SMV Project

Arun Autuchirayll, Manjulata Chivukula, Bhargava Konidena

April 28, 2009

Indiana University - Department of Computer Science
P515 Specification and Verification
For: Dr. Steven Johnson

Table of Contents

1 Technical Profile **4**

2 Requirements **5**

2.1 Functionality 5

2.1.1 Elevator 5

2.1.2 Combination Lock 6

3 Design **8**

3.1 Elevator SCAN model 9

3.2 Elevator FIFO model 11

3.2.1 Environment 12

3.2.2 Controller 12

3.3 Elevator nearest neighbor first model 14

3.3.1 Controller state machine 16

3.3.2 State machine for cDestination 17

3.3.3 State machine for Door state 17

3.3.4 State machine for cFloor variable 18

3.4 Combination Lock 19

4 Implementation **21**

5 Testing **23**

5.1 Test specification 23

5.1.1 Elevator 23

5.1.2 Combination Lock 24

5.2 Test results 25

5.2.1 Elevator Models 25

5.2.2 Combination Lock Models 26

5.3 Observation 30

6 Conclusion **33**

7 Future Work **33**

8 Appendix **34**

8.1 Verilog Model 34

1 Technical Profile

Symbolic model verification(SMV) [2] is a tool that is used for model checking. It is aimed at reliable verification of industrially sized designs. The model check is performed based on Binary Decision Diagrams(BDDs). The goal of this project is to use SMV tool for specifying and verifying the correctness of following models:

- Basic elevator model that scans from the first to fifth floor stopping at every floor and vice versa.
- Elevator model that serves the requests using the first in first out(FIFO) scheduling algorithm.
- Elevator model that serves the requests using the nearest neighbor first scheduling algorithm.
- Combination Lock model that opens the lock if the correct combination of the inputs is specified and the protocol is followed.

2 Requirements

2.1 Functionality

2.1.1 Elevator

The basic function of the elevator is to serve passenger requests. The requests can be served either using nearest neighbor first or first in first out basis. The requests are expected to arrive non-deterministically. A request is defined as a change in the state of the buttons either inside the elevator or on the floor, to a high.

2.1.1.1 Scan Algorithm The requirements of the algorithm are enumerated as following:

- *Functionality*: Model of a naïve elevator controller that functions based on SCAN scheduling algorithm. The elevator moves from the bottom floor to the top floor and then back again serving all the requests in its way. This is similar to the SCAN Disk Scheduling algorithm, where requests are served in one direction from start to end and then in the other.
- *Assumptions*: It is assumed that the elevator stops at every floor during its upward and downward motion. And this restates the fact that requests are not generated and there is no environment for this model.
- *Post conditions*: The elevator is guaranteed to serve all requests in finite time.
- *Invariants*: Safety and Liveness conditions are ensured by the Elevator. A passenger entering the elevator eventually exits it and passengers waiting to enter the elevator eventually enter it. But, in this algorithm there is no requirement to model the environment.

2.1.1.2 FIFO Algorithm The requirements of the algorithm are enumerated as following:

- *Functionality*: Model of a elevator controller that serves the request on a first come first serve basis.
- *Assumptions*
  - The environment is constrained in terms of number of requests that can be generated in a instant. Only one request is generated by the environment per clock cycle. The possibility of sending concurrent requests has been eliminated.
  - The queue size at the controller that holds the requests is chosen to be small.
  - The elevator and the controller execute synchronously i.e., a single step of execution of the model implies single step in execution of both the controller and the environment modules.

  - The elevator model is designed for a system of 4 floors.
  - There are no door buttons. The environment cannot set the status for the door buttons. The status of the door is controlled by the controller.
  - There are no sensors in the environment that determine the position of the elevator cabin.
  - The requests in the environment have no bearing on the output of the controller.
  - Cabin door always remains closed except when the cabin reaches the desired destination and the door is eventually opened.
  - Elevator engine is not expected to breakdown at any floor or when serving a request.
  - Buttons are clean pulses which remain high for finite duration.

- *Post conditions*: The elevator should serve all the requests coming in, in a finite amount of time.
- *Invariants*: Safety and Liveness conditions should be ensured in the model.Elevator door should remain closed until the cabin reaches the desired floor, a passenger entering the elevator should eventually leave and request coming in should be eventually served.

2.1.1.3 Nearest Neighbor Following are the set of requirements for the nearest neighbor scheduler algorithm for the elevator:

- *Functionality*: Model of an elevator controller that functions based on the nearest neighbor scheduling algorithm. The elevator moves from one floor to another serving all the requests on its way in a nearest request first fashion.
- *Assumptions*: For simplicity, it is assumed that the UP and DOWN buttons on each floor outside the elevator are represented by a single request, which indicates that some passenger outside the elevator wants to enter the elevator and thereafter will press the button he/she wants to from inside the elevator. Also, the order of generation of requests is non-deterministic with FAIRNESS added to all the requests.
- *Post conditions*: The elevator picks the nearest request first from the queue of requests and serves it.
- *Invariants*: Safety and Liveness conditions are ensured by the Elevator. A passenger entering the elevator eventually exits it and passengers waiting to enter the elevator eventually enter.

2.1.2 Combination Lock

The basic components of the combination lock are shown in the Figure (1).Tables (1) and (2) describes the inputs to and outputs from the box respectively. The protocol that opens the lock is explained in following steps:

- Push RESET

- Set C to c1
- Push Enter
- Set C to c2
- Push Enter
- Set C to c3
- Push Enter
- Push Try

The objective of this project is verify the correctness of the model, the model that opens the lock if the protocol is obeyed. The environment is assumed to be totally non-deterministic. The values for enter, combination bits c1,c2 and c3, reset and try buttons can be any value 0,1. Following are the design requirements of the model:

- *Functionality*:The combination lock model can be specified as a sequential system consisting of the following inputs and outputs
- *Postconditions*:
  - Once the lock is open, it would remain open until it is reset.
  - Once a combination fails to open the lock, the lock needs to be reset before another attempt is made to open it.
- *Invariants*:
  - The lock should never open until the try button is pressed and released.
  - The lock should eventually open.
  - The lock should fail to open if the try button is pressed and released before the combination input is entered.

Inputs	Description
A RESET button	To allow the system to transition to a predetermined state.
A single toggle switch labeled C (COMBINATION)	To enter combination bit sequence.
A push button E (ENTER)	Is an indication to the system when to read the next combination bit
A push button TRY	Indicates to the system when the user is trying to open the lock

Table 1: Inputs and their description

Output	Description
An OPEN signal	To indicate the releases of the physical lock. This can be thought as a light that is on when the lock is open.

Table 2: Output and its description

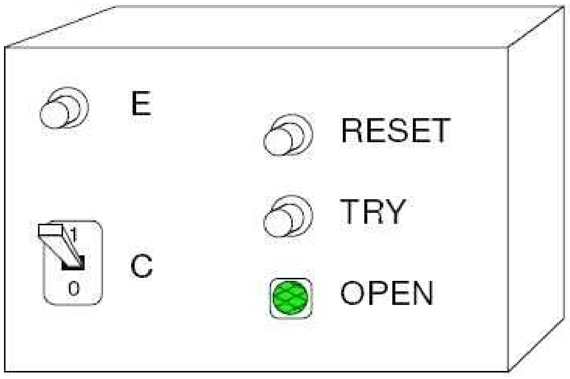


Figure 1: Combination Lock

3 Design

The design section is divided into various subsections, each of which describes the design of following SMV models whose requirements are described in section 2:

- Elevator Scan model;

- Elevator FIFO model;
- Elevator nearest neighbor first model;
- Combination lock model.

3.1 Elevator SCAN model

The system has a MAIN module which serves as the Elevator Controller. The controller is responsible for the movement of the elevator in upward or downward direction and also takes care of opening and closing of the doors. Following are the inputs to the controller:

1. cDestination: Input, indicates the value of the destination 1..5.
2. dState: Output, indicates the state of the door. Open,Close
3. cFloor: Output, indicates on which floor the elevator is currently is on can take either of the values in 1..5.
4. flag: Direction of motion of the elevator. The value of flag 0 indicates the upward motion and the value of the flag 1 indicates, lift is moving downwards.

The finite state automaton model of the controller contains following states and is described in the Figure (2).

1. Enter: Enter is the initial state of the controller. Automaton always unconditionally transitions to Leave state.
2. Leave: Indicating the elevator is ready to serve next request. From the Leave state based on the request the controller either transitions to MovingUp state or MovingDown state.
3. MovingUp: The state is used to indicate the movement of the elevator.The automaton transitions to Enter state if the difference between desired destination and the current floor is less than or equal to 1 indicating elevator has reached the destination floor.
4. MovingDown: The state is used to indicate the movement of the elevator. The automaton transitions to Enter state if the difference between the current floor and desired destination is less than or equal to 1 indicating elevator has reached the destination floor.

The automaton models for setting cDestination, dState, cFloor and flag is shown in the Figures (3),(4) and (5).

The cDestination determines the next destination for the elevator as described in Figure (3):

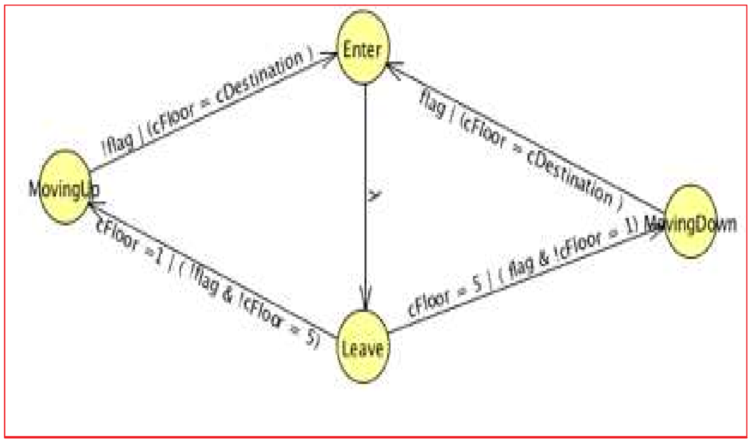

Enter
MovingUp
MovingDown
Leave
!flag | (cFloor = cDestination)
flag | (cFloor = cDestination)
cFloor =1 | (!flag & !cFloor = 5)
cFloor = 5 | (flag & !cFloor = 1)

