Hierarchical State Machine

Seminar Report

Submitted in partial fulfillment of the degree of
Bachelor of Technology
by

Abhishek Ranjan
(99005033)

under the guidance of
Prof. S. Ramesh

Department of Computer Science and Engineering
Indian Institute of Technology
Mumbai
Acknowledgement

October 18, 2002

I would like to thank Prof. S. Ramesh, for his constant support, encouragement and direction without which this report would not have been possible.

Abhishek Ranjan
99005033
## Contents

1 Introduction 1

2 Description of State Machines 1
   2.1 Communicating Hierarchical State Machine(CHM) 1
      2.1.1 Definition 1
      2.1.2 Semantics 3
   2.2 Kripke structure and HSM 6
      2.2.1 Definition Of Flat Kripke Structure 6
      2.2.2 Definition of Hierarchical Kripke Structure K over P 6
      2.2.3 Expansion Of Hierarchical Kripke Structure 7
   2.3 Unrestricted Hierarchical State Machine 9
   2.4 Recursive State Machine(RSM) 10
      2.4.1 Definition 10
      2.4.2 Semantics of RSM 11

3 Reachability Problem 13
   3.1 Reachability problem in HSM 13
      3.1.1 Reachability Algorithm 14
      3.1.2 Working and Analysis of Algorithm 14
   3.2 Reachability Problem in CHSM 15

4 Cycle Detection Problem 15
   4.1 Cycle Detection Problem in HSM 15
   4.2 Cycle Detection Algorithm for HSM 16

5 Automata Emptiness and Universality Problems 18
   5.1 Emptiness Problem 18
   5.2 Universality Problem 19

6 Succinctness 19

7 Appendix 20

8 Conclusion 21
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DAG representation for CHSM.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Hierarchy in HSM.</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Hierarchy Semantics.</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Kripke Structure.</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Kripke Structure Expansion.</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>Unrestricted HSM.</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>UHSM Semantics.</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Semantics of RSM.</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>HSM representation of a clock.</td>
<td>20</td>
</tr>
</tbody>
</table>
Abstract

This seminar studies the structure and properties of different classes of Hierarchical State Machines (HSM). Structure and behavior of Communicating Hierarchical State Machine, Hierarchical State Machine, Unrestricted Hierarchical Machine and Recursive State Machine have been studied. An introduction to several problems viz. Reachability, Cycle Detection, Automata Emptiness, Universality, LTL Model Checking and Branch Time Model Checking is also included in this seminar. Algorithms for some of the above problems have been analysed and their complexities have been appreciated.

1 Introduction

In any work of verification, we need to describe the system to be verified and then we apply our constraints which formally specify how system should work. Finite State Machines (FSMs) are widely used in the formal description of system and their modelling. Here we focus on an extension of ordinary FSMs which are Hierarchical in nature (called Hierarchical State Machine or HSM), in which vertices of the FSM can be ordinary states or super-states where super-states are states which are FSMs themselves.

Study of HSMs is motivated by the need to represent recursive and modular systems efficiently. For example, in verification of programs, while checking for logical errors, we first give a formal description of the program flow. In case of very large programs, which are divided into several modules and involve recursion, if an ordinary FSM is used, number of states will explode. But HSM could be an efficient tool to model such programs concisely. In this report we will see how these hierarchical structures can be used to represent and understand such complex systems.

2 Description of State Machines

In this section definition and semantics of various hierarchical and concurrent state machines are discussed. Here Concurrency means that the all machines make transitions from one state to another simultaneously on some input symbol. Hierarchy means that control flows from higher level machine to lower level machines through super-states of the higher level machine, in other words we can think of super-states of the higher level state machine as call to lower level machines.

2.1 Communicating Hierarchical State Machine (CHM)

This is the class of machines in which several state machines can run concurrently and hierarchically as well. We describe the structure and meaning of this state machine in this subsection.

2.1.1 Definition

A CHM can be represented as a DAG. Consider the Directed Acyclic Graph in Figure 1. Here, in figure, boxes represent nodes. A terminal node (leaf node) corresponds to an ordinary FSM (e.g. FSM2 in figure). An internal node may correspond to either a product expression or a hierarchical expression. A node for a product expression is labeled with the $\parallel$ operator. Semantics of $\parallel$ is discussed later, so, for the time being assume that in a product expression $\parallel$ operator combines the components in some special manner, and the children of the node corresponding to product expression are the components of the product.
expression. For instance, in Figure 1 consider $M$, it is $M_1 || M_2$, so its immediate children are $M_1$ and $M_2$, which themselves are CHMs.

A node for a hierarchical expression has a top level FSM associated with it, and the edges connect the states of the top level FSM with the other nodes. For instance, consider top level FSM in Box labeled $M_1$ with dotted boundary, it has 3 states $s_0$, $s_1$, $s_2$. $s_0$ is mapped to a CHM $M_3 || M_4$, $s_1$ and $s_2$, both are mapped to the same ordinary FSM ($FSM_1$).

After this illustration, CHM is being defined formally. CHMs are inductively defined as follows:

- **Base Case:** An FSM($Q$, $\Sigma$, $q^I$, $q^F$, $T$) is a CHM. Here Q is the set of states, $\Sigma$ is the alphabet, $q^I$ is the starting state, $q^F$ is the final state, and $T$ is the transition function.

- **Concurrency:** If $M_1 \ldots M_k$ are CHMs then $M_1 || M_2 \ldots || M_k$ is a CHM. (semantics of || will be defined later).

- **Hierarchy:** let,
  - $\xi$ = finite set of CHMs.
  - $N$ = FSM ($Q$, $\Sigma$, $q^I$, $q^F$, $T$).
  - $\mu : Q \rightarrow \xi$.
  
  the tuple $(N, \xi, \mu)$ is a CHM
Note that Base case FSM and FSM N in Hierarchy, both, have single entry and single exit nodes. While discussing the properties of HSM, we use the term “size of CHM”, which is the size of the DAG representation of the CHM. Two other important parameters of this DAG representation are described below:

\textbf{Width:} It is the maximum number of components in the product nodes.

