Embedded System - Design Verification and Test

Dr. Santosh Biswas
Ptof. Jatindra Kumar Deka
Dr. Arnab Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

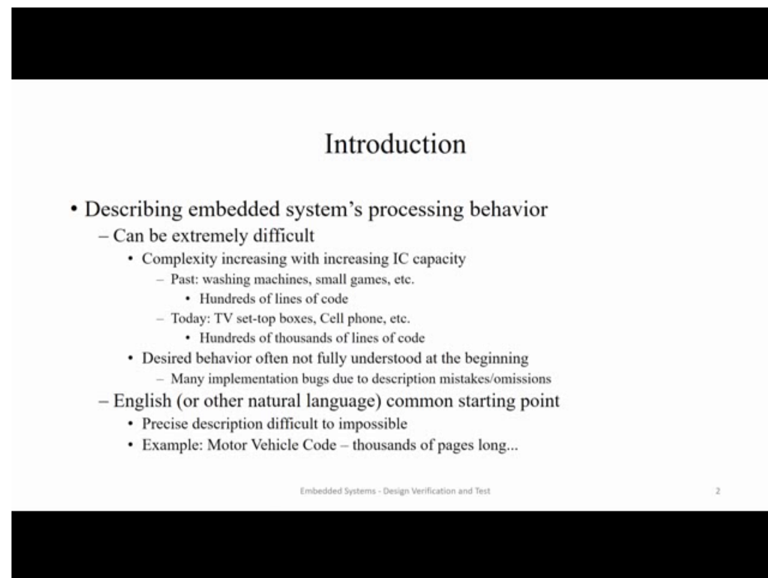
Lecture -02

Modelling Embedded Systems

Hello, welcome to lecture 2 of Embedded Systems Design. In this lecture we will be talking in details about Modelling of Embedded Systems. So, in the first lecture we talked about all the steps right from modelling to design to analysis and it is iterative structure to ultimately arrive at the complete design and finally, implementation we said last day. So, today we will deal with the modelling of embedded systems in more detail.

So, before starting to design an embedded system, we have a rough sketch or idea of what we want to develop. That idea has to be in a structured form specified so that we can do analytics on that understand if the specifications we just wrote down, we just specified you know we in the whatever way we want to specify them are correct and verifiable. So, this is why formal techniques to modelling embedded systems are required. The major source for this lecture today will be embedded systems design a unified hardware software approach Frank White's book.

(Refer Slide Time: 01:59)



So, describing embedded systems processing behaviour can be extremely difficult. Why? Because nowadays we have started to build very complex embedded systems with very complex functionalities in them, with complex interactions within these functionalities, why? Because with the increasing IC capacity, the capability of in designing functions very complex functions have increased.

And so, we have embedded systems with comparatively simple functionalities like the embedded computing systems in washing machines, small games etcetera, to which contains maybe say 100s of lines of code, to very complex embedded systems like say TV set-top boxes, cell phones to controllers in automobiles aircrafts etc, these may contain 100's of 1000 of lines of code.

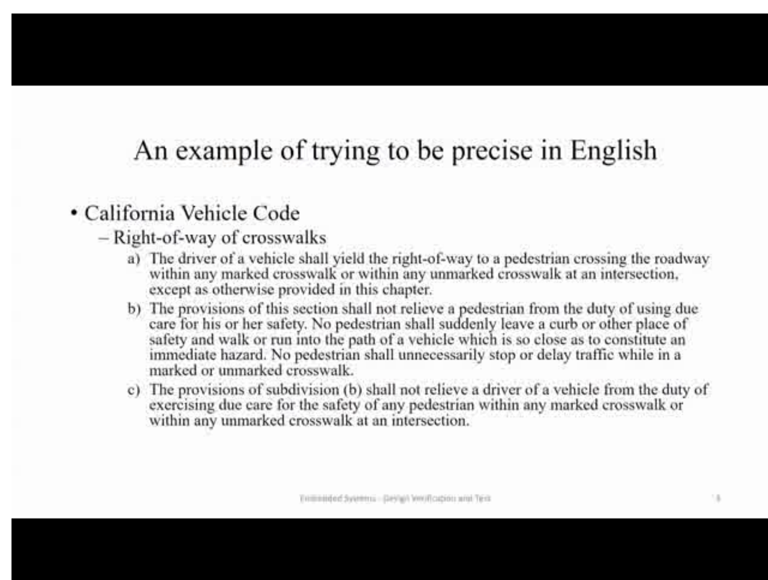
And when we have this crappy idea of what we want to develop in our head, it is not formally specified the desired behavior is often not fully understood; that if I implement this behavior in a certain way what will be the repercussions on other components of the system? We do not understand that fully, may not understand that fully. So, if we have this specification in this way, are we going to always get a response within? Can we add through any implementation get a response within say a requirement of 10 milliseconds? We do not know. So, therefore, formal and complete specification of the system is necessary.

Many implementation bugs due to description of mistakes and omissions; now because if we do not specify formally and completely an embedded system, there are chances that there will be a lot of bugs in the description, omissions and mistakes in the later when we want to further take this specification into design and implementation. So, that effort so, if we have such omissions and mistakes, we have to come back and again change the specification and redesign and do this many times.

Now, this effort in design and implementation again and again due to errors and omissions can be saved if the specification of the behavior is much is in a very structured format. Now, when we have this initial idea of what we want to do the embedded application in mind, we first generally write in English or some natural language as we take that as the starting point; however, through English precise description of what we want is impossible.

For example, suppose we want to write a motor vehicle code, we want to have an embedded system for a motor vehicle code, as to what are the rules that the vehicle should follow when moving on the streets, what are the rules that the pedestrians should follow on roads where vehicles are moving, those set of rules can go up to thousands of lines thousands of pages of English statements. And still you may not be able to formally and crisply state the exact requirements and specifications.

(Refer Slide Time: 05:41)



For example, again let us say that you want to just if we check a part of the vehicle code of the California vehicle code for crosswalks, we will see that you have to write statements like this. The driver of a vehicle shall yield the right-of-way to a pedestrian crossing the roadway within any mark cross way or within any unmarked cross way at an intersection except as otherwise provided in this chapter.

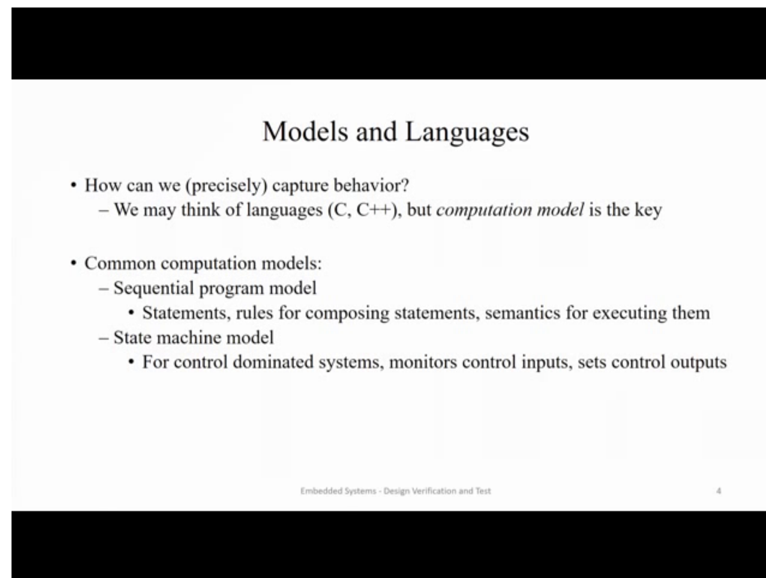
So, these are the ways in which people have tried to write crisp English language specifications; which is almost always is futile, you cannot write statements which is sound as specifications which are sound and complete. The provisions of this section shall not relieve a pedestrian. So, the first one says that the driver of a vehicle will allow a pedestrian to cross the roadway, either in a marked crosswalk or an unmarked crosswalk at an intersection, except otherwise provided. So, there will be exceptions to these rules, if those exceptions which are which will be stated later say I do not hold, then the pedestrian should be allowed to cross the road at the junction.

The provisions of this section shall not relieve a pedestrian from the duty of using due care for his or her safety; however, in doing so means although the vehicle will allow, this does not relieve the pedestrians duty. So, he cannot just close his eyes and cross an intersection. No pedestrian shall suddenly leave a curb or other place of safety and walk or run into the path of a vehicle which is so close as to constitute an immediate hazard.