Figure 2: Controller state machine

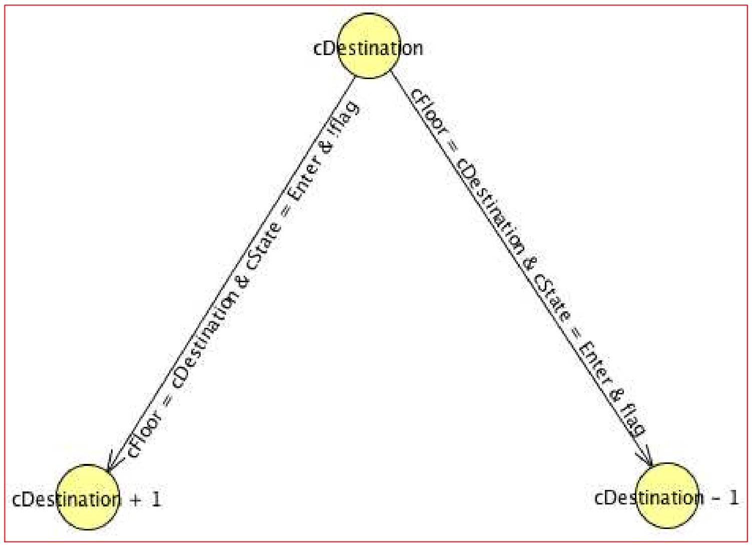

cDestination
cDestination + 1
cDestination - 1
cFloor = cDestination & cState = Enter & !flag
cFloor = cDestination & cState = Enter & flag

Figure 3: State machine automaton for cDestination variable

- It is incremented by 1 as long as the elevator is moving in the upward direction and doesn't reach the extreme end of that direction i.e. to the floor 5.
- It is decremented by 1 as long as the elevator is moving in the downward direction and doesn't reach the extreme end of that direction i.e. to the floor 1.
- The increments and decrements take place only when the controller is in Enter state and previous request has been served.

dState determines the door status of the elevator. It can be either in Open or Close state. As indicated in the Figure (4)

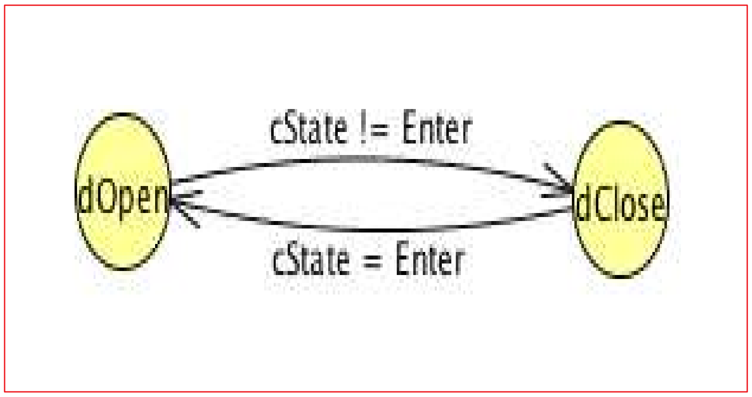


Figure 4: State machine automaton for dState variable

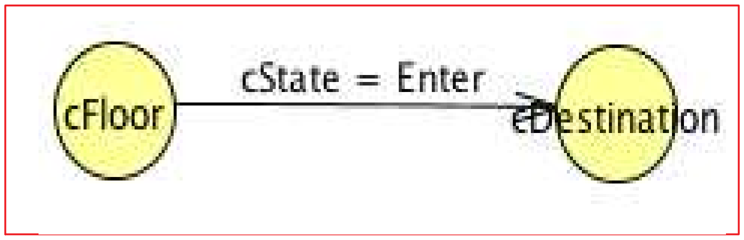


Figure 5: State machine automaton for cFloor variable

- dState remains Open as long as cState is in Enter state which tells us that the Elevator has just entered a floor and is not moving.
- dState gets the value of Close whenever the elevator is moving, which is indicated by the fact (cState ! = Enter).

cFloor defines the status of the floor in which the Elevator is currently on. And it attains the value of destination whenever cState equals Enter , i.e as soon as the Elevator reaches a floor. flag variable is used to determine the next direction of the elevator at every floor. It remains 0 throughout its journey from floor 1 to floor 5 and changes back to 0 at floor 5 to floor 1 and vice versa.

3.2 Elevator FIFO model

The components in the design of elevator model that serves the requests on basis of first in first out is shown in Figure (6). As illustrated in the figure there are three main components in the model:

- Environment;
- Controller;

- FIFO queue inside the controller.

3.2.1 Environment

The environment at any instant is responsible for non-deterministically selecting one of the value from

`{Up1,Up2,Dw2,Up3,Dw3,Dw4,1,2,3,4}`

the sets of values which act as a request to the controller.

3.2.2 Controller

Controller has following state machines:

- Inserting the value into the queue by checking the queue full condition and incrementing the tail pointer.
- Reading the value from the queue by checking the queue empty condition and incrementing the head pointer.
- Actual elevator controller state machine that is responsible to serve the request.

Controller contains two processes that are executing asynchronously to insert the value into the queue and reading the value from the queue. The writer process checks for **Queue Full** condition before inserting the request into the queue and the reader process checks for **Queue Empty** condition and reads the value from the queue. Figure (7) describes the state machine model of the writer process that inserts the value into the queue and the figure (8) describes the state machine model that retrieves the value from the queue. In addition, the value is read from the queue only after the controller state machine acknowledges that the previous request is served. **DestReach** variable is set to true when destination is reached.

Controller state machine: The state machine model for the actual elevator controller is described in the Figure (9). Controller state machine can be explained in following steps:

- Initial state is Waiting when the controller is waiting for the request and the door is closed. The destReach variable is set to 0, which states that the current request is yet to be served.
- Moves to Processing state when there is a request.

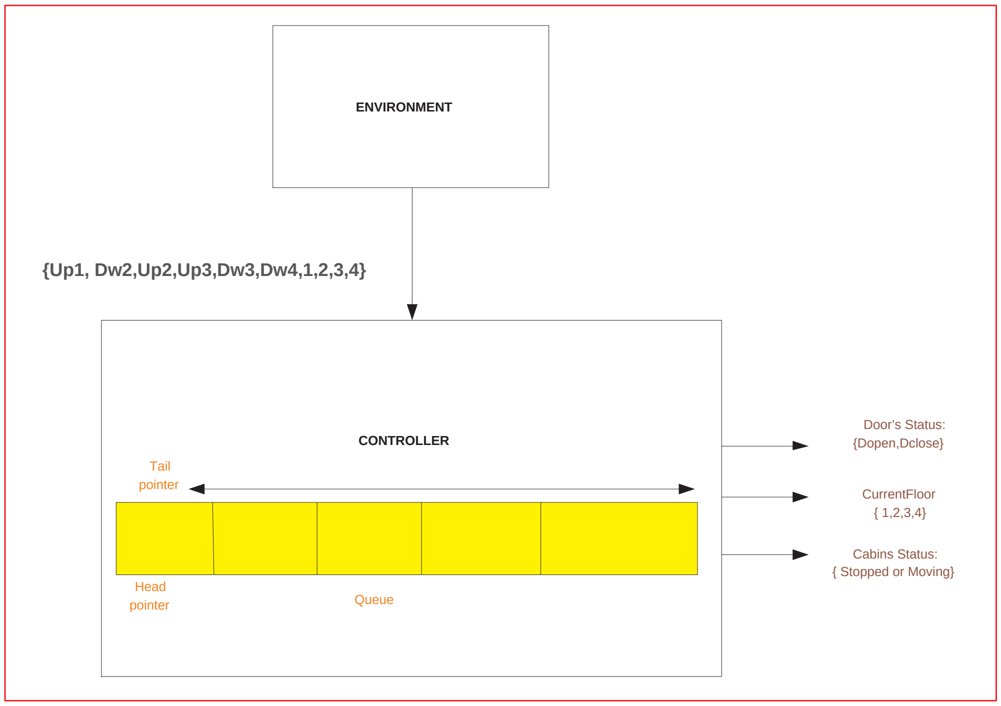

ENVIRONMENT
{Up1, Dw2,Up2,Up3,Dw3,Dw4,1,2,3,4}
CONTROLLER
Tail pointer
Head pointer
Queue
Door's Status: {Dopen,Dclose}
CurrentFloor { 1,2,3,4}
Cabins Status: { Stopped or Moving}

Figure 6: Components of Elevator FIFO model

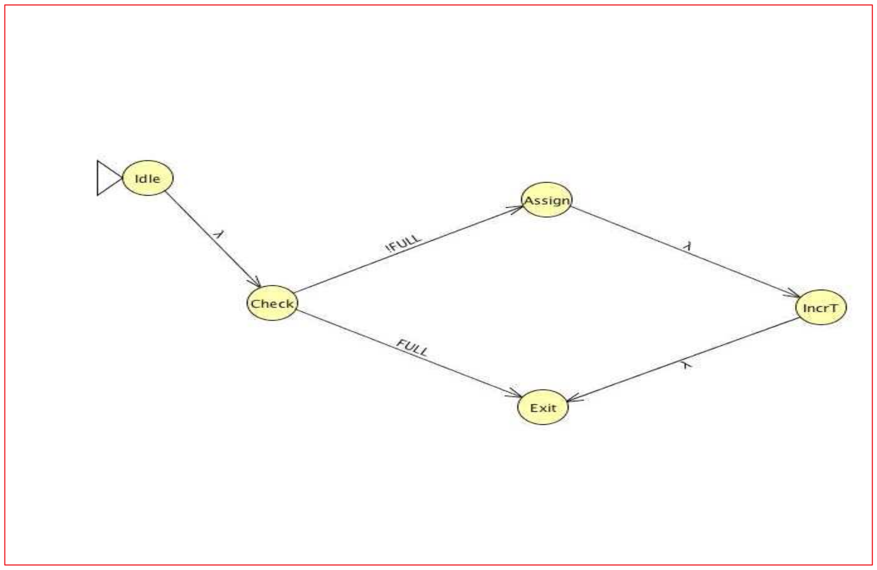


Figure 7: Queue insertion state machines

- Based on the request value and the current floor value the state machine moves to either MoveDown, MoveUp or Stop state. If the current floor and the destination are the same the next state is Stop. If the current floor is less then the destination the next state is MoveUp. If the current floor is greater than destination then the next state is set to MoveDown.

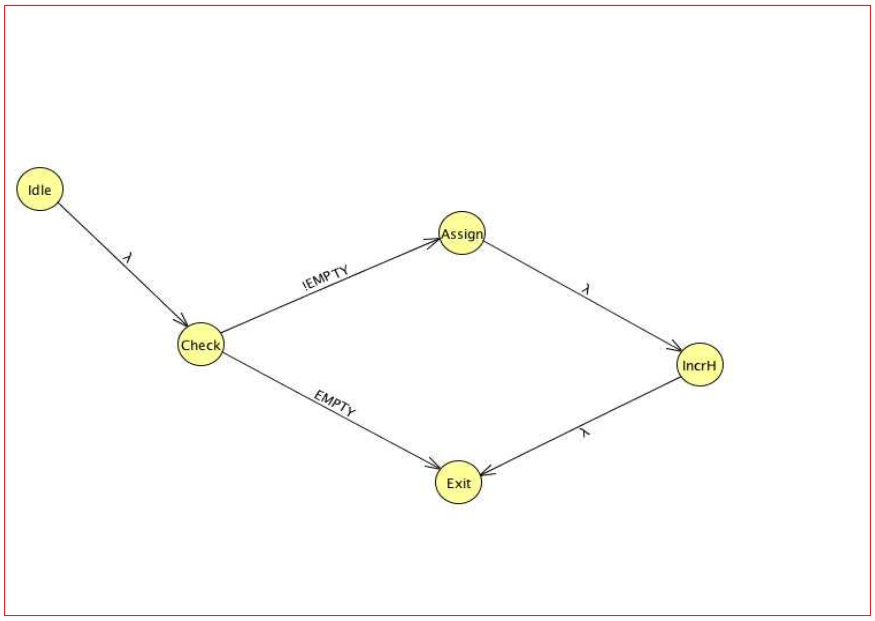


Figure 8: Queue reading state machines

- Moving state is used to capture actual movement of the elevator. Hence from MoveUp or MoveDown states, state machine transitions to Moving state unconditionally.
- In the Moving state the value of current floor is incremented or decremented until destination is reached.
- Once the current floor value is equal to or one less than the desired destination value the state machine transitions to Stop state.
- From Stop state the state machine unconditionally transitions to Open state to set the door status to Open.**destReach** variable and current floor value is updated with the destination floor to state the request has been served.
- From Open state the state machine goes back to initial state Waiting.

