in many related specification formalisms such as MODECHARTS [JM87] and RSML [LHHR94]. It is a central component of various object-oriented software development methodologies developed in recent years, such as OMT [RBPE91], ROOM [SGW94], and the Unified Modeling Language (UML [BJR97]). This capability is commonly available also in commercial computer-aided software engineering tools that are coming out, such as

Statemate (by i-Logix), ObjecTime Developer (by ObjecTime), and RationalRose (by Rational) (the last two are now combined into RationalRose-RealTime).

The nesting capability is useful also in formalisms and tools for the requirements and testing phases of the software development cycle. On the requirements side, it is used to specify scenarios (or *use cases* [Jac92]) in a structured manner. For instance, the new ITU standard Z.120 (MSC'96) for message sequence charts [RGG96] formalizes scenarios of distributed systems in terms of hierarchical graphs built from basic MSC's. The Lucent uBET toolset for behavioral requirements engineering uses a similar formalism and models requirements by hierarchical message sequence charts (uBET stands for Lucent Behavioral Engineering Toolset and is based on the MSC/POGA prototype tools [AHP96, HPR97]).

On the testing side, FSMs are used often to model systems for the purpose of test generation, and again the nesting capability is useful to model large systems. For example, Teradyne's commercial tool TestMaster [Apf95] is based on an extended hierarchical FSM model, i.e. it employs hierarchy and variables (but not concurrency).

In this paper we will summarize recent work on the impact of adding hierarchy to FSM, when added alone or in conjunction with concurrency. Most of the results come from the papers [AY98,AKY99,AY99] to which we refer for more details. In Section 2 we discuss hierarchical state machines: their expressive power and succinctness as compared to ordinary FSM's, the effect of nondeterminism vs. determinism in this context, the complexity of basic operations such as checking for emptiness, universality, equivalence etc, and the model checking of properties on hierarchical state machines. In Section 3 we discuss the effect of combining concurrency with hierarchy and address similar questions. In Section 4 we discuss hierarchical message sequence charts. In Section 5 we comment briefly on issues of testing, and we conclude in Section 6.

2 Hierarchical FSMs

There are many variants in definitions of finite state machines, depending on whether one views them as language acceptors/generators (automata), or machines that react with their environment and produce output (eg. Mealy and Moore machines, I/O automata), or logical models (Kripke structures) etc. For concreteness, let's settle with the automata definition. An ordinary *basic finite state machine* M consists of a finite set Q of states, an initial (or entry) state q_0, a set of final (or exit states) F, a finite input alphabet Σ, and a set E of transitions, $E \subseteq Q \times \Sigma \times Q$. Finite state machines are usually drawn as directed graphs with the states as nodes and the transitions as edges labeled by the inputs. The language $L(M)$ accepted by M is the set of strings over Σ that label paths from the initial state to a final state.

Hierarchical machines (HSM) are finite state machines whose states are themselves other basic FSM's or previously defined hierarchical machines. For simplicity we just give the formal definition for the single entry-single exit case. Formally, HSM's are defined inductively as follows. In the base case, a basic FSM is a hierarchical machine. Inductively, suppose that $\mathcal{M}$ is a set of HSM's. If N is a FSM with set of states Q and μ is a labeling function $\mu : Q \mapsto \mathcal{M}$ that associates each state $q \in Q$ with a HSM in $\mathcal{M}$ then the triple $(N, \mathcal{M}, \mu)$ is a HSM.

A hierarchical machine H represents in a compact notation a corresponding ordinary FSM $flat(H)$, the flattened version of H. The flat FSM can be constructed inductively as follows. In the base case, $flat(H) = H$. For the inductive step, if $H = (N, \mathcal{M}, \mu)$, then replace in the FSM N each state q by a copy of the flat version $flat(\mu(q))$ of its corresponding machine; note: if several states map to the same machine of $\mathcal{M}$ then we replace them by distinct copies of the flat machines with distinct states. Transitions of N coming into state q are directed to the entry state of the copy of $flat(\mu(q))$ and transitions of N coming out of q are now coming out of the exit state of $flat(\mu(q))$. If the entry state of N is q_0 and the exit state is q_f, then the entry state of $flat(H)$ is the entry state of $flat(\mu(q_0))$ and the exit state of $flat(H)$ is the exit state of $flat(\mu(q_f))$.

The definition of hierachical machines for the multiple entry - multiple exit case is similar, except that one has to specify in this case the set of entry and exit states, and to specify for every transition (q, a, q') of the top level FSM N in addition an associated exit state of $\mu(q)$ and an entry state of $\mu(q')$. We omit here the formalization.