No pedestrian shall unnecessarily stop or delay traffic while in a marked or unmarked crosswalk. Again the provisions of subdivision b shall not relieve a driver of a vehicle, from the duty of exercising due care for the safety of any pedestrian within any marked crosswalk or within any unmarked crosswalk at an intersection.

So, well what we just want to say is that, here through these sentences, when we are writing rules for how pedestrians should cross or how vehicles should cross in an English language we are trying to be very crisp. So, that exception to these rules do not occur, but we find we can find that these are never with through these sentences we can see that these are never complete and sound, we can always find exceptions to in whichever way we are trying to specify what we want to do in English. English cannot be used for these purposes this is what we want to say here. All that just for crossing the street the vehicle code is bigger and it will have much more things to do; so, English is not precise.

(Refer Slide Time: 08:50)

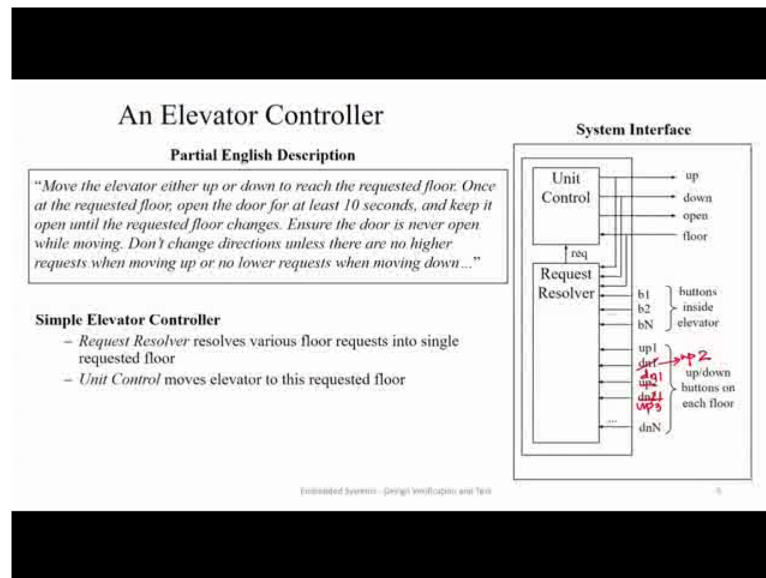


And therefore, specifically trying to specify English is futile. So, we need formal models and languages. So, how can we crisply or precisely capture behavior? We can think of languages like C, C plus plus, but computation model is the key. C and C plus plus are languages and using those languages we can write a behaviour. And that is a model which is called the sequential program behaviour.

So, if we write a C program, we are essentially write we have a formal model, we have instructions. So using C and C plus plus, we are using the sequential program model which has statements or instructions, rules for composing those statements how those statements can be composed to express a complete behavior and semantics for executing them. So, semantics meaning that when can certain sentences although they are grammatically correct can be used in a particular context. So, it allows statements and then it also allows us to iteratively and conditionally execute those statements, this is essentially the sequential program model.

Again another common computation model is the state machine model, it is generally used for control dominated systems, it has monitors for control inputs. So, it has many control inputs, it has states and it generates control outputs. This is the overall model of a state machine. There are many other computation models for example, the concurrent program model and then there are others such as the data flow model etcetera, which we will not go into.

(Refer Slide Time: 10:50)



So, in this chapter we will be primarily focusing on this example of an elevator controller, using this we will understand the different concepts that we will present in modelling Therefore, we first present the example here. So, the partial English description of an elevator controller will be like this.

Move the elevator so, what we are trying to do? The physical system is the elevator, and within this physical system there will be an embedded control system which will control the elevator, the movement of this elevator will be controlled by an embedded control system. And our job here is to design this embedded control system for the elevator. Now, therefore, first we have to specify in our mind; so, in our mind we have to first organize, what will be the behaviour for this embedded controller for the elevator.

So, it will be something like this. Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds and keep it open until the requested floor changes. Ensure the door, so move the elevator up or down until there will be a request, there will be a global request from somewhere some requests will be there for reaching a certain floor.

And there the elevator will move floor by floor and at each point at each time it will have knowledge of where, what is the current flowed it is in, and it will move to the requested floor. Once it reaches the requested floor, it will open the door for at least 10 seconds and keep it open, until there is the until we do not have a new request to go to another floor,

ok. Up to that the door will remain open. Ensure that the door is never open when the elevator is moving. Do not change directions unless there are no higher requests when moving up or lower requests when moving down.

So, we have a global request, right at any given point in time. When the current floor is not same as a requested floor, we are either moving down or moving up to the requested floor. Now the next what it says that while you are moving down? Moving up say to a certain requested floor above, we will not change direction unless there are no higher requests; so, you can take higher requests.

Suppose what happens is that you reach a floor, you reach a floor going up, and you have a new request which has which is higher than the current flow then you can go higher up. But you will otherwise you will go down. So, do not change directions unless there are no higher requests when moving up or no lower requests when moving down.

Now, with this English specification, we try to make it a more streamlined. Firstly, we see that it is best to design we decide within ourselves that it is best to design the elevator controller with 2 distinct modules. One, is a request resolver; which resolves various flow requests into a single requested floor. So, what happens? You have different floors and you have switches beside the lift door for going up or down, right? And you also have different switchboards within the lift to go to a certain requested floor.

Now so, therefore, request is coming from outside through your up and down requests outside the lift at various floors and you also have the request button inside the lift. All these requests together has to be resolved through an algorithm so that you finally have one requested floor. So, the request resolver resolves failures flow requests into a single requested floor. And you have the unit control which moves the elevator to this requested floor.

Now so, therefore, the request resolver here, if you see this one the request resolver, what are the inputs? There are various control inputs which are the buttons inside the elevator. So, b 1, b 2 up to b N, these are the N buttons for N floors say, within the elevator, ok. You also have buttons outside the lift just beside the lift door at each floor. What are these buttons? At the ground floor you do not have a down button. So, you have a up one, then you have sorry, you do not have a down one at the first floor.

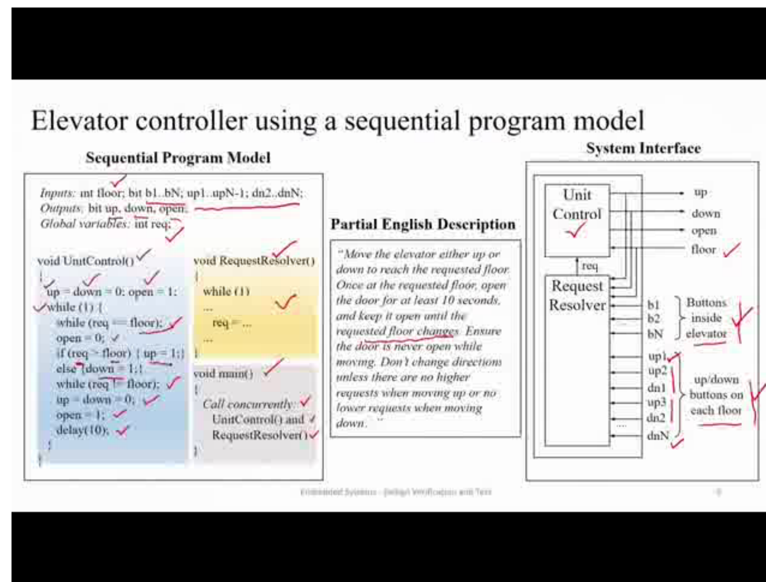
You have an up 2 and a down 1 in floor 1. So, in the ground floor you only have an up 1, because you cannot go down. And the first floor you can you can either go up to 2 or you can go down to 1. You have an up and a down. And at the highest floor, at the top floor you only have a down switch you do not have an up switch. So, these are the switches up and down buttons on each floor outside the lift door and these are the switches that you have inside the lift.

The modelling issues that we will be discussed in this lecture will be primarily based on the design of an embedded controller which we discuss here. So, what we want? We have a physical system an elevator; which is controlled by an embedded control system the elevator controller. Our job is to design an elevator controller. So, before the elevator controller is designed we just have what we want to do in our head, a scratchy idea. We first jot that down into in an English using an English language description.

