Elevator Control System (Version 0.5)

8 March 2003

1 Introduction

The purpose of the elevator control system is to manage movement of an elevator in response to user requests.

1.1 Basic elements

The elevator system has the following basic elements and parameters.

1.1.1 Number of elevators

Number of elevators in the system.

$$numElevators : \mathbb{N}_1$$

1.1.2 Floors

Floors serviced by the elevator system. Floors are numbered starting at one even though in some circumstances they might be labeled differently. (Have you noticed that many hotels and other buildings don't have a floor that is labeled "13", for example?)

Floors is modeled as a finite set, since we may need to apply the cardinality operator, which does not work with infinite sets.

$$nFloors : \mathbb{N}$$
$$Floors : \mathbb{F}\,\mathbb{Z}$$

$$\langle\!\langle \text{ grule TopFloorGE2 } \rangle\!\rangle$$
$$nFloors \geq 2$$
$$\langle\!\langle \text{ rule FloorsDef } \rangle\!\rangle$$
$$Floors = 1 \mathinner{..} nFloors$$

Proof note:

The $TopFloorGE2$ label marks the $nFloors \geq 2$ predicate as a theorem that the Z/EVES prover can assume to be true.

The $FloorsDef$ label marks the $Floors = 1 \mathinner{..} nFloors$ equality predicate so it can be used as a substitution rule. In other words, when the prover sees $Floors$, this rule means that it can rewrite that part of the expression to be $1 \mathinner{..} nFloors$ instead.

These proof rules are needed for use in later proofs.

1.1.3 Elevator status

An elevator may be in service or out of service.

$$ServiceStatus ::= InSvc \mid OutSvc$$

Proof note:

The following theorem is defined so that Z/EVES knows that the service status is binary; an elevator is either in service or out of service. This permits the theorem prover to infer, for example, that if an elevator is not in service it must be out of service. The theorem might seem obvious from the type definition, but Z/EVES doesn't automatically know this fact about free types.

theorem frule ServiceStatusDef
$\forall s : ServiceStatus \bullet s = InSvc \lor s = OutSvc$

1.1.4 Elevator direction

An elevator may be stopped, or it may be moving up or down.

$$Direction ::= DirUp \mid DirDown \mid DirHalt$$

Proof note:

The following theorem is defined to specify the enumerated directions, so the theorem prover can know that these are the only possible direction values.

theorem frule DirectionDef
$\forall d : Direction \bullet d = DirUp \lor d = DirDown \lor d = DirHalt$

1.2 Elevator

An elevator has a current location (floor) and direction of movement. It also has a set of floor requests that correspond to the floor buttons currently selected inside the elevator.

A finite set is used to model *requests*.

Elevator

$curFloor : Floors$
$status : ServiceStatus$
$curDir : Direction$
$requests : \mathbb{F}\ Floors$

The following schema describes the initial state of an elevator.

InitElevator

$Elevator$

$curFloor = 1$
$status = InSvc$
$curDir = DirHalt$
$requests = \emptyset$

The following theorem asserts that an elevator can be successfully initialized.

theorem InitElevatorOK
$\exists\, Elevator \bullet InitElevator$

Proof note:

The following proof steps demonstrate how the *InitElevatorOK* theorem can be proved. A faster alternative would be to use a single step of *prove by reduce*, which in effect combines all the steps into one operation.

proof
reduce;
invoke Elevator;
apply FloorsDef to expression Floors;
prove;
■

Proof notes:

Create an initialized elevator instance for later use in proofs.

$$elevator0 : Elevator$$
$$\exists\, InitElevator' \bullet elevator0 = \theta Elevator'$$

Create a sequence of initialized elevator instances for later proofs. The reason for this is that new "objects" cannot be created in the middle of a proof script, but existing ones can be used.

$$elevator0Seq : \mathrm{seq}\, Elevator$$
$\langle\!\langle$ Elev0SeqDef $\rangle\!\rangle$
$$elevator0Seq = (\lambda\, j : 1 \mathinner{..} numElevators \bullet elevator0)$$

