Devising a Formal Specification for an Elevator Controller

Technical Report UMCIS–1994–10

H. Conrad Cunningham, Viren R. Shah, and Shu Shen

cunningham@cs.olemiss.edu

Software Methods Research Group
Department of Computer and Information Science
University of Mississippi
302 Weir Hall
University, Mississippi 38677 USA

September 1994

H. Conrad Cunningham, D.Sc.
Assistant Professor
Department of Computer and Information Science
University of Mississippi
302 Weir Hall
University, Mississippi 38677
USA

cunningham@cs.olemiss.edu

Devising a Formal Specification for an Elevator Controller

H. Conrad Cunningham, Viren R. Shah, and Shu Shen

Department of Computer and Information Science
University of Mississippi
University, Mississippi 38677 USA

September 1994

Abstract

This is a working paper for the Software Methods Research Group. A full version of the paper is forthcoming.

1 Introduction

Natural language, with all of its nuances and ambiguities, is an effective medium for most aspects human communication, but software specification demands more precision than can be supplied by natural language alone [5]. It demands the kind of precision that only formal notations can provide. Formal notations naturally lead the specifier to identify important issues that might otherwise be obscured by the ambiguities of a natural language statement of a problem. It is better to confront such issues early, before design begins, rather than later when a costly design change may be required or later yet after a defective product has been delivered. Formal notations enable specifiers to record requirements precisely and communicate them unambiguously to software developers. In addition, the formal specification provides a framework for the systematic development and rigorous verification of the software product.

A precise specification is especially important when one is developing a *reactive program*, that is, a program, like an operating system, whose role is to maintain an ongoing interaction with its environment rather than just to compute some final value on termination [3]. Because of the complexity of the interactions of concurrent components, reactive programs are notoriously difficult to get right. The rigor promoted by formal notations can help tame this complexity.

In this case study, we examine an interesting problem and seek to devise an elegant formal specification for a reactive program to solve the problem. The problem concerns the requirements for the controlling mechanism for a system of N separate elevators, each of which can visit any of the M floors of a building [4].

1. Each elevator has a set of buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the elevator.

2. Each floor has two buttons (except the ground and top floors), an up button to request transport to a higher floor and a down button to request transport to a lower floor. These buttons illuminate when pressed. The illumination is cancelled when an elevator visits the floor and is either moving in the desired direction or has no outstanding requests. In the latter case, if both floor buttons are pressed, only one should be cancelled.

3. When an elevator has no requests to service, it should remain at its final destination with its doors closed and await further requests.

4. Each elevator has an emergency button that, when pressed, causes a warning signal to be sent to the site manager. The elevator is then deemed "out of service". Each elevator has a mechanism to cancel its "out of service" status.

This case study seeks to construct a specification of the elevator controller as an open, reactive system. That is, the case study seeks a formulation that gives the properties of the controller in terms of its interactions with an unspecified environment. The environment consists of the people, devices, and programs with which the elevator controller must interact. The controller specification must also state the assumptions it makes about its environment.

The problem poses several challenges that this case study should address:

- uncovering the sometimes subtle interactions among the operations of the system (e.g., pushing buttons, moving the elevator car, changing direction of movement, opening doors, etc.)
- considering the (often unstated) expectations that people who use the elevators have about the system's operation,
- stating the complex properties of the reactive system in a concise and modular way,
- devising a general specification that is as "weak" as is practical (e.g., allowing the designer to choose among many possible methods for scheduling elevators to service up and down requests rather than imposing a particular method a priori),
- identifying a reasonable set of assumptions that the elevator can make about its environment.

The remainder of the paper is organized as follows. Section 2 gives the specification model. Sections 3 and 4 develop the formal specification, first for a single elevator and then for a multiple elevator system.

2 Specification Model and Logic

This case study adopts Chandy and Misra's UNITY model [1] as the notation and logic for specification of the elevator controller. A UNITY program is, in essence, a nondeterministic program in Dijkstra's Guarded Commands notation [2] with the form

$$\textit{initialize variables}\ ;\ \mathbf{do}\ g_0 \rightarrow a_0\ [\!]\ g_1 \rightarrow a_1\ [\!]\ \cdots\ [\!]\ g_{n-1} \rightarrow a_{n-1}\ \mathbf{od}$$

where

- n is a finite constant,
- a_i (for $0 \leq i < n$) denotes an atomic, terminating, deterministic, multiple-assignment command that accesses a fixed set of variables,
- the execution is *fair* in the sense that, if a guard g_i holds at some point in an infinite computation, then there exists a later point at which either a_i is executed or g_i no longer holds.