\textbf{Depth:} It is the length of the longest path in the DAG.

In the machine shown in Figure 1 width is 3, depth is 3 and number of nodes is 11 (It should be noted here that FSM1, FSM2. have been counted once, though they appear in more than one contexts.)

\textbf{2.1.2 Semantics}

Here we consider how CHMs actually run. The behaviour of CHM is formally defined through its semantics, which is defined by mapping each CHM M to an FSM $[M]$. The FSM $[M]$ which is constructed from concurrent machine (say, $M_1||M_2$), has behaviour similar to that of a product machine (i.e $[M_1] \times [M_2]$). So, suppose $M = M_1||M_2$ and for the sake of simplicity assume that $M_1, M_2$ are ordinary FSMs, then behaviour of $M$ would be as if $M_1$ and $M_2$ were running concurrently. Both machines start running when an input symbol is given to their starting states. It should be noted here that both the machines get the same symbol as the input. On absorbing a symbol from the alphabet (which is the union of alphabets of $M_1$ and $M_2$), both machine make transition simultaneously according to their own transition rules. Thus machines run concurrently and a string is accepted by $M$, when it’s accepted by both $M_1$ and $M_2$.

Similarly when a machine is constructed hierarchically, then, Intuitively, it behaves as a program with procedure calls, and symbols of one machine are like local variables of that procedure. The machine starts running when the initial state of the top level FSM absorbs a symbol. Any input to the top level symbol causes the control to jump from one machine to another. While the symbols which belong to lower level machine’s alphabet cause transition in that machine itself. This has been illustrated through Figure 2 and Figure 3. In Figure 3, states are shown as pairs $(q, w)$, where first element $q$ is a state of $M_1$ and gives the context in which $w$ is being used and second element $w$ is a state of $M_2$. This second symbol $w$ actually shows the state for which transition is occurring. For a transition $(q_1, w_1) \xrightarrow{\sigma} (q_2, w_2)$, if $q_1 = q_2$ then it means that the transition occurs inside the same machine $M_2$ and this machine $M_2$ is one which is mapped to either of $q_1$ or $q_2$, and $\sigma$ is in the alphabet of $M_2$. Thus we see that for a symbol in the alphabet of the lower level machine (here $M_2$) causes transition in that machine only. But, if $q_1 \neq q_2$ then it means that a transition from one machine, which is mapped to $q_1$, to another machine, which is mapped to $q_2$, occurs and $\sigma$ belongs to $M_1$; and thus the symbol belonging to the upper level machine (here $M_1$) causes transition in the upper level machine (i.e. from state $q_1$ to $q_2$ of the upper level machine.)

Now in Figure 2, when $q^l$ gets symbol ‘a’, a transition occurs from machine mapped to $q^l$ to that mapped to $q^F$. Control leaves the previous machine through its final state and enters the new machine through its initial state. In figure, both states are mapped to the same machine so in FSM $[M]$ in semantic domain (Figure 3), a transition occurs $(q^l, p^1) \rightarrow (q^l, p^0)$. And for a symbol $c$ which belongs to the lower level machine only, transition occur inside the machine itself. For instance, $(q^F, p^0) \rightarrow (q^F, p^1)$. 

3
Figure 2: Hierarchy in HSM.

Figure 3: Hierarchy Semantics.
Thus through hierarchy all machines work in hierarchical order. Higher level machine gives control to lower level machines for inputs belonging to their alphabets, and for symbols belonging to higher level machine, it keeps the control to itself and performs the transition. Now we formally give the rules for the construction of the FSM from CHM:

- **Base Case:** If $M$ is an FSM then $[M] = M$.

- **Concurrency:** If $M = M_1||M_2||...||M_k$ then $[M] = [M_1][M_2]...[M_k]$. This product is defined as:

  $$[M_i] = (Q_i, \Sigma_i, q_i^0, q_i^f, T_i)$$

  - State space of $[M]$ is $Q_1 \times Q_2 \times ... \times Q_k$.

  - The alphabet $\Sigma$ of $[M]$ is $\Sigma_1 \cup \Sigma_2 \cup ... \cup \Sigma_k$.

  - The initial state of $[M]$ is $< q_1^0, q_2^0, ..., q_k^0 >$.

  - The final state of $[M]$ is $< q_1^f, q_2^f, ..., q_k^f >$.

  - For a symbol $\sigma \in \Sigma$, $[M]$ has $\sigma$ labeled transition from $< q_1, q_2, ..., q_k >$ to $< w_1, w_2, ..., w_k >$ if for every $i$ such that $\sigma \in \Sigma_i$, that FSM $[M_i]$ has a transition $< q_i, \sigma, w_i >$, and for every $i$ such that $\sigma \in \Sigma_i$, $w_i = q_i$. This transition rule just formalizes the notion that if $M_1$, $M_2$ form a concurrent system then, a symbol, common to both, causes a transition in both, while a symbol which is only in one of them (say in $M_1$) causes a transition only in that machine (i.e. in $M_1$), and the other concurrently running machine (i.e. $M_2$ here) just stays in its previous state.

- **Hierarchy:** If $M = (N, \xi, \mu)$, $N = (Q, \Sigma, q^l, q^f, T)$ then:

  - Set of states of $[M] = \{(q, w) \mid q \in Q, w$ is a state of the FSM $[\mu(q)] \}$.

  - A symbol $\sigma$ belongs to the alphabet of $[M]$ if: $\sigma \in \Sigma$ or $\sigma$ belongs to alphabet of $[\mu(q)]$ for some $q \in Q$.

  - The pair $(q^l$, initial state of $[\mu(q^l)])$ is the initial state of $[M]$.

  - The pair $(q^f$, final state of $[\mu(q^f)])$ is the final state of $[M]$.