Make a couple of assumptions about the sequence of initialized elevators. These assumptons are used in later proofs. Mark them *disabled* so that they have to explicitly referenced in proofs. (These assumptions should themselves be proved, but we'll defer that for now.)

theorem disabled grule Elev0SeqRangeIsElev0
$\mathrm{ran}\, elevator0Seq = \{elevator0\}$

theorem disabled grule Elev0SeqCardinality
$\# elevator0Seq = numElevators$

1.3 Elevator calls

An elevator call is a summons from a specific floor, which indicates that a user has signaled a desire to travel in a specified direction (up or down) from that floor.

The requested direction uses the same type as that used for an elevator's direction of travel, but the "halt" direction is excluded.

$$CallDirection == \{DirUp, DirDown\}$$

Proof note:

The following theorem allows the Z/EVES prover to assume the correct type of *CallDirection*; it is needed for later proofs.

theorem grule CallDirectionType
$CallDirection \in \mathbb{P}\, Direction$

All that is necessary to prove this theorem is to expand the definition of *CallDirection*.

proof
invoke CallDirection;
prove;
■

A call is represented by a pair that contains the originating floor and the desired direction of travel. The bottom floor has no "down" button and the top floor has no "up" button.

$$ValidCalls : \mathbb{P}(Floors \times CallDirection)$$

$$ValidCalls = (Floors \times CallDirection) \setminus \{(nFloors, DirUp), (1, DirDown)\}$$

Proof note:

The following theorem specifies the type of the domain of *ValidCalls* so that the Z/EVES theorem prover can assume this fact.

theorem grule ValidCallsDomType
$\forall c : \mathbb{F}\, ValidCalls \bullet \operatorname{dom} c \in \mathbb{F}(1 \mathinner{..} nFloors)$

A schema is used to model the set of pending elevator calls, to make it easier to define operations. A finite set is used to model *call*.

Calls
$calls : \mathbb{F}\, ValidCalls$

The following schema describes the initial state of the elevator calls.

InitCalls
Calls

$calls = \emptyset$

The following theorem asserts that the elevator calls can be successfully initialized.

theorem InitCallsOK
$\exists\, Calls \bullet InitCalls$

Proof note:

The following proof steps demonstrate how the *InitCallsOK* theorem can be proved. A single step of *prove by reduce* would also work.

proof
reduce;
invoke Calls;
prove;
■

1.4 Complete elevator system

The elevator system consists of the specified number of elevators and the elevator calls.

ElevatorSystem
Calls
$elevators : \mathrm{seq}\, Elevator$

$\# elevators = numElevators$

The following schema describes the initial state of the elevator system.

InitElevatorSystem
ElevatorSystem
InitCalls

$\mathrm{ran}\, elevators = \{elevator0\}$

Proof note:

The automatically generated domain check for *InitElevatorSystem* can be proved with a single step of *prove by reduce*.

The following theorem asserts that the elevator system can be successfully initialized.

theorem InitElevatorSystemOK
$\exists\, ElevatorSystem \bullet InitElevatorSystem$

Proof note:

The following proof steps demonstrate how the *InitElevatorSystemOK* theorem can be proved.

proof
prove by reduce;
instantiate elevators == elevator0Seq;
use Elev0SeqRangeIsElev0;
use Elev0SeqCardinality;
prove by reduce;
■

2 Elevator system operations

A number of operations are specified for the elevator system. Some apply to a single elevator, with or without information on elevator calls, and others apply to the elevator system as a whole.

2.1 Operation status

Operations return a status code to indicate their success or failure. The following set of status codes represents the values defined so far.

$$OpStatusCode ::= StatusOK \mid \\ StatusOutOfService \mid \\ StatusInvalidMovement$$

The following schema simply returns a success status. It is used in composite operations so its declaration and predicate don't need to be repeated.

