Elevator Logical Design Specification v1.0

Suresh Damodaran
Caroline Lee
Saurabh Mittal
Gabriel Pascualy
Andrew Yale

Sponsor: MITRE Innovation Program
Department No.: T8A7
Project No.: 5118MC18-GA

MP180076
Bedford, MA

MITRE

This page intentionally left blank.

Table of Contents

List of Figures

List of Tables

Introduction

This document describes the static structure and dynamic behavior of the software control aspects of an elevator. The description is intended to be useful for anyone trying to build such an elevator. The document also shows the log messages generated by the components of the elevator as it operates. A separate Monitoring and Analysis Dashboard (MonAD) tool displays these log messages for further analysis and debugging. MITRE built this elevator model to make available a shared, simple, yet complex enough, platform that cyber-physical security researchers can use for advancing their research envelope.

System Overview

An elevator is an example of a real-time cyber physical system (CPS)[1]. This elevator comprises of physical components such as doors, a motor, and an elevator car interacting with processors. The system description in this document does not include the physical models of the motors or pulleys. Instead, this document focuses on the software components of the elevator system, and how to extend and apply the software components with additional hardware.

System Description

Table 1 lists the elevator components and their functions. The following paragraph describes the physical or Live components of the elevator to aid the reader in understanding the components shown in Table 1 and Figure 1. The next section of this document describes the models of these components. The description in the next paragraph refers to the components by the ID given in Table 1.

The elevator described in this document has five floors and an elevator car, though Figure 1 shows only three floors. The number of floors can be as few as four to demonstrate some of the cyber effects. Each component in Figure 1 has input ports and output ports. The elevator car has a functioning door (F4.4), controlled by its motor (not modeled), and a sensor that detects whether the door is open or closed. Each floor of the building where the elevator is situated also has a door (F5), operated by its own motor. There are car hailing buttons (F5) on each floor, and the elevator car contains car buttons (F4.3) to request navigation to the respective floor. F4 is moved up and down by its motor (F4.1). The requests from Floor (F5) and CarBtn (F4.3) are routed through RequestProc (F2) to ElevatorCtrl (F1), which processes the door requests in a sequential manner, and generated commands for Car Ctrl (F4.2). DoorStatusProc (F3) processes the door status from the floor doors and car door.

[1] https://www.nist.gov/el/cyber-physical-systems

Table 1. Elevator Components

ID	Name	Function
F1	ElevatorCtrl	Implements the control logic for elevator movement
F2	RequestProc	Collects and forwards inputs such as destination floor requests from different floors and inside the car to F1
F3	DoorStatusProc	Provides the door status of both the floor and the car door to F1
F4	Car	This is the elevator car that moves between the floors
F4.1	Motor	A motor that activates the pulley
F4.2	CarCtrl	Controller that operates motor and car door under commands from ElevatorCtrl, and provides car position to F1
F4.3	CarBtn	Button inside the car that generates a floor request to F2
F4.4	CarDoor	Car door position sensor, sends position to F3
F5	Floor	Floors 1-5 with a floor button to send requests to F2 and a floor door

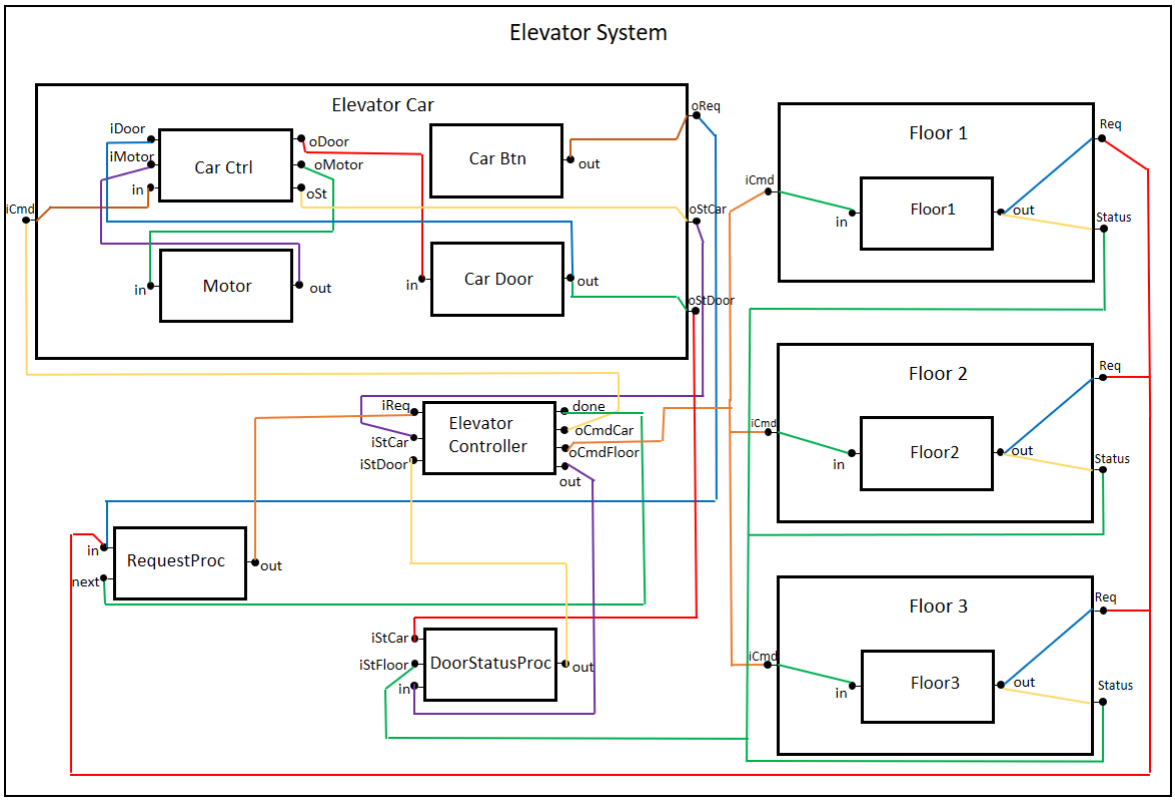

Figure 1. Elevator System Diagram

There are two types of components: one that does not contain another component, referred to as an *atomic* component in DEVS formalism (e.g., Car Door), and one that contains other components, referred to as a *coupled* component in DEVS formalism [1] (e.g., Elevator Car). A coupled component may contain atomic and coupled components. This document refers to the components within a coupled component as *sub-components.* It also uses the term *couplings* to denote connections among the components through their input and output *ports.* Each component has a state diagram, and as messages from through the system these messages form the inputs to the components. The components react to the input

through state transitions, and generate output messages through exection of output functions.