An HSM H can be represented by a rooted DAG, whose leaves are basic FSM's and each internal node u has an associated FSM M_u and a mapping μ_u from the states of M_u to the children of u. The size of the (representation of the) HSM H is the sum of the sizes (numbers of states and transitions) of all the machines M_u at the nodes of the DAG.

Note that the same machine M_j may be used by many other machines M_i and by many of their states. This reuse permits exponential succinctness as compared to the flattened machine. Consider for example the HSM depicted in Figure 1, over a unary alphabet $\Sigma = \{a\}$. At each level, each M_i has two states that are mapped to the machine M_{i-1} at the previous level. The size of the HSM in this example is $O(n)$. However, the flattened machine in this case is simply a path of length $2^n - 1$. Thus, the machine is a counter that accepts just one string, the string of length $2^n - 1$.

2.1 Expressibility

Hierarchical state machines are essentially pushdown machines with a pushdown stack that is bounded by the input. They capture of course the same languages as ordinary FSM (regular languages), they gain only in succinctness. If a HSM H is defined by a DAG D of r machines $\{M_i\}$, each with at most n states, and if the nesting depth is m (i.e. the maximum length of a path in the DAG), then the flattened FSM has at most $O(n^m)$ states: every state of $flat(H)$ can be represented as a tuple $(q_1, q_2, ...q_t)$, $t \leq m$, of states of the given machines M_i, where q_1 is a state of the top-level machine, q_2 is a state of $\mu(q_1)$, and so

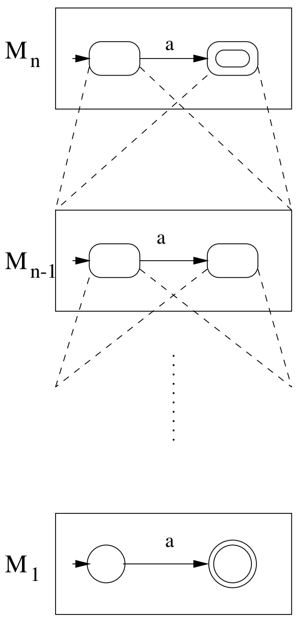

Fig. 1. An exponential counter

forth, i.e. every other state q_j of the tuple is a state of the machine to which the previous state q_{j-1} of the tuple is mapped. Thus, hierarchical machines can be translated to ordinary FSM's at an exponential cost. This is in general inherent.

We can consider the impact of hierarchy on the relation between determinism and nondeterminism. Recall that a FSM is deterministic if for every state q and every input symbol a there is at most one transition out of q labeled a. We say that a HSM is deterministic if its flat FSM is deterministic.

In the case of ordinary FSM's there is of course an exponential gap in succinctness between deterministic and nondeterministic FSM's. For hierarchical machines the gap becomes doubly exponential. That is, there are (nondeterministic) HSM's of size n such that the smallest equivalent deterministic HSM has size doubly exponential in n. An example of such a family of languages is the set L_n of strings w of length $2 \cdot 2^n$ such that there is an index i for which $w_i = w_{i+2^n}$. A nondeterministic HSM of linear size can guess i, record w_i, count 2^n as in Figure 1, and check that $w_i = w_{i+2^n}$. On the other hand, deterministic HSM's need doubly exponential size for this language [AKY99].

Of course, the gap between nondeterministic HSM's and deterministic HMS or even basic FSM's is no worse than doubly exponential: one exponential suffices to translate from HSM to nondeterministic basic FSM, and another exponential to deterministic FSM.

Comparing deterministic hierarchical machines with nondeterministic basic FSM's, there is an exponential gap in both directions. That is, one the one hand, there is a family of languages that is accepted by linear size nondeterministic

FSM's but needs exponential size for deterministic HSM's; one such language is the set of strings w of length $2n$ such that $w_i = w_{i+n}$ for some i. On the other hand, there is a language that can be accepted by a linear size deterministic HSM but needs exponential size for nondeterministic FSM's; one such language is the set of strings of the form $w\#w^R$, i.e. the second half w^R is the reverse of the first half w of the input string.

2.2 Operations

Usual operations of interest for a state machine and its language L are emptiness (is $L = \emptyset$?), universality (is $L = \Sigma^*$), and complementation (construct a machine that accepts the complement). For pairs of state machines M_1, M_2, operations of interest include intersection (is $L(M_1) \cap L(M_2) = \emptyset$?, and compute a machine that accepts the intersection), and comparison of the two machines: equivalence (is $L(M_1) = L(M_2)$?) and containment or inclusion (is $L(M_1) \subseteq L(M_2)$?) We discuss the complexity of these operations for hierarchical machines.