The union of two UNITY programs consists of a program in which the initialization is formed by the union of the two initializations (assuming no inconsistency exists) and the **do** loop is formed by the union of the two sets of guarded commands.

UNITY's operational model represents a program as the set of all maximal execution sequences of the corresponding **do** program. A maximal execution sequence records the sequence of states corresponding to a possible execution of the program. The transition from one state to the next corresponds to the execution of an atomic action. These execution sequences are either infinite or end in a state in which all guards of the **do** program are false. This operational model allows program properties to be stated in terms of temporal logic [3].

This case study uses UNITY's simple subset of temporal logic [1] to specify the properties that a program must be constructed to satisfy. For the purposes here, we consider the logical relations **initially**, **unless**, **stable**, **invariant** (abbreviated as **inv**), **constant**, and $\longmapsto$ (read "leads-to"). Informally, for arbitrary predicates p and q on program states:

- **initially** p means that p must hold for the initial state of every execution sequence. (In the **do** loop above, the predicate p must hold between the initialization and the beginning of the loop.)
- p **unless** q means that, for any execution sequence, if p holds for some state, then either q never holds and p continues to hold for all succeeding states or q holds eventually and p holds at least until q holds. (If $p \wedge \neg q$ holds before the execution of an action in the loop body, then $p \vee q$ must hold after its execution.)
- **stable** p means that, for any execution sequence, if p holds for some state, then p must continue to hold for all succeeding states of the sequence. That is, **stable** p means that p **unless** $false$ holds. (A stable predicate is preserved by the actions in the body of the **do** loop above.)
- **invariant** p means that p must hold for all states of all execution sequences. That is, both **initially** p and **stable** p hold. (UNITY invariants are loop invariants of the **do** loop above.)
- **constant** p means that both **stable** p and **stable** $\neg p$ hold. That is, for any execution sequence, p must either remain true forever or remain false forever.
- $p \longmapsto q$ means that, if p holds for any state of any execution sequence, q must also hold within a finite number of steps in the execution sequence.

The annotation *prop* **in** P denotes that *prop* is a property of program P considered in isolation and $P \,[]\, Q$ denotes the union of programs P and Q. A UNITY conditional property specified a property of a reactive program that is dependent upon properties of the program's environment. The specification

Hypothesis: *Property_List_1*
Conclusion: *Property_List_2*

means that the program must satisfy *Property_List_2* whenever *Property_List_1* holds.

3 One Elevator

The elevator controller, as described in Section 1 has many facets that must be captured in a specification. Handling the entire specification at once would thus be quite unwieldy. For the purpose of clarity and modularity, it was decided to first specify a single-elevator system and then build upon it to achieve the required specification.

The single-elevator system is a restricted case of the multi-elevator system, in which N, the number of elevators, is 1.

In specifying any system, there are usually a few logical *entities* that are inherent in the description. For this case in particular, the elevator buttons, the up buttons and the down buttons on each floor, the position and direction of the elevator, and the emergency stop button are all functionally discrete units that need to be represented in the formal specification.

The above *units* can be used as a basis upon which to construct our specification. As required by UNITY, the variables and constants representing these units first need to be declared.

M	: integer constant, $M \geq 1$, number of floors
$b[1..M]$	: boolean, internal button, light on if true
$up[1..M-1]$	: boolean, external up button, light on if true
$dn[2..M]$	: boolean, external down button, light on if true
pos	: integer, floor location for elevator car in $[1..M]$
dir	: integer $-1 \leq dir \leq 1$, DOWN, HOLD, UP, direction of movement
$door[1..M]$	: boolean, true if elevator, floor doors open
$emstop$	: boolean, true if emergency stop button on

For convenience, we assume that all free variables are universally quantified. We also assume that out-of-range references to boolean arrays have the value false. Within this section, we shall refer to the single-elevator system as *Elev* and its environment as *User*.

Once the variables have been declared, constraints on the scope of the variables need to be stated. The interactions among these variables should also be elaborated upon. This serves the dual purpose of explicitly excluding unwanted scenarios, and reiterating implicit properties that might be otherwise overlooked.