State Diagram Notations

The following subsections describe the syntax used within the state diagrams shown in the Behavior of Components section.

State

The states in a diagram are shown as bubbles, and contain the state name. Java language syntax is used to describe the *conditions* and *assignments* in transitions below, and *condition** indicates zero or more conditions.

Internal Transition

The internal transition is the change in state, triggered internally when some conditions are true. It is shown as a dotted arrow between two states. The diagrams show descriptions of internal transitions above the arrow in the state diagrams. The syntax of the description is:

[<*condition*>**]** <*assignment**>

Semantics of this description: if <*condition*> is true, make assignments zero or more <*assignment**> to state or component variables, and then move to next state. If <*condition*> is not true, log error and passivate.

Example: **[***done==true && !q.isEmpty***]**

External Transition

The external transition is the state change triggered by an external input; for example, if a message arrives at an input port. This input may trigger a state change, shown as a solid arrow between states. Descriptions of the external transitions are shown as a brown message over the arrow in the state diagrams. The syntax of the description is:

<*input port* **?** *message type*>
[<*condition*>**]** <*assignment**>

Semantics of this description: Process the *message type* from *input port* and assign to local state variables. Arrival of any message will trigger the evaluation of the specified *condition.* If <*condition*> is true, make zero or more assignments <*assignment**> to state or component variables, and then move on to next state. If <*condition*> is not true, log error and passivate.

Example: iStDoor ? MsgDoor
[door==closed && Car == ready_to_move && isGoUp]

In this example, isGoUp is a Boolean variable, and there are no assignments specified.

Output Function

An output function is triggered after an internal transition is completed. If an output message needs to be generated after an internal transition, that message is specified in the output function, shown in blue over the dotted arrows in state diagrams. Syntax is:

outport : **!** <message type*>

Semantics: The message with *message type* is output on port *outport*. There many be multiple message types that can be sent via the same output port.

Logs

Logs are generated when events such as sending a message or receiving a message occurs in each component. These logs are useful to understand the behavior of the system when it is executing. There are two types of log messages, those that record the sending or receiving of a message at a port, or logged by a component when some interesting interbal event occurs. The structure of a log record is best described with examples.

Here is an example of a log of a message received.

1517924508.128,74.3,RequestProc,Elevator Controller,R,{"port":"iReq", "value":"[REQ:5]"}

1517924508.128:	*Simulation Time*
74.3:	*Real Time*
RequestProc:	*The component that sent this message*
Elevator Controller:	*The component that received this message*
R:	*Received (S indicates sent)*

{"port":"iReq", "value":"[REQ:5]"}: Message content received at port “iREQ”, with the value “[REQ:5]”

Here is an example of a component log.

1518451514.894,136.5,Elevator Controller,,C,{"statusDoor":"CLOSED"}

1517924508.128:	*Simulation Time*
74.3:	*Real Time*
RequestProc:	*The component that sent this message*
<empty>:	*There is no recipient*
C:	*Component*

{ "statusDoor":"CLOSED"}: variable statusDoor with value CLOSED

System Design

The components of the system communicate with each other using messages. This section describes the types of these messages and the contents of each. Note that these message types remain the same for the elevator system, irrespective of whether the components are physical components or their sofware models.

Message Types

The message types described in the following subsections are sent out or to the component ports. The subsection for each message type defines message variables and the data types of each of the message variables.

MsgCar

MsgCar contains commands and status related to the elevator car. A message can contain only one message variable: either CommandCar or StatusCar.

A MsgCar command appends [pos:<curPos>][dest:<destPos>] to signify the source and destination floor of the elevator car. An example of this is *CarMOVE[pos:0][dest:1]*.

- CommandCar: CarUP, CarDOWN, CarMOVE, CarSTOP
- StatusCar: CarREADYTOMOVE, CarMOVING, CarOPENING, CarSTOPPED.

MsgDoor

MsgDoor contains commands and statuses related to the car and floor door. A message can contain only one message variable: either CommandDoor or StatusCar.

When MsgDoor is sent from FloorXDoor, the message values specify which door that message is from Floor by appending “FloorX” to each value, where X denotes the floor ID.

- CommandDoor: DoorFloorXOPEN, DoorFloorXCLOSE
- StatusDoor: DoorFloorXOPENED, DoorFloorXCLOSED.

When MsgDoor is sent from CarDoor, the message values specify that the message is from Car by appending “Car” to each value.

- CommandDoor: DoorCarOPEN, DoorCarCLOSE
- StatusDoor: DoorCarOPENED, DoorCarCLOSED.

MsgMotor

MsgMotor contains commands and statuses related to the car motor. A message can contain only one message variable: either CommandMotor or StatusMotor.

- CommandMotor: MotorFORWARD, MotorBACKWARD, MotorSTOP
- StatusMotor: MotorMOVING, MotorREACHED

MsgReq

MsgReq contains floor requests through the message variable “dest”, where the integer in the request designates which floor is being requested.

- CommandDest: [REQ:1], [REQ:2], [REQ:3]…

MsgElev

MsgElev contains elevator messages. Message values include destination floors as integers.

- msg: [ELEV:1], [ELEV:2], [ELEV:3],…

MsgFloor

MsgFloor contains floor requests from the FloorX, where X is the floor ID.