3.3 Elevator nearest neighbor first model

The system has a MAIN module and a CONTROLLER module which serves as the Elevator Controller. MAIN module is responsible for generation of two requests Req1 and Req2 in every cycle non-deterministically. Req1 represents all the requests from outside the elevator. As mentioned in the ’Assumptions’ section, both the UP and DOWN buttons are represented by a single request for simplicity. Req2 represents the requests from inside the elevator.
The CONTROLLER module is responsible for deciding which request to choose next as per the nearest neighbor scheduling and moves the elevator to the appropriate floor. It is also accountable for opening and

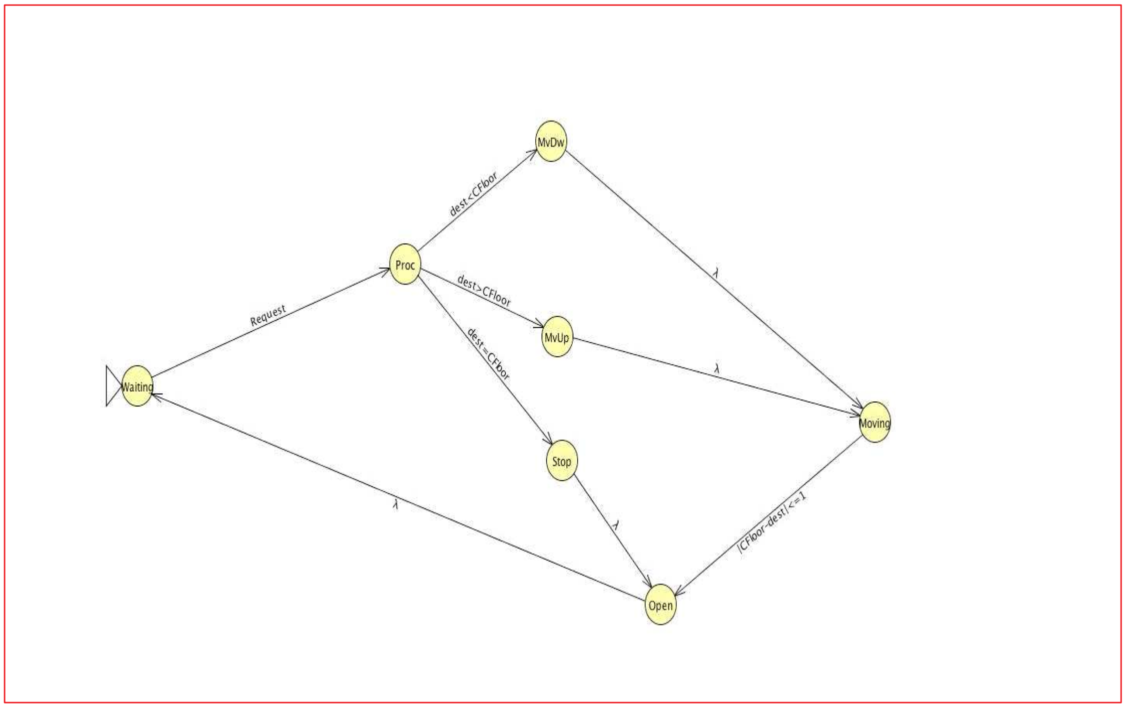


Figure 9: Finite Automaton model of the controller

closing of the doors. This module receives Req1 and Req2 as inputs from the MAIN module. Following state variables are defined for the main

- Req1
- Req2

Following state variables are defined for the controller.

- cState
- cDestination
- dState
- cFloor

In addition, the controller has following condition variables.

- REQ1: condition variable which has the absolute difference of the Req1 with cFloor.

- REQ2: condition variable which has the absolute difference of the Req2 with cFloor.
- ISREQ1: is set to 1 if Req1 is nearer to cFloor than Req2 is or otherwise set to 0.
- ISREQ2: is set to 1 if Req2 is nearer to cFloor than Req1 or otherwise set to 0.

Each state variable's automaton is described in detail below.

3.3.1 Controller state machine

Figure describes the finite state automaton for the controller (10). cState is responsible for determining the

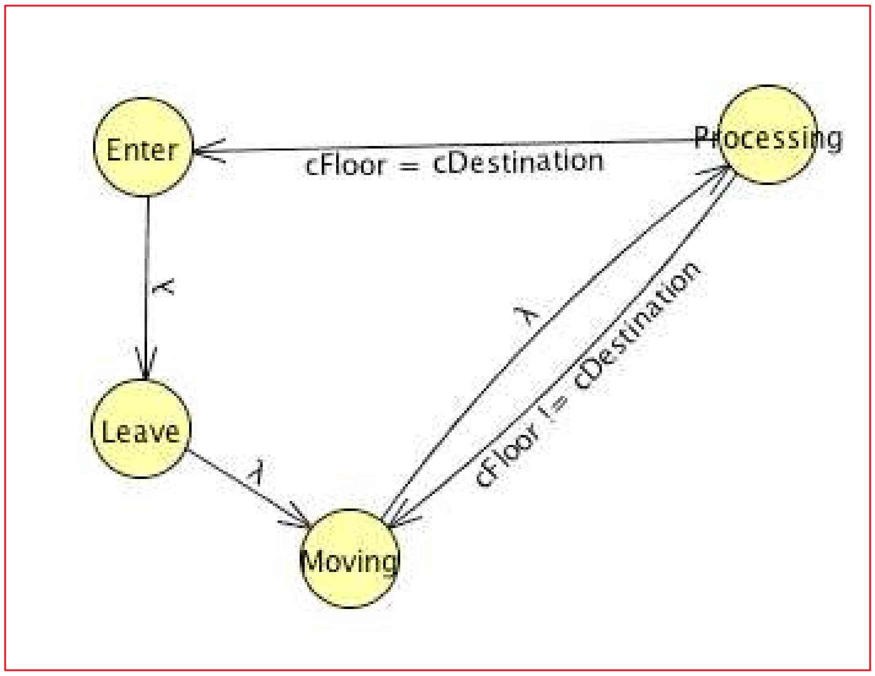


Figure 10: Finite State Automaton for the Controller

state of the controller. The controller has the following states:

- Enter
- Moving
- Processing

- Leave

Conditions that cause state transitions are described below:

- Enter is the initial state of the controller.
- The Elevator always transitions from Enter to Leave.
- Leave is always followed by Moving
- Processing follows Moving and determines whether to stop at the floor, i.e. checks if cFloor equals cDestination and decides whether to move to Enter state or continue with Moving state until cDestination is reached. It also sets cFloor to an appropriate value.
- Processing is followed by Enter if cFloor equals cDestination and if it not then is followed by Moving.

3.3.2 State machine for cDestination

cDestination determines the next destination for the elevator based on Req1 and Req2. Figure (11) describes the finite state automaton which sets the next state value for cDestination.

- It takes the value of Req1 as long as the elevator's state is Enter and the floor has reached its previous destination and ISREQ1 is 1.
- It takes the value of Req2 as long as the elevator's state is Enter and the floor has reached its previous destination and ISREQ2 is 1.

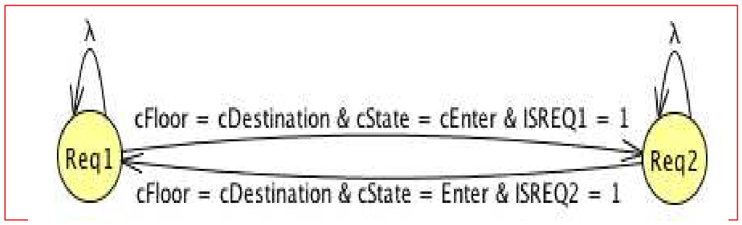

λ
λ
cFloor = cDestination & cState = cEnter & ISREQ1 = 1
Req1
Req2
cFloor = cDestination & cState = Enter & ISREQ2 = 1

Figure 11: Finite State Automaton for setting cDestination variable

3.3.3 State machine for Door state

dState determines the door status of the elevator. And it has two values

- Open
- Close

Figure (12)) finite state automaton.

- State remains Open as long as cState is in Enter state which tells us that the Elevator has just entered a floor and is not moving.
- dState gets the value of Close whenever the elevator is moving which is indicated by the fact cState ! = Enter.

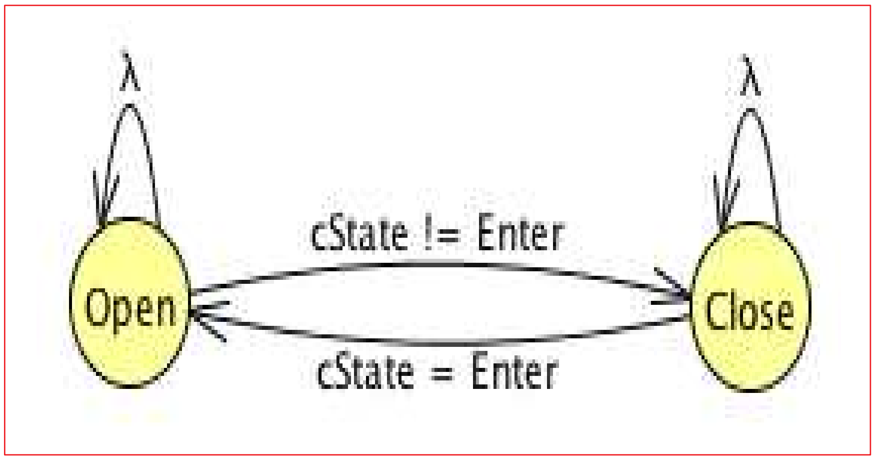


Figure 12: Finite State Automaton for setting Door status variable

3.3.4 State machine for cFloor variable

cFloor defines the status of the floor in which the Elevator is currently in. Figure (13) shows the finite state automaton that sets the value for cFloor variable.

- And it attains the value of destination whenever cState equals Processing , i.e and cFloor has indeed reached cDestination.
- It is incremented by 1 as long as the elevator is in its Processing state and cFloor is less than cDestination and it has not reached the extreme end of that direction yet, which is indicated by the cFloor ¡ 4.
- It is decremented by 1 as long as the elevator is in its Processing state and cFloor is greater than cDestination and it has not reached the extreme end of that direction yet, which is indicated by the cFloor ¿ 1.