  - For transition:

    * For a transition of $< q, \sigma, q^0 >$ in $N$, $[M]$ has the transition from the pair $(q$, final state of $[\mu(q)])$ to the pair $(q^0$, initial state of $[\mu(q^0)])$.

      Note that, in this case control leaves one machine and enters another machine.

    * For $q \in Q$, if $< w, \sigma, w^0 >$ is a transition of $[\mu(q)]$ then $< (q, w), \sigma, (q, w^0) >$ is a transition in $[M]$.

A hierarchical structure without concurrency is shown in Figure 2 and the corresponding structure in semantic domain is shown in Figure 3.
It is interesting that hierarchy can cause non-determinism in the corresponding FSM in semantic domain, even though the components of the hierarchical structure are deterministic. While, in concurrency, if components are deterministic then FSM in semantic domain can not be non-deterministic. For instance, consider the HSM $M_1$ without concurrency in Figure 3. Now in FSM $[M_1]$, corresponding to $M_1$, in state $(q^I, p^I)$, on input symbol $a$, there is a transition from $(q^I, p^I)$ to $(q^f, p^0)$ and also there will be a jump from structure $M_2$ to structure $M_1$ (i.e. a transition from $(q^I, p^I)$ to $(q^f, p^0)$). So nondeterminism is introduced.

**Language of HSM:** The language of HSM $M$, $L(M) = L([M])$. For example the language of the FSM in Figure 2 is the language of the FSM corresponding to it, which is shown in Figure 3. It should be noted here that language of HSM is Regular Language.

### 2.2 Kripke structure and HSM

A CHM without concurrency is called Hierarchical State Machine. Hierarchical States Machines can also be represented in terms of Kripke Structure. Due to prevalent use of Kripke Structure in Model Checking literature, various results related to the Hierarchical State Machine are presented here in terms of Hierarchical Kripke Structure (Model Checking has been discussed briefly in Appendix).

#### 2.2.1 Definition Of Flat Kripke Structure

A Flat Kripke Structure $M$ over set of atomic propositions $P$ is defined as consisting of following:

- $W$: A finite set of States
- $i_0 \in W$: Initial State
- $R$: A set of transitions, each transition is a pair of states
- $L$: A labeling function $L: W \rightarrow 2^P$

#### 2.2.2 Definition of Hierarchical Kripke Structure $K$ over $P$

Intuitively a Hierarchical kripke Structure is quite similar to CHM (without concurrency), described in the last section. Here we have boxes as references to other structures, and each node satisfy a set of atomic propositions.

A Hierarchical kripke structure over a set of atomic propositions $P$ is a tuple $K = \langle K_1, K_2, \ldots, K_n \rangle$ Each of $K_i$ has following components:

- $N_i$: A finite set of Nodes .
- $B_i$: A finite set of Boxes .
- $i_0 \in N_i$: Single initial node
- $O_i \subseteq N_i$: Set of exit nodes
- $X_i: N_i \rightarrow 2^P$. This is a map which assigns a set of propositions to each node in $N_i$.
- $Y_i: B_i \rightarrow \{i + 1, i + 2, \ldots, n\}$. A map which assigns an index to each box. This map causes the hierarchy in the structure.
Figure 4: Kripke Structure.

Figure 5: Kripke Structure Expansion.

- $E_i$: Edge relation $E_i$ is set of pairs $(u, v)$ satisfying of the following conditions on $u$ and $v$.
  - $u \in N_i$ or $u = (w_1, w_2)$ such that $(w_1 \in B_i \land w_2 \in O_{V_i[w_1])}$. In this condition, pair form of $u$ contains the name of the box $w_1$, which causes the control to enter a new component (say $K_i$) and $w_2$ is the name of the node of $K_i$ through which controls exits from $K_i$.
  - $v \in B_i$ or $N_i$

2.2.3 Expansion Of Hierarchical Kripke Structure

With each Hierarchical Kripke Structure an expanded Flat Kripke Structure can be associated. In this expanded structure each box is recursively substituted by the Kripke structure indexed by that box.

Before going to formally define the expansion of this hierarchical kripke structure we illustrate the process through a very practical example. In fact, hierarchical kripke structure quite naturally models several practical situations. Consider Figure 4, where we have a structure which models a protocol over a network where one machine attempts to connect
to another machine. Machine starts from \textit{start} state, and then goes to box \textit{try1}. \textit{try1} itself is pointer to a sequence of steps which can be taken as a module. So, instead of continuing the state transitions from \textit{start} in the same machine a different machine is used to introduce modularity, which causes reusability of a component in the representation. And now control is one level below in hierarchy. Here a usual sequence of steps are followed, like \textit{send} a request, \textit{wait} for acknowledgment (\textit{ack}), if \textit{ack} is received then connection is established but still process might fail, so two transitions are taken from here. In case it’s \textit{ok}, control reaches one level above and \textit{success} is declared. In case a \textit{fail} is observed, system goes for another try, i.e. \textit{try2}. In second try also, same steps will be followed so, the same module is reused, which is Hierarchical Kripke Structure’s advantage over ordinary FSM. This time if a \textit{fail} is observed then process is aborted. Otherwise process succeeds.

Thus we observe here that a box takes the control to the initial state of the structure it is pointing to, then the lower level machine runs. And the box is exited through the exit node of the structure pointed by that box. These features will be reflected in the formal definition through a pair structure of states.

Having discussed the intuition behind the expansion now a formal description of the expansion is given below. Each step is followed by an example to illustrate the method.

For each structure $K_i$ in $K = < K_1, K_2 \ldots K_n >$, there corresponds a kripke structure $K_i^F$. The structure $K_i^F = < W_i, \text {init}_i, R_i, L_i >$ is a flat kripke structure over $P$, and is called \textit{expanded structure of} $K_i$. $K_i^F$ is also denoted as $K^F$ and is the expanded structure of $K$.