Emptiness. This is equivalent to a reachability problem in the flattened machine: can the initial state reach a final state? This problem can be solved in linear time in the size of the HSM by a simple dynamic programming search algorithm. In the case of flat FSM's the problem is in NL (nondeterministic logspace). In the case of hierarchical machines, the problem becomes P-complete.

Universality. As in the case of basic FSM's, this is a harder problem. As is well-known, for basic FSM's universality is PSPACE-complete. Hierarchy costs an additional exponential in this case: universality for HSM's is EXPSPACE-complete. Note that universality is emptiness for the complement, and emptiness is easy (hence complementation is hard). For deterministic HSM's we can complement them easily, and hence we can solve universality in linear time.

Intersection. For basic FSM's intersection is a polynomial operation. This holds also for intersection of a HSM with a FSM: that is, given a HSM H and a (basic) FSM M, we can define their product, which is another HSM H' such that $L(H') = L(H) \cap L(M)$. The size of H' is bounded by $|H||M|^2$; basically, the product involves at worst pairing every machine in the representation of H with a copy of M initialized at each state of M. However there is no such simple product construction for two HSM's: determining whether the intersection of the languages of two HSM's is empty turns out to be PSPACE-complete.

Equivalence and Inclusion. Since universality is EXPSPACE-hard, it follows that equivalence and containment are at least as hard. Indeed they are both in EXPSPACE, by flattening and complementing one of the machines, and forming the product with the other machine. In the case of deterministic HSM's (for which universality as well as complementation is easy), inclusion turns out to be PSPACE-complete. As for equivalence of deterministic HSM's, it can be done in PSPACE but the precise complexity is open.

2.3 Verification

A hierarchical state machine H is considered now as modeling a system. The model checking problem asks whether a given system model satisfies a given property P. The model checking problem has been studied for a variety of classes of properties, and there are several software tools to automatically perform the verification for finite state systems (eg. SPIN [Hol97], COSPAN [Kur94], SMV [McM93]). Typically, properties are defined either using automata, or some form of logic (linear time or branching time). Usually models are finite state structures that have labelled nodes instead of edges; specifically, there is a finite set of propositions, and the nodes (states) are labelled by the propositions satisfied at the state. We defined above hierarchical machines using edge labels (to conform with language generation), but one can equally well define them using node labels; there is not a significant difference between edge-labelled and node-labelled machines. We say that the hierarchical machine H satisfies a property P if $flat(H)$ does.

The simplest kind of property is a state invariant, i.e., the property that starting from the initial state, the system stays within a specified subset of states. This amounts to a simple reachability problem and can be solved in linear time in the size of the HSM. So-called 'safety' properties that depend on finite computations can be also reduced to a reachability problem.

A much richer class of properties is the class of linear time properties expressed by linear temporal logic (LTL) [Pnu77] or by Buchi automata. These are properties on the infinite computations of the system (i.e. paths of the state machine). The model satisfies a given property if all of its paths starting from the initial state satisfy the property. A Buchi automaton A is like a usual finite string automaton (i.e. a basic FSM); the only difference is that its language is defined to be a set of infinite strings, namely the strings that label infinite paths (starting from the initial state) which go infinitely often through one of the accepting states. We will not define LTL here, but suffice it to say that every LTL formula ϕ can be translated to an equivalent Buchi automaton A of size $O(2^{|\phi|})$ [VW86]; and in the automata theoretic approach (and in tools like SPIN) this is the way that LTL properties are checked, i.e. by conversion to automata. This does not hurt the complexity, because the exponential dependence on the size of the formula is unavoidable (but formulas are typically quite small, so this is tolerable). It is more convenient to use a Buchi automaton to specify the *bad* computations, those that signify an error, rather than the correct computations. Then the model checking problem for a model M amounts to the question of nonemptiness of the intersection $L(M) \cap L(A)$, where the model M is now viewed as generating a set of infinite strings.

The core algorithmic problem in this case is not just reachability, but the following *Cycle Detection problem*: Given a directed graph (a basic FSM) with an initial node and a set of specified 'special' nodes, can the initial node reach a cycle that contains one of the special nodes? The model checking problem for a flat FSM M and a Buchi automaton A is reduced to this cycle detection problem, by simply taking the product of M and A and letting the special nodes be those that correspond to accepting nodes of A. The cycle detection problem can be solved in linear time, either by computing the strongly connected components of the graph, or by an alternative "nested depth-first search" algorithm that is useful for on-the-fly verification [CVWY92].