- cmd: FloorXDOWN, FloorXUP

Behavior of Components

The subsections below describe the dynamic behavior of each elevator component. The description of each component’s dynamic behavior contains the following subsections.

1. State diagram of the component, as applicable.
2. The input and output ports and the message types that pass through each.
3. Component variables whose values form a part of the state of each component, and message types used to communicate the state of the component to other components.
4. Example logs that may be sent or received from the component. *Italicized text* indicates log text examples.

Elevator Controller (ElevatorCtrl)

The Elevator Controller implements the control logic for elevator movement. It processes floor requests originating from the car or the floor. At a given time, only one floor request is processed. The cycle begins with “idle” and ends in “idle” phase. At the end of the cycle, ElevatorCtrl communicates to the RequestProc that it is done with the cycle, which then prompts the RequestProc to send the next queued destination. If the elevator is operational and the requested destination is not the same as the current floor, the elevator goes into “move” state; otherwise it goes to “stop.” The “move” state notifies the Elevator Car, the Floor(s) and DoorProc to get ready for movement. Once these components acknowledge the notification the elevator goes to the “moveUP” or “moveDOWN” phase, depending on the direction of move. After the move is completed and the Elevator Car notifies ElevatorCtrl that it has reached the destination, the elevator stops and waits for the Elevator Car to open the doors. This is the “operating” state. Once notified by the Elevator Car and the Floor that the doors have cycled through an open-close cycle, ElevatorCtrl goes into “reset” mode. This completes the cycle. Figure 2 shows the state diagram for ElevatorCtrl.

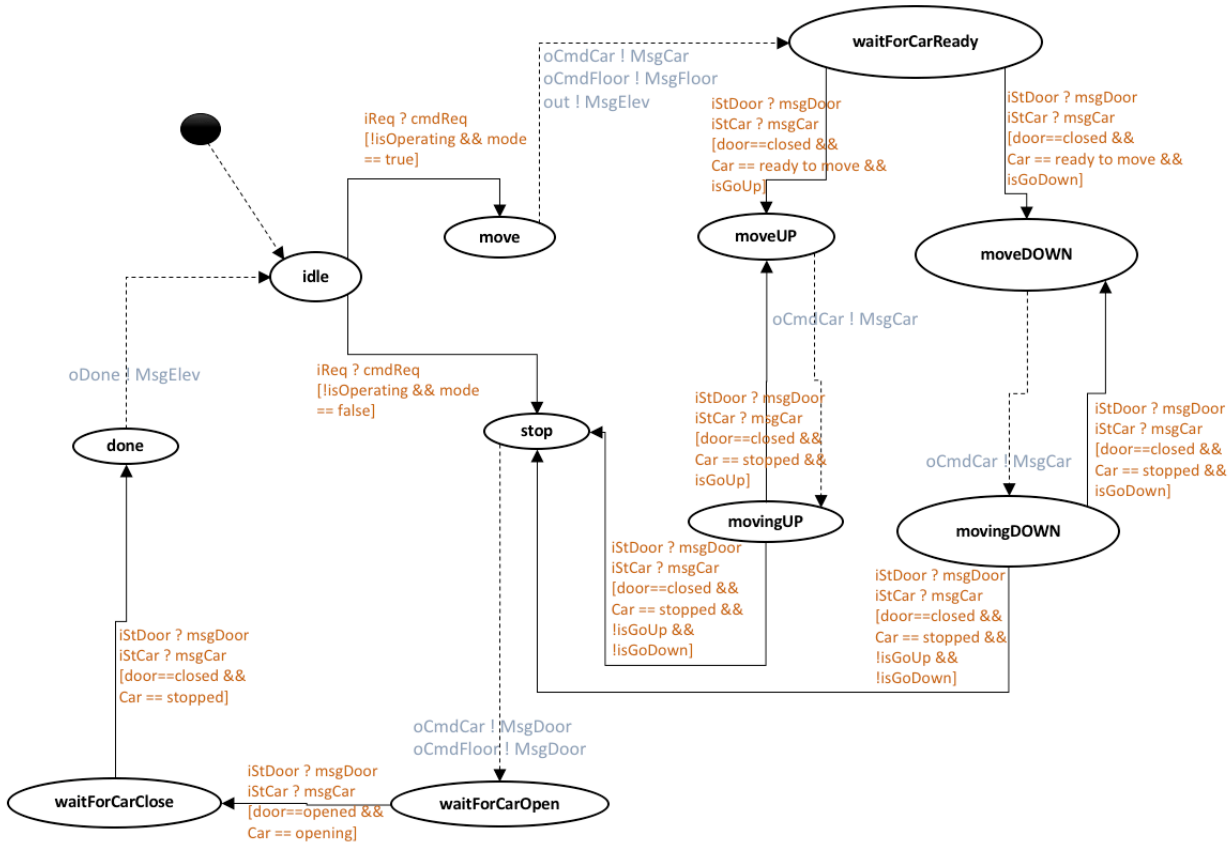

Figure 2. ElevatorCtrl State Diagram

Ports

Input

- iReq: From RequestProc, receives floor request.
  - Message Type: MsgReq
- iStCar: From Car Ctrl, receives the state of the elevator car.
  - Message Type: MsgCar
- iStDoor: From DoorStatusProc, the state of both the floor door and Car Door
  - Message Type: MsgDoor

Output

- oCmdCar: Sends commands to CarCtrl of the action it needs to perform
  - Message Type: MsgCar
- oCmdFloor: Sends commands to Floor Door
  - Message Type: MsgDoor
- out: Sends to DoorStatusProc the floor of the door to open

    - Message Type: MsgElev
- done:
    - Message type: MsgElev

Component Variables

- State Variables
    - curFloor (int)
        - Description: Current floor of Elevator Car
    - destFloor (int)
        - Description: Destination floor
    - isGoUp (boolean)
        - Description: Is Elevator Car going up
    - isGoDown (boolean)
        - Description: Is Elevator Car going down
    - operating (boolean)
        - Description: Is the Elevator Car operating
    - statusDoor (string)
        - Description: State of car door
    - statusCar (string)
        - Description: State of elevator car

