Report Number: WUCS-87-2

1987-02-01

System Specifications and Flow Control

Authors: Gruia-Catalin Roman and Michael E. Ehlers

We started with an approach intended for the formalization of software/hardware interactions in distributed systems and applied it to an elevator control problem. The emphasis on physical relevance, intrinsic to the approach, has resulted in a new treatment of the elevator problem, one which reflects faithfully the structural and behavioral properties of the system components and which allows the designer to work on the algorithm for elevator movement and its proof in the realistic context of the total system. In addition to presenting the model we discuss several issues important in ensuring the physical relevance of software specifications: (1) boundary validation, (2) failure analysis, and (3) design-rules formulation and enforcement.

SYSTEM SPECIFICATIONS AND FLOW CONTROL

Gruia-Catalin Roman and Michael E. Ehlers

WUCS-87-2

February 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Proceedings of the 4th International Workshop on Software Specification and Design, April 1987, pp. 118-125.

SYSTEM SPECIFICATIONS AND PHYSICAL RELEVANCE

Gruia-Catalin Roman and Michael E. Ehlers

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

We started with an approach intended for the formalization of software/hardware interactions in distributed systems and applied it to an elevator control problem. The emphasis on physical relevance, intrinsic to the approach, has resulted in a new treatment of the elevator problem, one which reflects faithfully the structural and behavioral properties of the system components and which allows the designer to work on the algorithm for elevator movement and its proof in the realistic context of the total system. In addition to presenting the model we discuss several issues important in ensuring the physical relevance of software specifications: (1) boundary validation, (2) failure analysis, and (3) design-rules formulation and enforcement.

1. Introduction

Although there is a general consensus regarding the nature of distributed systems, distributed processing models do not generally capture the dependency that exists between distributed software and the characteristics of the operating system and hardware architecture. Approaches as diverse as data flow, Petri nets, exchange functions, calculus of communicating processes, CSP, actors, and others [1] exhibit this important limitation. While compatible with the notion of distribution, these formal systems do not attempt to model distributed software but distributed computation, an abstraction somewhat removed from the realities of distributed processing.

Our research is based on a distributed system model, called *Virtual System* [2], that enables the designer to formulate and answer questions regarding the logical correctness and performance characteristics of distributed software when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delays, hardware selection, scheduling policies, etc., must be considered. By capturing the functionality, architecture, scheduling policies, and performance attributes of the system it models, the Virtual System enables us to reformulate the traditional distributed software verification concerns by addressing the issue of establishing the correctness and performance characteristics of software running on a particular distributed hardware architecture and using a particular operating system.

In contrast to other distributed system models where the architecture is treated only as a collection of computational sites, the Virtual System attributes arbitrary computational characteristics and behavior to processors. Furthermore, the model permits independent elaboration, analysis, and modification of its components while allowing easy identification of the entities affected by particular changes. For instance, the architecture can be modified without changing the functionality, to determine how best to implement a required set of functions. Alternatively, the functionality can be restructured without changing the architecture, to determine how best to take advantage of a particular architecture or feature of the system hardware.

We argue here that such specifications force designers to give more careful consideration to the physical relevance of the design being produced, thus making it easier for two implementors looking at the same design to interpret it in the same manner. After all, design involves not the formulation of elegant mathematical entities but the unambiguous and correct representation of postulated system components together with their interactions and processing environment. The concept of physical relevance is the basis for validating the implementation against the design.

The remainder of this paper provides support for this position by showing how the treatment of an elevator control problem is affected by by the emphasis on physical relevance intrinsic to the Virtual System approach. Section 2 introduces our reformulation of the elevator problem. Section 3 provides a review of the notation used to specify Virtual Systems. Section 4 presents our requirements specification for the elevator control problem and discusses three methods useful in ensuring the physical relevance of the model: environment/system boundary validation, failure analysis, and design-rules formulation and enforcement. The complete

specification is included in the Appendix. The paper also shows the applicability of our specification technique to modeling control type systems.