The model checking problem for hierarchical machines H can be similarly reduced to the cycle detection problem for HSM's: As we mentioned earlier, we can form the product of the HSM H with the automaton A, which is another HSM H'. Then H and A have a nonempty intersection iff $flat(H')$ has a reachable cycle that contains a special node. Finding strong components of $flat(H')$ is not convenient (there is too many of them), but the cycle detection problem can still be solved in linear time in the size of H' by suitable adaptation of the nested depth first search algorithm, see [AY98]. This yields an algorithm for model checking of an HSM H with a Buchi automaton A that is linear in the size of the HSM and quadratic in the size of A (usually the system is much larger than the property, so the dependence on the system size is the more critical measure). Thus, we can verify a herarchical machine without flattening it, i.e., there is essentially no penalty for the exponential succinctness of hierarchy in this case.

The other major brand of temporal logic is branching temporal logic. In the case of basic FSM's, checking a model M for a formula ϕ in branching time logic CTL is easier than LTL: it can be done in time $O(|M||\phi|)$, i.e. linear in both the size of the model and the specification. In the case of hierarchical systems, if we flatten the hierarchical machine and apply the pure FSM algorithm, the complexity of the algorithm will be exponential in the size of the hierarchical model and linear in the size of the formula $|\phi|$. Typically, formulas are small and models are large, so this is not a good trade off. There is an alternative better algorithm, which makes the trade off in the other way, and has complexity exponential in the size of the formula. The dependence on the model size is affected by the number d of exit nodes allowed in the nested machines. Namely, the model checking problem for a HSM H and a CTL formula ϕ can be solved in time $O(|H|2^{|\phi|d})$, and also in polynomial space. Thus, in general we can again avoid flattening the HSM. In the single exit case, the time complexity is linear in the size of the HSM, though in the multiple exit case it grows exponentially with the number of exit states. These dependencies are probably inherent: In the single-exit case, if the formula is part of the input then the problem is PSPACE-complete, so the complexity presumably has to be exponential in either the model size or the formula size, and of course we are better off having the exponential dependence on the formula rather than the model. In the multiple exit case, there is a fixed formula for which the problem is PSPACE-complete (so the model size has to enter exponentially somehow). We summarize in the table of Figure 2 the complexity of the various analysis problems in the checking of properties for HSM's, as compared to FSM's.

	Ordinary FSM's	Hierarchical FSM's
Reachability	$O(\|M\|)$	$O(\|M\|)$
Automata-checking	$O(\|M\| \cdot \|A\|)$	$O(\|M\| \cdot \|A\|^2)$
Ltl model checking	$O(\|M\| \cdot 2^{\|\phi\|})$	$O(\|M\| \cdot 4^{\|\phi\|})$
Ctl model checking	$O(\|M\| \cdot \|\phi\|)$	$O(\|M\| \cdot 2^{\|\phi\|^d})$

Fig. 2. Summary of FSM and HSM model checking

3 Concurrency and Hierarchy

A *concurrent (or communicating) hierarchical state machine*, CHM, combines concurrency and hierarchy in an arbitrarily nested manner. We define it formally here for the single entry -single exit case; the definition for the multiple entry/exit case is similar.

A CHM is defined again inductively. In the base case, every ordinary FSM is a CHM. For the induction step, a CHM H is either a hierarchical combination of previously defined CHM's exactly as before, or it is a concurrent (parallel) combination $M_1 \parallel M_2 \parallel \cdots \parallel M_k$.

The flat basic FSM, $flat(H)$ is defined again inductively. It is the same as before in the basis case and in the case of a hierarchical combination. In the concurrent combination, $flat(H)$ is the product of the FSM's $flat(M_i)$, defined as follows. Assume that $flat(M_i)$ has set of states Q_i, alphabet Σ_i, initial state q_0^i, final state q_f^i, and transition relation E_i. Then the state space of H is $Q_1 \times \cdots \times Q_k$, the alphabet is $\Sigma_1 \cup \cdots \cup \Sigma_k$, the initial state is $(q_0^1, \cdots, q_0^k)$, the final state is $(q_f^1, \cdots, q_f^k)$, and the transition relation has a transition labelled a from state $(u_1, \cdots, u_k)$ to state $(v_1, \cdots, v_k)$ if every component machine M_i, whose alphabet Σ_i contains the symbol a, has a transition from u_i to v_i labelled a. The language $L(H)$ of a CHM H is the language of its corresponding FSM $flat(H)$.