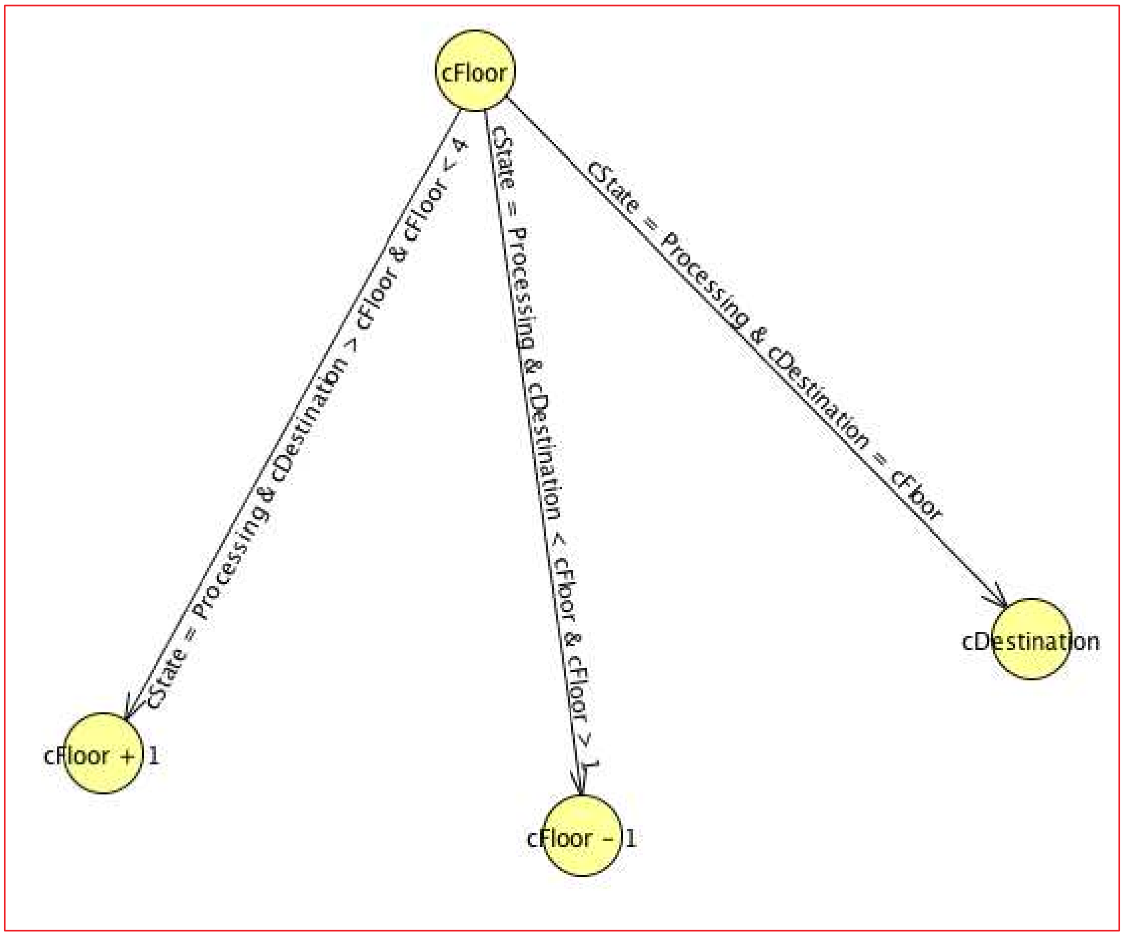

cFloor
cState = Processing & cDestination > cFloor & cFloor < 4
cState = Processing & cDestination < cFloor & cFloor > 1
cState = Processing & cDestination = cFloor
cFloor + 1
cFloor - 1
cDestination

Figure 13: Finite State Automaton for setting cFloor variable

3.4 Combination Lock

Different combination lock models have been devised in order to analyze, understand and demonstrate the lock's behavior as a black box. The description of each of the models is enumerated below:

- Combination lock model 1: The model is shown in Figure (14). This is a correct model which means it would open when the correct combination is entered and steps of the protocol are followed.
- Combination lock model 2: The model is shown in Figure (15). This is a flawed model. The flaw is the illustrates by possibility for opening the lock irrespective of the values of input combination bits. The model is simulated by using a 64 bit counter, that acts as a delay for opening a lock irrespective of the path chosen.
- Combination lock model 3: The model is shown in Figure (16). This is also a flawed model. The flaw is the non-deterministic possibility of the lock failing to open irrespective of entered input combination.

- Combination lock model 4: The model is shown in Figure (17). This is a flawed model. The flaw is the non-deterministic possibility of either the lock opening or failing to open and this happens irrespective of the combination entered.
- Combination lock model 5: The model is shown in Figure (14). This is a correct model in which the environment is constrained to generate the correct combination which would eventually open the lock. The environment in this model can be treated as a component which has observed the user entering the right combination and opening the lock. Therefore it is presumed that the environment produces the same combination which would eventually open the lock.

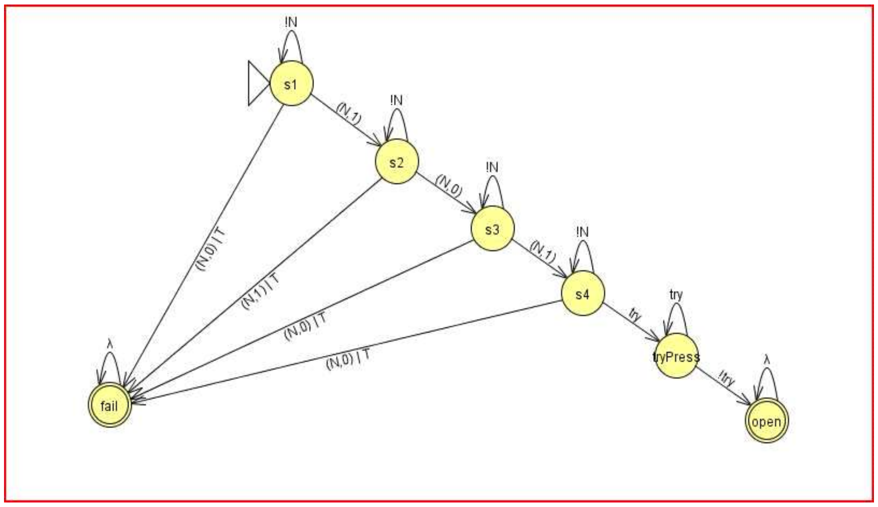


Figure 14: Combination Lock Model 1

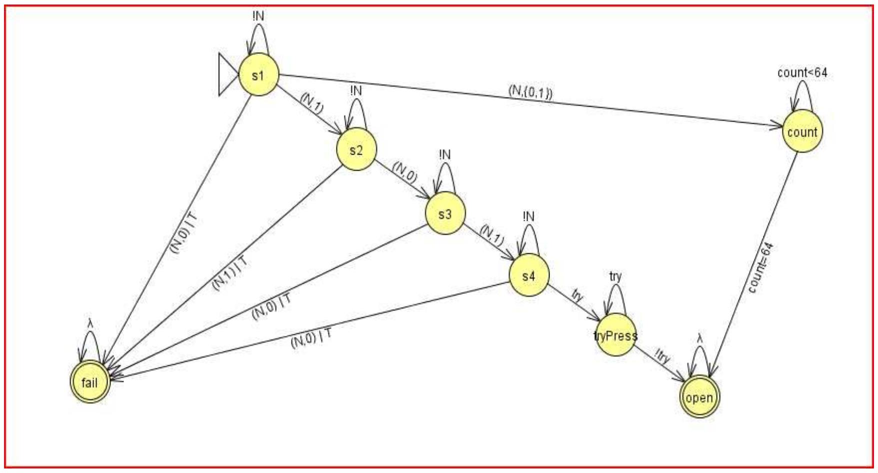


Figure 15: Combination Lock Model 2

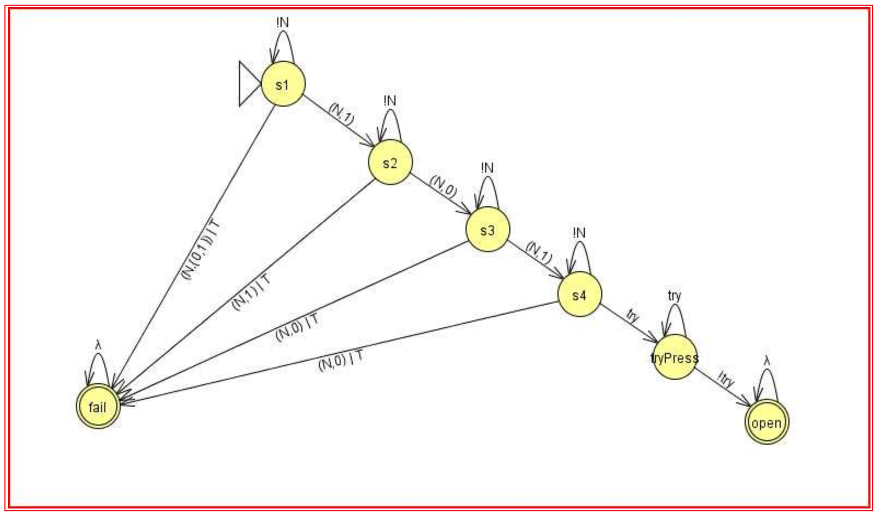


Figure 16: Combination Lock Model 3

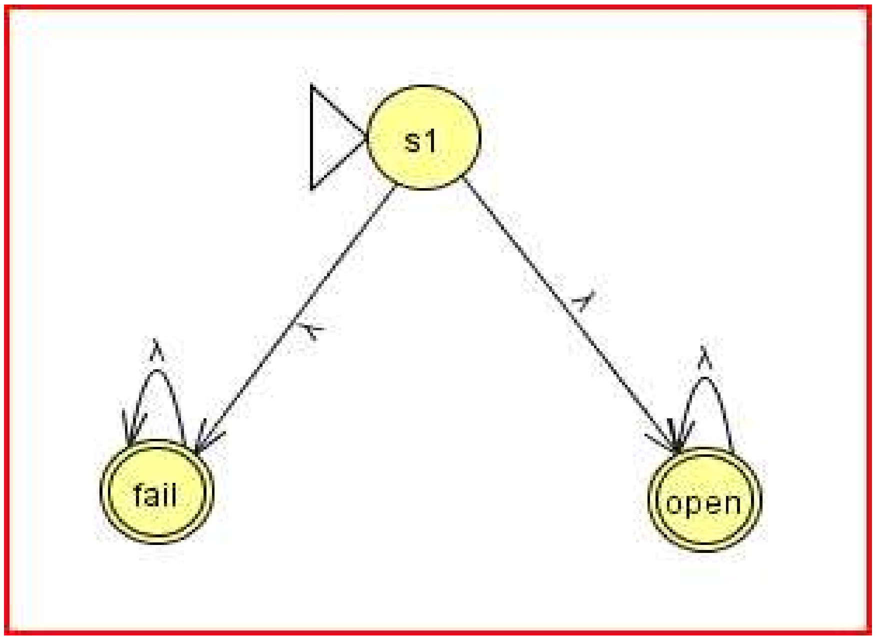


Figure 17: Combination Lock Model 4

4 Implementation

SMV source code files [7] [8] and their description is shown in the Table (3) and (4)for the directory Elevator and Combination Lock. Inaddition, the source files contains Computational tree logic(CTL) [5] specifications statements that are used verify the model behavior.

