8 An Application: The Case of the Elevator System

"In order to get some kind of limit to this enormous subject [elevators] it seems sensible to restrict this study to those devices which have land as their starting point, leaving aside the larger question of aviation and rocketry."

[Jean Gavois, in the Preface to his Going Up: An Informal History of the Elevator from the Pyramids to the Present, Otis Elevator Co., New York, 1983]

8.1 Introduction

Our modelling approach is illustrated in this chapter by a fairly complex application, an Elevator System (ELS). A brief review of this frequently used case study starts the presentation, then the application is defined in terms of general and temporal requirements. In particular, the timing constraints imposed on ELS are shown to provide a comprehensive coverage of the Dasarathy constraints discussed in Chapter 5. Following the specification steps presented in Chapter 7 the elevator system is subjected first to UML modelling, then the formalisation algorithms described in Chapter 6 are applied. The need of enhancing the formal specification and of precisely expressing the temporal requirements placed on systems is emphasised and, based on this application, observations regarding the modelling process proposed in Chapter 7 are included.

8.2 On the Elevator Case Study

The elevator example constitutes one of the preferred case studies of software engineering authors, its extraction from a daily life reality (everybody knows what an elevator is) doubled by its intrinsic complexity --which allows the illustration of various modelling concepts and techniques, including the treatment of temporal constraints-- accounting for its popularity and frequent employment. The origins of this case study can be traced back to Donald Knuth's first volume on the Art of Computer Programming, where a simulation program of Caltech's Mathematics building's elevator was included to exemplify coroutine-based implementation techniques [Knuth73, pp. 280-295]. Since then, many other authors have resorted to the elevator problem as a means of illustrating new software development approaches; to point out only a few reports focused on an elevator system, we refer to Glenn Coleman et al's paper on simulating concurrent systems using Statemate specifications and automatic prototyping [Coleman90], Zhang and Mackworth's formal description of embedded real-time systems using Constraint Nets [Zhang93], Dong et al's approach on specifying parallel and distributed systems in Object-Z [Dong97b], Duval and Cattel's PROMELA and Synchronous C++-based method for developing safe process control applications [Duval97], and Schach's textbook on software engineering [Schach99]. The latter author points out that the elevator case study is non-trivial ("the problem is by no means as simple as it looks" [Schach99, pp. 347]), and can be of great value when illustration of software development techniques is intended. In fact, Schach makes the elevator system one of the two main case studies recurrent in his book. Recently, a detailed, comprehensive Object-Z description of an elevator has been proposed [Mahony00] and although it is one of the few that employs an object-oriented variant of Z, it differs from ours in several major aspects: firstly, it is purely formal, and we combine semi-formal graphical descriptions with formal specifications; secondly, they use CSP and we employ RTL as primary instrument for capturing temporal properties of systems; and thirdly, the OO extension of Z they employ is different from ours. In addition, our goal in this chapter has been to illustrate the steps and the artefacts of the modelling process proposed in Chapter 7, without giving comprehensive

details on a specific application. Thus we have been less ambitious with our Elevator System: it is not a multiple elevator-system (although the modelling resources employed in our approach, in particular UML and RTL, allow the expressing of concurrent behaviours), and it is not specified in all details.

The starting point for our example was provided by the multiple-elevator system presented by Robert Holibaugh in his special report on Joint Integrated Avionics Working Group's Object-Oriented Domain Analysis Method (briefly denoted JODA) [Holibaugh93]. However, in order to make it illustrative for our purposes, we modified the problem statement in numerous places, in some cases by adding new requirements or by providing supplemental details to the existing ones, while in others by eliminating stipulations that would have had only limited significance for exercising our approach (for instance, we renounced providing the elevators with back doors). For the same illustration purposes, we have added a set of temporal constraints (time conditioning was non-existent in Holibaugh's case study) and described the solution of the problem in a fair level of detail. In this way, our example has departed significantly from its starting point, and acquired a “personality” of its own. Although fictive, without a precise correspondent in real life, the elevator described below is sufficiently general to be easily imagined working around the clock in the concrete, shadowing high-rise office building across the street.