1. The set $W_i$ of states of $K_i^F$ is defined inductively:

- Every node of $K_i$ belongs to $W_i$.
- States labeled \textit{start}, \textit{abort}, \textit{success} are examples of this type in the Figure 5
- If $u$ is a box of $K_i$ with $Y_i(u) = j$ and $v$ is a node of $K_j^F$, then $(u, v)$ belongs to $W_i$.
- States labeled (\textit{try1}, \textit{send}), (\textit{try1}, \textit{wait}), (\textit{try2}, \textit{send}) (\textit{try2}, \textit{wait}) etc. are examples of this type in the Figure 5

2. The set $R_i$ of transitions of $K_i^F$ is defined inductively:

- For $(u, v) \in E_i$, if the sink $v$ is a node then $(u, v) \in R_i$, for instance $s1$, $s2$ in Figure 4 correspond to $t1$, $t2$ in expanded structure in Figure 5. And if $v$ is a box with $Y_i(v) = j$ then $(u, (v, \text {init}_j)) \in R_i$, for instance, $s3$ in Figure 4 corresponds to $t3$ in Figure 5.

- If $w$ is a box of $K_i$ with $Y_i(w) = j$, and $(u, v)$ is a transition of $K_j^F$, then $((u, v), (w, v))$ belongs to $R_i$. For example, transition $t4$, $t6$ in Figure 5 correspond to transition $s4$ in Figure 4, while $t5$, $t7$ in Figure 5 correspond to $s5$ in Figure 4.

3. The labeling function $L_i : W_i \rightarrow 2^p$ of $K_i^F$ is defined inductively as :

- if $w$ is a node of $K_i$, then $L_i(w) = X_i(w)$
- if $w = (u, v)$, where $u$ is a box of $K_i$ with $Y_i(u) = j$, then $L_i(w)$ equals $L_j(v)$.

Consider the Hierarchical Kripke Structure as shown in the Figure 4. This Hierarchical Kripke Structure is expanded to the structure shown in Figure 5.
2.3 Unrestricted Hierarchical State Machine

Hierarchical State Machine described above using *Hierarchical kripke structure* has one restriction that if \( K = < K_1, K_2, \ldots, K_n > \) is the *Hierarchical kripke structure* then any \( K_i \) can call (from its boxes) structures with indices greater than \( i \) i.e. their boxes can be mapped to the set \( K_{i+1}, \ldots, K_n \). If we remove this restriction then the resulting HSM becomes Unrestricted HSM. In this machine components can call each other arbitrarily. Modified formal definition follows:

An Unrestricted HSM over a set \( P \) of atomic propositions, is a tuple of component structures \( K = < K_1, K_2, \ldots, K_n > \) Each of \( K_i \) has following components

- \( N_i \) : A finite set of Nodes .
- \( B_i \) : A finite set of Boxes.
- A non empty \( I_i \subseteq N_i \) : Called the *entry-nodes* of \( N_i \)
- \( O_i \subseteq N_i \) : Called the *exit-nodes* of \( N_i \).
- \( X_i : N_i \to 2^P \) : A labeling function that labels each node to a subset of \( P \).
- \( Y_i : B_i \to \{1, 2, 3 \ldots i + 1, i + 2, \ldots n\} \) : An indexing function that maps each box of \( M_i \) to the index of some structure \( M_j \). Note the difference from the HSM.
- A set \( C_i \) of pairs of the form \((b, e)\) where \( b \) is a box in \( B_i \) and \( e \) is an entry-node of \( M_j \) with \( j = Y_i(b) \), called *call-nodes* of \( B_i \).
- A set \( R_i \) of pairs of the form \((b, x)\) where \( b \) is a box in \( B_i \) and \( x \) is an exit-node of \( M_j \) with \( j = Y_i(b) \), called *return-nodes* of \( B_i \).
- \( E_i \) : Edge relation \( E_i \) is set of pairs \((u, v)\) such that:
  - Either \( u \in N_i \) or \( u \in R_i \) and
  - \( v \in N_i \) or \( v \in C_i \)

The transitions in UHSM are quite similar to those of HSM. Here two new terms *call-nodes* and *return-nodes* are introduced to show that transition occurs from return-nodes to call-nodes. Intuitively it's like transfer of control from one component of UHSM (say \( K_i \)) to another (say \( K_j \)), where transition can occur only if first component \( K_i \) is exited via an exit-node of it, and second component is entered via an entry-node. It can also be compared with procedure calling in programs (see Section 2.4.2 for a similar comparison for RSM).

Semantics of UHSM is similar to that of HSM. The only difference is due to the unrestricted calls in the case of UHSM. An Unrestricted Hierarchical State Machine is shown in Figure 6 and the corresponding structure in semantic domain is shown in Figure 7. In the Figure 6, if box \( B_i \) calls component structure \( M_j \) then, in expanded structure (Figure 7) \( B_i \) is written above the nodes of \( M_j \) to denote the context in which that node is being used. In case of HSM, same thing was shown by denoting the nodes as the pair \((B_i, \text{node of } M_j)\). In the first level call \( B_1 \) calls \( M_2 \) and so states *send, wait, ack* etc are used in context of \( B_1 \). Inside \( M_2, B_2 \) again calls \( M_2 \), so, at this second level call, context of the call becomes \( B_1 B_2 \) and similarly at third level, context becomes \( B_1 B_2 B_2 \) and so on.

9
Figure 6: Unrestricted HSM.

In Unrestricted HSM, component structures can be called in arbitrary order, and this can possibly make the expansion infinite. For example, in UHSM in Figure 6 the expanded UHSM becomes infinite since box $B_2$ calls $M_2$ and $M_2$ itself contains the box $B_2$. But it should be noted here that context in which component $M_2$ is called changes (denoted by $B_1$, $B_1B_2$ etc.)