2. Problem Formulation

In the simplest terms, the elevator control problem requires one to specify the logic for moving n elevators in a building with m floors. Solutions to this problem provide the designer with three important challenges: (1) how to capture the functionality of the system; (2) how to design an algorithm for elevator movement in a manner that limits the waiting time by passengers; and (3) how to prove that all requests for service are satisfied in a fair manner. The last two problems remain unaffected by our unorthodox viewpoint and we plan to deal with them in a peripheral way. The first one, however, had to be reformulated as follows: *How can we capture the functionality involved in the elevator problem in a manner which is faithful to the structural and behavior properties of the physical components of some proposed system design? Can the Virtual System model help?*

Because we are talking about design specifications (not problem definition) we must have a design we want to express and a specification method that must be used to express it. Unfortunately, the design is greatly affected by the controls and capabilities already available in the elevators as supplied by their manufacturer. For simplicity sake, let us assume that (1) all buttons may be turned on and off, (2) elevators may open/close doors, change direction and move up/down one floor at a time, and (3) a controller monitors the status of the buttons and the elevators and commands certain changes in their state. With regard to the specification method, we capture the above design outline in a language called CSPS *(Communicating Sequential Processes with Synchronization)* which we used to specify a number of Virtual Systems some of which included dynamic reallocation of software modules to hardware processors [3]. In the next section we provide a brief overview of the CSPS notation.

3. CSPS Review

An extension of Hoare's CSP [4], CSPS allows synchronization between multiple processes in addition to the I/O commands of CSP. The I/O commands are used to express communication between software or hardware components which are modeled as CSP processes. Software processes are called modules and hardware processes are called processors. A module and a processor may not use I/O commands to interact with each other. The synchronization commands, on the other hand, are employed as a mechanism for identifying the occurrence of the *same* event at the hardware and software level. In this manner the interactions between software and hardware may be formalized and later evaluated by employing proof procedures developed for CSP programs, e.g., [5].

The simplest form of a synchronization command consists of a *synchronization label* and a *synchronization operator:* a **$**;. A synchronization command may occur many times in the text of a single process. Any process whose text includes a particular synchronization command must always participate in the corresponding synchronization, i.e., a synchronization takes place only when each participating process is ready to execute the same synchronization command. If two or more synchronization and I/O commands appear together separated by blanks (forming a *composite synchronization command*), all the commands must occur together—this feature enhances modularity of the models but adds no extra power to CSPS. CSPS extends the CSP distributed termination convention to cover event synchronization commands. The last element in a guard may be an I/O command, a synchronization command, or a composite synchronization command. A guard fails if the boolean part is *false* or a terminated process is involved in any of the synchronization or I/O commands appearing on the guard; a guard is passable if the boolean part is *true*, the I/O command may be executed, and all synchronizations may take place.

N-way synchronization, as introduced so far, is unable to simulate the data transfers accomplished by I/O commands. The *n-way synchronization with pattern match* is a mechanism for accomplishing what may be seen as data transfer.

Here is an example of how to simulate an I/O exchange using synchronization with pattern match:

```
P::     ... PtoQ $ (x); ...
Q::     ... PtoQ $ (y'); ...
```

PtoQ is the synchronization label which is used to determine (a priori) the set of processes that must synchronize. The pattern definition appears as a list following the synchronization operator. For a synchronization to be successful, each pattern must contain the same values. A single quote indicates that the value of the particular variable is part of the pattern for the respective synchronization but it is *indeterminate*, i.e., the variable will assume any value (within the restrictions of the variable type) that renders the pattern match successful.

4. Model Specification and Validation

A design specification is a model, at some level of abstraction, of relevant structural and behavior properties of some (software) system and its environment. A requirements specification is an extreme case of a design specification, one in which the designer focuses on the behavior of the interface between the system and its environment and ignores, to the greatest possible extent, the internal structure of the system. Clarity and precision are desirable properties of a requirements specification. They should not be achieved, however, at the expense of a loss in the physical relevance, i.e., validity, of the model. In this section we discuss three methods useful in ensuring the validity of software specifications: (1) boundary validation, (2) failure analysis, and (3) design-rules formulation and enforcement. For the sake of simplicity and without loss of generality, we explain our ideas in the context of requirements specification. The presentation starts with an overview of our treatment of the elevator problem.