File	Description
Elevator/ElevScan.smv	File contains SMV model of the Elevator Scan algorithm and CTL SPEC statements that verify the model
Elevator/ElevFIFO.smv	File contains SMV model of the Elevator FIFO algorithm and CTL SPEC statements that verify the model behavior
Elevator/ElevatorNN.smv	This file contains the description of the SMV model for the Elevator Nearest Neighbor Scheduling algorithm.

Table 3: Source files and their descriptions: Elevator Directory

File	Description
model1/C-Lock1.smv	File contains the correct model of the combination lock
model2/C-Lock2.smv	File contains the flawed model of the combination lock. From state s1 there is a non-deterministic transition which opens the lock after some delay (modelled by a counter)
model3/C-Lock3.smv	File contains the flawed model of the combination lock. From state s1 there is a non-deterministic transition which fails to open the lock
model4/C-Lock4.smv	File contains the flawed model of the combination lock. From state s1 there are two non-deterministic transitions possible - One which opens the lock and the other which fails to open the lock. All this happens irrespective of the input combination
model5/C-Lock5.smv	File contains a model of combination lock in which the environment is constrained to generate a sequence of inputs which open the lock
Verilog/Clock.v	File contains the verilog model of the combination lock.

Table 4: Source files and their descriptions: Combination Lock Directory

5 Testing

5.1 Test specification

5.1.1 Elevator

5.1.1.1 Elevator Scan Following are the CTL specification statements that verify the behavior of the elevator scan algorithm.

- If the state machine is in the enter state it will eventually move to MovingUp or MovingDown state to serve the request. The state transitions indicates the elevator is serving the request.

  ```
  (AG ((cState=Enter) ->(AF (cState=MovingUp))))
  ```

- If the door is Open on reaching the destination it will eventually gets closed to process next request.

  ```
  (AG ((dState=Open) ->(AF (dState=Close))))
  ```

- The lift is floor 1 will move to floor 5 eventually.

  ```
  (AG ((cFloor=1) ->(AF (cFloor=5))))
  ```

- If there is request for floor 2 the lift will eventually move to floor 2 and state is opened.

  ```
  (AG ((cDestination=2) ->(AF ((cFloor=2)&(dState=Open)))))
  ```

- The lift is floor 5 will eventually come down to floor 1.

  ```
  (AG ((cFloor=5) ->(AF (cFloor=1))))
  ```

5.1.1.2 Elevator FIFO Following CTL specification statements are written to verify the behavior of the model.

- If there is a requests from Floor 2 the request is eventually served and the door is opened.

  ```
  AG(nextReq = Up2 -> AF(CFloor=2 & doorStatus=Open))
  ```

- Door will be open globally always.

  ```
  !AG(doorStatus=Open)
  ```

5.1.1.3 Elevator Nearest Neighbor CTL Specifications used to test and verify the Elevator nearest neighbor scheduling model are:

- SPEC

  ```
  AG(elev.cState = Enter -> AF(elev.cState = Moving))
  ```

  Description - Whenever the Elevator is Ready to move, it moves. (It doesnt fail to move)

- SPEC

  ```
   AG(elev.dState = Open -> AF(elev.dState = Close))
  ```

  Description - Whenever the Door of the elevator is open, it is closed infinitely immediately often.

- SPEC

  ```
   AG(elev.cFloor = 1 -> AF(elev.cFloor = 3))
  ```

  Description - In all the paths in which the elevator is currently in 1 floor, it reaches floor 3 sometime later.

- SPEC

  ```
  AG(!(elev.dState = Open & elev.cState = Moving ))
  ```

  Description - The elevator's door is never open while moving.

5.1.2 Combination Lock

Specification statements used to test and verify all the combination lock models are:

- There exists a possibility that the lock would never open.
  EF(!lockOpen)
- The lock never opens.
  AG(!lockOpen)

- The specified combination opens the lock.
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try)))))))))$\rightarrow$ AF(lockOpen))
- The lock opens only after try is pressed and released.
  AG(lockOpen $\rightarrow$ A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

5.2 Test results

5.2.1 Elevator Models

5.2.1.1 Elevator Scan All the CTL spec statements returned *true* describing the correctness of the model.

5.2.1.2 Elevator FIFO

- It was found the first SPEC statement returned false. Trace in the Figure (18 shows an infinite loop where reader next adds the value to the queue for the reader to read the value from the queue and process. Debugging from counter scenario was found to be difficult in this model as it is limitation of the SMV to model queue behavior.
- Second spec statement returned false. The counter scenario is shown in the Figure (19) which shows door goes to closed state in some path.

5.2.1.3 Elevator Nearest Neighbor

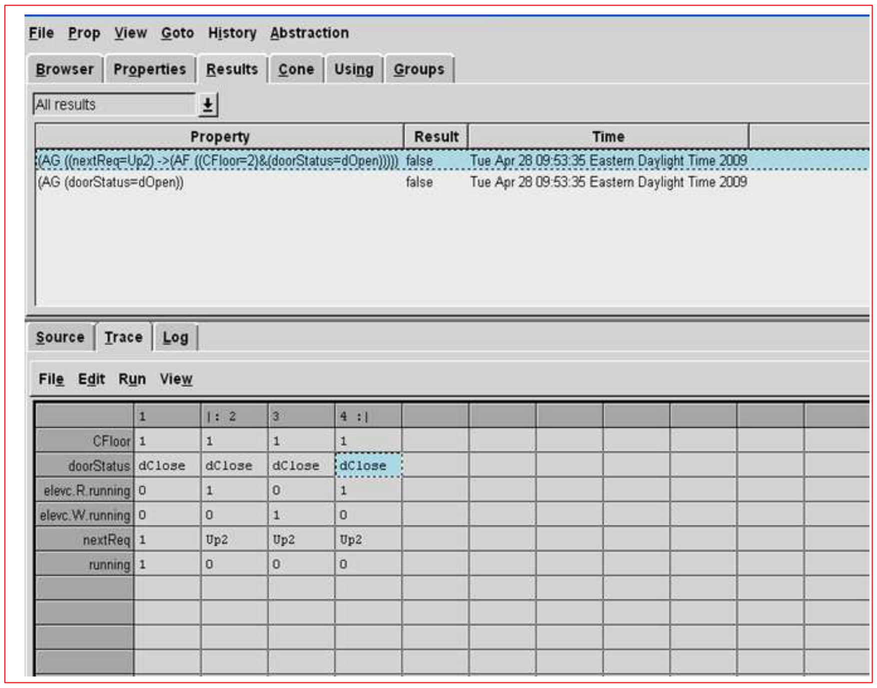
- SPEC

  ```
  AG(cState = Enter -> AF(cState = MovingUp))
  ```

  Result - SMV verifies the result to be true.
- SPEC

  ```
  AG(dState = Open -> AF(dState = Close))
  ```

  Result - SMV verifies the result to be true.


File Prop View Goto History Abstraction
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View

Property	Result	Time
(AG ((nextReq=Up2) ->(AF ((CFloor=2)&(doorStatus=dOpen)))))	false	Tue Apr 28 09:53:35 Eastern Daylight Time 2009
(AG (doorStatus=dOpen))	false	Tue Apr 28 09:53:35 Eastern Daylight Time 2009

	1	|: 2	3	4 :|
CFloor	1	1	1	1
doorStatus	dClose	dClose	dClose	dClose
elevc.R.running	0	1	0	1
elevc.W.running	0	0	1	0
nextReq	1	Up2	Up2	Up2
running	1	0	0	0

Figure 18: SPEC 1 Failed Result of Elevator FIFO model

- SPEC

  `AG(cFloor = 1 -> AF(cFloor = 5))`

  Result - SMV verifies the result to be False. The circled area in the Figure (20) shows a counter scenario where the elevator keeps on looping between floor 1 and floor 2. Because floor1 and floor2 alternatively happen to be the nearer requests at each cycle, other requests are totally ignored and they might starve to death.

- SPEC

  `AG(cFloor = 5 -> AF(cFloor = 1))`

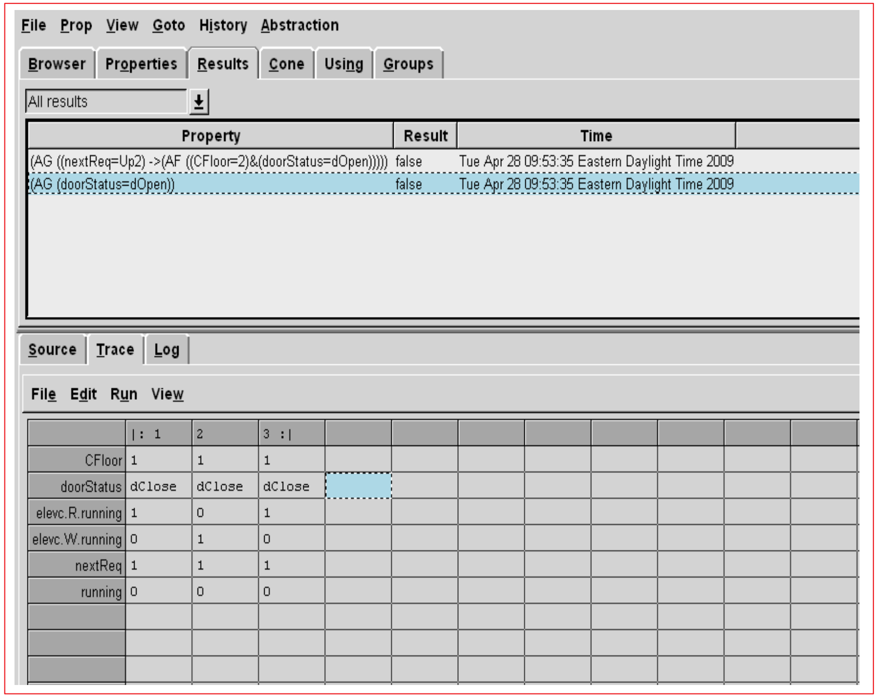
  Result - SMV verifies the result to be true.