A concurrent hierarchical machine H can be represented by a rooted DAG (directed acyclic graph) corresponding to the way it is built from basic FSM's. The leaves are basic FSM's and each internal node is either labelled by the concurrent combinator $\parallel$, or corresponds to a hierarchical combination and is labelled by a machine N and a mapping μ from the states of N to the children of the node. Since $\parallel$ is an associative operator, we can assume that the children of $\parallel$ nodes are not themselves $\parallel$ nodes (otherwise we can combine them). The size of the CHM M is the size of its DAG representation and all the machines in it. Two important parameters for a CHM is the *depth* m of the DAG (length of the longest path) and the *width* w which is the maximum number of components of a concurrent node of the DAG (i.e. node labelled $\parallel$).

3.1 Expressiveness

Concurrent hierarchical state machines still define of course only regular languages. If a CHM H has width w and depth m, and every FSM in its definition has n states, then $flat(H)$ has $O(n^{w^m})$ states, that is, flattening causes now in general a double exponential blow up.

This is in general unavoidable. In fact concurrency adds two exponentials even compared to hierarchical state machines: There is a family of languages that can be recognized by linear size CHM's but which require HSM's (and hence also basic FSM's) of double exponential size. Furthermore, there is a triple exponential gap in the worst case between CHM's and deterministic hierarchical state machines (and hence also deterministic FSM's). An example of such a language family is $L = \{w_0\#w_1\#\cdots\#w_k||w_i| = 2^n$ for each i and $w_i = w_j$ for some $i, j\ \}$.

The following figure summarizes the expressibility relationships between the different kinds of machines.

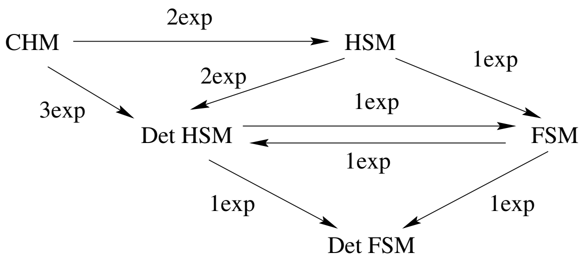

Fig. 3. Summary of succinctness relations

3.2 Operations

Emptiness (Reachability). This cannot be done efficiently any more. Even for a simple concurrent combination (intersection) of basic deterministic FSM's we know that the problem is PSPACE-complete [Koz77], and we saw in the previous section that the same is true for the concurrent combination of just two HSM's. Mixing concurrency with hierarchy adds another exponential: The reachability (and emptiness) problem for CHM's is EXPSPACE-complete.

There are two exponential contributions to this complexity, one due to concurrency and one due to hierarchy. For certain classes of machines, we can avoid one of the exponentials, reducing the complexity to 'only' PSPACE. For example, one case is when all the concurrency is at the top. That is, we have a set of concurrent interacting components, where each component is represented by a hierarchical state machine. Reachability is in this case in PSPACE (and is of course PSPACE-complete). Another case is when the hierarchical construction at a level does not expose the internals of the submachines to higher levels. Note that in the concurrent composition the alphabets of the components are critical. Informally, call a CHM H "well-structured" if the hierarchical nesting always hides the alphabet of the nested machines as far as further concurrent

	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	∈ PSPACE
CHM	EXPSPACE	EXPSPACE	2EXPSPACE	2EXPSPACE	2EXPSPACE

Fig. 4. Summary of complexity of operations

combinations with other machines are concerned, that is, in every concurrent combination $M_1 \parallel \cdots \parallel M_t$ in the definition of H (where M_i are without loss of generality hierachical nodes), only the alphabets of the top level components M_i enter in the composition; i.e., the alphabet of the nested submachines are hidden. Then reachability can be solved in PSPACE and in time $O(kn^w)$, where k is the number of operators (internal nodes of the DAG), w is the width, and n is the maximum number of nodes of a FSM in the definition of H.

Universality. The universality problem turns out to be again harder, and is as bad as it can get: It is complete for double exponential space.

Intersection. The intersection of two CHM's H_1, H_2 is itself another CHM $H_1 \parallel H_2$. Emptiness can be done in EXPSPACE (and is complete).

Equivalence, Inclusion. Since universality is hard, the same is true of equivalence and inclusion, which are also complete for double exponential space.

Finally, model checking for a CHM and a Buchi automaton is, as in the case of simple reachability, complete for exponential space (with respect to the size of the model), i.e., if we nest arbitrarily concurrency and hierarchy we have to pay in general for both the concurrency and the hierarchy. It is worth noting however, that these lower bound constructions that give this pessimistic complexity, take advantage of the fact that one can define succinctly (because of the reuse) systems with exponentially many parallel components; this may be unrealistic for significant sizes, and hence the lower bounds may be unduly pessimistic.