A partial English language description for elevator controller can be as follows. Move the elevator either up or down to reach the requested floor. Once at the requested floor open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Do not change directions unless there are no higher requests when moving up or no lower requests when moving down.

Now, with this description in head we start our modelling. In our modelling we first understand and decide that this lift controller should have 2 distinct modules. One is the request resolver, which results various flow requests into a single requested floor and two and unit controller which actually moves to the requested floor which actually moves the elevator to the requested floor.

(Refer Slide Time: 18:24)



Now, this description this particular specification English language specification, can now be should be transformed through a model to a more concrete specification. And as we said we have various modelling strategies, one such is the sequential program model, suppose we want to implement or specify the behaviour of this elevator controller through a programming language such as C.

If we do so we have to first specify, what are the input outputs and data variables associated with this controller? Here we see that we have a data we have a data input floor. So, this data input floor will be provided to the embedded controller from the physical system. So, whenever the lift is moving or not moving, at any point in time the elevator controller, the embedded controller for the elevator will have the knowledge of what is the current floor the elevator is in.

It will also have bits b 1 to b N. So, these are binary inputs bits b 1 to b N corresponding to the buttons inside the lift. So, b 1 to b N are the buttons inside the lift corresponding to each floor. So, if we want to go to say the third floor we will press b 3, the switch b 3. These are the switches these are corresponding to the switches inside the lift. At each floor outside the door of the lift, we also have up and down buttons, right.

So, at floor 3, we will if you want to go up from floor 3, we will press the up button and wait for my lift to come. So, these are the up and down buttons on each floor outside the lift door. So, at the ground floor you have an up 1 button, you do not have a down button

because you are already at the ground floor. At the first floor you will have an up 2 and a down 1 button, at floor 3 you will have an up 3 and a down 2 button and so on at the top floor you will only have a down button, you will not have an up button.

So, these are inputs again binary inputs corresponding to these switches outside the door of the lift at each floor. Now, the elevated controller will output, what will it output? It will output it will generate control outputs; up, down or open. So, up will mean that the lift has to go up, the elevator has to go up, down will mean that the elevator has to go down, open will mean that the door of the elevator has to open.

It will also use the global variable req to which will hold at a given time the requested floor, meaning that the lift has to move to a certain requested floor, this global variable req will hold at any time to which floor does the lift has to move. This req will be output by the function request resolver which we do not completely specify here, but what it will do is that it will resolve from various requests made at the different floors by different people and within the lift.

So, we are not bothered about what particular algorithm, it will use. It will need to certainly use certain algorithm, and we do have to specify that we are not specifying it here. But ultimately at the end of this algorithm finally, we will resolve into a single requested floor where the lift has to move, ok. This will be done by the request resolver.

The other is the unit control function. So, we will write a function or sequential program for the request resolver, we will also write a function for the unit control. This unit control part. So, what will the unit control do? Initially, for the unit control, the output up and down are 0 and the door is open, why? Because initially the it will be at the IDLE position, at the IDLE position the requested floor will be equal to the current floor, and up and down with it will neither go up it will neither go down therefore, it will be stationary the lift will be stationary.

So, up and down or 0, and the door of the lift is open. So, this is what was required. Move the elevator either up or down to reach the requested floor, once set the requested floor open the door for at least 10 second then keep the door open until the requested floor changes. So, till the time the requested floor is equal to the current floor, it will just stay in that floor and keep the door open.

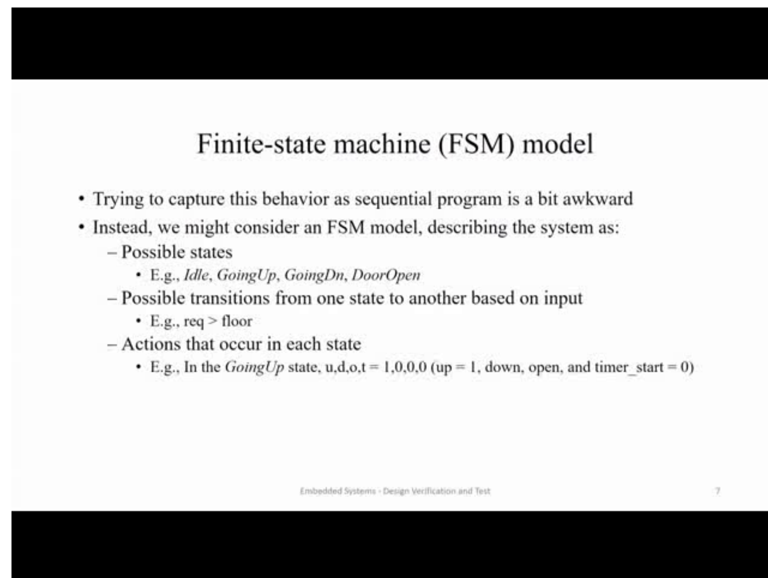
Now, after that it will move into an infinite loop. While the requested floor still remains as the still remains still remain same as the current floor, it will just wait in a loop. Then open equals to 0, so, when this is not true, when request when the requested floor has changed from the current floor and the lift has to move, the elevator has to move to a certain floor open is set to 0; that means, the door of the elevator has to close.

Now it has to decide whether to move up or down. If the requested floor is greater than the current floor it has to move up, then up equals to 1, else down equals to 1. So, if req is greater than it is not equal to req, it is not equal to floor that is known. If now req is greater than floor it is going to move up, if req is less than floor it is going to move down.

Now, after that while the lift has not reached a requested floor, unit control will wait. So, it will wait in a loop, this while loop until request is not equals to floor, until the lift has not elevated has not reached to the requested floor. Once it has reached the requested floor, again up equals to down equals to 0. Because the lift has to go in the lift will again be stationary. And the door will open; so, open is 1 and then it will wait for at least 10 seconds delay of 10 seconds and then it will remain idle for at least 10 seconds it will the open is then not set to 0. So, open is going to still remain open. But at least 10 seconds it is going to wait there. It will it is not going to move to a newly requested floor at least before 10 seconds. It is not going to do that for 10 seconds, after that it can again move.

Now, what will main do? The main function of the lift controller, the elevator controller will concurrently call in parallel both unit control and request resolver. So, request resolver will work independently and concurrently and it will go on resolving requests and generate a console it finally, consolidated single request. Based on this request the unit controller will move to the requested floor, and by generating control outputs, unit controller will generate instructions to go to the requested floor rather through control outputs up down open etcetera. In this process, the unit controller will always keep this floor input floor data input it will keep in it is knowledge the value of the floor data input.

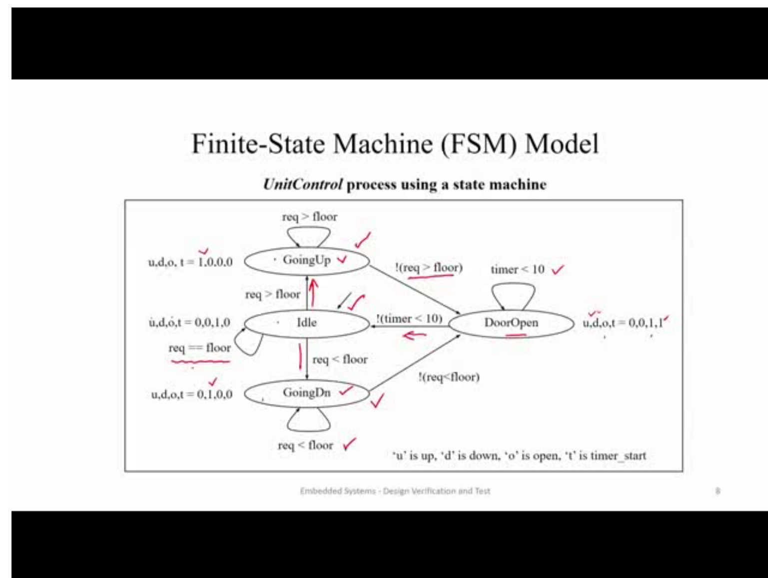
(Refer Slide Time: 26:40)



So, trying to capture this behaviour as a sequential program is a bit awkward. So, instead, because why is this so? Because, this is primarily a control dominated problem. This embedded controller here for this case is a control dominated software that we want to build software or in hardware whatever it is. It is a control dominated specification that we want to specify. So, instead we might consider an FSM model, describing the system as possible states, their transitions and actions that should occur at each state. This is more easy because this is how the things are actually happened for our elevator controller. It has states idle when the lea when the elevator is not moving, it can go up, it can be in a state going up it can be in a state going down, it can be in a state door open.