2.4 Recursive State Machine (RSM)

Recursive state machine is a powerful way to model programs in which recursive calls to procedures are involved. Intuitively, description of RSM can be thought of as describing a program. In this subsection RSM is defined formally, and then an analogy is illustrated between the semantics of RSM and a control flow in recursive programs.

2.4.1 Definition

A recursive state machine $A$ over a finite alphabet $\Sigma$ consists of a set of component machines $A_1, A_2, \ldots, A_k$. Each component machine $A_i$ has a set of $nodes$, $boxes$, $entry$ and $exit$ nodes for interfacing. So, $A$ is given by tuple $< A_1, A_2, \ldots, A_k >$ and each component machine $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$

- $N_i$ : A set of Nodes.
- $B_i$ : A disjoint set of Boxes.
- $En_i \subseteq N_i$ : Called the $entry$-nodes of $N_i$
- $Ex_i \subseteq N_i$ : Called the $exit$-nodes of $N_i$.
- $Y_i : B_i \rightarrow \{1, 2, 3 \ldots k\}$ : An indexing function that maps each box of $A_i$ to the index of some structure $A_j$.
\( \delta_i \): A set of transition relations of the form \((u, a, v)\) such that:

- source \(u\) is either a node of \(N_i\) or a pair \((b, x)\), where \(b\) is a box in \(B_i\) and \(x\) is an exit node in \(Ex_i\) for \(j = Y_i(b)\);
- The label \(a\) is either \(\epsilon\), a silent transition, or in \(\Sigma\) and
- The destination \(v\) is either a node in \(N_i\) or a pair \((b, e)\), where \(b\) is a box in \(B_i\) and \(e\) is an entry node in \(En_j\) for \(j = Y_i(b)\).

The entry and exit nodes of the RSMs are interface of the component by which it can communicate with other components. Intuitively RSMs can be taken as procedures, an edge entering a box is like invoking the procedure corresponding to the box and an edge leaving a box is like returning from the procedure.

### 2.4.2 Semantics of RSM

To define the execution of an RSM first a global state (or a general state) has to be defined. A global state is of tuple form \(<b_1, b_2, \ldots, b_r, u>\), where \(b_1, b_2 \ldots\) are boxes and \(u\) is a node. We consider only those tuples for which if \(b_i\) is mapped to \(A_j\) then \(b_{i+1}\) is in \(A_j\). So intuitively, boxes in the tuple form a recursive call sequence, and each state stores the context in which they are being executed (such states are called well-formed states). And the node \(u\) is in the component mapped to \(b_r\). Intuitively this node represents the current statement to be executed.

Now we define a global transition relation \(\delta\). Intuitively, Global transition relation describes the possible cases which might arise during the execution of a program. Suppose \(s = <b_1, b_2, \ldots, b_r, u>\) be a state with \(u \in N_j\) and \(b_r \in B_m\). Then \(<s, a, s'> \in \delta\) if one of the following holds:
Figure 8: Semantics of RSM.
1. \((u, a, u^0) \in \delta_j\) for a node \(u^0\) of \(A_j\), and \(s^0 = \langle b_1, b_2, \ldots, b_r, u^0 \rangle\).

In this case, the control stays within within the component \(A_j\). Only the last component of the tuple changes in \(s^0\) corresponding to a transition in \(A_j\) (as shown in the Figure 8: case 1). Intuitively it’s similar to executing statements in the same procedure.

2. \((u, a, (b^0, e)) \in \delta_j\) for a box \(b^0\) of \(A_j\), and \(s^0 = \langle b_1, b_2, \ldots, b_r, b^0, e \rangle\).

In this case a new component \(A_k\) is entered via a box of \(A_j\), and a new box of \(A_j\) is also included in \(s^0\) (as shown in Figure 8: case 2). Intuitively it’s similar to yet another procedure call from the current recursion level, so this is the only case in which procedure depth or number of boxes increases.

3. \(u\) is an exit node of \(A_j\), \((b_r, u), a, u^0) \in \delta_m\) for a node \(u^0\) of \(A_m\), and \(s^0 = \langle b_1, \ldots, b_r, a, u^0 \rangle\).

In this case the control exits \(A_j\) through exit node \(e\) and returns back to \(A_m\) at node \(u^0\) (as shown in Figure 8: case 3). Intuitively, in this case procedure \(A_j\) returns and control goes to next statement labeled \(u^0\).

4. \(u\) is an exit node of \(A_j\), \((b_r, u), a, (b^0, e)) \in \delta_m\) for a box \(b^0\) of \(A_m\), and \(s^0 = \langle b_1, \ldots, b_r, a, b^0, e \rangle\). And in this final case (as shown in Figure 8: case 4) the control exits \(A_j\) and enters a new component \(A_k\) via a box \(b^0\) of \(A_m\). Intuitively here control returns from procedure \(A_j\) but again enters into a new procedure \(A_k\) via another procedure call through \(b^0\).

After defining and illustrating various state machines we now look at the properties of these machines. In the subsequent sections several problems are taken one by one and discussed for different machines.

3 Reachability Problem

Informally, Reachability problem for a finite state machine is to determine whether final state is reachable from the initial state or not. So, this problem can also be viewed as checking if the language of the machine is empty. Solving Reachability problem for a state machine is quite important because several problems can be reduced to it. For example, suppose we formally define a system \(S\) by a state machine \(M\). Now to check whether \(S\) has any undesirable property or not, we mark as undesirable those states of \(N\) which correspond to undesirable property. Now the problem to be solved is to decide if any state, marked undesirable, can be reached from the initial state. So, in various practical situations, where we model a system using HSM, we need to solve the Reachability problem for HSM.

In this section Reachability problem for different kinds of hierarchical state machines has been discussed separately. Reachability problem for simplest Hierarchical Structure (i.e. Hierarchical state machines) has been discussed in details first. Later it is analysed for some other hierarchical structures.