Consider the variable *pos*, the position of the elevator. As the building has M floors, the valid values for the position of the elevator has to be confined to those M floors. This constraint can be stated as an invariant property of the elevator program.

$$\textbf{inv} \quad 1 \le pos \le M \ \textbf{in} \ Elev \tag{1}$$

Similarly, the elevator's direction of movement, recorded by the *dir* variable, can only be changed by the elevator controller, and, then, only among the three valid directions — up, down, and hold. This requirement can be formalized with the use of three statements, i.e. an invariant and two constant properties. The invariant restricts the valid values for the *dir* variable, while the constant properties define and constrain the functionality of *Elev* and *User*.

$$\textbf{inv} \quad dir \in \{\mathrm{UP}, \mathrm{DOWN}, \mathrm{HOLD}\} \ \textbf{in} \ Elev \tag{2}$$
$$\textbf{constant} \quad pos = k \ \textbf{in} \ User \tag{3}$$
$$\textbf{constant} \quad dir = d \ \textbf{in} \ User \tag{4}$$

We also need to constrain the doors on each floor to open only when the elevator is present on that floor. The simplest way to do this is to add an invariant property for *Elev*.

$$\textbf{inv} \quad door.j \ \Rightarrow \ pos = j \ \textbf{in} \ Elev \tag{5}$$

Another property that is convenient from a user standpoint is that the door should not be opened at floor j unless a request has been made for that floor—this will prevent the elevator from stopping and opening the door at each floor.

$$\begin{aligned} \neg door.j \quad \textbf{unless} \quad & (b.j \wedge dir \neq \mathrm{HOLD}) \ \vee \\ & (up.j \wedge dir = \mathrm{UP}) \ \vee \\ & (dn.j \wedge dir = \mathrm{DOWN}) \ \textbf{in} \ Elev \end{aligned} \tag{6}$$

However, for the above property to hold true, we need to assume that the *User* is not allowed to open a closed door, i.e., a closed door is stable in the *User* environment.

$$\textbf{stable} \quad \neg door.j \ \textbf{in} \ User \tag{7}$$

Once the more obvious properties have been defined, each logical entity in the system can be looked at in detail. For the elevator, we have already listed these entities. These "groupings" of atomic objects into "entities" are based upon the functionality of the objects and their interactions with other entities. It would be redundant and unnecessary to classify each up button on a floor as a separate unit. Also, since the response of the elevator to the up button and the down button, is effectively different, it would be counterproductive to group the up and down button on each floor together as an unit. The goal in grouping objects is to be able to generalize properties to a class of similar objects, without weakening a specification to the point of uselessness.

We begin by examining the functioning of the internal buttons of the elevator. An implicit requirement for a button is that an elevator should not be able to generate its own requests—all requests must come from the elevator's environment. We formalize this as follows:

$$\textbf{stable} \;\; \neg b.j \;\textbf{in}\; \mathit{Elev} \tag{8}$$

Another requirement for the internal buttons is that the pressing of a button should result in the elevator visiting the corresponding floor. The term "visiting" can be split into three atomic actions: the button must be cleared, the position of the elevator must be the appropriate floor, and the door at that floor should be open. Formalizing this, we obtain,

$$b.j \;\longmapsto\; \neg b.j \wedge pos = j \wedge door.j \;\textbf{in}\; \mathit{Elev} \,[\!]\, \mathit{User} \tag{9}$$

Upon further reflection, however, the above property seems to have several inconsistencies. First, it allows the illumination of the button to be cancelled before the elevator visits that floor. This could result in confusion on part of the user. More seriously, from an user standpoint, the above property does not require the elevator to visit that floor on the first pass, nor does it force the elevator to clear the button on its first visit to that floor.

The first two objections can be overcome by weakening the right hand side of the property, and then adding safety properties as follows

$$b.j \;\longmapsto\; \neg b.j \;\textbf{in}\; \mathit{Elev} \,[\!]\, \mathit{User} \tag{10}$$
$$b.j \;\;\textbf{unless}\;\; pos = j \wedge door.j \;\textbf{in}\; \mathit{Elev} \tag{11}$$

This weakening of the property allows requests to be cancelled by the environment, and ensures that the elevator cannot clear the light before visiting the floor. In order to force the elevator to clear the button on its first visit, we add

$$pos = j \wedge door.j \;\;\textbf{unless}\;\; \neg b.j \;\textbf{in}\; \mathit{Elev} \tag{12}$$

A property that will make the elevator visit the floor on its first pass is also needed,

$$b.j \wedge pos = j \wedge dir \neq \text{HOLD} \;\;\textbf{unless}\;\; door.j \;\textbf{in}\; \mathit{Elev} \tag{13}$$

Thus, the original property (9) has been modified into the set of properties (10) (11) (12) (13), which specify exactly how we want the elevator to react in that situation. It was due to the formal specification process that we were able to point out and overcome potential problem areas, before the cost of handling them became significant.