Possible transitions from one state to another based on input. It has so if the requested floor is if the requested floor is greater than the current floor then you go up. So, you are you are in an idle state, and then say the requested floor is greater than the current floor, then you go to the state going up using the transition request floor greater than using the transition labelled request greater than floor. Actions that occur in each state so, corresponding to each state you also have actions. For example, in the going upstate, the up act the up control output has to be one the down control output has to be 0, the o control output which is the door open has to be 0, and the timer is also 0. Why we require the timer? We just we are just coming to.

(Refer Slide Time: 28:33)



So, using this we can specify an FSM like this. So, in this FSM, let us see that the FSM starts at the initial state of the FSM is this one idle, it has the name idle. In this state, it stays until the requested float is equal to the current floor. So, in the self-loop it moves, it goes on in this the self-loop it goes on staying in this idle state, until the requested floor is different from the current floor. And in this state you have up and down variables; u and d corresponding corresponds to up and down outputs, they are 0 and the open door open output is 1 and the timer is 0 again.

And then if the requested floor is greater than the current floor, you move to the going up state, and then you stay in the going upstate until requested floor is greater than the current floor and in this state, your up variable is 1, down variable is 0, open variable open door variable is 0. So, the while going up or going down the door of the elevator should remain closed.

Similarly, when request is less than floor, you go down to the going down state, you go to the going down state, and in this state you remain until the requested floor is less than the current floor. And in the, and this one the output the output that it will produce in this state is down equals to 1 all other control outputs are 0. Now in going up or going down who are after when at this state when the requested floors finally, becomes equal to the current floor you go to the door open state.

So, when go when you are in going up, and then the requested floor is no more greater than the current floor; that means, the requested floor has become equal to the current floor you go to the door open state. Similarly, in going down if the requested floor is no more less than the current floor then you go to the door open state. In the door open state, you are up and down both are 0 so, the lift is not moving. Your door is open so, it is 1, and what you do is you set the timer t. So, the timer was 0 in all these 3 states going up idle and going down.

Now you set the timer and you move into the self-loop until timer is less than 10. So, the door should remain open for at least 10 seconds before it takes fresh requests, ok. So, it will remain in this state, and then when the timer becomes equals to 10 which means that the timer is not less than 10 anymore it goes back to the idle state. And in this idle state, it will still the door will still remain open until till the time the requested floor and the current floor are same. Again when the requested floor becomes different based on what the request resolver has told you, then you then the FSM again moves to either going up or going down.

(Refer Slide Time: 31:57)

Now this was an informal definition of how the FSM model works. So, how you specify a behavior using the FSM model? Now to formally define this model we do as follows. An FSM is a 6 tuple, where S is a set of states every FSM will have a set of states s 0 s 1 dot, dot, dot, a set of inputs i 0, i 1 dot, dot, dot, a set of outputs o 0, o 1, a next state

function which goes from which is a function of what the prod the cut the which goes from the set of states and set of inputs to a set of states.

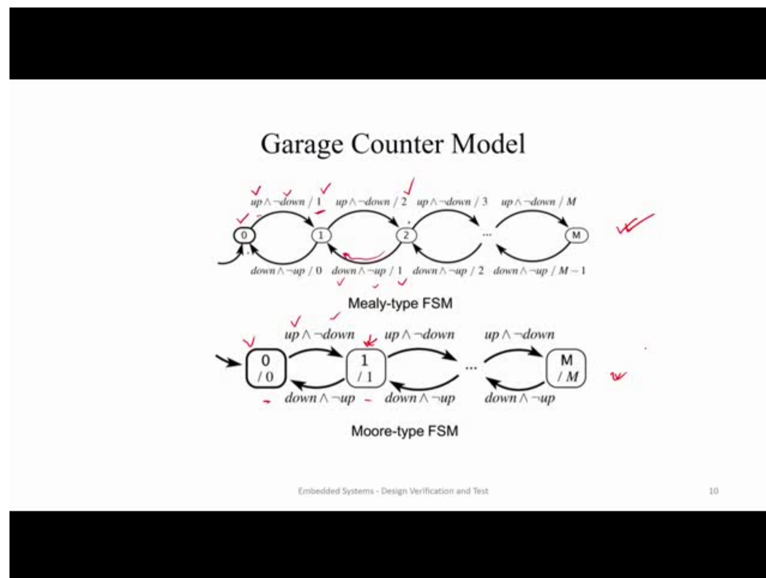
And it has a set of output, so, it has an output function from each state it generates an output. So, at each state based on certain inputs it goes to a next state. At each state based on certain inputs it goes to a next state. That is why it is S cross I to S. H is an output function which goes from a state to output, and s 0 is the initial state.

And now, the FSM we just built is a mood type FSM, why? Because it associates the outputs with states. A more FSM is an FSM where the outputs are so only associated with states. As against mini FSM's where the outputs are associated with transitions. So, in a mini type FSM, the outputs are associated with transitions. So, if you are at current state S and you have some inputs depending on that you will produce some output. So, you have an S, you have a transition condition based on inputs based on that you go to a certain next state and produce certain output, in a Mealy machine. In a Moore machine, you move to a new state and at that state you produce certain outputs.

Now, to simplify definitions of FSM several shorthand notations can be used, a few of them are as follows. Shorthand notations to simplify descriptions, you can implicitly assign 0 to all unassigned outputs in a state. So, if you have not assigned, here we have explicitly assigned a 0 to all out all outputs at a given state. Now if you do not want to do that because it is tedious to do only certain output will be 1 and all other output will be 0, you can implicitly assign 0 to all unassigned outputs in a state.

You can also implicitly and every transition condition with the clock edge. Meaning that many of these FSM's will act will actually end up being defined as hardware circuitry. And in a hardware circuitry typically things are synchronized with the clock pulse of a timer. So, you can say that implicitly you can add every transition, every transition with the clock edge of a timer. And we do not you do not need to explicitly say that and timer.

(Refer Slide Time: 35:28)



As an example of how more and many machines are defined, we again go back to the garage counter example that we had disk we had discussed in the last lecture here, you had garage parking lot where the embedded the job of the embedded system is to keep an account at any time of the number of cars in the garage. Now you can design this as a mealy FSM or a Moore FSM.

Now, the first one here is a mealy FSM. Now here the at any time the job of the embedded controller is to produce as output the number of cars. So, if you are at stage 0, and you your up is 1; that means, a new car has entered and down is not 0, then what happens? Then you output a 1, because up is 1 you have one car in the garage. At state one again you have an up one and a down and down 0 then you produce an output 2; that means, you have 2 cars in the garage.

Now at state 2 you moved and you move to state 2. So, a state tells me what is the current number of 4 cars in the garage, at state 2 say again what happens is that one car goes out. So, down is 1, and up is 0, if it so happens, then you produce an output of 1. Because you go back to state one and the number of cars actually is one in the garage at this time. So, this is how you can define a mealy FSM for the garage counter.

A corresponding Moore FSM will be as follows. At state 0 you output 0. At state 0 you know that the number of cars is 0, and you output 0. Then you have if you have an up and not down, you straightaway go to 1, and you know that you are at 1; that means, an

of cars in the garage is 1 so, you output 1. So, here you see that the output is associated with states and not transitions, and hence this is a Moore type FSM, here in this case the output is associated with a state and a condition. So, here the output is associated and the inputs if the state is 0, and the up is 1, and down is 0 then the output is 1. So, here the output is associated with a state and input and the inputs here the output is associated only with the current state. So, the first one is mealy and the second one is a Moore type FSM.

Now, the FSM description that we just provided here was not actually a simple FSM, but was a extended FSM or a finite state machine with data path FSMD.