- Message Types
    - MsgReq
    - MsgCar
    - MsgDoor
    - MsgElev

Example Logs

- *1517924508.128,74.3,RequestProc,Elevator Controller,R,{"port":"iReq", "value":"[REQ:5]"}*
- *1517924897.625,75.9,Car Ctrl,Elevator Controller,S,{"port":"oSt", "value":"CarMOVING[pos:1][dest:5]"}*
-

Request Processor (RequestProc)

The RequestProc collects and forwards inputs such as destination floor requests from different floors and inside the car to ElevatorCtrl. It receives floor destination requests from ElevatorCar and from different floors, and maintains a FIFO [first in, first out] queue. When notified by the ElevatorCtrl of cycle completion, it sends the next floor in the queue to ElevatorCtrl and then removes that floor from the queue. RequestProc has a default processing time for a request to be completed that is refreshed once that cycle has elapsed. Once this cycle has elapsed, RequestProc is in the "busy" state. Until the request is processed, it passivates in "busy" (i.e., infinity). Figure 3 shows the state diagram for RequestProc.

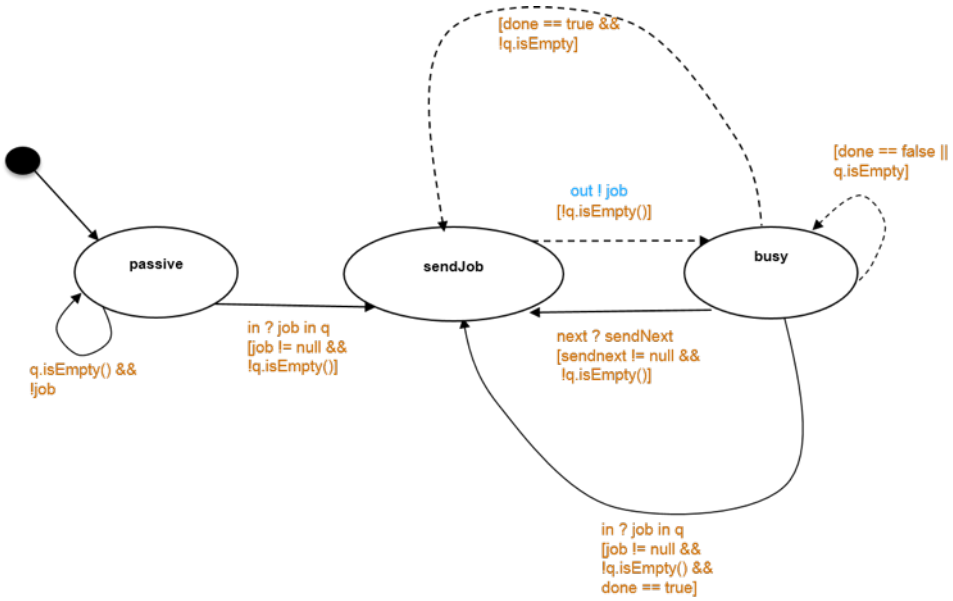

Figure 3. RequestProc State Diagram

Ports

Input

- in
  - Message Type: MsgReq, MsgFloor
- next
  - Message Type: MsgElev

Output

- out: Sends floor requests to ElevatorCtrl
  - Message Type: MsgReq

Component Variables

- State Variables

- q (Queue<entity>)
  - Description: A queue of jobs to perform
- job (entity)
  - Description: The current job
- processing_time (double)
  - Description: Processing time of the job

- Message Types
  - MsgReq, MsgFloor, MsgElev

Example Logs

- *1517924896.746,75.0,RequestProc,Elevator Controller,S,{"port":"out", "value":"[REQ:5]"}*
- *1518451514.894,136.5,Elevator Controller,,C,{"statusDoor":"CLOSED"}*

Door Status Processor (DoorStatusProc)

DoorStatusProc provides the door status of both the floor and the car door to ElevatorCtrl. Its primary function is to synchronize the door status of Elevator Car and the appropriate destination floor. Synchronization occurs when both the doors are either closed or open. Once they are synchronized, DoorStatusProc notifies the ElevatorCtrl to proceed with the next step. It ensures that the destination door (communicated to it by ElevatorCtrl) is indeed the same floor that is sending the door status message. Figure 4 shows the state diagram for DoorStatusProc.

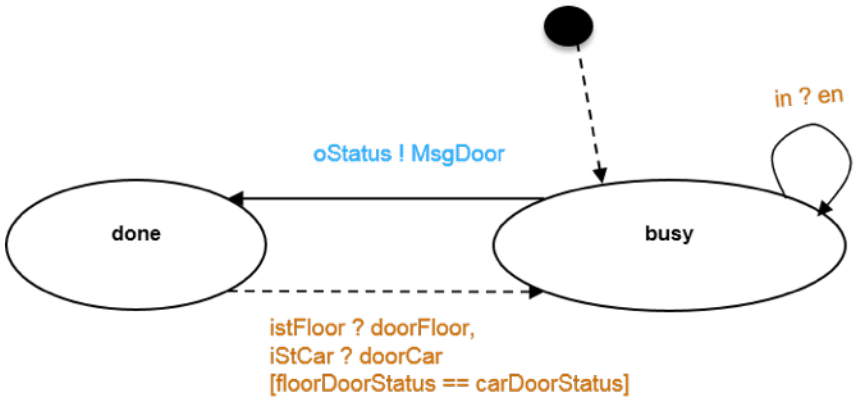

Figure 4. DoorStatusProc State Diagram

Ports

Input

- iStCar: From Car Door, the state of the elevator car door.
  - Message Type: MsgDoor
- iStFloor: From FloorXDoor, the state of the floor X door.