The table of Figure 4 summarizes the complexity results of basic operations for the different kinds of machines.

4 Hierarchical Message Sequence Charts

These represent the extreme case where all the concurrency is at the bottom of the hierarchy. A *basic message sequence chart* (MSC) represents the partial order of the message exchanges in a concurrent execution of a set of processes, see Figure 5. The vertical lines represent the processes, time proceeds down vertically for each process, and the arrows represent the messages. Send and receive events are labelled from some alphabet Σ (for example, the messages sent or received, or any other labels). An MSC M represents a set of linear executions, namely all the linearizations of the partial order, and thus it has a corresponding finite language $L(M)$, the strings that label these linearizations.

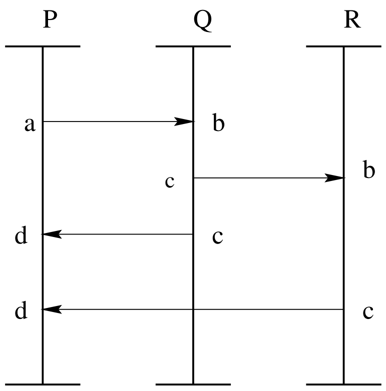

Fig. 5. A basic message sequence chart

Using simple MSC's as basic building blocks, one can combine them in a hierarchical fashion, the same way as with hierarchical state machines. We give for simplicity the definition for the single entry - single exit case (which can be extended to the multiple entries and exits). An *MSC-graph* is a directed graph, with a distinguished initial (entry) node and a final (exit) node, along with a mapping μ that associates with every node a basic MSC (labelled from an alphabet Σ). More generally, we can define inductively a hierarchical MSC graph (HMSC) to be a hierarchical combination of previously defined HMSC's. An HMSC can be represented by a rooted DAG, where each leaf is labelled by a basic MSC's, and each internal node u is labelled by a graph G_u and a mapping μ_u from the nodes of G_u to the children of u. A hierachical MSC H can be of course flattened to a simple MSC-graph $flat(H)$ (at an exponential blow up), the same way that a hierarchical machine is flattened to an FSM.

The executions of a system modeled by an MSC-graph (or HMSC) correspond to the paths of the graph starting at the initial node. There are two interpretations for the concatenation of the MSC's of the different nodes along the path. In the synchronous interpretation, the concatenation of two nodes u, v is viewed as consisting of executing all actions of the MSC of u followed by the actions of the MSC of v; that is, when executing a path, every node represents a block of activity that completes before going on to the next block. In the asynchronous interpretation, the MSC's of the nodes along the path are pasted together, separately process by process, i.e. by identifying for each process the end of its line in one node with the beginning of its line in the next node, thus forming a long MSC spanning all the actions of all the nodes. The synchronous concatenation is probably closer to the way system engineers and designers think when they draw HMSC's partitioning the activities into blocks. The asynchronous interpretation is however closer to what one would get if one was to implement directly the HMSC, i.e. without any other coordination or constraints than what is depicted explicitly in the HMSC.

For each of the interpretations, we can define the language of an HMSC, by considering all the paths starting at the initial node and taking the labelings of all the corresponding linear executions. (We can either define a language of finite strings, if we require that the paths terminate at a final node, or define a language of potentially infinite strings).

In the synchronous interpretation, the language of an HMSC H is regular. A hierarchical state machine H' can be constructed from H by replacing in the representation of the HMSC each basic MSC (ie. leaf of the DAG) by a FSM that recognizes the language of the MSC. Then $L(H) = L(H')$. This is a very benign and convenient case, because we can use directly the results for hierarchical state machines; it means in particular that the graphs and the hierarchy of the HMSC's do not add anything to the complexity of problems like reachability, model checking etc. beyond what's in the simple basic MSC's. If an individual basic MSC is a 'large' one, then the translation to an automaton, as well as answering basic questions about the MSC can be expensive because of the concurrency. Namely, if an MSC M has k processes with l events in each process, then the equivalent FSM can have l^k states, and in fact answering a simple question such as, 'does a given string x belong to $L(M)$' is NP-complete. However, in many cases in hierarchical MSC's most of the size of the HMSC is in the use case graph; the individual MSC's are not large and the number of processes is often small (for example in telephony, processes correspond to entities such as originating and terminating switches and users).