Success

$opStatus! : OpStatusCode$

$opStatus! = StatusOK$

2.2 Elevator movement

In this model, elevator movement is broken down into the following components:

- Movement up or down by one floor.
- Visiting a floor (opening doors, exchanging passengers, closing doors, accepting requests from passengers).
- Choosing (calculating) an updated direction of movement, taking into account the pending requests and calls.

Other operations (e.g., deciding whether to visit a floor when an elevator moves past it or is halted there) still need to be defined.

These operation components are specified in the following sections.

2.2.1 Single-floor movement

An elevator may move up one floor or down one floor, depending on its current direction. An elevator may not move if it is currently halted, if it has reached the limit of travel in its current direction, or if it is out of service.

The following operation moves an elevator up one floor.

MoveElevatorUp

$$\Delta Elevator$$

$$\begin{aligned}
&curDir = DirUp \\
&curFloor < nFloors \\
&status = InSvc \\
&curFloor' = curFloor + 1 \\
&curDir' = curDir \\
&status' = status \\
&requests' = requests
\end{aligned}$$

The following operation moves an elevator down one floor.

MoveElevatorDown

$$\Delta Elevator$$

$$\begin{aligned}
&curDir = DirDown \\
&curFloor > 1 \\
&status = InSvc \\
&curFloor' = curFloor - 1 \\
&curDir' = curDir \\
&status' = status \\
&requests' = requests
\end{aligned}$$

The following operation specifies that an elevator cannot move if it is out of service.

MoveElevatorOutOfSvc

$$\begin{aligned}
&\Xi Elevator \\
&opStatus! : OpStatusCode
\end{aligned}$$

$$\begin{aligned}
&status = OutSvc \\
&opStatus! = StatusOutOfService
\end{aligned}$$

The following operation handles the case where movement is not valid because the elevator is halted or cannot move farther in the current direction.

MoveElevatorInvalid

$$\begin{aligned}
&\Xi Elevator \\
&opStatus! : OpStatusCode
\end{aligned}$$

$$\begin{aligned}
&status = InSvc \\
&(curDir = DirHalt) \lor \\
&(curDir = DirUp \land \neg\,(curFloor < nFloors)) \lor \\
&(curDir = DirDown \land \neg\,(curFloor > 1)) \\
&opStatus! = StatusInvalidMovement
\end{aligned}$$

The above partial operations are now combined into a total operation.

$$MoveElevator \mathrel{\widehat=} \\ (MoveElevatorUp \lor MoveElevatorDown) \land Success \lor \\ MoveElevatorOutOfSvc \lor \\ MoveElevatorInvalid$$

The following theorem asserts that the total operation is in fact total, meaning that it can correctly handle any elevator state. Technically, the theorem asserts that the total operation precondition is met in all cases.

theorem MoveElevatorIsTotal
$\forall Elevator \bullet \mathrm{pre}\ MoveElevator$

Proof note:

To prove the theorem, it is necessary to identify all the cases, so that the theorem prover can attempt each one separately. Each of the "split" steps specifies one binary condition; together, they partition the system states.

proof
split status = InSvc;
split $(curDir = DirHalt) \lor$
$(curDir = DirUp \land \neg\ curFloor < nFloors) \lor$
$(curDir = DirDown \land \neg\ curFloor > 1)$;
split curDir = DirHalt;
split curDir = DirUp;
prove by reduce;
■

2.2.2 Visiting a floor

When an elevator visits a floor, it opens its doors, permits entry and exit of passengers, closes its doors, and accepts new floor requests. The elevator's current floor and direction are not changed by this operation.

The normal case is handled first, followed by the case of the elevator being out of service. These cases are then combined in the total operation.