    - Message Type: MsgDoor
- in: From Elevator Controller, floor of door to open
    - Message Type: MsgElev

Output

- out: To Elevator Controller, the status of both Floor Door and Car Door
    - Message Type: MsgDoor

Component Variables

- State Variables
    - doors (Map<Integer, StatusDoor>)
        - Description: Describes the floor doors and their statuses
    - curFloor (integer)
        - Description: The current floor of Elevator Car
- Message Types
    - MsgDoor

Example Logs

- *1517924896.941,75.2,Elevator Controller,DoorStatusProc,R,{"port":"in", "value":"[ELEV:5]"}*
- *1517924897.468,75.8,DoorStatusProc,Elevator Controller,S,{"port":"out", "value":"DoorCarCLOSED"}*
- *1518451514.894,136.5,DoorStatusProc,,C,{"carDoorStatus":"CLOSED"}*

Elevator Car

An Elevator Car is a coupled component that moves between the floors. The Elevator consists of four components: CarBtn, which generates new floor requests; Car Door, which opens or closes the doors when the car reaches the requested destination; CarCtrl, which controls the operation of the Motor and Car Door; and Motor, which moves the elevator up or down. Figure 5 shows the components of the Elevator Car.

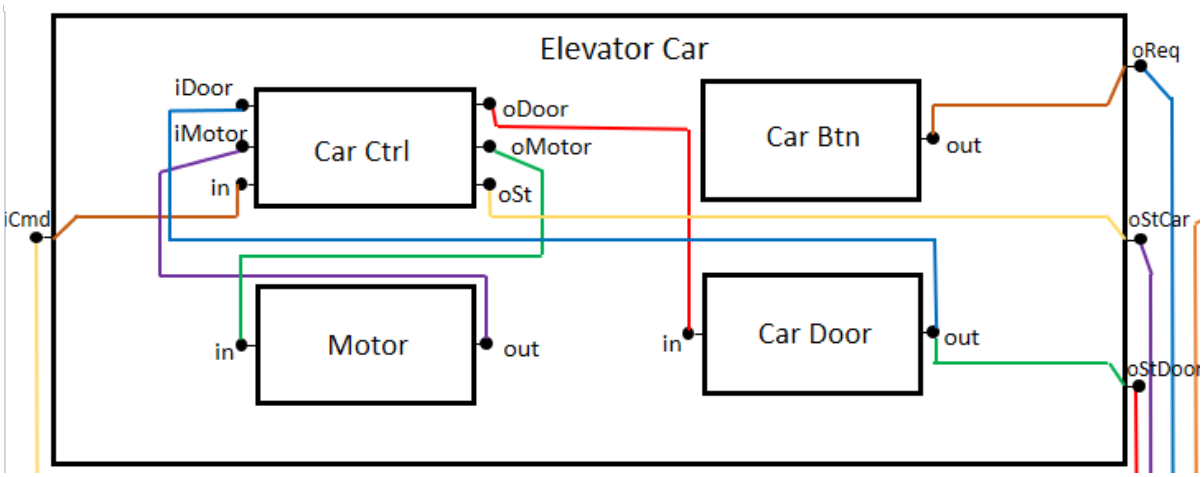

Figure 5. Elevator Car Components

Subcomponents

- Car Ctrl
- Car Btn
- Motor
- Car Door

Ports

Input

- iCmd
  - Coupled with carCtrl, port: “in”

Output

- oReq
  - Coupled with carBtn, port “out”
- oStDoor
  - Coupled with carDoor, port “out”
- oStCar
  - Coupled with carCtrl, port “oSt”

Motor

The Motor activates the pulley. It is a simple processor that models the movement of the Elevator Car. It takes 5 units of simulation time to travel to one floor (any direction). For example, a move from floor 1 to floor 4 takes 15 units of time. The unit of time can be seconds or, minutes or any desired unit. The motor is in direct control of CarCtrl, which provides the direction (forward or backward). Accordingly, the motor moves forward

when the car is going up or backward when the car is moving down. It must be notified by the CarCtrl to maintain the direction after each floor is passed when heading toward the final destination. Figure 6 shows the state diagram for the Motor.

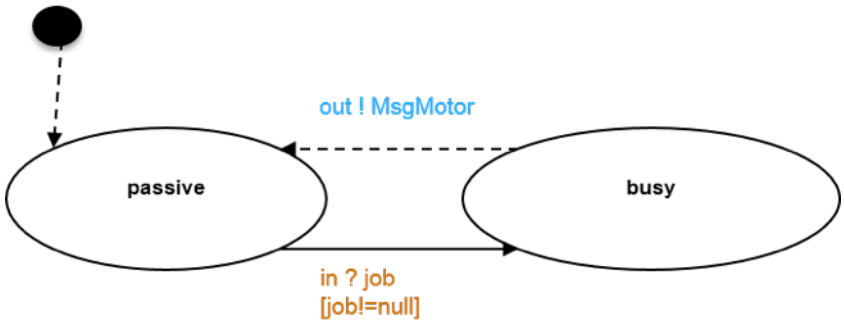

Figure 6. Motor State Diagram

Ports

Input:

- in: From CarCtrl, some action for motor to perform.
  - Message Type: MsgMotor

Output:

- Out: To Car Ctrl, the status of the car motor.
  - Message Type: MsgMotor

Component Variables

- Message Types
  - MsgMotor

Example Logs

- *1517924897.625,75.9,Car Ctrl,Motor,S,{"port":"oMotor", "value":"MotorFORWARD"}*
- *1517924898.714,77.0,Motor,Car Ctrl,S,{"port":"out", "value":"MotorREACHED"}*
- *1518451514.894,136.5,Cat Ctrl,,C,{"statusMotor":"REACHED"}*

CarCtrl

The CarCtrl is the controller that operates Motor and Car Door following commands from ElevatorCtrl, and provides car position to ElevatorCtrl. It receives input from the ElevatorCtrl and updates ElevatorCtrl with the status of the Car. The process of executing a given destination request begins with the ElevatorCtrl asking the CarCtrl to prepare for moving (if the destination floor is not the same as the current floor). The CarCtrl then commands the door to close. Once notified by the door that it is closed, CarCtrl notifies the ElevatorCtrl, which then notifies the Car to start moving. CarCtrl then commands Motor to move in the appropriate direction. On passing through each of the intermediate floors, it communicates the status position of the Car to the ElevatorCtrl, and verifies the direction of

the motor after each passing floor. Once the Car reaches the destination floor, CarCtrl stops the motor and notifies the ElevatorCtrl that it has reached the destination, and waits for acknowledgement. ElevatorCtrl then notifies the Car and the Floor to open the doors. The doors then go through an open-close cycle. This completes one cycle for a destination request. Figure 7 shows the state diagram for CarCtrl.