(Refer Slide Time: 38:41)

Finite-state machine with datapath model (FSMD)

- FSMD extends FSM: complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- FSMD: 7-tuple <S, I, O, V, F, H, s_0>
  - S is a set of states $\{s_0, s_1, \ldots, s_l\}$
  - I is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - O is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - **V is a set of variables $\{v_0, v_1, \ldots, v_n\}$**
  - F is a next-state function ($S \times I \times V \rightarrow S$)
  - H is an **action** function ($S \rightarrow O + V$)
  - s_0 is an initial state
- I,O,V may represent complex data types (i.e., integers, floating point, etc.)
- F,H may include arithmetic operations
- H is an action function, not just an output function
  - Describes variable updates as well as outputs
- Complete system state now consists of current state, s_i, and values of all variables

We described UnitControl as an FSMD

So, let us define what an FSMD is, it is not a simple FSM that we just discussed because we will just discuss now. So, FSMD extends FSM, it allows complex data types and variables for storing data. FSM's use only Boolean data types and operations no variables. So, in our definition of FSM that we used earlier, we did not have any variables. In the definition of the FSM, so, in the definition of the FSM we did not have variables and all our inputs and outputs were binary, we did not have complex data types. So, these are the 2 extensions that have been added to the simple FSM to for better modelling of embedded systems.

So, FSMD is a 7 tuple, it has a set of states, it has a set of inputs it has a set of outputs, it has variables. Now which is nu, it has the next state function which is a function of the

state, the inputs and the variables and then it goes to new state. It has an action function which is which goes from a state to a set of outputs and variables. So, the action function now not only produces output control actions, it also updates variables.

Now, in our case we see that we have variables, in this FSMD we have variables. We have complex comparisons on transitions such as comparisons. Here these are not binary, Boolean you see that request and floor they can have data types which are integers they are not primary. So, therefore, you have allowed complex mo data types. And you have allowed variables.

So, I, O, V, may represent complex data types integers floating points etcetera. F and H now include arithmetic operations. So, in our previous case these were just Boolean expressions F and H where this s cross I cross V could only be Boolean expressions. Now you can have arithmetic operations for example, subtraction from subtraction some integer variable from another integer variable and update that variable etcetera. So, these are now included as part of the FSMD, ok.

H is an action function now and just an output fact output function, why because along with the production of outputs it can also update variables. So, we do not say that this to h to be an output function if in FSMD we say that as an action function.

The complete state now consists of the current state s i and the values of the variables. So, the state in if in an FSMD or an extended FSM consists of the state and the values of the variables. So, the elevator controller state machine that we had drawn is actually an FSMD and not a simple FSM.

(Refer Slide Time: 42:22)

Describing a system as a state machine

1. List all possible states
2. Declare all variables
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning

Now we will look at we understood what is the formal definition of an FSMD and an FSM, we have already this we have already drawn an FSM for the embedded controller. But we now want to see we want to now see systematically how for any embedded such embedded system and FSM can be drawn. So, what are the steps that are required to draw an FSM for a given specification? From an English language specification, how do we come about drawing a structure systematic FSM?

So, first is to list all possible states. And so, what are the my states in my in the current FSM? My states are idle, going up, going down and door open. So, from the English language specification, first we have to think what could be the states of my FSM. There could be different ways of designing the same designing the same functionality. There could be different ways of not designing, there could be different ways of specifying the same functionality.

So, we can say that the meaning of the states to be different, but this one is a possible way in which to do. So, we have 4 states here and idle going up going down and door open. We declare all variables. So what are my variables? My variables here were the floor and the others were inputs and outputs. For each state, we list all possible transitions with conditions to other states. So, for example, for the idle state, we have 2 transitions, sorry we have 3 transitions, one is when request equals to floor request greater than floor and request less than floor; these are the 3 transitions. So, for the idle

state we have listed down possible we have listed down possible transitions with conditions to the other states or to the same state rather when we have a self-loop. This for the going down transition similarly we have these 2 transitions and for the going up transition for the going up state we have these 2 transitions again and then for the door open we have further 2 transition. So, we have listed down or we have drawn all the transitions separately for all the states with conditions.

Then for each state or transition we have to list the associated actions. So, what are the actions with each state? The actions are specified for our FSMD through the u, d, o, t very involved in the u, d, o, t outputs, right. So, u is the up control action, d is the down control action, o being the door open control action and t it being the timer control action. Now after that for each state ensured exclusive and complete exciting conditions exciting transition conditions. What do we mean by these to ensure exclusive and complete? So, what happens if you do not have exclusive transition conditions?

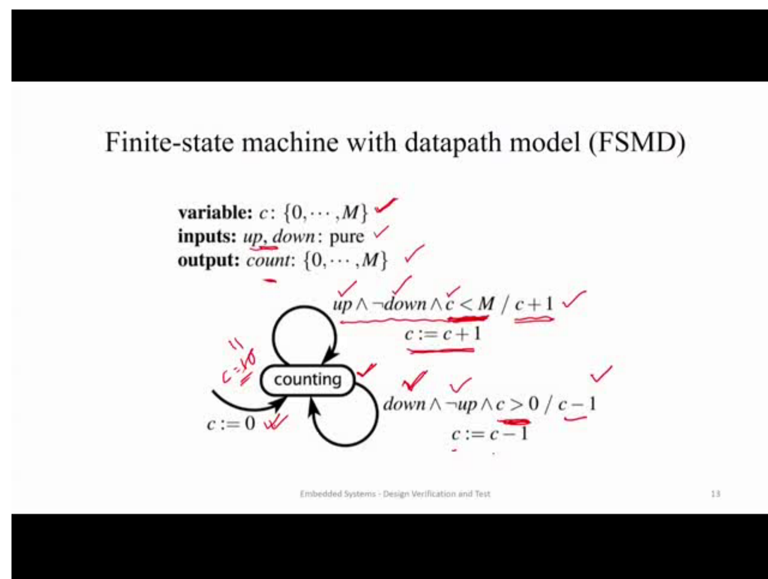
Then 2 conditions can be true at the same time. So, exclusive transition conditions meaning that with respect to a set of control units, there can be 2 there cannot be 2 transitions from the same state which hold true. For example, req great in req can be either greater than floor, req can be either equal to floor or req can be less than floor. So, you have 3 transitions none of which can be true simultaneously. So, if req get greater than floor, req cannot be equal to floor if req greater than floor they cannot be less than floor. So, this transition will be exclusively excited.

So, no 2 existing conditions can be true at the same time. Otherwise what will we have? We will have a non-deterministic state machine; that means, you have 2 possible ways to move from a state for the same inputs. We have you have 2 possible ways to move from a state to another state. And it has to be complete, and one condition must be true at any given state. Otherwise, the state machine becomes incomplete.

So, it is not that design cannot be cannot be non-deterministic or design cannot be incomplete, but then for a starter we should avoid having non-deterministic state machines, although later on with further practice we will see that non-deterministic state machines allows us certain optimizations corresponding to the to the to the design that we will get from this specification, may make allow a certain in this one. But it also open up doors for errors which for a first time or it may be difficult to handle.

Similarly, the need for a complete transition function and not a partial one; that means, that at least one of the transitions must be taken at any point in time. So, if I do not have req greater than floor door req less than floor, at least req equals to floor which means that I will have a self-loop moving into the idle state again and again. So, one condition must be true at any given time that is why it is complete reducing explicit transitions should be avoided when first learning.

(Refer Slide Time: 48:35)



Now, we said that the milling machine that we considered for the garage counter, the Mealy and the Moore machine that we consider for the garage counter was a was an FSM. Now we said that FSMD allows certain succinctness in the in the in the in the specification of the models. For example, for our garage counter example we can have that that whole big state machine that we had.

In that state machine if you go if we go back in that state machine, we see that we have let us say that the garage has space for M cars, then in this one we will have a big state machine consisting of M states both for the Mealy machine and for the Moore machine which is a big representation. How can we simplify this representation make it more succeed? By using an FSMD a variable.

So now, we have a variable which can count from 0 to M, we have an integer variable which one counts for 0 to M. We have up and down pure inputs which are Boolean inputs for our garage counter. We have an output count we have an output count which is

again an integer, right. So, we have integer outputs, we have Boolean inputs and we have variables so, this one is an FSMD. So, we have a state the initial, initial state is this one at this state C equals to 0. So, the variable is 0 C represents the number of cars.

Now, how do we output? So, at this state if up is 1 and down is 0, and you have and you have C and C is less than M, then you output C plus 1. So, the at each point in time at this state C will have a certain value. Initially C 0 there are no cars in the garage. But at a certain point in time when cars have entered and cars have gone out at a certain point in time, this C value will load let us say C equals to 10 at a given time you have 10 cars in the garage.