The hierarchy does not introduce any additional complications beyond the basic MSC's under the synchronous interpretation. Thus, for example reachability, or membership, or the model checking problem for a hierarchical HMSC H of size n and a Buchi automaton A of size a is still "only" NP-complete, and can be solved in time $O(a^2nl^k)$, where k and l are as above the number of processes and number of events per process in a basic MSC. Thus, hierarchical MSC's can be analyzed under the synchronous interpretation without paying a penalty for the hierarchy, and if the basic MSC's are small or do not have too much concurrency (which is often the case) then the complexity is low. Furthermore, for properties that are linearization-independent (i.e., if the property is such that one linearization of an ordinary MSC satisfies the property iff any other one does), then the HMSC H can be checked in linear time in its size: substitute each basic MSC in the definition of H by an automaton (just a path) that accepts just one linearization (instead of all of them); the resulting hierarchical machine satisfies the property iff the given hierarchical MSC does (for linearization-independent properties, there is no difference between synchronous and asynchronous interpretation as far as satisfaction of the properties is concerned, so this holds also for the asynchronous case).

In the asynchronous interpretation, the language of an HMSC is in general not regular, and in fact model checking an MSC-graph for a property given by a Buchi automaton is undecidable. There is a syntactic condition that ensures that the language is regular, and which leads to decidability. Given a set S of basic MSC's, we can construct a corresponding *communication graph* $CG(S)$ among the processes: this is a directed graph with the processes as nodes and with an arc $P_i \to P_j$ if process P_i sends a message to process P_j in some MSC of S. Given an MSC-graph G (respectively, a hierarchical HMCS H) we say that it is *bounded* if for every cycle K of G (resp. of $flat(H)$) that is reachable from the initial state, the corresponding communication graph $CG(K)$ consists of one strongly connected component and possibly some isolated nodes. If an HMCS is bounded then its language is regular [AY99]. Conversely, it is shown in [HMNT00] that every regular MSC language that is definable by an MSG graph (or equivalently, is finitely generated) then it can be defined by a bounded MSC graph. Boundedness of an HMSC can be determined without flattening the hierarchy in time $O(nl2^k)$ where n is the size of the HMSC, k is the number of processes and l is the number of events per process in a basic MSC (this mild exponential dependence on k is unavoidable, as the boundedness problem is coNP-complete). We can analyze algorithmically bounded HMSC's by constructing an equivalent FSM. The complexity is higher in this case: model checking a Buchi automaton property is PSPACE-complete for bounded MSC graphs and EXPSPACE-complete for HMSC's (the problem is hard even for a fixed property, a fixed number of processes k, and fixed number of events l per basic MSC).

5 Testing

In testing, we have a design model M (eg., a finite state machine) and the actual system S (the implementation under test) and we wish to generate a suitable set of test cases based on the model, which will be applied to the system S to test its correctness, i.e., whether it conforms with the model. There is extensive theoretical and practical work on test generation for FSM's, see for instance [LY96] for a survey. There is a variety of testing criteria that can be used depending on the extent of testing that can be afforded; it can range from checking sequences that provide certain guarrantees on the relation between M and S (usually at a rather high cost), to a more commonly applied simpler set of coverage criteria, such as covering all the states and transitions of the model.

Suppose that we have a hierarchical finite state machine model and let's consider the most common criterion of 'transition coverage', i.e. finding a set of test sequences (paths from the initial state) that cover all the transitions. There are two interpretations of this requirement: in the first interpretation we seek to cover all transitions of M and its nested submachines in all possible contexts, i.e. if a submachine is reused several times, we want to cover all its transitions in all the uses; this is equivalent to covering all the transitions of the flattened machine $flat(M)$. A covering test set can be computed in time polynomial in the size of the flat FSM using flow techniques to minimize the number of test sequences and the total length of the tests. Such an algorithm has been implemented in Lucent's uBET toolset; after flattening the hierarchical MSC graph, an optimal test set is generated, and for each test path, the basic MSC's of the nodes along the path are combined to form a test scenario in the form of an MSC.

An alternative (and more economical) interpretation of the transition coverage requirement is to cover each transition of the hierarchical state machine and each nested submachine at least once overall, i.e., if a submachine is used several times, we only need to cover each transition in only one occurrence (and different transitions could be covered in different copies). One justification for this could be for example if the same implementation of the submachine is used in the different copies, in which case testing a transition in one instance suggests that it will likely work properly in the other instances as well (although that is not necessarily guarranteed). Of course, the number of tests needed under this weaker requirement is generally much smaller than the full coverage requirement; for example, the number of test cases needed is certainly bounded by the size of the hierarchical machine, as opposed to the potentially exponentially larger size of the flattened machine. Computing an optimal (minimum cardinality) test set for the weaker coverage is much harder than for simple FSM's. For example, even in the 2-level case the problem is at least as hard as Set Cover (hence it is hard to compute or to approximate the smallest test set within a constant factor).