3.1 Reachability problem in HSM

For a Hierarchical structure \(K\) a state \(v\) is reachable from \(u\) if there is a path from state \(u\) to state \(v\) in the expanded structure \(K^F\). The input to the Reachability problem is a Hierarchical structure \(K\) and a set \(T\) s.t. \(T \subseteq \bigcup_i N_i\) called target-region. And the problem is to determine whether or not some state whose last component is in the target-region \(T\) is reachable from top level entry-node \(m_i\).
3.1.1 Reachability Algorithm

The Reachability algorithm is shown below. After giving algorithm, working of algorithm has been discussed. It has been assumed that sets of nodes and boxes of all structures are disjoint.

---

**Algorithm 1**

**Input**: A Hierarchical structure $K$ and a target region $T$.

**Output**: The answer to the Reachability problem $(K, T)$.

**visited**: set of nodes and boxes, initially empty

```plaintext
procedure DFS(u)
    if $u \in T$ then
        print("Target Reachable")
        Break
    end if
    visited := visited $\cup \{u\}$
    if $u \in N$ then
        for all $(u, v)$ such that $(u, v) \in E$ do
            if $v \notin visited$ then
                DFS(v)
            end if
        end for
    else
        $i := Y(u)$
        if $in_i \notin visited$ then
            DFS(in$_i$
        end if
        for all $((u, v), w)$ such that $((u, v), w) \in E$ do
            if $v \in visited$ and $w \notin visited$ then
                DFS(w)
            end if
        end for
    end if
end DFS
```

DFS(in$_1$

print("Target is not reachable")

---

3.1.2 Working and Analysis of Algorithm

It has been assumed that given a node $u$ the test $u \in T$ can be performed in time $O(1)$. Now the working of the algorithm has been described here.

The algorithm performs a depth first search using the global data structure visited to store the nodes and boxes. While processing the box $b$ with $Y(b) = i$, the algorithm checks if the entry-node $in_i$ of the $i$th structure was visited before (This is done because boxes can be considered to be a reference to other structure and when we encounter a box, it means we must go down the hierarchy to check for reachability). The first time the algorithm visits some box $b$ with index $i$, it searches the $K_i$ by invoking the $DFS(in_i)$. At the end of
this search, the set of exit-nodes of $K_i$ that reachable form $in_i$ will be stored in the data structure visited. If the algorithm visits subsequently some other box $c$ with the index $i$, it does not search $K_i$ again, but simply uses the information stored in visited to continue the search. It should be noted that, in the case of boxes we need to do two things, first is to go down the hierarchy and the second is to continue search for the neighbours of box on the same level. The algorithm takes care of these two things (see the loop in the part of the algorithm which deals with boxes).

It can be observed that Algorithm 1 invokes, for every $u \in N \cup B$, $DFS(u)$ at most once. The cost of processing a node or a box $u$ equals the number of edges with source $u$. Consequently, the running time of the algorithm is linear in the size of the input structure.

3.2 Reachability Problem in CHSM

The Reachability Problem for ordinary FSMS is in NL. On introducing hierarchy in FSMS, the problem for HSMs can be solved in linear time, and is $P$-complete. On the other hand introducing concurrency in FSMS is expensive: Reachability problem for product of FSMS is PSPACE-complete[2]. In CHMs, the concurrency and hierarchy operators are arbitrarily nested, and the product components can synchronize with each other at different level of hierarchy. These features make reachability problem significantly difficult to solve.

Now some results are quoted to analyse the effect of introducing both hierarchy and concurrency.

**Proposition 1.** Reachability problem for $M_1||M_2$, where $M_1$ and $M_2$ are HSMs, is PSPACE-complete.[2]

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

**Theorem 1.** Reachability problem for CHMs is EXPSPACE-complete.

(for terms depth and width refer to sec. 2.1.1 ).
Proposition 2 implies that reachability problem for a CHM can be solved in time $O(n^dm)$, that is doubly exponential in the depth in the worst case.

4 Cycle Detection Problem

Cycle detection problem is to determine if there exists a cycle in the structure and which is reachable from the source or entry-node. Cycle detection problem comes into picture in several situations like in detecting a loop in a program or in determining if a system can get stuck in some cycle or deadlock. Also, problem of Model checking and problem of language emptiness, where strings are of infinite length, can be reduced to this problem (Model checking is briefly discussed in Appendix : Section 9). In this section we define the problem and then discuss the algorithm for this important problem of cycle detection.

4.1 Cycle Detection Problem in HSM

As in Reachability problem, the input to the cycle-detection problem consists of a hierarchical Kripke Structure $K$, and a target region $T \subseteq N$. Given $(K,T)$, the cycle-detection problem is to determine whether there exists a state $u$ whose last component is in the target region $T$ such that $u$ is reachable from the entry-node $in_1$ and $u$ is reachable from itself.
4.2 Cycle Detection Algorithm for HSM

The Cycle Detection Algorithm involves two searches, a primary and a secondary. Global data-structure \( \text{visited}_p \) is used to store the nodes visited in the primary search and \( \text{visited}_s \) is used to store nodes encountered during secondary search. Algorithm starts running by calling \( DFS_p \) with the entry node of the topmost hierarchical structure. Algorithm takes two cases of nodes and boxes separately.

First case in \( DFS_p \) is when a node is to be checked. Procedure \( DFS_p \) checks if the node is reachable from source node. Once it encounters a node it places it in \( \text{visited}_p \) Data-structure, implying that it has been encountered and reachable. It also pushes this node on a global \( \text{Stack} \). This \( \text{Stack} \) is later used in secondary search. If the node is one of target-nodes, then it calls Procedure \( DFS_s \), which checks whether the node is reachable from itself (i.e. is there a cycle containing that node). Secondary procedure does this by using the global \( \text{stack} \). Primary search pushes nodes onto \( \text{Stack} \) after encountering it. Suppose secondary search, while processing node \( n \) encounters a node \( m \) which is also on the \( \text{Stack} \), it means that there is a path from \( n \) to \( m \) and another disjoint path from \( m \) to \( n \), so a cycle is found containing \( n \) and \( n \in T \), so “Cycle found” is declared and algorithm terminates.