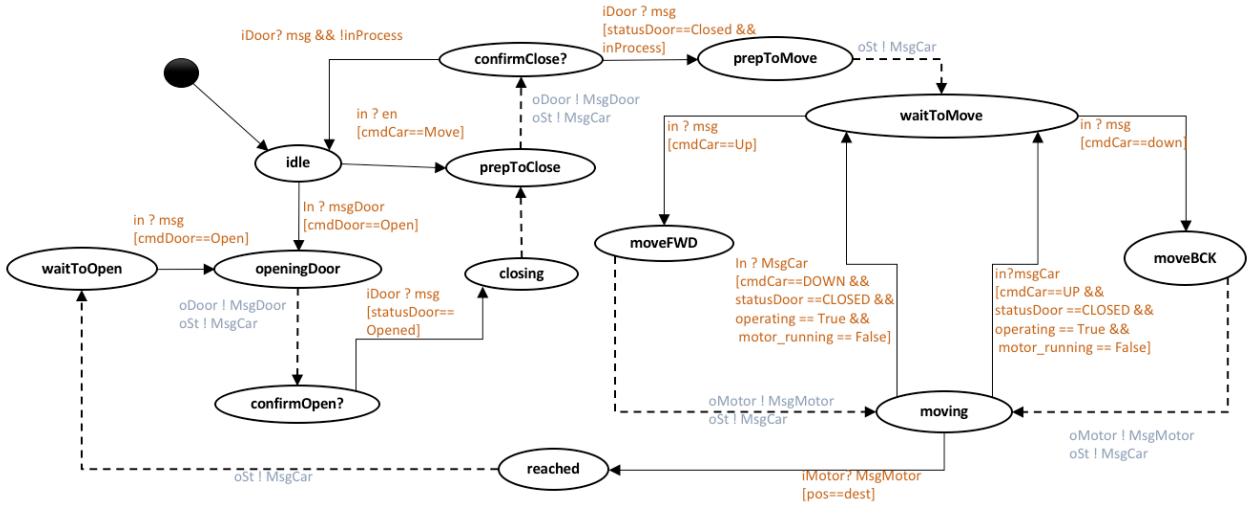

Figure 7. CarCtrl State Diagram

Ports

Input

- iDoor: From CarDoor, state of the car door.
  - Message Type: MsgDoor
- iMotor: From Motor, the state of the car motor.
  - Message Type: MsgMotor
- in: From Elevator Controller, command to perform some action.
  - Message Type: MsgDoor, MsgCar

Output

- oDoor: To Car Door, action Car Door is to perform.
  - Message Type: MsgDoor
- oMotor: To Motor, command for car motor to perform some action.
  - Message Type: MsgMotor
- oSt: To Elevator Controller, send status of the elevator car.
  - Message Type: MsgCar

Component Variables

- State Variables
  - doorStatus (string)
    - Description: closed if false and open if true
  - motorStatus (string)
    - Description: stopped if false and moving if true
  - curFloor (int)
    - Description: Current floor position for car
  - destFloor (int)
    - Description: Destination floor position for car
  - operating (boolean)
    - Description: inactive if false and activated if true
- Message Types
  - MsgCar
  - MsgDoor
  - MsgMotor

Example Logs

- *1517924897.011,75.3,Elevator Controller,Car Ctrl,R,{"port":"in", "value":"CarMOVE[pos:1][dest:5]"}*
- *1517924897.064,75.3,Car Ctrl,Car Door,S,{"port":"oDoor", "value":"DoorCarCLOSE"}*
- *1517924897.490,75.8,Car Door,Car Ctrl,R,{"port":"iStDoor", "value":"DoorCarCLOSED"}*
- *1517924897.556,75.8,Car Ctrl,Elevator Controller,S,{"port":"oSt", "value":"CarREADYTOMOVE[pos:1][dest:5]"}*
- *1518451514.894,136.5, Car Ctrl,,C,{"carDoorStatus":"CLOSED"}*

Car Btn

CarBtn generates floor destination requests from within the car. It sends the request to the Car, which then forwards it to the RequestProc. Figure 8 shows the state diagram for Car Btn.

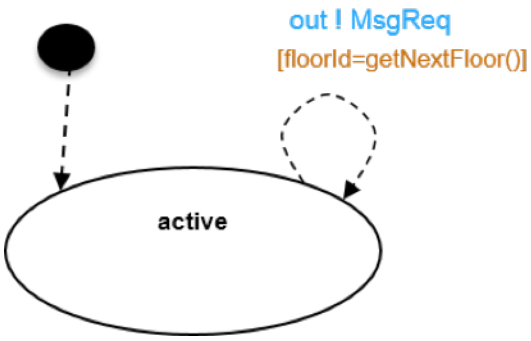

Figure 8. Car Btn State Diagram

Ports

Output

- out: Sends out requests for floor numbers
  - Message Type: MsgReq

Component Variables

- State Variables
  - carCall (boolean)
    - Description: If car is in use
  - arrivedAtFloor (boolean)
    - Description: If car has fully arrived at destination floor
- Message Types
  - MsgReq

Example Log

- *1517924896.670,74.9,Car Btn,RequestProc,S,{"port":"out", "value":"[REQ:5]"}*

Car Door

Car Door is a position sensor that sends the car's position to DoorStatusProc. A door passivates if it is in the state "closed." When Car Door receives a command to open, it goes into the "opening" state and then on to the "opened" state. The opening and closing takes motion_time to complete the physical door movement (in the model). Once that time is elapsed Car Door goes into "opened" or "closed" state. It stays opened for processing_time, after which it automatically goes into closing state, and eventually to closed state. Figure 9 shows the state diagram for Car Door.