In the elevator problem, the environment is represented by elevators and buttons, generically called devices, while the system consists of a software controller. The physical distinction between controller and devices is captured well by the module/processor dichotomy present is CSPS. The controller can be represented by one or more CSPS *modules* while the devices in the system can be modeled by CSPS *processors*. The interactions between the controller and devices may be expressed using CSPS *synchronizations* between modules and processors.

For this particular example of the elevator model, we assume that there are three types of devices: elevators, floor buttons (to request an elevator on a particular floor), and elevator buttons (to request a floor from inside an elevator). These devices are primitive in nature, they do not interact with each other, but only report events and accept commands from the controller. We further assume that the controller software is centralized. The software is represented by a single CSPS module, called **elevator_control**. The devices are each represented by a different CSPS processor. If the system has k elevators and n floors, then there are k elevator processors (**elevator.i**, $1 \le i \le k$), k elevator button processors (**elevator_buttons.i**, $1 \le i \le k$), and n floor button processors (**floor_buttons.i**, $1 \le i \le n$). The only interactions present in the model are communications between the controller and the devices. There are no software to software communications since the software is a single entity. There are no hardware to hardware communications since the devices were assumed not to interact directly. Next we examine more closely the structure of the module and processors that make up the model.

First we consider the processors used to model the devices. We specifically look at **elevator.1**, the processor model for the first elevator. The elevator has several different actions it can perform repeatedly and, for the most part, in any order. This type of behavior is modeled in CSPS by a guarded repetition statement, where each guarded command corresponds to one of the actions that may be undertaken, as shown by the specification of **elevator.1** given in Figure 1.

```
PROCESSOR elevator.1 ::
[
    dir: direction := up;
    lev : floor_number := 1;
    doors_closed : BOOLEAN := true;

    *[
        change_dir.1 $→
            [   dir = up→dir := down
            #   dir = down→dir := up]
    #
        open_doors.1 $→doors_closed := false
    #
        doors_closed; go.1 $→
            lev := max(1, min(Num_floors, lev + dir));
    #
        NOT doors_closed→doors_closed := true
    ]
]
```

Figure 1: Elevator modeled as a CSPS processor.

The four guarded commands correspond to the elevator actions of changing direction, moving one floor in the current direction, stopping and opening the doors at the current floor, and closing the doors, respectively. The first three are actions taken only upon command from the controller, and this interaction is represented by the synchronization on each of the guards. In the first two, the synchronization is alone on the guard, representing the fact that the elevator is always ready to perform either of these commands. The third one (**go.1**) also includes a boolean expression which expresses a local condition that must exist for the command to be accepted, in this case the elevator doors must be closed before the elevator will move to the next floor. The last guard contains no synchronization, which corresponds to an independent local action, in this case closing the doors at some point after they have been

opened.

The effect of each action is captured by the statement portion of the guarded command, as exemplified by setting the boolean *doors_closed* true in the local action corresponding to closing the elevator doors. Another example is the movement of the elevator expressed by incrementing or decrementing *lev*. This model of the elevator movement implicitly prevents the elevator from changing direction or opening the doors between floors. In a more detailed model, the movement could be modeled by several actions. Then, explicit booleans on the guards would be needed to prevent any inappropriate actions. It is important to note that we require the model to remain faithful to the actual physical characteristics of the device, not in absolute terms, but relative to a specific level of abstraction.

The processors corresponding to the buttons have a similar structure and interpretation as the elevator (see Appendix). In addition to executing commands issued by the controller these processors also report events taking place in the devices, e.g., the pressing of the buttons to generate new requests for service. The event reporting is represented by a synchronization on the guard, with the boolean expression (not present in examples) representing the conditions under which the event may take place. Event reports appear syntactically the same as commands in the controller, but represent communications initiated by the device, whereas commands are initiated by the controller.