Second case is when a box is being checked. Since boxes are references to the other hierarchical structures, so the structure, referenced by the box, must be first checked for cycle. This is done by calling \( DFS_p \) with \( in_i \). After search backtracks from the structure referenced by the box, neighbours of the box are checked by following the edges of the form \((u, v), w\). (refer to section 2.2.2 for recalling the type of edges HSM can have). If, exit node \( v \) is reachable and the neighbouring structure \( w \) hasn’t yet been checked, then \( DFS_p \) has been called with this structure. Now if node \( v \) has been checked for cycle, and \( w \) is still on the stack, it means the control has reached \( w \) again, so “Cycle Found” has been declared. If \( w \) is not on the stack and it hasn’t yet been checked for cycle then \( DFS_s \) has been called with \( w \) to check for a cycle.

Now complete algorithm is given below

---

**Algorithm 2**

**Input**: A Hierarchical structure \( K \) and a target region \( T \).

**Output**: The answer to the Cycle Detection problem \((K, T)\).

\( \text{visited}_p, \text{visited}_s \) : set of nodes and boxes, initially empty

\( \text{Stack} \) : Stack of nodes and boxes, initially empty

**procedure** \( DFS_p(u) \)

Push(\( u \), stack)

\( \text{visited}_p := \text{visited}_p \cup \{u\} \)

if \( u \in N \) then

for all \((u, v)\) such that \((u, v) \in E\) do

if \( v \notin \text{visited}_p \) then

\( DFS_p(v) \)

end if

end for

if \( u \in T \) and \( u \notin \text{visited}_s \) then

\( DFS_p(v) \)

end if
else
  \( i := Y(u) \);
  if \( in_i \notin visited_P \) then
    \( DFS_P(in_i) \)
  end if
  for all \( ((u, v), w) \) such that \( ((u, v), w) \in E \) do
    if \( v \in visited_P \) and \( w \notin visited_P \) then
      \( DFS_P(w) \)
    end if
    if \( v \in visited_S \) then
      if \( w \in Stack \) then
        print("Cycle found");
        break;
      end if
      if \( w \notin visited_S \) then
        \( DFS_S(w) \)
      end if
    end if
  end for
end if
end \( DFS_P \).

procedure \( DFS_S(u) \)
\( visited_S := visited_S \cup \{u\} \)
if \( u \in N \) then
  for all \( (u, v) \) such that \( (u, v) \in E \) do
    if \( v \in Stack \) then
      print("Cycle found"); break;
    end if
    if \( v \notin visited_S \) \( DFS_S(v) \) then
      end if
  end for
else
  \( i := Y(u) \)
  for all \( ((u, v), w) \) such that \( ((u, v), w) \in E \) do
    if \( v \in visited_P \) then
      if \( w \in Stack \) then
        print("Cycle found"); break;
      end if
      if \( w \notin visited_S \) \( DFS_S(w) \) then
        end if
    end if
  end for
end if
end \( DFS_S \).

To get the complexity of the above algorithm, we observe that for every \( u \in N \cup B \), \( DFS_P(u) \) is invoked at most once and \( DFS_S(u) \) is invoked at most once, so this gives linear time-complexity in the size of the input structure.
5 Automata Emptiness and Universality Problems

These two problems involve the language or trace corresponding to the automaton. First is to determine whether the language is empty or not and the other one is to determine whether the complement of the language is empty or not.

Automata emptiness problem is used in detecting bad behaviour of models. Suppose we construct an FSM $F$ which accepts all undesirable strings, i.e. this automata captures all the undesirable behaviours of the system $S$. Now we create 2 automata, one is $M$ to represent the system $S$, and the other is $A$ which accepts the strings which are accepted by both $F$, and $M$. So if $L(A)$ has some strings then it means that there are some undesirable behaviours in our model $M$ of the system $S$. So, our system is error-free if $L(A)$ is empty, and thus we can reduce the problem of verifying the model of the system to the problem of Automata emptiness.

In this section these problems have been defined for HSMs and approach to solve these problems are given. Details of the methods[3][2] have been omitted in this report.

5.1 Emptiness Problem

The Automata Emptiness problem for an automata $A$ is to find out whether the Language of the automata $L(A)$ is empty or not. The emptiness problem for HSMs can be solved in $P$ by using reachability algorithm.

In this report automata-emptiness has also been considered in terms of trace of HSM, defined using kripke Structure $M = (W, in, R, L)$ over proposition set $P$. Automata emptiness problem for this structure is to determine whether the trace corresponding to $M$ is empty or not. A trajectory is an infinite sequence of states from $W$, $w_0w_1w_2....$. And trace corresponding to this trajectory is the infinite sequence $L(w_0)L(w_1)L(w_2)....$ over $2^P$. Trace is obtained by replacing each state in the sequence by its label in $2^P$. The language $L(M)$ consists of all the traces corresponding to all trajectories starting with $in$.

To check automata emptiness Buchi Automata need to be defined. A Buchi Automata $A$ over $P$ consists of a Kripke Structure $M$ over $P$ and a set $T$ of accepting state. Intuitively, Buchi Automata is a formal way of dealing with strings of infinite length. Ordinary FSM and their acceptance conditions can not be applied to strings of infinite length. Informally, Buchi Automata says that an infinite string is accepted if it causes some accepting states to occur infinitely many times when that string is input to the automata. Based on the above intuition, we define an Accepting Trajectory. An accepting trajectory of $A$ is a trajectory $u_0u_1..$ starting with $u_0$, of $M$ such that $u_i \in T$ for infinitely many $i$’s. The language $L(A)$ consists of all traces corresponding to accepting trajectories of $A$.