5.2.2 Combination Lock Models

- Combinations lock model 1

  - SPEC1

    `EF(!lockOpen)`


File Prop View Goto History Abstraction
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View

Property	Result	Time
(AG ((nextReq=Up2) ->(AF ((CFloor=2)&(doorStatus=dOpen)))))	false	Tue Apr 28 09:53:35 Eastern Daylight Time 2009
(AG (doorStatus=dOpen))	false	Tue Apr 28 09:53:35 Eastern Daylight Time 2009

	|: 1	2	3 :|
CFloor	1	1	1
doorStatus	dClose	dClose	dClose
elevc.R.running	1	0	1
elevc.W.running	0	1	0
nextReq	1	1	1
running	0	0	0

Figure 19: SPEC 2 Failed Result of Elevator FIFO model

Result - SMV verifies the specification to be *true*

– SPEC2 AG(!lockOpen)

Result - SMV verifies the specification to be *true*

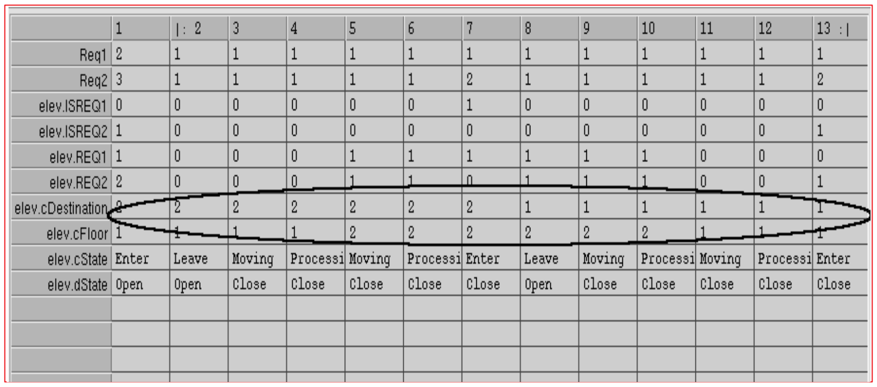
- SPEC3
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try)))))))))$\rightarrow$ AF(lockOpen))

  Result - SMV verifies the specification to be *true*

- SPEC4
  AG(lockOpen $\rightarrow$ A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

  Result - SMV verifies the specification to be *true*

Combinations lock model 2



	1	|: 2	3	4	5	6	7	8	9	10	11	12	13 :|
Req1	2	1	1	1	1	1	1	1	1	1	1	1	1
Req2	3	1	1	1	1	1	2	1	1	1	1	1	2
elev.ISREQ1	0	0	0	0	0	0	1	0	0	0	0	0	0
elev.ISREQ2	1	0	0	0	0	0	0	0	0	0	0	0	1
elev.REQ1	1	0	0	0	1	1	1	1	1	1	0	0	0
elev.REQ2	2	0	0	0	1	1	0	1	1	1	0	0	1
elev.cDestination	2	2	2	2	2	2	2	1	1	1	1	1	1
elev.cFloor	1	1	1	1	2	2	2	2	2	2	1	1	1
elev.cState	Enter	Leave	Moving	Processi	Moving	Processi	Enter	Leave	Moving	Processi	Moving	Processi	Enter
elev.dState	Open	Open	Close	Close	Close	Close	Close	Open	Close	Close	Close	Close	Close

Figure 20: Failed Result leading to Deadlock

- SPEC 1

  `EF(!lockOpen)`

  Result - SMV verifies the specification to be *true*

- SPEC 2

  `AG(!lockOpen)`

  Result - SMV verifies the specification to be *false*. The counter scenario is shown in (21)and (22). Please note that the trace is too big to fit in one figure, hence is captured in two screen shots - (21) and (22), in that order. In (22), the cells highlighted in green represent the looping and the cells highlighted in blue indicate when lockOpen becomes true.

- SPEC 3
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try)))))))))$\rightarrow$ AF(lockOpen))

  Result - SMV verifies the specification to be *true*

- SPEC 4
  AG(lockOpen $\rightarrow$ A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

  Result - SMV verifies the specification to be *true*

Combinations lock model 3

- SPEC 1
  EF(!lockOpen)
  Result - SMV verifies the specification to be *true*

- SPEC 2
  AG(!lockOpen)

  Result - SMV verifies the specification to be *true*

- SPEC 3
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try)))))))))→ AF(lockOpen))

  Result - SMV verifies the specification to be *true*

- SPEC 4
  AG(lockOpen → A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

  Result - SMV verifies the specification to be *true*

Combinations lock model 4

- SPEC 1
  EF(!lockOpen)
  Result - SMV does not show any result for the given specification. As can be seen from Figure8 there is no result available for this specification.
- SPEC 2
  AG(!lockOpen)
  Result - SMV verifies the specification to be *false*. The counter scenario is shown in Figure8. In Figure8, the cells highlighted in blue indicate when lockOpen becomes true.
- SPEC 3
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try)))))))))→ AF(lockOpen))

  Result - SMV does not show any result for the given specification. As can be seen from Figure8 there is no result available for this specification.

- SPEC 4
  AG(lockOpen → A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

Result - SMV does not show any result for the given specification. As can be seen from (23) there is no result available for this specification.

Combinations lock model 5

- SPEC 1
  EF(!lockOpen)
  Result - SMV verifies the specification to be *true*

- SPEC 2
  AG(!lockOpen)
  Result - SMV verifies the specification to be *false*. The counter scenario is shown in Figure9. In Figure9, the cells highlighted in green represent the looping and the cells highlighted in blue indicate when lockOpen becomes true.

- SPEC 3
  AF(enter & input & !try & AX(!enter & input & !try & AX(enter & !input & !try & AX(!enter & !input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & !try & AX(!enter & input & !try & AX(enter & input & try & AX(!enter & input & !try))))))))))→ AF(lockOpen))

  Result - SMV verifies the specification to be *true*

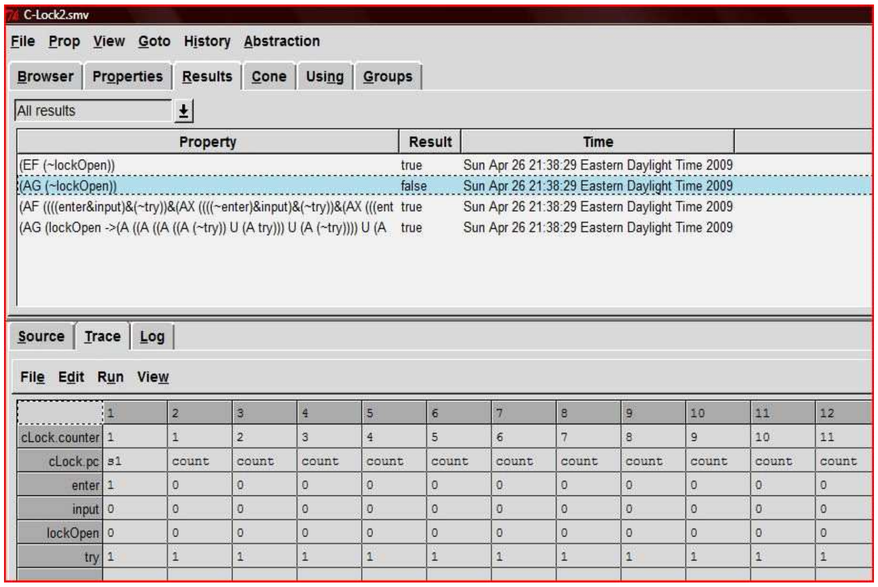
- SPEC 4
  AG(lockOpen → A[A(A(A(!try) U A(try)) U A(!try)) U A(lockOpen)])

  Result - SMV verifies the specification to be *true*

5.3 Observation

All the SMV models developed as a part of the project are used to highlight the model's behavior and help in verification of the model. SMV is not meant to estimate the performance of the model in terms of time and space. Following are the observations which we noticed during the development of the project:

- Among all the models that were developed, elevator SCAN and Combination lock were mostly straight forward in terms of modeling.
- **In elevator FIFO model-** The most realistic model for this would use the synchronous approach. But it was noticed during the course of development that it is next to impossible to model synchronous


C-Lock2.smv
File Prop View Goto History Abstraction
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View

Property	Result	Time
(EF (~lockOpen))	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AG (~lockOpen))	false	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AF ((((enter&input)&(~try))&(AX ((((~enter)&input)&(~try))&(AX (((ent	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AG (lockOpen ->(A ((A ((A ((A (~try)) U (A try))) U (A (~try)))) U (A	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009

	1	2	3	4	5	6	7	8	9	10	11	12
cLock.counter	1	1	2	3	4	5	6	7	8	9	10	11
cLock.pc	s1	count	count	count	count	count	count	count	count	count	count	count
enter	1	0	0	0	0	0	0	0	0	0	0	0
input	0	0	0	0	0	0	0	0	0	0	0	0
lockOpen	0	0	0	0	0	0	0	0	0	0	0	0
try	1	1	1	1	1	1	1	1	1	1	1	1

Figure 21: SMV trace for SPEC AG(!lockOpen)

behavior for environment and controller, since they share common queue. Following Queue models have been modeled:

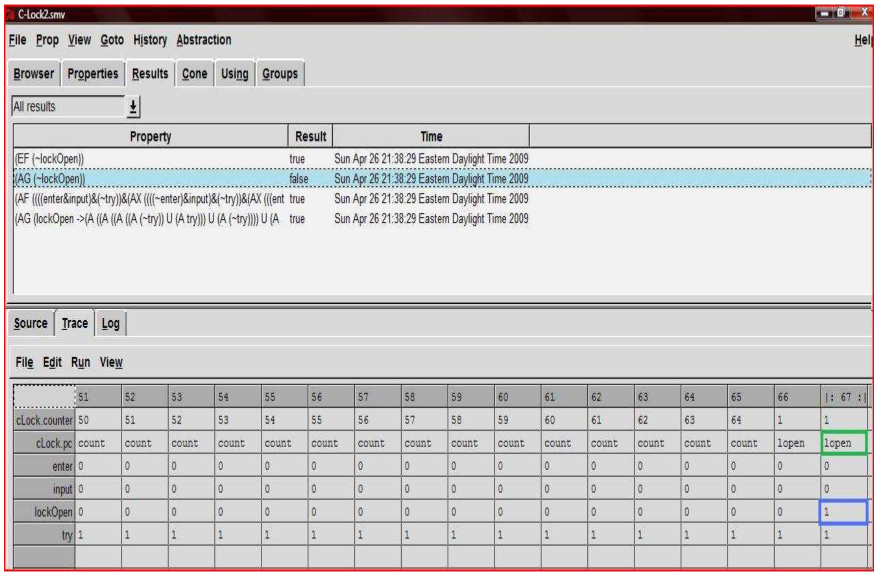
– Peterson's algorithm for mutual exclusion.
– Reader Writer approach.
– Shifting queue elements.

To work around this we made the queue to exist only in the controller while the environment executes synchronously generating a request every clock cycle. But this was not very fruitful because it was found difficult to model a queue behavior with reader and writer processes asynchronously running in the controller. Couple of times we also encountered segmentation faults.

- **Nearest Neighbor model**: Since we found modeling a queue was a limitation in SMV, we modeled without queue by generating two requests every clock cycle and developed a scheduler model that picked the nearest neighbor. Since nearest neighbor causes only nearest requests to be served frequently, it might lead to starvation of the floors which are farther from the cabin's current position. One of the test results showed deadlock scenario between two floors causing starvation to other floors requests which eventually led to the failure of the test case.