The **elevator-control** has two major functions to perform. It accepts requests from the buttons and issues commands to dispatch elevators in response to service requests and to reset buttons when services are rendered. To accept a request from a button, the module must synchronize with that processor, and, similarly, to issue a command to an elevator or button, it must synchronize with that processor. Since any order of requests is possible and the arrival of new requests can be interspersed with servicing of previous ones, the controller must accept requests and issue commands in any sequence. This is similar to the way the devices behave. Hence, the structure of the module **elevator_control** is similar to that of the devices, namely a single guarded repetition, where each guarded command represents an action of the controller. Each action is either a response to an event in one of the devices or a command for an action by one of the devices.

Every event report is represented by a synchronization in the processor, so the module must have a matching synchronization to accept the report. This synchronization appears in the guard of the guarded command representing the response to such an event report. In general, these guards have true (empty) boolean portions because the software is not allowed to ignore the reporting of the events (even though it may take no action as a result of the particular event). As an illustration, consider pressing the button on floor 5 for an elevator to go up. When the controller accepts this event report, it updates its internal state so the request can be serviced later. This is represented by the following guarded command in the module:

elev_up_req.5 $ → up_req(5) := true

Commands are handled analogously, with the controller having a matching synchronization to that in the processor for each command. Again they appear on guards, but in this case the boolean expression is non-empty and represents the condition under which this command should be executed. The statement potion of the guarded command represents the actions taken by the software as a result of the command being performed. As an example, consider the controller's command to an elevator to change direction when it is on the first floor and heading in the downward direction. Upon command execution, the controller updates its state to reflect the change in the direction of the elevator movement. This is captured by the following guarded command (the elevator involved is number 3):

elev_dir(3) = down AND on_floor(3) = 1;
change_dir.3 $ → elev_dir(3) := up;

The other guarded commands for the controller module are similar. A complete specification of the module is contained in the Appendix.

Environment/system boundary validation. Most frequent specification errors are interface specification errors. Any misunderstanding regarding the boundary between the system and its environment or regarding the protocol followed by the two leads to an incorrect specification. Although these precepts are universally accepted, the selection of the system/environment boundary is not always straightforward and should be subject to explicit validation during the specification review.

In the elevator example, for instance, one is initially tempted to draw the boundary at the floor and elevator button level. This choice, although intellectually very exciting, is, nevertheless, wrong. It provides the opportunity for developing interesting algorithms aimed at optimal servicing of the passengers but fails to acknowledge that the elevators available on the market may have built-in controls that are incompatible with the choice of algorithm. The correct boundary is the point of interaction between the software controller and the

available hardware devices (elevators and buttons) and a proper model of their capabilities ought to be constructed (at an appropriate level of abstraction) before attempting to specify the desired software behavior.

In our model we made *explicit* assumptions about the elevators' capabilities. The potential for drastic changes in the model is a consequence of altering these assumptions. For instance, consider the hardware to be composed of two types of devices: elevators and floor monitors. The elevator includes what was previously the elevator buttons and automatically services requests from those buttons, as long as they are in the direction it is currently heading. To change direction or to continue moving in a direction for which it has no service requests requires direction from the controller. The floor monitors include the floor buttons, but they are also capable of communicating with the elevators as they approach so that, if the floor has a request and the elevator is going in the right direction, it will stop at the floor. The floor monitors will signal when a request is made and also when a request is serviced by an elevator. The elevators signal each time they move one floor.

In this system the requirement on the controller is to arrange for elevators to go to all floors that have generated a request. The controller must accept requests and service notifications from the floor monitors and movement data from the elevators. All three of these are simple event reports; they correspond to the first three guards in Figure 2, which contains the new controller. The controller also provides limited directions to the elevator. First, the elevator is allowed to change direction when requested and if there are no floor requests in the direction it is currently heading. Secondly, the controller could instruct the elevator to continue in the direction its heading (so that some floor requests will be serviced). Finally, the controller can request the elevator to change direction (to meet some floor requests), but it is assumed the elevator will not accept this unless it has no requests of its own in the current direction. The above commands are represented by the last three guards in Figure 2, in the order given.