The next unit to be specified is the *up* buttons. Again, there are several properties, implicit to the basic functioning of the *up* buttons, that we need to state. First, as for the internal buttons, we do not want the elevator to be able to illuminate an *up* button,

$$\textbf{stable} \;\; \neg up.j \;\; \textbf{in} \;\; Elev \tag{14}$$

An illuminated *up* button also needs to result in a visit to that floor

$$up.j \;\; \longmapsto \;\; \neg up.j \wedge pos = j \wedge dir = \text{UP} \wedge door.j \;\; \textbf{in} \;\; Elev \;[\,]\; User \tag{15}$$

This is analogous to the property we had for the internal buttons. It also, suffers from the same shortcomings. So, we weaken the property to result in the following

$$up.j \;\; \longmapsto \;\; \neg up.j \;\; \textbf{in} \;\; Elev \;[\,]\; User \tag{16}$$
$$up.j \;\; \textbf{unless} \;\; pos.j \wedge door.j \;\; \textbf{in} \;\; Elev \tag{17}$$
$$pos = j \wedge door.j \;\; \textbf{unless} \;\; \neg up.j \;\; \textbf{in} \;\; Elev \tag{18}$$
$$up.j \wedge pos = j \;\; \textbf{unless} \;\; door.j \;\; \textbf{in} \;\; Elev \tag{19}$$

These properties constrain the up buttons interactions with the other entities, in the same manner that the corresponding properties did for the internal buttons. Properties (16) and (17) ensure that the up requests can be cancelled by the environment and that the elevator can't clear the up button before visiting that floor. In turn, (18) and (19) make the elevator visit the floor on its first pass, and force it to clear the button on its first visit. However, this set of properties still does not constrain the interactions between the units as strenuously as needed. As can be seen by a careful perusal of the above properties, there are no restrictions on the direction in which the elevator is moving when answering a request. This could conceivably result in an up request being answered by the elevator when it is moving downwards. It would also prevent the progress property (16) from holding, since with the properties as they stand, it is not possible to guarantee that an *up* request will be answered. This problem can be rectified by strengthening the left hand side of each of the properties in the *up* unit. As the problem is caused by a lack of a direction vector, we strengthen the properties by inserting a direction clause. We want to restrict the elevator to answering up requests only if its direction is up, and down requests only when its direction is down. So, the modified properties are:

$$up.j \;\; \longmapsto \;\; \neg up.j \;\; \textbf{in} \;\; Elev \;[\,]\; User \tag{20}$$
$$up.j \;\; \textbf{unless} \;\; pos.j \wedge door.j \wedge dir = \text{UP} \;\; \textbf{in} \;\; Elev \tag{21}$$
$$pos = j \wedge door.j \wedge dir = \text{UP} \;\; \textbf{unless} \;\; \neg up.j \;\; \textbf{in} \;\; Elev \tag{22}$$
$$up.j \wedge pos = j \wedge dir = \text{UP} \;\; \textbf{unless} \;\; door.j \;\; \textbf{in} \;\; Elev \tag{23}$$

As the up and down buttons perform identical tasks, albeit in opposite directions, the properties for the down buttons will be similar to the above properties with two obvious exceptions. All *up.j* clauses will become *dn.j* clauses, and all $dir = \text{UP}$ clauses will become $dir = \text{DOWN}$ clauses.

We now continue on to the properties for the direction unit. Although direction has already been used in some of the properties stated above, it served as a secondary variable;

the main focus of the properties was on other entities. We now want to verbalize those properties that are based primarily on *dir*.

The first rule for constraining the value of *dir* involves specifying under what circumstances a change in direction can take place. One obvious constraint is that we do not want to change directions in an elevator when a door is open. Also, if an elevator is moving up (or down), and has outstanding requests for floors above (or below) it. Then we do not want it switching direction before all the requests that are above (or below) it are fulfilled. In order to formalize this, a variable for outstanding requests needs to be declared.