- **Combination Lock model**: Model5 was peculiar as it involved constraining the environment to generate the correct combination which eventually opens the lock. Verifying the correctness of the model using the CTL SPEC statements was found to be a difficult task than modeling. It gave us the experience to see that sometimes it is easier to model but difficult to verify a model's behavior.

- **Two Elevator Scheduler model** [10]:What scheduling approach to take in case there are multiple elevators present in a building? Consider a scenario in which there are 2 elevators present in a building having N floors. All the paths which the elevator can traverse can be represented in a graph as shown



C-Lock2.smv

Property	Result	Time
(EF (~lockOpen))	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AG (~lockOpen))	false	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AF ((((enter&input)&(~try))&(AX ((((~enter)&input)&(~try))&(AX (((ent	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009
(AG (lockOpen ->(A ((A ((A ((A (~try)) U (A try))) U (A (~try)))) U (A	true	Sun Apr 26 21:38:29 Eastern Daylight Time 2009

	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	|: 67 :|
cLock.counter	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	1	1
cLock.pc	count	count	count	count	count	count	count	count	count	count	count	count	count	count	count	lopen	lopen
enter	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
input	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lockOpen	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
try	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 22: SMV trace for SPEC AG(!lockOpen)

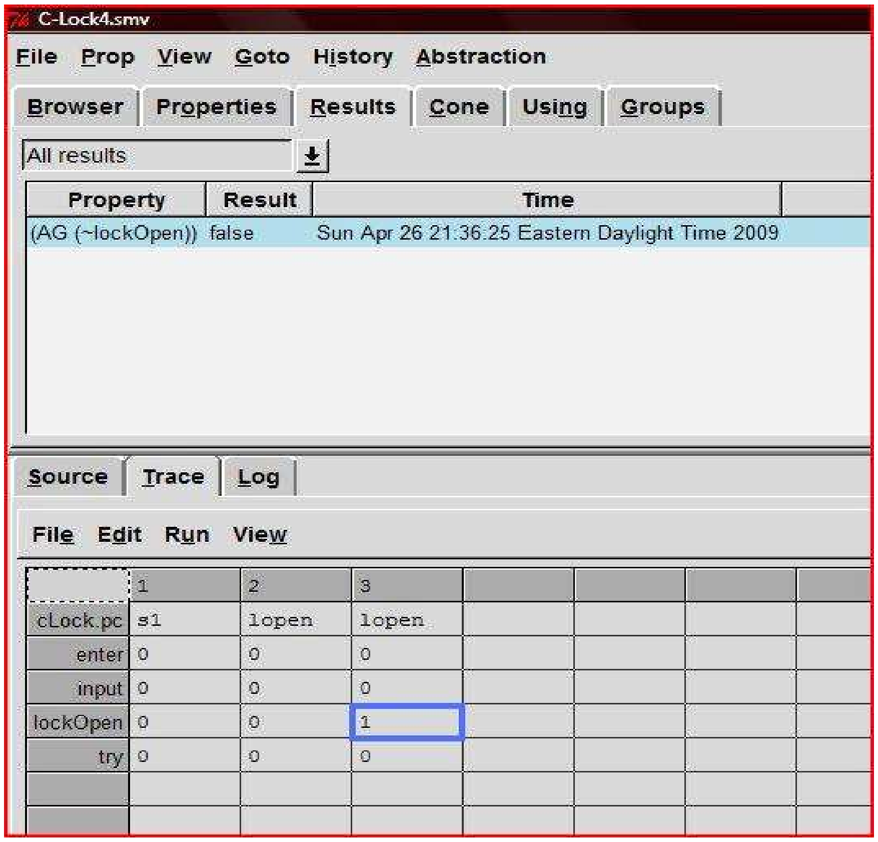
in Figure (25). In the graph the elevator position at floor 1, 2, 3, . . . , N-1, N is shown by vertices V_1, V_2, V_3, . . . , V_{N-1}, V_N
The weight between each node of the graph can be represented as

$$\omega|j-i| = (|j-i|)T_T + T_S \tag{1}$$

where, i : elevator present floor position, j : elevator destination floor position
$|j-i|$: total no. of floors of elevator movement
T_T :elevator traveling time between two consecutive floors
T_S: elevator stopping time at a floor
The output of the graph, given by sum of the graph weights thus represents the total traveling time of the elevator, i.e.

$$G(E) = \sum j - 1 = 1N - 1\omega_{|j-i|}(2)$$

For a building with 2 elevators, 2 similar graphs as shown in Figure (25) can be duplicated representing all 2 elevator travel paths. The total traveling time of all the elevators can thus be calculated by summing up each of the elevator traveling time as G(E1, E2)= G(E1) + G(E2). The optimal travel path is thus given by the minimum total traveling time of both the elevators with all initial conditions and requirements satisfied, i.e. Optimal Travel Path = G (E1, E2)min.


C-Lock4.smv
File Prop View Goto History Abstraction
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View

Property	Result	Time
(AG (~lockOpen))	false	Sun Apr 26 21:36:25 Eastern Daylight Time 2009

	1	2	3
cLock.pc	s1	lopen	lopen
enter	0	0	0
input	0	0	0
lockOpen	0	0	1
try	0	0	0

Figure 23: SMV trace for SPEC AG(!lockOpen)

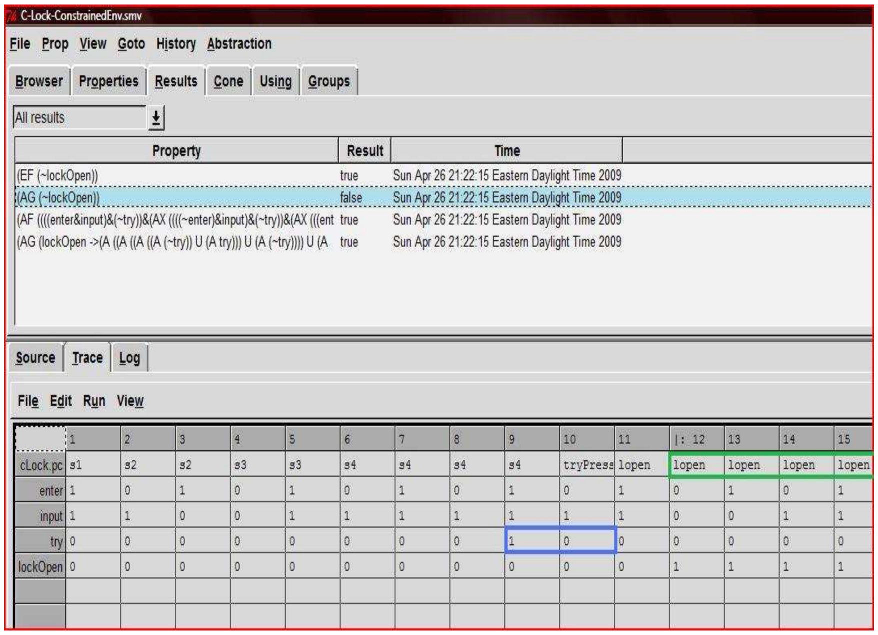
6 Conclusion

This project in SMV proved to be a good learning experience. We were able to comprehend the abilities and limitations of SMV. Modelling and verifying the models by writing CTL specification statements was a rich experience in its own. We also found NuSMV which is another symbolic model verification tool and it gave a better readability in terms of the trace of the counter scenarios [3].

7 Future Work

On completion of this project we felt following could be tasks that could be further developed in future:

- Elevator model with FIFO queue could be refined further by ensuring all the requests that are inserted in the queue are served. There could be further research on why SMV fails to model the queue by using strong CTL statements that support the argument.


C-Lock-ConstrainedEnv.smv
File Prop View Goto History Abstraction
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View

Property	Result	Time
(EF (~lockOpen))	true	Sun Apr 26 21:22:15 Eastern Daylight Time 2009
(AG (~lockOpen))	false	Sun Apr 26 21:22:15 Eastern Daylight Time 2009
(AF ((((enter&input)&(~try))&(AX ((((~enter)&input)&(~try))&(AX (((ent	true	Sun Apr 26 21:22:15 Eastern Daylight Time 2009
(AG (lockOpen ->(A ((A ((A ((A (~try)) U (A try))) U (A (~try)))) U (A	true	Sun Apr 26 21:22:15 Eastern Daylight Time 2009

	1	2	3	4	5	6	7	8	9	10	11	|: 12	13	14	15
cLock.pc	s1	s2	s2	s3	s3	s4	s4	s4	s4	tryPress	lopen	lopen	lopen	lopen	lopen
enter	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
input	1	1	0	0	1	1	1	1	1	1	1	0	0	1	1
try	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
lockOpen	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Figure 24: SMV trace for SPEC AG(!lockOpen)

- An ideal elevator scheduling algorithm could be developed after investigating the failed cases of both FIFO and Nearest Neighbor algorithm that are modeled. This can be done by using advantages of Nearest neighbor and FIFO and overcoming the limitation of Nearest Neighbor i.e. farthest request gets starved and limitation of FIFO i.e. closest request might need to wait long before it gets served, are eliminated.
- An elevator which treats all floors between the top and bottom floor as same can be modeled and verified.
- Combination Lock model can be verified by testing on FPGA(Field Programmable Gate Array) by programming.

8 Appendix

8.1 Verilog Model

Verilog model of the Combination lock problem is implemented simulated to verify the model. The state machine of the verilog model can be described in the Figure (26).Table (5) discusses the description of the states used in the modeling the problem in verilog.

Two simulation results are described.

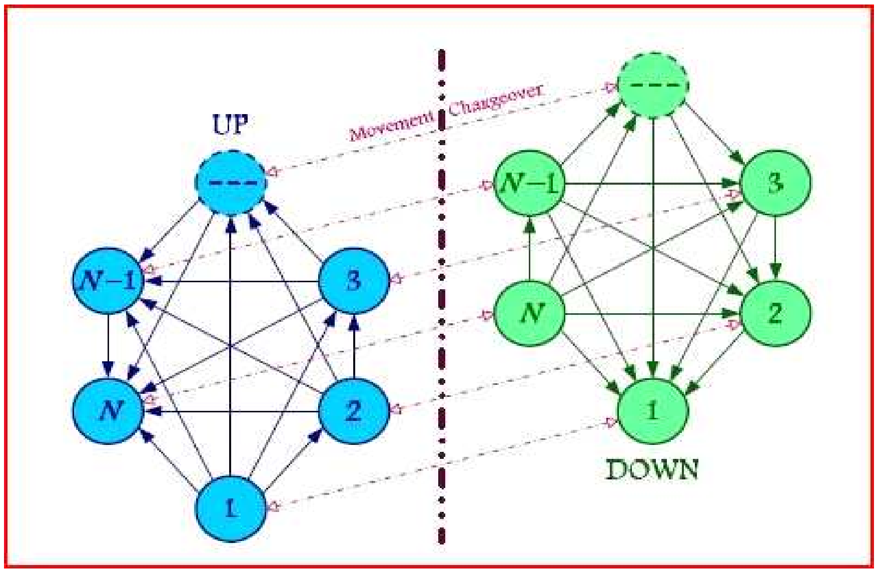


Figure 25: Two Elevator Scheduler

- Figure (27) shows the simulation result when protocol steps are followed and the box has opened. display_open variable goes to high indicating lock is open.
- Figure (28) shows the simulation result when the protocol is violated where after the second combination bit is pressed but the enter is not pressed. display_close remain on high always stating lock is closed.