The risks associated with neglecting the boundary validation check should become obvious when one contrasts this version of the controller with the one given in the Appendix. Although their general structure is similar, the actual actions and conditions are different. In particular, the actions of the alternate controller are much simpler since only updates to the status of the elevators and the buttons are required. If the elevators were made more complex so that they could detect the floor requests and respond in a manner like the controller is instructing them to, then the controller would be reduced to mere report gathering, and could be eliminated entirely. Hardware/software tradeoffs normally captured by the virtual system are manifest in the elevator problem as tradeoffs between environment and system capabilities.

```
MODULE elevator_control
[
    dir: direction(Num_elev) := up;
    lev : floor_number(Num_elev) := 1;
    e : elevator;
    f: floor;

    *[
        elev_move.e" $ [1]→lev(e) := lev(e) + dir(e)
    #
        elev_req.f" $→req(f) := true
    #
        floor_serv.f" $→req(f) := false
    #
        (dir(e") = up AND NOT request_above(lev(e)))
        OR (dir(e") = down AND
            NOT request_below(lev(e)))
        elev_chng.e" $→dir(e) := - dir(e)
    #
        (dir(e") = up AND request_above(lev(e))) OR
        (dir(e") = down AND request_below(lev(e)))
        continue.e" $→skip
    #
        (dir(e") = up AND request_below(lev(e))) OR
        (dir(e") = down AND request_above(lev(e)))
        chng_dir.e" $→dir(e) := - dir(e)
    ]
]
```

Figure 2: Alternate controller specification

Failure analysis. It is common practice to ignore the potential for failure when constructing a requirements specification. This is due to both the desire to keep the specifications simple and the technical difficulties raised by the failures' design dependent manifestations. Nonetheless, failures may be used to analyze the validity of the specifications. If the model is true to physical

[1] In CSPS the double quote indicates that the guard and the corresponding action are replicated once for each possible value of the variable marked by a double quote. In this instance, a version of this line is conceptually present for each possible floor number.

reality and if a particular failure may be characterized faithfully at the level of abstraction used by the model, then the failure's potential impact on the system should be the observable in the model.

We found this kind of analysis to be a very effective tool both in uncovering modeling errors and in sharpening our understanding of the level of abstraction at which the model is constructed. Although proper characterization of the failures is non-trivial, the payoffs make the effort worthwhile.

Our strategy focused on identifying potential device failures and on establishing the consequences of these failures on other devices and the controlling software. In other applications the effects of software failures may prove to be equally important to consider. To account for software failures, however, one must refine the system specification part of the model to include additional detail, i.e., a partial design of the system must be carried out. The danger of overspecifying the requirements becomes now a serious concern.

In our model it is reasonable to view a failure in a device as the abortion of the corresponding processor, so that in effect there is no partial failure of a system component. Even at this level of detail, several types of failures are possible. Devices could fail while idle (awaiting next event or command), while performing an action, or while performing a communication. The first two types do not involve any other component, so there is no direct effect on others, but the failure during a communication could leave the other components involved in an unknown state. To effectively study this, a more detailed specification of the communication (including such details as protocol, etc.) would be necessary. Hence, we must assume that failures do not occur during a communication. Due to the structure of the device models, a failure could be modeled by the addition of another guarded command with a true guard and statement portion containing a single abort statement. This is consistent with the reliable communication assumption and covers the other types of failures.

Once a processor has aborted, any synchronizations it previously could participate in are prevented from happening henceforth. Any other process that tries one of these synchronizations will either block, if it is inline (a stand-alone statement), or fail to pass the guard containing the synchronization. As an example of the latter case, consider the abortion of **elevator.1**. The controller can no longer issue commands to it. The guards containing the appropriate synchronizations fail. This prevents the statement portion from being executed also, so the **floor_arr.1** synchronization is never performed again, resulting in the **elevator_buttons.1** processor leaving the button lighted. However, further requests from the elevator buttons will be acknowledged by the controller since the reporting of these events is independent of the **elevator.1** operation. Visualized in the real device, the elevator is stuck somewhere, incapable of further movement. However, the buttons in the elevator can be pressed and, as a result, remain lighted (acknowledging the request). Hence, the model matches well the physical situation. Inline synchronizations do not occur in the example, but consider the following alternate and straightforward specification in the controller for the open_doors command to **elevator.1** when it is going up and at level 4 (boolean portion of guard omitted):

```
open_doors.1 $ →
    doors_closed := false;
    floor_arr.1 $ (4);
    elev_up_arr.4 $
```