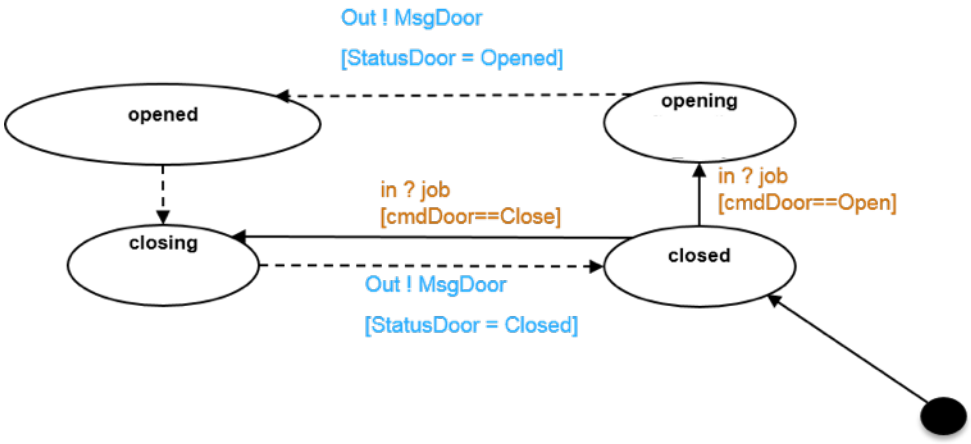

Figure 9. Car Door State Diagram

Ports

Input

- in: From CarCtrl, action to perform by Car Door.
  - Message Type: MsgDoor

Output

- Out: To DoorStatusProc, send status of Car Door.
  - Message Type: MsgDoor

Component Variables

- State Variables
  - id (int)
    - Description: ID for the door. The Door component is used both in Car and the Floor coupled component. To differentiate a Car door with a Floor door, the ID of Car door is set to 100. The floor id is the same as the floor number.
  - job (entity)
    - Description: The current command message
  - processing_time (double)
    - Description: Time to keep the doors open. It is set at 5 units of time.
  - motion_time (double):
    - Description: Time to physically close/open the door. It is set at 3 units of time.
- Message Types
  - MsgDoor

Example Logs

- *1517924897.115,75.4,Car Ctrl,Car Door,R,{"port":"in", "value":"DoorCarCLOSE"}*
- *1517924897.115,75.4, Car Door, Car Ctrl,S,{"port":"in", "value":"DoorCarCLOSED"}*
- *1518451514.894,136.5,Car Door,,C,{"cmdDoor":"CLOSE"}*

Floor

A Floor is a coupled component that consists of two components. The system can have more than one floor. Figure 10 shows the state diagram for Floor.

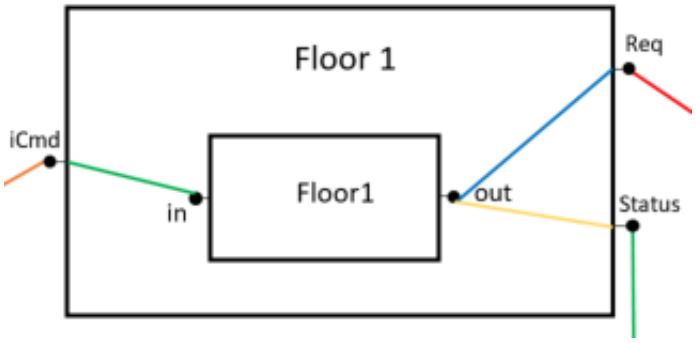

Figure 10. Floor Component Diagram

Sub-components

- FloorXDoor (e.g., Floor3Door for the door of the 3rd floor)

Ports

Input

- iCmd
  - Coupled with floorDoor, port “in”

Output

- oReq
  - Coupled with floorBtn, port “out”
- oStatus
  - Coupled with floorDoor, port “out”

FloorDoor

Floor door. Description and behavior is same as Car Door.

Ports

Input

- in: From Elevator Controller, command to open/close the door.
  - Message Type: MsgDoor

    - Output
- out: To DoorStatusProc, send the status of FloorXDoor
    - Message Type: MsgDoor

Component Variables

- State Variables
    - id (int)
        - Description: The floor ID.
    - job (entity)
        - Description: The current job
    - processing_time (double)
        - Description: Processing time of the job
    - motion_time (double)
        - Description: Time to move from one floor to another
- Message Types
    - MsgDoor

Example Log

- *1517924918.655,96.9,Elevator Controller,Floor5Door,R,{"port":"in", "value":"DoorFloor5OPEN"}*
- *1518451514.894,136.5,Floor3Door,,C,{"cmdDoor":"CLOSE"}*

Extending Python Model with Live Components

MITRE used Python to implement the elevator design described in this document, referred to as Python model, and ran simulations of this model. MITRE also implemented a Live model of the elevator that contains physical components such as motor, doors, and sensors. This section describes the configuration of theses two types of models using examples, including logging options. A separate document describes the physical construction of the Live model.

Floor Request Process in Python Model

In the Python model, the Car Button controller receives a floor request from the user via stdin. The Car Button controller then sends a message over a socket connection to the Request Processor. The Request Processor queues the request and dispatches it once the Elevator Controller has fulfilled the preceding requests. Each socket message is logged before it is sent and after it is received. The previous sections showed examples of these logs. Figure 11 shows the message flow in the floor request process when the elevator is fully simulated.