The input to automata-emptiness problem consists of a hierarchical structure $K$ over $P$ and an automaton $A$ over $P$. Given $(K, A)$, the automata-emptiness problem is to determine whether the intersection $L(A) \cap L(K^F)$ is empty. This is the automata-theoretic approach to verification: if the automaton $A$ accepts undesirable or bad behaviour, checking emptiness of $L(A) \cap L(K^F)$ corresponds to ensuring that model has no bad behavior.

The automata-emptiness problem $(K, A)$ has been solved by reduction to a cycle-detection problem for the hierarchical structure obtained by constructing the product of $K$ with $A$.  

18
The automata emptiness problem \((K, A)\) can be solved by reducing it to the cycle-detection problem in time \(O(a^2|A| |K|)\), where \(a\) is the number of states in \(A\). [3]

5.2 Universality Problem

*Universality Problem* for HSMs \(M\) is to determine if the complement of \(L(M)\) is empty. This problem turns out to be much harder. *Universality Problem for HSMs is EXPSPACE-complete.* [2]. A similar problem is of *Trace Equivalence*. Two HSMs are trace-equivalent if their languages are identical. This problem has the same complexity as *Universality Problem*. [2].

6 Succinctness

In this section expressive power of hierarchical constructs are discussed (both in presence and absence of concurrency). First, expressiveness has been defined formally as *Succinctness*. We consider a family of languages \(\Gamma = \{L_n = 1, 2, \ldots\}\) and the number of states \(\#(\Gamma, n)\) needed by a machine that accepts \(L_n\) in some particular formalism. Also \(\#_{FSM}(\Gamma, n)\) denotes the number of states necessary and sufficient for an FSM to recognize \(L_n\). We will say that formalism \(F_1\) can be exponentially more succinct than formalism \(F_2\) if there is a family of languages \(\Lambda\) for which \(\#_{F_1}(\Gamma, n) = O(\log[\#_{F_2}(\Gamma, n)])\) [2]. Greater succinctness implies better expressiveness.

Both HSMs and FSMs define regular languages, but HSMs can be much more succinct than ordinary FSM. It should be noted here that there is an exponential translation from HSM to FSM. To illustrate the possible advantage of HSMs over ordinary FSMs in expressing the system concisely, we consider an example of a clock. First we try to represent the clock using ordinary FSM. We will represent the time of the day as states. There are 24 hours in a day, each hour has 60 minutes, and each minute has 60 seconds. So, we require \(24 \times 60 \times 60 = 86400\) states. Now we represent the same system using HSMs. Here we create one top level FSM with 24 super-states and map each of these to another HSM, with 60 states, and each of these 60 states is mapped to another FSM with 60 states. Now this HSM runs as follows. The moment we enter the first state of \(H\) we jump down the hierarchy, and now reach the initial state of \(M\), from there we jump further down to \(S\), and there machine makes transitions from \(S1\) to \(S60\) and then control moves from \(M1\) to \(M2\) and again \(S\) is called, and so on till control reaches \(M60\) after which control goes from \(H1\) to \(H2\) in the top level FSM. Thus control transfer exactly depicts the clock and with size \(24 + 60 + 60 = 144\). But as far as expressive power of non-deterministic FSM and HSM is concerned, it is interesting to note that nondeterminism and hierarchy are incomparable extensions, and we have these propositions:

**Proposition.** Deterministic HSMs can be exponentially more succinct than nondeterministic FSMs [2].

**Proposition.** Nondeterministic FSMs can be exponentially more succinct than deterministic HSMs [2].

So, HSMs can be advantageous in several cases (not all cases) when representing the system.
7 Appendix

Problem of model checking has been studied in this seminar, but due to space and time constraints the detailed discussion has been omitted in this report. In this appendix we give a brief discussion on Model Checking of HSM.

State machines define the systems formally. Once a system is formally defined we need some practical tool to check the logical errors in the system. Model checking is one of such tools. The requirements are first defined in some formal logic system, and then the logical formula is checked for correctness, on the state machine. HSMs are quite useful for modelling recursive and modular programs. So, checking for some specific requirements in such programs need model-checking of HSMs.

Linear Temporal Logic (LTL) Model Checking:
Several requirements can not be expressed in terms of simple propositional logic, for instance statements like “p will be true sometime in the future”, can not be represented in propositional logic but can be expressed in Linear Temporal Logic. In LTL model checking of HSM, we represent a system as HSM, express our requirements as LTL formula, and then check the model for the requirement. For instance, consider the network interpretation of Figure 4 where one system tries to connect to another system. It tries and waits for acknowledgment, if it receives the acknowledgment then process succeeds, otherwise it waits till timeout and then tries again. The same sequence is followed again and this time if timeout occurs, process is aborted. If such a model is given, and we are interested in finding out if the process will be aborted some time. Then we express our requirement in LTL, and then using LTL model checking of HSM we solve this problem. Detailed method is discussed in [4].

Branching Time Logic Model (CTL) Checking
Branching Time Logic formula can represent the sense of possibility. For instance, statements like “Along some computation p” can be expressed in CTL. Thus, if we have some requirement in which future has several possibilities, then to formally express such a re-
requirement, we need CTL. Once requirement is specified, we apply CTL Model checking of HSM [3].

8 Conclusion

HSM is a special class of FSMs which can have two added features: concurrency and hierarchy. These features add to the expressive power of FSM, of course, at some cost. As the complexity of systems (e.g. software and hardware programs) is increasing, we need more powerful methods to represent the system and at the same time we need to perform several actions (e.g. checking errors) on them efficiently. Since HSMs can represent complex systems concisely, and problems of reachability and cycle-detection can also be solved in linear time, they can be efficiently used for representation and verification of complex systems involving modularity and recursion. Also, a more general form of HSM, i.e. an RSM, can naturally represent cumbersome recursive programs. In this seminar we studied the structure and properties of the HSMs, and using them we tried to discuss some of its applications in practical problems.

References