Once the open_doors command has been given, the elevator buttons and floor buttons must be notified so they can turn off their lights. However, in this simpler version, if either of the button processors has failed, then the module will block and the system will come to a halt. This is most likely a modeling error, since this is not a realistic manner for such a controller to behave, though it does serve to point out the danger of casual placement of synchronizations inline. As seen in the complete specification in the Appendix, the use of a repetition statement with the synchronizations on guards in such a manner that they are executed once if the processor is still alive and are skipped if the processor has terminated avoids this pitfall.

Design-rules formulation and enforcement. It is not sufficient to understand the nature of the system/environment interactions, one must also capture it correctly in the specification language being used. Functionally equivalent formulations, when subjected to an analysis of their physical implications, often reveal distinct properties. One way to ensure the model's desired physical relevance is by formulating design rules which, when followed, guarantee the faithful modeling of physical reality. There are many ways of capturing the design rules. They range in sophistication from the use of macros, templates, and user-defined language extensions, at one end of the spectrum, to the formulation of provable assertions about the model, at the other.

The idea of formulating the design rules before building the specification and checking the specification for adherence to the design rules

seems to be the only way to reconcile the apparent conflict between specification language generality and problem specificity. It also has the advantage that it forces the designer to gain a deeper understanding of the physical realities with which he/she must cope.

An analysis of the types of interactions taking place between the software and the controlled devices led to the following set of design rules:

(1) each processor consists of a single guarded repetition, with synchronizations occurring only on the guards of this repetition;
(2) each module consists of an outer guarded repetition in which guards corresponding to event reporting from devices must always be serviceable, i.e., no boolean condition may appear on these guards; and
(3) no inline synchronizations may appear in any of the modules in order to eliminate the possibility that a terminated processor might block the module.

Adherence to these rules guarantees that all interactions are consistent with the physical system but does not imply that the software requirements are properly specified. In certain cases, however, knowledge about the design rules may lead to simplified proofs of correctness for the software specifications.

5. Conclusions

We started with an approach intended for the formalization of software/hardware interactions in distributed systems and applied it to the elevator control problem. The emphasis on physical relevance has resulted in a new treatment of the elevator problem, one which is design oriented, which reflects faithfully the structural and behavioral properties of the system components, and which allows the designer to work on the algorithm for elevator movement and its proof in the realistic context of the total system. This means that the algorithm development will not involve a strategy which, optimal in theory, may never be implemented because of some peculiar built-in elevator controls which have not been considered by the abstract study of the system!

Another interesting result is the fact that we have been forced to reexamine the interpretation of the synchronization commands. While in the Virtual System context a synchronization command represented the occurrence of the same event at two distinct levels of abstraction (software and hardware), in the elevator problem we used them to model reliable signal exchanges between the software and the controlled devices. It should be noted, however, that the ways in which we used the synchronization commands have precise physical interpretation! (Other uses permitted by the language do not have such interpretations.) It is our hope that this simple exercise will prompt others to give serious consideration to the issue of how to limit a specification language so as to ensure physical relevance of the proposed designs.

6. References

[1] Filman, R. E. and Friedman, D. P., *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[2] Roman, G.-C. and Day, M. S., "Multifaceted Distributed Systems Specification Using Processes and Event Synchronization," *Proceedings of the 7th International Conference on Software Engineering*, pp. 44-55, March 1984.

[3] Roman, G.-C., Ehlers, M. E., Cunningham, H. C., and Lykins, R. H., "Toward Comprehensive Specification of Distributed Systems," Technical Report WUCS-86-8, Department of Computer Science, Washington University, Saint Louis, Missouri 63130.

[4] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21, No. 8, pp. 666-677.

[5] Soundararajan, N., "Axiomatic Semantics of Communicating Sequential Processes," *ACM Trans. on Prog. Lang. and Sys.* 6, No. 4, pp. 647-662, 1984.

Acknowledgement: The authors are indebted to H. C. Cunningham, R. H. Lykins, and W. Chen for their review of the elevator model. Their participation in the development of the CSPS language and models is also acknowledged.