State	Description
st_reset	Initial state of the system. The state is also reached when reset button is pressed
st_checkc1	System is ready to accept the first combination bit
st_ewait1	System is waiting for the user to press enter acknowledging the first bit entered
st_check2	System is ready to accept the second combination bit
st_ewait2	System is waiting for the user to press enter acknowledging the entry of the combination bit
st_checkc3	System is ready to accept the third combination bit
st_twait	System is waiting for the user to press try button
st_open	User has succeeded and the system will eventually signal to open the lock
st_close	User has failed and the box remains closed

Table 5: State description in verilog model

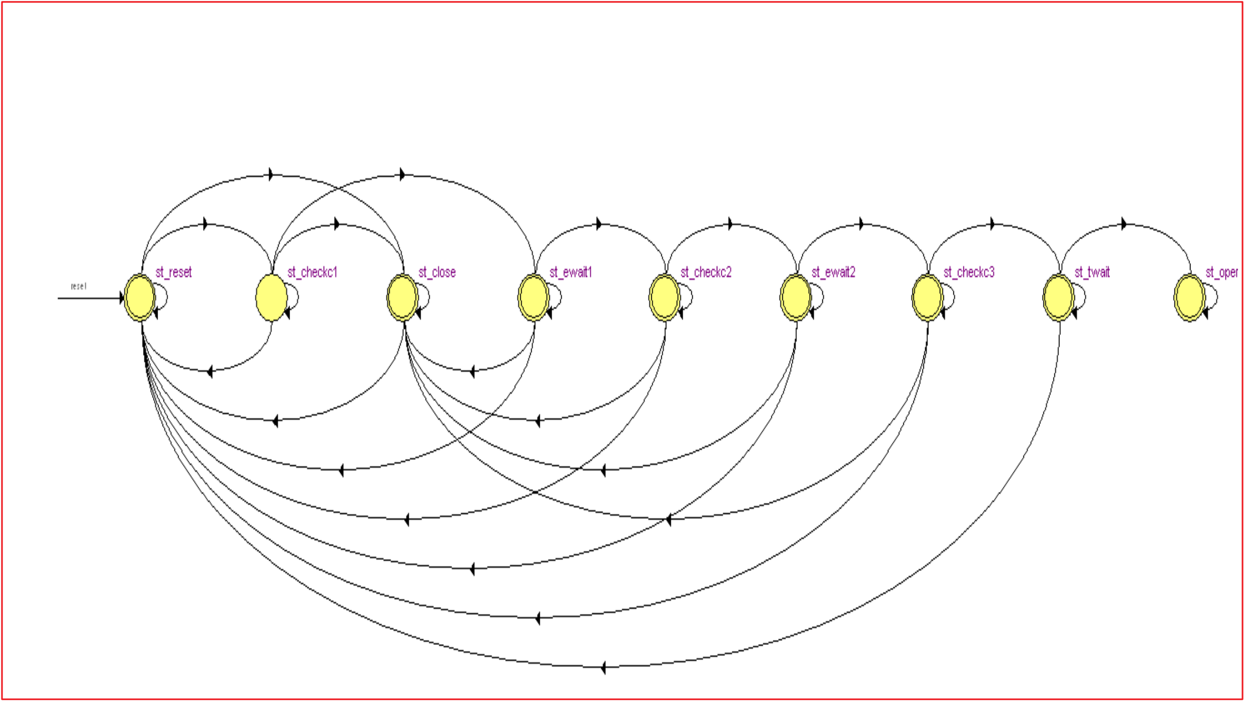


Figure 26: Combination Lock model in state machine

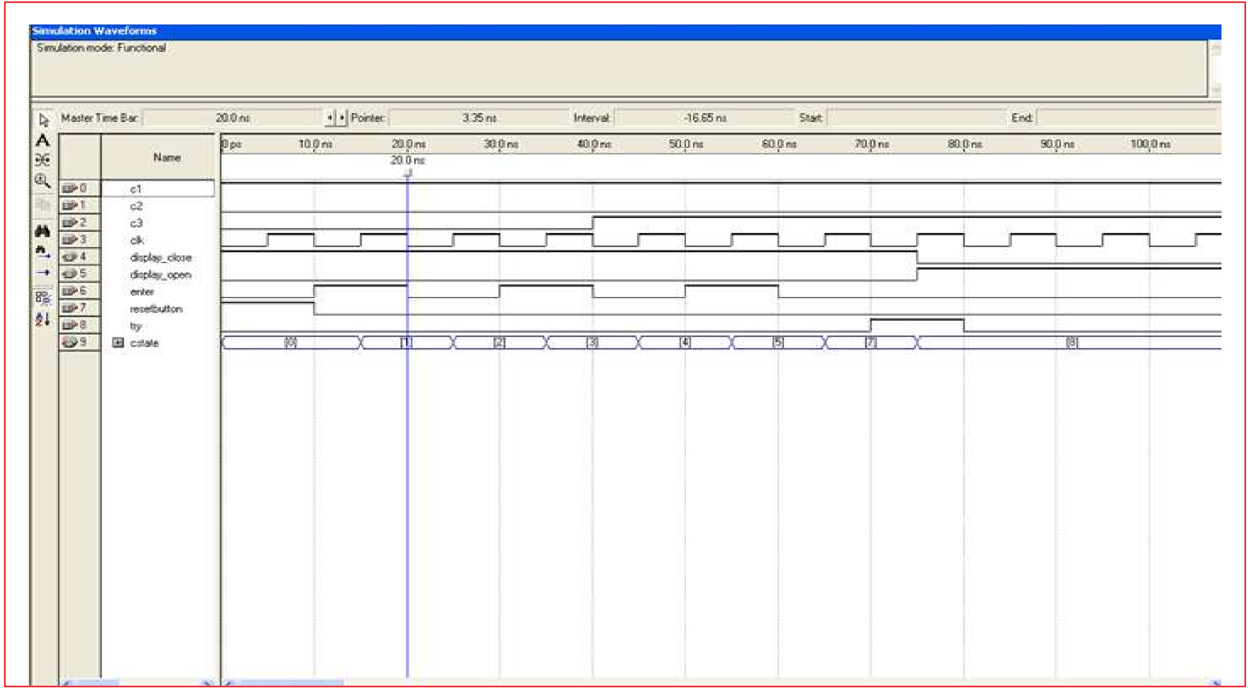


Figure 27: Simulation result for successful lock open

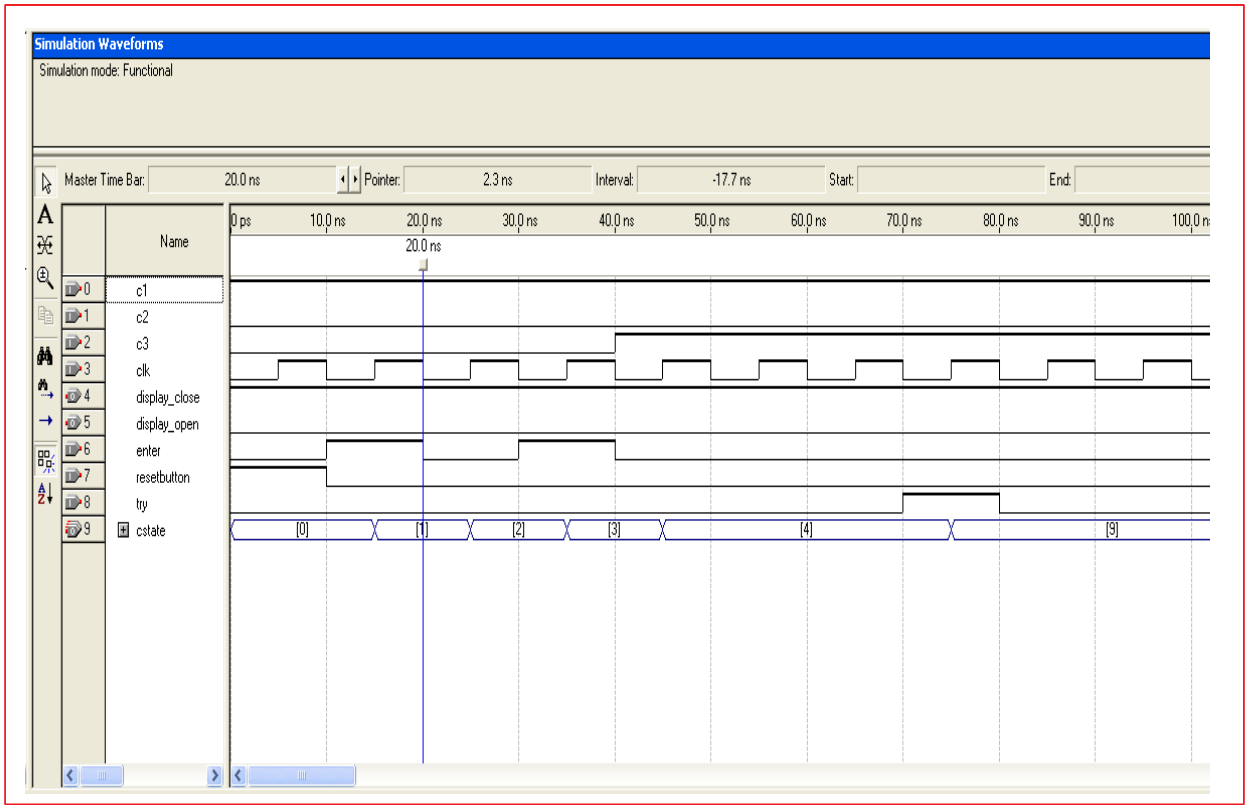


Figure 28: Simulation result for failure case where lock is closed

Bibliography

[1] `http://www.icis.ntu.edu.sg/scs-ijit/1203/1203_10.pdf`

[2] `http://en.wikipedia.org/wiki/Model_checker`

[3] `http://nusmv.irst.itc.it/`

[4] `https://www.cs.indiana.edu/classes/p515/`

[5] `http://nicta.com.au/__data/assets/pdf_file/0005/14747/lecture2-ctl.pdf`

[6] H. M. Deitel’s textbook, Operating Systems (Addison Wesley, 2nd ed., 1990)

[7] `http://www.itu.dk/courses/ISOT/E2003/doc/tutorial/tutorial.html`

[8] SMV language reference pdf

[9] `http://en.wikipedia.org/wiki/Producers-consumers_problem`

[10] `http://www.icis.ntu.edu.sg/scs-ijit/1203/1203_10.pdf`