Generally, very little systematic work has been done on test generation for hierarchical state machines.

6 Conclusions

Hierarchy is a useful construct in enhancing the expressive power of finite state machines and facilitating the modeling of large systems. We summarized recent work on the theory of hierarchical finite state machines and related specification mechanisms (HMSC's). We discussed the effect of concurrency and nondeterminism, the classification of their expressive power, and the complexity of various algorithmic problems. The picture that emerged is rather comprehensive. One specific problem that remains open is the equivalence of deterministic HSM's. More broadly, we did not touch on the interaction with variables (extended HSM's). Also, as we mentioned more work remains to be done on the testing of hierarchical FSM's.

Another issue concerns design problems, such as the structuring and modularization of the behavioral aspects of systems, to form a suitable hierarchical model. For example, suppose that we have a legacy system which we want to capture formally in a requirement model like uBET. We have available or can generate a large number of executions of the system to obtain a library of MSC scenarios. How do we go from this collection of MSC's to a more higher level, structured view of the system, in the form of a hierarchical use case graph built from basic MSC's?

Similarly, suppose that we have a test model of a system in the form of a large FSM; for example, we produced recently automatically such a model for a large Lucent enterprise switch from the test platform in use for the testing of the switch [EY00]; the FSM model has hundreds of thousands of states. The model can be used for targeted test generation to meet a variety of criteria. However, the machine is obviously too large for a person to look at, or to manipulate directly, for example to modify as the system evolves, to add new features etc. Can we structure such an FSM and obtain an equivalent (hopefully much smaller) hierarchical FSM? There is an elegant classical decomposition theory of FSM's [HS66] which may be useful in this context.

References

[AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[AKY99] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. *Proc. 26th Intl. Coll. on Automata, Languages and Programming*, 1999.

[AY98] R. Alur, and M. Yannakakis. Model checking of hierarchical state machines. In *Proc. Sixth Symp. on Foundations of Software Engineering*, pp. 175–188. 1998.

[AY99] R. Alur, and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th CONCUR*, Springer Verlag, 1999.

[Apf95] L. Apfelbaum. Automated functional test generation. In *Proc. IEEE Autotestcon Conference*, 1995. See also, www.teradyne.com/prods/sst/product_center/t_main.html.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Conference*, LNCS 818, pages 142–155, 1994.

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

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. *Proc. CONCUR'97*, LNCS 1243, 1997.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. *Proc. CONCUR'92*, LNCS 630, pp. 123-137, 1992.

[CE81] E.M. Clarke and E.A. 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.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I: finite automata. JACM 41(3), 1994.

[EY00] K. Etessami, and M. Yannakakis. From rule-based to automata-based testing. submitted, 2000.

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

[HMNT00] J. G. Henriksen, M. Mukund, K. Narayan Kumar, P. S. Thiagarajan. On message sequence graphs and finitely generated regular MSC languages. *27th Intl. Coll. on Automata, Languages and Programming*, 2000.

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

[HPR97] G.J. Holzmann, D. A. Peled, M. H. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 2(1):86–95, 1997.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[Jac92] I. Jacobson. *Object-oriented software engineering: a use case driven approach.* Addison-Wesley, 1992.

[JM87] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Trans. on Computers*, C-36(8):961–975, 1987.

[Koz77] D. Kozen. Lower bounds for natural proof systems. *Proc. 18th Annual IEEE Symp. on Foundations of Computer Science*, pp. 254-266, 1977.

[Kur94] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.

[LHHR94] N.G. Leveson, M. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process control systems. *IEEE Trans. on Software Engineering*, 20(9), 1994.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84 (8), pp. 1090-1126, 1996.

[McM93] K. McMillan. *Symbolic model checking: An approach to the sate explosion problem.* Kluwer Academic, 1993.

[MSC96] ITU-T Recommendation Z.120. *Message Sequence Chart (MSC).* 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. *Proc. of the 5th Intl. Symp. on Programming*, LNCS 137, pp. 195–220, 1982.

[RBPE91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[RGG96] E. Rudolph, J. Grabowski, P. Graubmann. Tutorial on Message Sequence Charts (MSC'96). *FORTE/PSTV'96*, 1996.

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

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. *Proc. of the First IEEE Symp. on Logic in Computer Science*, pp. 332–344, 1986.