There are, however, three classes into which outstanding requests can be split based upon the elevator's current position. The first case consists of simply the event in which the internal button for the current floor, *b.j* is pressed. The second case is one in which the floor button on the current floor, corresponding to the elevator's current position is pressed, i.e., if $dir = \mathrm{UP}$ and $pos = j$, and the *up* button on floor j is pressed. The last case consists of any occurences of the floor buttons on, or the internal buttons for any floor lying in the elevators current direction, being pressed. We could define one single variable for all these cases, as below,

$$\begin{aligned}\textbf{inv}\quad req.d.p \;\equiv\;& b.p \vee ((d = \mathrm{DOWN} \vee d = \mathrm{UP}) \;\wedge\\ & (d = \mathrm{DOWN} \Rightarrow (dn.p \vee (\exists k : 1 \le k < p : up.k \vee dn.k \vee b.k))) \;\wedge\\ & (d = \mathrm{UP} \Rightarrow (up.p \vee (\exists k : p \le k < M : up.k \vee dn.k \vee b.k))))\end{aligned} \tag{24}$$

This makes the property obscure and bulky. Another disadvantage is that we can not separately reference each of the cases mentioned above. We should thus declare two different variables for the two cases. For the first case, we already have concise formalisms—*b.j* and *up.j* (or *dn.j*)—and don't require variables for them.

$$\begin{aligned}\textbf{inv}\quad req^{+}.d.p \;\equiv\;& (d = \mathrm{DOWN} \vee d = \mathrm{UP}) \;\wedge\\ & (d = \mathrm{DOWN} \Rightarrow (\exists k : 1 \le k < p : up.k \vee dn.k \vee b.k)) \;\wedge\\ & (d = \mathrm{UP} \Rightarrow (\exists k : p < k \le M : up.k \vee dn.k \vee b.k))\end{aligned} \tag{25}$$

We build upon req^{+} by grouping it with the second case to arrive at

$$\begin{aligned}\textbf{inv}\quad req.d.p \;\equiv\;& (d = \mathrm{DOWN} \vee d = \mathrm{UP}) \;\wedge\\ & (d = \mathrm{DOWN} \Rightarrow dn.p \vee b.p \vee req^{+}.\mathrm{DOWN}.p) \;\wedge\\ & (d = \mathrm{UP} \Rightarrow up.p \vee b.p \vee req^{+}.\mathrm{UP}.p)\end{aligned} \tag{26}$$

Thus, *req* stands for any outstanding request, i.e., any request from one of the floors in the elevators direction, or any request from the current floor to a floor in the elevator's direction of movement.

We can now state the two properties for *dir*

$$dir = d \;\;\textbf{unless}\;\; \neg door.j \;\;\textbf{in}\;\; Elev \tag{27}$$

$$dir = d \wedge d \neq \mathrm{HOLD} \;\;\textbf{unless}\;\; dir = \mathrm{HOLD} \wedge \neg req.d.pos \;\;\textbf{in}\;\; Elev \tag{28}$$

Another property of direction is that if the elevator is holding (dir = HOLD) and there is an up request, we do not want the elevator to move until its direction has been changed to up. A similar constraint holds if there is a down request.

$$\begin{aligned} &dir = \text{HOLD} \quad \textbf{unless} \quad (dir = \text{UP} \wedge req.\text{UP}.pos) \;\vee \\ &\qquad\qquad\qquad\qquad (dir = \text{DOWN} \wedge req.\text{DOWN}.pos) \;\textbf{in}\; Elev \qquad (29) \\ &pos = j \quad \textbf{unless} \quad dir \neq \text{HOLD} \wedge req^{+}.dir.j \wedge pos = j + dir \;\textbf{in}\; Elev \qquad (30) \end{aligned}$$

Upon examining the specification obtained so far, we have taken into account most possible scenarios for an elevator. Nevertheless, we need to keep in mind that this is an open system specification. As such, we are supposed to make minimal assumptions about the system's environment. There are two possible occurences that are not accounted for by the above specification. The first involves an unknown factor in the environment blocking the elevator door indefinitely or an user pressing the *up* or *down* button infinitely while the door is open, and thus preventing it from closing. The second one concerns a person inside the elevator continuously pressing the internal button for the current floor. This would prevent the elevator from ever moving since it would be answering these requests forever. So we declare a variable for the second case: These events, while improbable, are still possible, and therefore need to be dealt with in the formal specification. Both the aforementioned events would stop the progress properties (10) (20) from being satisfied, and thus void the entire specification.