Appendix

Simplified treatment of an elevator problem.

```
Num_floors : INTEGER CONSTANT;
Num_elevators: INTEGER CONSTANT;
on: BOOLEAN CONSTANT := true;
off: BOOLEAN CONSTANT := false;
up: INTEGER CONSTANT := 1;
down: INTEGER CONSTANT := -1;
TYPE button IS BOOLEAN;
TYPE floor_num IS (1..Num_floors);
TYPE elev_num IS (1..Num_elevators);
TYPE direction IS (up,down);

PROCESSOR elevator.i (i : elev_num) ::
|
    move: direction := up;
    level : floor_num := 1;
    doors_closed : BOOLEAN := true;
    *|
        /* Commands from controller */
        change_dir.i $      →
            |   moving = up    →  moving := down
            #   moving = down →  moving := up|
    #
        doors_closed; go.i $ →
            level := max(1, min(Num_floors, level + move))
    #
        open_doors.i $      →  doors_closed := false
    #
        /* Local actions */
        NOT doors_closed    →  doors_closed := true
    |
|

PROCESSOR Elevator_buttons.i (i : elev_num) ::
|
    B : ARRAY(floor_num) OF button := off;
    f : floor_num;
    *|
        /* Event reports to controller */
        floor_req.i $ (f)     →  B(f) := on;
    #
        /* Commands from controller */
        floor_arr.i $ (f')    →  B(f) := off
    |
|

PROCESSOR Floor_buttons.i (i : floor_num) ::
|
    up,down : button := off;
    *|
        /* Event reports to controller */
        elev_up_req.i $     →  up := on;
    #
        elev_down_req.i $ →  down := on;
    #
        /* Commands from controller */
        elev_up_arr.i $     →  up := off
    #
        elev_down_arr.i $ →  down := off
    |
|

MODULE elevator_control ::
|
    up_req : ARRAY (floor_num) OF BOOLEAN := false;
    down_req : ARRAY (floor_num) OF BOOLEAN := false;
    elev_dir : ARRAY (elev_num) OF direction := up;
    on_floor : ARRAY (elev_num) OF floor_num := 1;
    B : ARRAY (elev_num,floor_num) OF BOOLEAN := false;
    f : floor_num;
    e : elev_num;
    b1, b2 : BOOLEAN;
    *|
        /* Event reports from devices */

        elev_up_req.f" $       →  up_req(f) := true
    #
        elev_down_req.f" $     →  down_req(f) := true
    #
        floor_req.e" $ (f')    →  B(e,f) := true
    #
        /* Commands to devices */

        elev_dir(e") = down AND on_floor(e) > 1 AND
          NOT (B(e,on_floor(e)) OR down_req(on_floor(e)));
          go.e $
            →  on_floor(e) := on_floor(e) + down
    #
        elev_dir(e") = up AND on_floor(e) < Num_floors AND
          NOT (B(e,on_floor(e)) OR up_req(on_floor(e)));
          go.e $
            →  on_floor(e) := on_floor(e) + up
    #
        elev_dir(e") = down AND f" = on_floor(e)
          AND (B(e,f) OR down_req(f));
          open_doors.e $
            →  down_req (f) := false;
               B(e,f) := false;
               b1 := true; b2 :=true;
               *|  b1; elev_down_arr.f $ → b1 := false
                #  b2; floor_arr.e $ (f)    → b2 := false
                |
    #
        elev_dir(e") = up AND f" = on_floor(e)
          AND (B(e,f) OR up_req(f));
          open_doors.e $
            →  up_req (f) := false;
               B(e,f) := false
               b1 := true; b2 := true;
               *|  b1; elev_up_arr.f $ → b1 := false
                #  b2; floor_arr.e $ (f) → b2 := false
                |
    #
        elev_dir(e") = down AND on_floor(e) = 1;
          change_dir.e $
            →  elev_dir(e) := up
    #
        elev_dir(e") = up AND on_floor(e) = Num_floors;
          change_dir.e $
            →  elev_dir(e) := down
    |
|
```