VisitFloorOK

$\Delta Elevator$
$\Delta Calls$
$newRequests? : \mathbb{F}\ Floors$

$status = InSvc$

$calls' = calls \setminus \{curFloor \mapsto curDir\}$
$requests' = (requests \cup newRequests?) \setminus \{curFloor\}$

$status' = status$
$curDir' = curDir$
$curFloor' = curFloor$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{VisitFloorOutOfSvc}} \\
\hline
\Xi Elevator \\
\Xi Calls \\
newRequests? : \mathbb{F}\ Floors \\
opStatus! : OpStatusCode \\
\hline
status = OutSvc \\
opStatus! = StatusOutOfService \\
\hline
\end{array}
$$

$$VisitFloor \mathrel{\widehat{=}} (VisitFloorOK \land Success) \lor VisitFloorOutOfSvc$$

The following theorem asserts that the total operation covers all possible cases of system state and input.

theorem VisitFloorIsTotal
$\forall Elevator;\ Calls;\ newRequests? : \mathbb{F}\ Floors \bullet \mathrm{pre}\ VisitFloor$

Proof note:

The proof separates the two cases corresponding to the operation schemas above.

proof
split status = InSvc;
prove by reduce;
■

2.2.3 Choosing a direction to travel

When an elevator is at a floor, perhaps after visiting it, the system must decide what direction (up, down, or halt) is appropriate for the next movement. This decision depends on the current floor and direction, as well as the pending requests and calls. The result is a new current direction. The elevator's current floor, status, and requests are not changed by this operation. The elevator calls are also not affected.

The first part of the operation identifies the floors above and below the current floor for which there are pending requests or calls. The output parameters (*above*! and *below*!) are piped to the schemas that actually implement the choice of direction. The reason for defining this preliminary operation is to "factor out" the part of the specification that would otherwise be repeated in several partial operations.

The *ChooseDirectionCommon* schema also specifies that the elevator must be in service. The "out of service" condition is handled as an exception.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ChooseDirectionCommon}} \\
\hline
\Delta Elevator \\
\Xi Calls \\
above! : \mathbb{F}\ Floors \\
below! : \mathbb{F}\ Floors \\
\hline
status = InSvc \\
above! = (requests \cup \mathrm{dom}\ calls) \setminus (1 \mathrel{..} curFloor) \\
below! = (requests \cup \mathrm{dom}\ calls) \setminus (curFloor \mathrel{..} nFloors) \\
\hline
\end{array}
$$

The following partial operation schema handles the situation where the elevator direction is up or down (not halted) and there is still a reason to proceed in the current direction. For example, an elevator moving upward continues to do so if there are requests or calls on higher floors.

$$
\begin{array}{|l}
\textit{ChooseDirectionSame} \\
\hline
\Xi \textit{Elevator} \\
\textit{above}? : \mathbb{F}\, \textit{Floors} \\
\textit{below}? : \mathbb{F}\, \textit{Floors} \\
\hline
(\textit{curDir} = \textit{DirUp} \land \textit{above}? \neq \emptyset) \lor \\
(\textit{curDir} = \textit{DirDown} \land \textit{below}? \neq \emptyset) \\
\hline
\end{array}
$$

The following partial operation handles the case where an elevator is moving up or down, there is no reason to continue in the current direction, and there is a reason to go in the opposite direction.

$$
\begin{array}{|l}
\textit{ChooseDirectionReverse} \\
\hline
\Delta \textit{Elevator} \\
\textit{above}? : \mathbb{F}\, \textit{Floors} \\
\textit{below}? : \mathbb{F}\, \textit{Floors} \\
\hline
(\textit{curDir} = \textit{DirUp} \land \textit{above}? = \emptyset \land \textit{below}? \neq \emptyset \land \textit{curDir}' = \textit{DirDown}) \lor \\
(\textit{curDir} = \textit{DirDown} \land \textit{below}? = \emptyset \land \textit{above}? \neq \emptyset \land \textit{curDir}' = \textit{DirUp}) \\
\\
\textit{status}' = \textit{status} \\
\textit{curFloor}' = \textit{curFloor} \\
\textit{requests}' = \textit{requests} \\
\hline
\end{array}
$$

The following partial operation handles the case in which a halted elevator now has a reason to move up or down. If there are requests or calls in both directions, an arbitrary choice is made to move upward.