The first event is a purely environmental one, having no relation to any of the elevator functions. Hence, specifying any property that would overcome this problem is impossible. So, we make an environmental assumption stating that none of the doors for the elevator will be infinitely blocked. We also can formalize this generally with

$$door.j \;\longmapsto\; \neg door.j \;\textbf{in}\; Elev \,[]\, User \qquad (31)$$

The second event relates to the manner in which the functions of the elevator are being handled by the system. In this case, it is possible to prevent the endless opening and closing of the elevator doors which can be caused by repetitively pressing the internal button for the current floor. This can be achieved by preventing the elevator from consecutively satisfying this kind of request more than a fixed number of times. It also engenders making the elevator have knowledge of having visited a floor. A boolean flag would suffice in this respect. The flag will be true when an elevator just arrives at a floor, and will be set to false if the door at that floor is opened.

$$\begin{aligned} &\textbf{initially} \quad novisit \;\textbf{in}\; Elev \qquad (32) \\ &novisit \quad \textbf{unless} \quad (\exists k :: door.k) \;\textbf{in}\; Elev \qquad (33) \\ &door.j \quad \textbf{unless} \quad \neg novisit \;\textbf{in}\; Elev \qquad (34) \end{aligned}$$

The *novisit* variable will remain false as long as the the elevator remains at the same floor and its direction remains the same.

$$\neg novisit \wedge pos = j \wedge dir = d \ \ \textbf{unless} \ \ pos \neq j \vee dir \neq d \ \textbf{in} \ Elev \tag{35}$$

The inclusion of a new variable means that some of the earlier properties will have to be changed. The changes will occur mainly in properties constraining the arrival and departure of the elevator and also those properties dictating when the elevator door should be opened.

$$b.j \wedge pos = j \wedge dir \neq \text{HOLD} \wedge novisit \ \ \textbf{unless} \ \ door.j \ \textbf{in} \ Elev \tag{36}$$

$$up.j \wedge pos = j \wedge dir = \text{UP} \wedge novisit \ \ \textbf{unless} \ \ door.j \ \textbf{in} \ Elev \tag{37}$$

$$dn.j \wedge pos = j \wedge dir = \text{DOWN} \wedge novisit \ \ \textbf{unless} \ \ door.j \ \textbf{in} \ Elev \tag{38}$$

$$\begin{aligned} dir = d \wedge d \neq \text{HOLD} \ \ \textbf{unless} \ \ & \neg req^{+}.d.p \wedge novisit \ \wedge \\ & ((dir = \text{HOLD} \wedge \neg req.(-d).pos) \ \vee \\ & (dir = -d \wedge req.(-d).pos)) \ \textbf{in} \ Elev \end{aligned} \tag{39}$$

$$\begin{aligned} pos = j \ \ \textbf{unless} \ \ & dir \neq \text{HOLD} \wedge req^{+}.dir.j \ \wedge \\ & pos = j + dir \wedge novisit \ \textbf{in} \ Elev \end{aligned} \tag{40}$$

With this, the specification for the single elevator is nearly complete. The elevator has all the functionality it is required to have, and we have effectively blocked the environment from causing any problems with the elevator. The only remaining addition to this specification is the emergency stop button.

The functionality of the emergency stop button is simple. As soon as the emergency button is pressed, nothing about the physical condition of the elevator should change until the button is cleared. Thus, the *dir*, *pos*, and *door* variables need to be kept constant for the duration of the emergency. To formalize this, we utilize two properties, a **constant** property and an **unless** property.

$$\textbf{constant} \ \ emstop \ \textbf{in} \ Elev \tag{41}$$

$$pos = j \wedge door = p \wedge dir = d \ \ \textbf{unless} \ \ \neg emstop \tag{42}$$

The emergency stop button was easy to specify since it does not interact with any of the other units; rather, it supersedes the other units during the time period in which it is pressed. Such units should always be left till the end as they are relatively simple to add on to an existing specification.

This was the specification for the single elevator system. Using this we can construct the specification for a multi-elevator system.

4 Multiple Elevators

The single elevator specification in Section 3 served to focus our attention on the fundamental properties, actions and functionality of an elevator. Now that these have been specified within the single elevator specification, we can use it as a foundation upon which to construct the specification for the overall multi-elevator problem. It allows us to concentrate on functionality that is a step higher than that which we examined in the previous sections.