Now at that point in time, you have up equals to 1 down equals to 0, and let us say the capacity of the garage is 20 cars so, C is less than M. So, if these conditions if this total condition the all they said well this and the up and not down and C less than M is true, then you output C plus 1; that means, the number of cars will be C plus 1 because up is one new car has come. And you update also the variable C equals to C plus 1, ok.

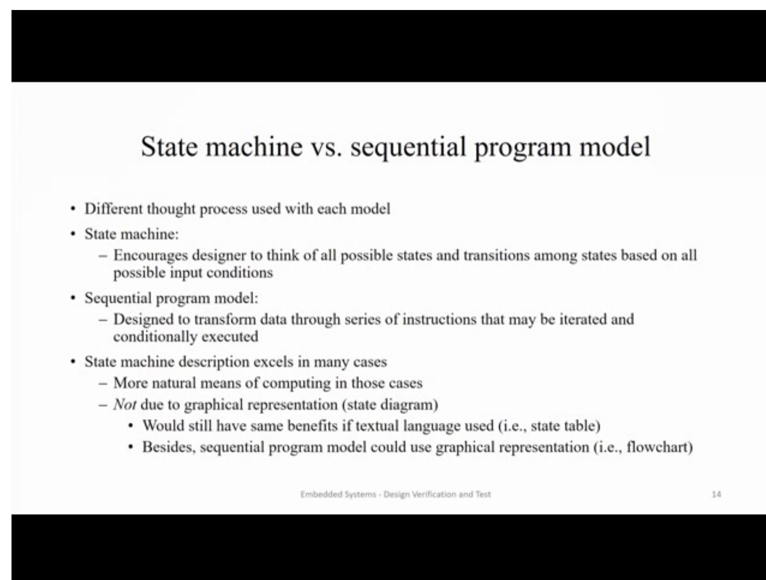
Similarly, if you are at C equals to 10. So now, C plus 1 is going to be 11 and the updated value of count will be C equals to 11, ok? Similarly, now let us say C equals to 11 down is 1, up is 0, and C greater than 0, ok. Then the output will be C minus 1; that means, one new car has gone out therefore, therefore, the output should be one less C should be equal to 10; that means, the currently the garage has 10 cars after one car has gone out, right.

Now what happens when C equals to M? And so, when C equals to n, even if one cars tries to enter and up is 1, there is no capacity, the garage is full. So, it is not going to output C equals to C plus 1 and update the variable because the garage can hold at most M. Similarly, if the garage is empty and C is equal to 0, down equals to 1 has no meaning therefore, it is not going to out output C minus 1 and update the variable C to C minus 1; it is not going to do it. So, that hole M state machine FSM if FSM machine, now just can be expressed as a one state FSMD or an extended FSM. So, this is how FSMD extends the power of simple FSM's.

So, we saw 2 ways of designing of specifying embedded systems, 2 ways of modelling embedded systems. So, one way is the state machine model, and the other is the sequential program model. How does these 2 models compare? Actually 2 different

thought processes go into the design of these 2 models. The in the state machine model, the state machine model encourages the designer to think in terms of possible states that the embedded system can be in and transitions among these states so that you can you can find out how these states how the state changes happen.

(Refer Slide Time: 54:07)



And transitions among these states on all possible input conditions this is how the state machine model allows us to think. On the other hand, the sequential program model is designed to transform data through a series of instructions. So, we have instructions in the sequential. So, this think of a C program, you have a series of instructions. So, what does it do? It transforms the data by the statements, and it allows us to iterate over these statements using loops, and it also allows conditional execution of statements through ifs for loops cases while loops etcetera. So, a combination of iterations and conditional execution is possible through the if for loops while loops case switch cases etc. And the statements themselves go on transforming the data. So, the sequential program model is designed to transform data through a series of instructions that may be it iterated and conditionally executed.

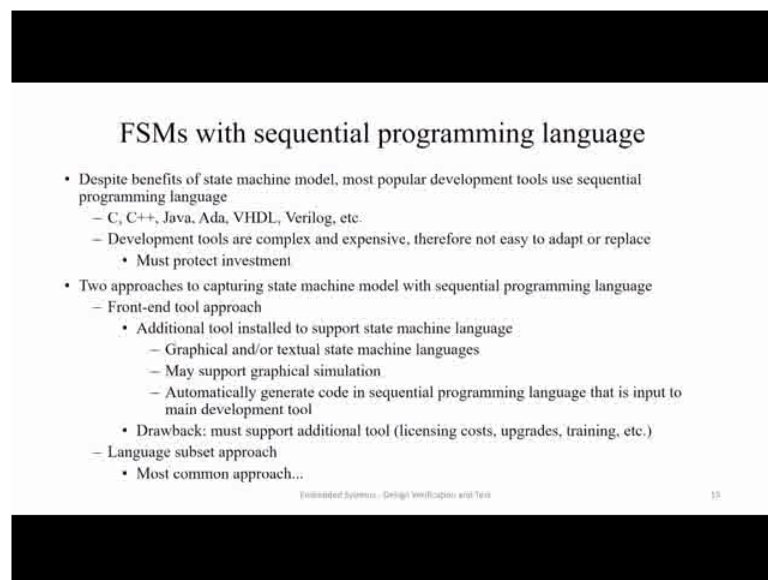
So, state machine description excels in many cases. When you have control dominated systems, state machine is a very easy succinct way of it is simple way of expressing. And it becomes much less cumbersome than a sequential model. So, state machine often are a more natural means of computing in those cases, not due to graphical representation. So,

it so, a state machine would still have the benefits of simple repress of simplifying the way you can specify the behaviour of an embedded system even if it is used in a textual form.

So, we did need not have circulars states, it has some in circles as states and arcs as transitions we do not need to draw it like that. We can draw it in the form of a state table, where we have an entry for each table with inputs outputs a specified, right? We can represent it in the form of a state table, and it will still have the same expressive power right.

On the other hand, the sequential programming program can be represented through a graphical representation for a for a flowchart. So, it does not matter whether you are representing in textually through a textual language or a graphical language, but the for certain types of embedded systems, state machines are a very natural way of expressing such systems expressing the behaviour of such systems. And therefore, in those system state machines will be more beneficial than sequential program model.

(Refer Slide Time: 57:01)



So, FSM's so constructing, but, but it is it is the sequential program model has traditionally being been used. So, there are many tools for sequential program based specifications, and further compilation from that. So, you have high-level synthesis tools which allow you to specify in VHDL for a hardware description, let us say, and then there are automated tools that will take the behaviour to the complete design of the

circuit infer to the hardware circuit up to the fabrication level. So, once you specify the VHDL code, it there are a set of tools which will take you to the hardware circuit finally, ok.

Similarly there are other tools which start from the sequential program models. But then so despite the benefits of the state machine model, most popular development tools use the sequential programming model which works with C, C plus, Java, Ada, VHDL, Verilog etcetera. Development tools are complex and expensive, therefore, not easy to adapt and replace. So, if we now suddenly say that we will use this diagrammatic FSM and we will draw diagrams and from there you build it is again a separate process, and we have to do away with all the tools we have designed thus far.

So, what we do instead is that we have we use 2 approaches. We have a front-end tool in which we draw it, and we draw an FSM based model using a diagrammatic approach. And then there will be an automated mechanism we use a subset of the sequential program model itself of a sequential programming language to represent a a state machine textually in the form of a program. So, 2 approaches to capturing state machine model with sequential programming language.

Use a front-end tool approach, additional tool installed to support state machine model, graphical and or textual state machine languages. So, we will use a graphical and or state machine languages and that may support graphical simulation. So, what why graphical simulation? Because at that level itself we can add up to some level we can understand whether the whether what we have specified is correct or not. And there will be an automated automatically generate sequential programming language code, that is input to the main development tool, ok.

So, we have this front-end tool; which takes it to the sequential program model automatically, which compiles the front-end tool is a diagrammatic tool, you have a compiler for this diagrammatic front-end tool, which will automatically generate the sequential program model corresponding and that sequential program can then be fed to the main tool which will take it to the further levels of design.