$$
\begin{array}{|l}
\textit{ChooseDirectionRestart} \\
\hline
\Delta \textit{Elevator} \\
\textit{above}? : \mathbb{F}\, \textit{Floors} \\
\textit{below}? : \mathbb{F}\, \textit{Floors} \\
\hline
\textit{curDir} = \textit{DirHalt} \\
(\textit{above}? \neq \emptyset \land \textit{curDir}' = \textit{DirUp}) \lor \\
(\textit{above}? = \emptyset \land \textit{below}? \neq \emptyset \land \textit{curDir}' = \textit{DirDown}) \\
\\
\textit{status}' = \textit{status} \\
\textit{curFloor}' = \textit{curFloor} \\
\textit{requests}' = \textit{requests} \\
\hline
\end{array}
$$

The following partial operation handles the situation where there is no reason for movement, and the direction is set to halted. (Note that there might be a reason to visit the current floor, because of a call or request, but that is not part of choosing a direction of travel.)

$$
\begin{array}{|l}
\hline
\textit{ChooseDirectionHalt} \\
\Delta \textit{Elevator} \\
\textit{above}? : \mathbb{F}\ \textit{Floors} \\
\textit{below}? : \mathbb{F}\ \textit{Floors} \\
\hline
\textit{above}? = \emptyset \\
\textit{below}? = \emptyset \\
\\
\textit{curDir}' = \textit{DirHalt} \\
\\
\textit{status}' = \textit{status} \\
\textit{curFloor}' = \textit{curFloor} \\
\textit{requests}' = \textit{requests} \\
\hline
\end{array}
$$

The following partial operation handles the case of an elevator that is out of service. No state change takes place, but an error status is returned.

$$
\begin{array}{|l}
\hline
\textit{ChooseDirectionOutSvc} \\
\Xi \textit{Elevator} \\
\Xi \textit{Calls} \\
\textit{opStatus}! : \textit{OpStatusCode} \\
\hline
\textit{status} = \textit{OutSvc} \\
\\
\textit{opStatus}! = \textit{StatusOutOfService} \\
\hline
\end{array}
$$

The partial operations are combined in the following total operation. Note that the common preparation schema pipes its output to the choice operations.

$$
\begin{array}{l}
\textit{ChooseDirection} \mathrel{\widehat=} (\textit{ChooseDirectionCommon} \gg \\
(\textit{ChooseDirectionSame} \lor \textit{ChooseDirectionReverse} \lor \textit{ChooseDirectionHalt} \lor \textit{ChooseDirectionRestart})) \\
\land \textit{Success} \lor \\
\textit{ChooseDirectionOutSvc}
\end{array}
$$

The following theorem asserts that the *ChooseDirection* operation covers all possible cases of system state and input.

theorem ChooseDirectionIsTotal
$\forall \textit{Elevator}; \ \textit{Calls} \bullet \text{pre } \textit{ChooseDirection}$

Proof note:

The theorem is proved by splitting across cases handled by the various partial operations.

proof
split $(\textit{requests} \cup \text{dom } \textit{calls}) \setminus (1 \mathrel{..} \textit{curFloor}) = \emptyset$;
split $(\textit{requests} \cup \text{dom } \textit{calls}) \setminus (\textit{curFloor} \mathrel{..} \textit{nFloors}) = \emptyset$;
split $\textit{curDir} = \textit{DirUp}$;
split $\textit{curDir} = \textit{DirDown}$;
split $\textit{status} = \textit{InSvc}$;
prove by reduce;
■

3 Remaining work

This specification is currently incomplete. At least the following issues need to be dealt with:

- Deciding whether to visit a floor when an elevator has moved to it or is halted there.
- Handling new calls from waiting passengers.
- Implementing policies such as cancelling all pending requests when an elevator reaches the top or bottom floors.
- Managing the overall elevator system by invoking operations on individual elevators. This will likely involve promoting elevator-level operations to the aggregate (sequence) of elevators.
- Taking elevators out of service and returning them to service.