It will be necessary to extend the specification arrived at in Section 3 to a slight degree, in order to be able to use it in a multi-elevator system. The process of *extending* the specification will involve mapping the original variables to other variables, so as to allow for multiple elevators. The changes are trivial and will not diminish the effectiveness of the single-elevator specification. While, in Section 3, there were only two programs, *Elev* and *User*, we now have three programs, namely the actual user (*User′*), the manager (*Mgr*), and the elevators(*Elev.i*). The programs *User′* and *Mgr* made up the *User* module discussed in Section 3. We will also need to change the variables *up.j* and *dn.j* to *up.i.j* and *dn.i.j*, where *i* refers to the elevator being referred to, and *j* stands for the floor the button is on.

M	: integer constant, $M \geq 1$, #floors
N	: integer constant, $N \geq 1$, #elevators
$up[1..N][1..M-1]$	: booleans where *up.i.j* is *up.j* of *Elev.i*
$dn[1..N][2..M]$	: booleans where *dn.i.j* is *dn.j* of *Elev.i*

Also, in this section, we will be looking at *up* and *dn* from a slightly different perspective. In the previous section both variables were assumed to have been direct representations of the actual up and down buttons. Now, however, we have an additional component in the system, namely the manager, that acts as the liaison between the user and each of the elevators. Owing to the fact that the link between the elevators and the user goes through the manager, the functionality of the *up* and *dn* buttons has to change. They no longer represent the physical floor buttons but, in fact, become virtual buttons that are an interface between the elevators and the manager.

Due to this development, we need to have variables representing the real up and down buttons. We declare these as shown.

$rup[1..M-1]$	: boolean, real external up button/light on if true
$rdn[2..M]$	: boolean, real external down button/light on if true

As needed for all new variables, we impose constraints on them, as well as upon the *up* and *dn* buttons.

$$\textbf{stable}\ \ rup.j\ \ \textbf{in}\ User' \tag{43}$$

$$\textbf{constant}\ \ rup.j\ \ \textbf{in}\ Elev.i \tag{44}$$

$$\textbf{stable}\ \ rdn.j\ \ \textbf{in}\ User' \tag{45}$$

$$\textbf{constant}\ \ rdn.j\ \ \textbf{in}\ Elev.i \tag{46}$$

$$\textbf{constant}\ \ up.j\ \ \textbf{in}\ User' \tag{47}$$

$$\textbf{constant}\ \ dn.j\ \ \textbf{in}\ User' \tag{48}$$

We also need to ensure that the manager cannot *conjure up* a request.

$$\textbf{stable} \;\; \neg rup.j \;\textbf{in}\; Mgr \tag{49}$$

$$\textbf{inv} \;\; up.i.j \;\Rightarrow\; rup.j \;\textbf{in}\; Mgr \tag{50}$$

The working of the entire elevator system also needs to be modified to accommodate the new developments. Each elevator will still respond to the pressing of its own internal buttons. However, the method for fulfilling up and down requests (i.e. floor requests) will have to change. It is now the manager that decides which elevator satisfies each floor button request. Whenever a floor button request is made, i.e, a *rup.j* or *rdn.j* button is pressed, the manager, using an allocation algorithm, will signal the appropriate elevator(s) to fulfill that request. As soon as the request is satisfied by an elevator, the manager clears all the signals it had sent to the elevators and also clears the appropriate *rup.j* or *rdn.j* button.

There are several algorithms that can be used by the manager for allocation of requests. The simplest way would be to propagate a floor button request to all elevators, and clear the signals as soon as the first elevator going in the right direction visits the floor. This is a rather simplistic and inefficient routine. Another method would be for the manager to send the request to the elevator which is in a situation to satisfy the request the fastest. This could be done by examining various aspects of each elevator such as its proximity to the request generating floor, its direction, the number of requests outstanding for that elevator, and other such relevant factors.

These allocation routines will differ according to the implementation chosen, and as such, they should not be decided upon at this stage. The purpose of a formal specification, such as the present one, is to arrive at a set of properties that act as a skeleton which the implementors can flesh out according to the requirements of the situation. Thus, we need to build a specification for the manager which will ensure that it will do its job, and one that will not narrow the options available to the implementors. Once an implementation policy has been decided upon, the specification can be augmented to reflect the implementation-specific details by adding or modifying properties. Based on this, we need to select properties that allow a degree of flexibility in the implementation of the manager.