Now, what is the drawback of this approach? It must support an additional tool. Number 1, licensing cause upgrades training etcetera that will be required. Another may be another bottleneck could be that is the interfacing with the main tool. So, the sequential

program models can be in different languages. Suppose the main tool supports a language which is not output by the front-end tool, so, that interfacing between the output of the front-end and the input to the main tool has still to be provided, ok. So, these are the 2 drawbacks. So, what do we use instead? Instead of using a front-end tool we use a language subset approach.

(Refer Slide Time: 60:55)

And how does that approach go about? So, in the language subset approach, we use it follows rules or templates for capturing state machine constructs in equivalent sequential language constructs, ok. So, we have equivalent sequential language constructs to capture different state machine constructs, can be used with software example C and hardware languages like VHDL.

So, for our unit control for example for our unit control FSMD, you can, we can you can capture this whole unit control FSMD using a sequential program like the following that we have specified here. So, in this we first enumerate all the states through hash defined constructs. So, idle the state idle is hash defined enumerated as 0, going up is 1 going down is 2 and door open is 3.

Then you declare, state variable initialized to the initial state. So, you have then you. So, what then you have a subroutine for the unit controller you go into that subroutine, and you have a state variable. So, the current state is captured is held at any time through this

variable state. And this state variable this variable is initialized to idle. So, declared state variable initialized to the initial state idle for our case.

There is a single switch statement which branches to the current states case. Now at each point in time, we use a switch case to branch to the current states case. So, suppose the current state of the FSM is going up, then it is 1. So, switch going up or switch 1 will take you to the going upstate. So, there is a single switch statement which branches to the current states case. Now, for if it is going up you go up. In each case what do you have? You have actions.

So, first in after you go to going up, what are the actions? Up has to be 1, down is 0, open door should be closed and the timer should be 0. So, these are the actions at this state. And you also have what? There is a set of if then else statement, if statements; each case checks transition conditions to determine next state. So, what are these ifs doing? These ifs tell you what are the conditions for going to the next state? For example, if you are at going up then it says that if req is great is still req if req is still greater than the current floor then you remain in going up. If req is not greater than current floor you go to door open state, ok.

Now, you instead of this if statement you can make it an explicitly an if-then-else, you can make it an if-else statement, so that even if by mistake, you have you do not have exclusive conditions. So, we said that so, I at any state only one transition should be true. There cannot be a set of inputs which give you a condition where more than one transition becomes true, you cannot have that case.

So, suppose by mistake, your conditions are such that your design is such that your inputs as such that it allows 2 transitions to be true. Now if you in this specify this you capture this through and if else and not if then only the first of these 2 transitions will be true and not the subsequent ones, ok. So, you can use an if explicit if-else instead of a if. So, this if condition then next state this structure gives you the way how to go to a next state based on condition.

(Refer Slide Time: 65:24)

General Template

```
#define S0       0
#define S1       1
...
#define SN       N
void StateMachine() {
   int state = S0; // or whatever is the initial state.
   while (1) {
      switch (state) {
         S0:
            // Insert S0's actions here & Insert transitions Ti leaving S0:
            if( T0's condition is true ) {state = T0's next state; /*actions*/ }
            if( T1's condition is true ) {state = T1's next state; /*actions*/ }
            ...
            if( Tm's condition is true ) {state = Tm's next state; /*actions*/ }
            break;
         S1:
            // Insert S1's actions here
            // Insert transitions Ti leaving S1
            break;
         ...
         SN:
            // Insert SN's actions here
            // Insert transitions Ti leaving SN
            break;
      }
   }
}
```

Embedded Systems - Design Verification and Test 17

So, what is the general template for designing any state machine using the programming language structure? Again, you have to define or enumerate all the different states in the state machine, then you have to define a state variable your state, which has to be first initialized to the initial state, then you have to go to an infinite loop which in which you will have a single switch case statement. In this switch statement you will go to the case for the current state based on the value of state. At each case statement what do you have? You have the actions that you have to output at that state and you have the transitions for conditions. So, insert transitions for leaving that state, and you break so this is the general template.

(Refer Slide Time: 66:29)

HCFSM and the Statecharts Language

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - *With hierarchy* has identical functionality as *Without hierarchy*, but has one less transition (z)
    - Known as OR-decomposition
  - States can execute concurrently
    - Known as AND-decomposition
- Statecharts
  - Graphical language to capture HCFSM
  - *timeout*: transition with time limit as condition
  - *history*: remember last substate OR-decomposed state *A* was in before transitioning to another state *B*
    - Return to saved substate of A when returning from B instead of initial state

Without hierarchy

With hierarchy

Concurrency

Now to further succinctly and nicely represent even co even more complex embedded systems further extensions to FSMD were specified. One such very important extension is a hierarchical concurrent state machine model or the HCFSM. So, HCFSM is an extension to state machine model to support to support hierarchy and concurrency.

Now in this states can be decomposed into another state machines or you can say that a state machine can be a whole state machine can be can be abstracted as a single state, so, this allows. So, this is the hierarchy in the hierarchical state machine. States can be decomposed into another state machine. So, have a single state and that get decomposed into a complete new state machine. With hierarchy and identical with hierarchy has identical functionality as without hierarchy, but one less transition z in this case. So, let us see here we have a state machine without hierarchy.

This state machine has 3 states; A 1, A 2 and B. The initial state is A 1 and you move to state B whenever you have this input it z. So, on input z, whenever you have this input z, you go to state B and at state B whenever you get this input w you go to A 1 state, ok. Now this one is a general state machine that we have studied so far without hierarchy.

How do we represent this same state machine with hierarchy? Now we have clubbed the states A 1 and A 2 into a composite state A, ok. The where now so, you have now at the outer level you have 2 states A and B, A is the initial state. Within A, A 1 is the so, within A, this A is basically this state machine composed of 2 states A 1 A 2 of which A 1 is the

initial state, ok. Now from this state A, whenever you have this state z, you move to state B and whenever you have this state whenever.

So, whenever you have this input z you move to state B, and that state B whenever you have this input w you move back to state A and to the initial state of A which is A 1. So, in this one from corresponding to this design with hierarchy it has a same functionality as without hierarchy, but it has one less transition for z. So, here you had 2 transitions from A 1 and A 2 to B, now it has a single transition from the entire state A. So, this one is known as an OR decomposition. Because either A or B will be executed at a given time not both concurrently.

So, A and B we could either we could further we could possibly be another state machine internally. Inside B, because B can either also be a composite state, and when you have an all decomposition as in here you say this to be an OR decomposition, because either A or B the system will be in either state A or state B and not both together concurrently.

However, HCFSM's can also have concurrent states. And this is called an and-decomposition. For example, in this case here you have a state B which has 2 sub states C and D each of them happens to be a composite state. C is a composite state which contains a state machine composed of C 1 a composed of state C 1 and C 2 and these transitions here. And also you have this composite state D which is again a state machine composed of states D 1 and d 2 and these 2 transitions, ok.

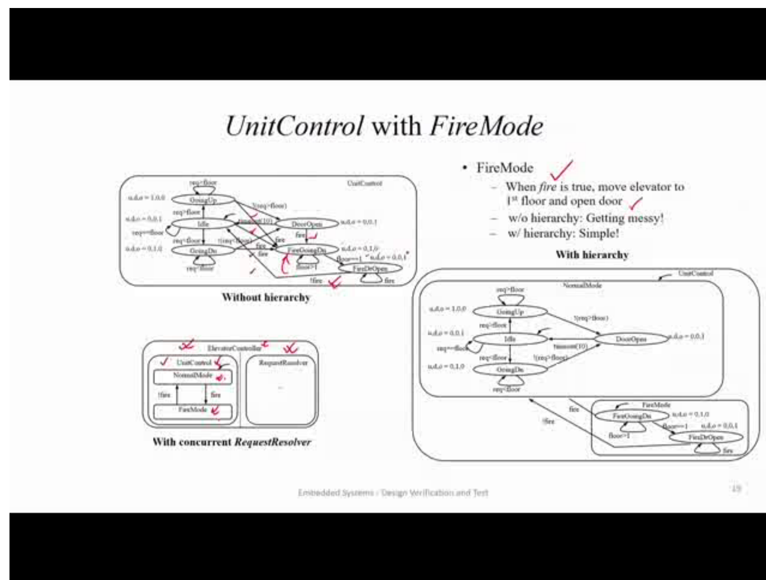
And this is a composite this represents concurrency because C AND D execute concurrently together. So, both C AND D, execute concurrently with in state B. And because both of them is executing concurrently we say these to be an AND-decomposition. So, C AND D execute together, A OR B executes together in the in this example A OR B, this is C AND D.