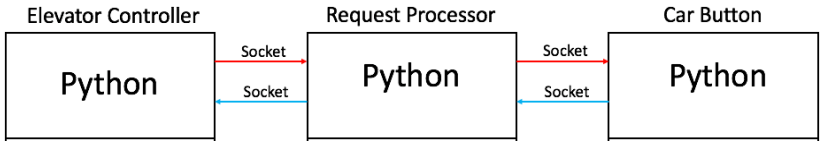

Figure 11. Diagram of Floor Request Process in Simulation

Floor Request Process in Live Model

In the *live* or physical model, the floor request process utilizes Arduinos that control physical components. Figure 12 shows how these Arduinos integrate with the corresponding Python model of a component, but does not show the physical components themselves. When a floor or car button is pressed, a digital signal is sent from Car Button's Arduino controller to the Request Processor Arduino. The request is queued and dispatched to the elevator controller Arduino when the elevator has processed the previous request. When the floor request reaches the Elevator Controller, it is propagated up to the Python model over a serial connection.

Each socket message is logged before it is sent and after it is received. The socket messages are only used for logging, and not used for any state changes in the component that is receiving it by the Python model. The reason behind this arrangement is that currently there is no option to log Digital I/O based messages.

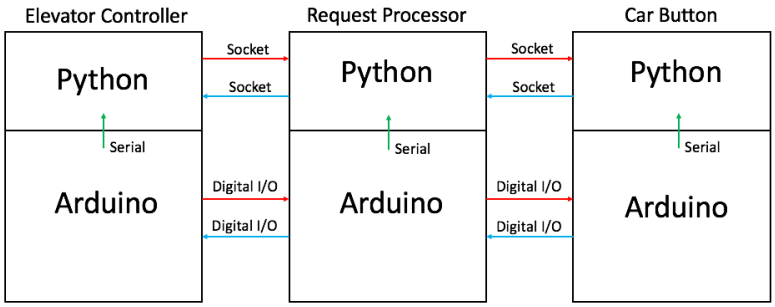

Figure 12. Floor Request Process Message Flow in the Live Model

Motor Control in Python Model

In the Python model, each motor controller simulates motor action by inserting a delay. Each socket message is logged before it is sent and after it is received. Figure 13 shows the simulated message flow.

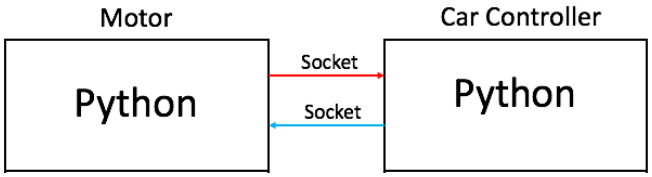

Figure 13. Car Motor Control Message Flow in Simulation

Car Motor Control in Live Model

In the Live model, the car motor is controlled by an Arduino, which in turn is controlled by a Python model that communicates with it over a serial connection (see Figure 14). When the Python model receives a message over a socket from the Car Controller, it sends the appropriate command over a serial connection to the Arduino motor controller. Each socket message is logged before it is sent and after it is received. The serial messages are not logged.

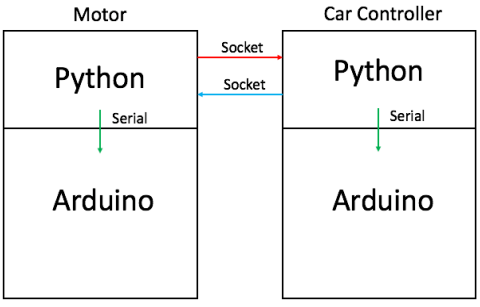

Figure 14. Car Motor Control Message Flow in Live Model

References

[1] Zeigler, Bernard P., Herbert Praehofer, and Tag Gon Kim. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic Press, 2000.

Appendix A: Message Type Detector

The message variable names corresponding to the message data are not encoded in the elevator logs. The Message Type Detector performs the detection of the message type from message data. This appendix describes the details of the Message Type Detector.

Message Signatures

Each type of message has a defined set of possible message variables that can be sent, and each variable has a defined set of possible values. Using the value in the message, the Message Type Detector can identify which variable is being used and can then determine the message type from the variables being sent. This works because there is no overlap in possible values for the different variables and no overlap in the variables being used.

MsgCar

CommandCar: CarUP, CarDOWN, CarMOVE, CarSTOP

StatusCar: CarREADYTOMOVE, CarMOVING, CarOPENING, CarSTOPPED

MsgDoor

CommandDoor: DoorFloor1OPEN, DoorFloor1CLOSE, DoorFloor2OPEN, DoorFloor2CLOSE, DoorFloor3OPEN, DoorFloor3CLOSE, DoorFloor4OPEN, DoorFloor4CLOSE, DoorFloor5OPEN, DoorFloor5CLOSE, DoorCarOPEN, DoorCarCLOSE

StatusDoor: DoorFloor1OPENED, DoorFloor1CLOSED, DoorFloor2OPENED, DoorFloor2CLOSED, DoorFloor3OPENED, DoorFloor3CLOSED, DoorFloor4OPENED, DoorFloor4CLOSED, DoorFloor5OPENED, DoorFloor5CLOSED, DoorCarOPENED, DoorCarCLOSED

MsgMotor

CommandMotor: MotorFORWARD, MotorBACKWARD, MotorSTOP

StatusMotor: MotorMOVING, MotorREACHED

MsgReq

CommandDest: [REQ:1], [REQ:2], [REQ:3], [REQ:4], [REQ:5]

MsgElev

CommandElev: [ELEV:1], [ELEV:2], [ELEV:3], [ELEV:4], [ELEV:5][ELEV:done]

MsgFloor

CommandFloor: Floor1DOWN, Floor1UP, Floor2DOWN, Floor2UP, Floor3DOWN, Floor3UP, Floor4DOWN, Floor4UP, Floor5DOWN, Floor5UP