The best course seems to be to allow the manager to signal as many elevators as it wants to. We also allow the manager to arbitrarily clear an elevator's *up* and *dn* buttons, even though the corresponding *rup* or *rdn* button has not been cleared. This gives the manager the ability to dynamically change the request allocation to the elevators. We formalize the above for the up floor requests, as seen below. The properties for down floor requests are similar.

$$rup.j \;\;\textbf{unless}\;\; (\exists i :: up.i.j) \;\textbf{in}\; Mgr \tag{51}$$

$$rup.j \;\longmapsto\; (\exists i :: up.i.j) \;\textbf{in}\; Mgr \,[]\, Elev' \,[]\, User' \tag{52}$$

$$\textbf{stable} \;\; (\exists i :: up.i.j) \;\textbf{in}\; Mgr \tag{53}$$

These three properties state that when a *rup.j* button is pressed, there will eventually be an elevator for which the *up.i.j* button is set. Thus there will eventually be at least one elevator responding to that request. Also, once the manager sends the signal to an elevator, the manager must ensure that until the request is fulfilled, there is always at least one elevator which has that request pending.

We now address the question of how, once a request has been satisfied, the manager should propagate the result to the other elevators as well as to the appropriate floor button. In order to do so, the manager needs to have some way to keep track of the allocations that it makes. The most convenient way to do this is by maintaining a boolean array that maps the virtual up and dn buttons to the real rup and rdn buttons. We shall discuss the variables needed for only the up floor requests, since the variables needed for down floor requests are similar.

For the up and rup buttons, we declare a variable called mup.

$mup[1..N][1..M-1]$: boolean, true if $up.i.j$ mapped to $rup.j$

$$\textbf{inv} \quad mup.i.j \equiv up.i.j \ \textbf{in} \ Mgr \tag{54}$$

The above property states that the mapping in mup will always represent the actual state of the corresponding up buttons. We also want to make sure that mup remains internal to the manager. This is done by making *mup* a constant in *User′* and *Elev.i*.

$$\textbf{constant} \quad mup.i.j \ \textbf{in} \ User' \tag{55}$$

$$\textbf{constant} \quad mup.i.j \ \textbf{in} \ Elev.i \tag{56}$$

Once a floor button request has been satisfied, the elevator involved clears its appropriate *up* button, and the manager needs to propagate this to the *rup* button, as well as to the other elevators.

$$mup.i.j \wedge \neg up.i.j \quad \textbf{unless} \quad \neg rup.j \ \textbf{in} \ Mgr \tag{57}$$

$$mup.i.j \wedge \neg up.i.j \ \longmapsto \ \neg rup.j \ \textbf{in} \ Mgr \tag{58}$$

The above specification is still not complete. While it allows the manager to do what we require, namely to dynamically change allocations, it does not constrain it enough. Given the above properties, it is conceivable that the manager could keep changing the allocations in such a manner as would prevent the request from ever being satisfied. This would violate the progress properties within the specification. We need to come up with a general solution to this that would enable the specification to be as flexible as we want. To solve this, we introduce another auxiliary variable *lup*. This variable implements a locking functionality in *mup*.

$lup[1..N][1..M-1]$: boolean, true if $up.i.j$ mapping is locked

We want to ensure that the progress properties hold. So, we need to make certain that ultimately there will be one elevator that will stay allocated to a request until that request has been satisfied. We use *lup* to perform this function, which is namely that of, after an arbitrary amount of time, locking one elevator to a particular request.

$$\textbf{inv}\ \ lup.i.j\ \Rightarrow\ \ mup.i.j\ \textbf{in}\ Mgr \tag{59}$$

$$lup.i.j\ \ \textbf{unless}\ \ \neg rup.j\ \textbf{in}\ Mgr \tag{60}$$

$$\begin{aligned}(\exists i :: up.i.j)\ \longmapsto\ & (\exists i :: lup.i.j) \vee \\ & (\exists i :: \neg up.i.j \wedge mup.i.j)\end{aligned} \tag{61}$$

The down floor requests have *mdn* and *ldn* as their counterparts to *mup* and *lup*. Each of the above properties concerning *mup* and *lup* is also duplicated for *mdn* and *ldn*.

Acknowledgements

The authors' work was supported by the National Science Foundation under Grant CCR-9210342. The government has certain rights in this material.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[3] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[4] D. Marca and M. T. Harandi, editors. *Proccedings of the Fourth International Workshop on Software Specification and Design.* IEEE, 1987.

[5] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.