So, a language which actually represents such an HCFSM is the state charts. It is a graphical language to capture HCFSM. It has this all the functionalities that we just described, and it also along with this it also allows the function a function called timeout which so, what how does it help? Instead of using an explicit external timer, it will remain in a state for a certain amount of time till the timeout happens, and then move out of that state to the state where this timeout label indicates to so it will move to that state after the timeout happens.

And it also remembers history so it remembers the last sub states or decomposed state A was in before transitioning to state. For example, this is with respect to this example that we were discussing. Now you can move to state B either from A 1 or from A 2 because on transition z you have moved to state B. Now by remembering history, you remember the last sub state, sub state from which state? In state A, in this state A, that it that you were in before moving to B, before moving to state B, suppose you move to state B from A 2 through using this transition.

Now, because you remembered the last sub state from which you move to B after you go back to A, you can move directly to A 2 and not to A 1. So, the original HCFSM did not remember history. So, whenever you move from B to A it will actually go to the initial state of a which is A 1. But state charts allow provide you the option of remembering history. So, you can go back to either A 1 or A 2. So, return to saved sub state of a when returning from B instead of the initial state. This is allowed by the state charts.

(Refer Slide Time: 74:10)



To show an example which uses such hierarchy, we extend the specification of our elevator controller by one step. So, let us say that we have a fire mode. And it is in this fire mode, when fire is true, when fire is true, we move the elevator to the first floor and open the door, this is what we have to do.

So, how do you capture it? Capture this without concurrency, you have explicit fire transitions, you have explicit fire transitions see fire transitions you have explicit fire

transitions from all the states of our original unit control FSM to a new state called fire going down.

So, when you have a fire, you go to fire going down, in whichever floor you are you go to the first floor and you open the door after you go there. So, when up till you are going down to the first floor, your open is 0 your down is 1, ok ypour up is 0. So, you cannot go up when you have a fire, you have to go down to the first floor, your lift door or the elevator door will remain shut so, open is 0, but down is 1.

Now, from there when floor equals to 1 is reached. So, till floor is greater than 1 you move in this way you move in this self-loop and remain in fire going down. So, when you have you have come down to the first floor, then you open the door so, you are up is 0 you are down in 0. So, you remain stationary in the first floor, and you open the door and so, open is 1, ok. Until you have fire so after you do not have a fire then what happens? Then when the fire is extinguished you go back to the IDLE state of the original state machine.

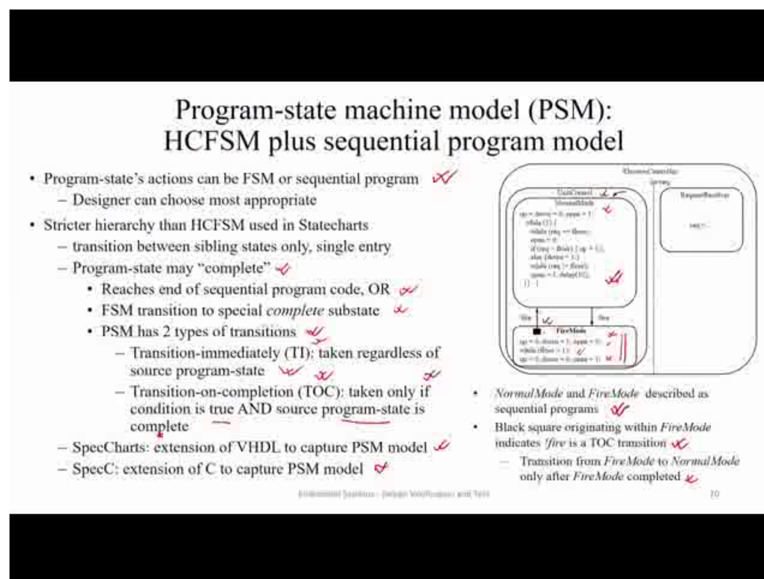
So, when you have fire, you will have explicit fire transitions from each of the states of the original state machine to the fire going down, in which case in the fire going down it will provide a lift or the elevator will progressively go down to the first floor, with the door remaining closed it will go down to the first floor, then it will remain stationary, open the door and remain in that state until fire is on, when the fire is extinguished it can take fresh requests so it will go back to the IDLE state of the original state machine.

So, without hierarchy, this whole thing so many transitions going on it is a bit make it is a bit messy. Now when we have hierarchy, we can model this whole FSM with 2 hierarchical states. So, you have we now have 2 composite states, one is the normal mode and the other is the fire mode. So, the normal mode is our original machine, and we move to the fire mode from the normal mode, whenever there is a fire we move from the normal mode to the fire mode. And whenever the fire is extinguished we go back to the initial state of the normal mode from the fire mode, ok. So, with hierarchy things become much more simpler to represent.

Now, how do we represent concurrently? This whole elevator controller with this unit control this so this is the unit control and request resolver, these 2 execute concurrently, because these are 2 modules which executes concurrently. Unit control and request

resolver execute concurrently together. Within unit control you have again have this normal mode and fire mode. So, this normal mode and fire mode these is an OR decomposition of unit control. And you can see off the elevated controller the unit control and req request resolver is an AND decomposition of the elevator controller because unit control and request resolver executes concurrently within, unit control normal mode and fire mode are or decomposed.

(Refer Slide Time: 78:30)



Now after HCFSM comes the program state machine model, it is an extension of the HCFSM, now within a state we can have sequential programs we can write programs instead of a state diagram. We need not have a state diagram inside a state now, the state can now constitute a program. This is how the program state machine extends HCFSM. Programs actions can be FSM or a sequential program. So, it can still be an FSM like HCFSM or it can be a program. The designer can choose whatever he wants.

It has a stricter hierarchy than the state charts in the in the sense that transition is now only allowed within sibling states. So, you if you have between sibling states; that means, you suppose you have a a particular. So, so unit control is a composite state within which sibling states are normal state normal mode and fire mode, these are 2 sibling states. So now, transitions can happen within only these sibling states. So, transition cannot happen between non sibling states. You cannot go in within inside a state within normal mode. Say, suppose this normal mode was implemented as an FSM,

and it had multiple states. You cannot go from a state from fire mode to a state inside the normal mode, you can only move between sibling states.

A program state may complete; that means, you go to the execute the end statement of the program reaches end of the sequential program or FSM transition can be a special complete sub state, where you mean that you want to immediately exit based on a condition. So, therefore, PSM has 2 types of transitions one is transition immediately. TI taken regardless of the source program state, or transition on completion where the program completes which would mean for us it goes to so suppose in fire mode the program completing will mean, that you go to the first floor, and only then if the fire is extinguished, then you move from after the program completes you can move to the to the next state.

So, transition can only be on completion so taken only if condition is true and source program is complete. So, fire is extinguished and you reach to the first floor, only then you can go to the normal mode otherwise not, if you take user TOC extension.

So, normal mode and fire mode here if you can see are described as sequential programs not as state machine. So, in the fire mode also you have up equals to 0 down equals to 1 open equals to 0, these variables initialized, while floor less while for floor get greater than 1, you wait and the elevator progressively moves down. So, when this condition becomes false that is floor one is reached, you make up equals to 0 down equals to 0; that means, you stay in the first floor and you open the door. So, this is a sequential program representation of the fire mode, ok.

The black square originating within fire mode indicates that not fire is a TOC transition. It says that not fire is a TOC transition, meaning that this transition will be will be excited only when the program completes; that means, you reach the first floor, you reach the you reach floor one and only then only then can this transition be excited. Transition from fire mode to normal mode only after fire mode is computed. So, this is what it means.

State chart is an extension of VHDL to capture the so state charts of the language which extends VHDL to capture the PSM model. Similarly, spec C is a newer language which extends C to capture the PSM model. So, with this we have learned the essential ways of modelling using state machines and the sequential program model. We have seen how

concurrent behaviour can be represented using hierarchical concurrent FSM and the program state machine and how things can be nicely all the entire embedded system behaviour can be represented in a structured and systematic manner. With this we come to the end of this lecture.