8.3 The Problem

In our initial source of inspiration, the [Holibaugh93] report, the elevator system was part of an Office Building Transportation System that also encompassed an escalator system and a set of staircases. Since the focus of this application is on the elevator system, detailed requirements on the building's escalators and stairs are not considered below. The general requirements for the elevator are denoted Rx (where x is a number provided for easier referencing) while the requirements that explicitly impose timing constraints on the system are denoted Tx. The correspondence between the temporal constraints placed on the Elevator

System and the basic Dasarathy timing constraints to which they can be related is given in Subsection 8.3.3 and both types of requirements are consequently treated by the problem's solution, presented in Sections 8.4.

8.3.1 General Requirements for the Elevator System

The general, non time-related requirements for the Elevator System are the following:

[R1] The elevator serves two or more floors;

[R2] The elevator contains on board a set of destination buttons (car buttons), one for each floor served by the elevator. When pressed, a destination button becomes illuminated and remains so until the elevators arrives at the corresponding floor;

[R3] The elevator has on board a set of lights (floor indicators), in one-to-one correspondence with the floors. At any given moment, exactly one of these indicators is lit, showing the floor the elevator is currently at;

[R4] The elevator has on board two door buttons (a close door button and, respectively, an open door button) which, when the elevator is stopped at a floor, can be pressed by the passengers to close the door earlier than otherwise done automatically and, respectively, to keep the door open longer than otherwise allowed by the elevator's preset timeout;

[R5] The elevator contains an Alarm button that, when pressed, will generate an intense audio signal (further details are given in constraint [T5]);

[R6] Each floor except the top and bottom floors has two request buttons, one for requesting the elevator to go up, and the other to go down. When pressed, such a button becomes illuminated and remains so until the elevator arrives at the floor and then moves in the requested direction. The terminal floors (the bottom floor and the top floor) have only one request button;

[R7] The elevator has a single elevator door, which is either closed or open. The door cannot be opened while the elevator is moving and, reciprocally, when open it will prevent the elevator from moving;

[R8] On each floor, there is a floor door that will work in tandem with the elevator's door and, for safety reasons, a floor door can be open only if the elevator is stopped at that particular floor;

[R9] When an elevator has no requests, it will remain idle at the last visited floor (the last target floor at which the elevator has stopped), with its door closed;

[R10] On board of the elevator there is a special Stop button, which when pressed will stop the elevator's movement. It will not be possible to open the doors when the elevator is stopped in between the floors.

8.3.2 Temporal Constraints for the Elevator System

The elevator is also required to satisfy the following timing constraints:

[T1] <Open Floor Door> After the elevator has stopped at a particular floor, the elevator's door will open no sooner than OPEN_MIN_TIME seconds and no later than OPEN_MAX_TIME. Practical values for these constants can be, for instance, 1.0 seconds and, respectively, 3.0 seconds;

[T2] <Stay Open Floor Door> After the elevator has stopped at a given floor the elevator's door will normally stay open for a STAY_OPEN_NORMAL_TIME period of time (e.g., 12.0 seconds). However, if the Close Door button on board of the elevator is pressed before this timeout expires, the door will close but no sooner than STAY_OPEN_MIN_TIME (e.g., 2.0 seconds);

[T3] <Resume Elevator Movement> After the door is closed, the movement of the elevator can resume, but no sooner than CLOSE_MIN_TIME seconds and no later than CLOSE_MAX_TIME seconds (possible values can be, for instance, 1.0 seconds and, respectively, 3.0 seconds);

[T04] <Elevator Speed Constraints> The movement of the elevator between destination floors should be continuously monitored, and a minimum and maximum speed limits should be considered. Two preset values, SPEED_LIMIT_LOW and SPEED_LIMIT_HIGH will serve the detection of abnormal moving conditions (too slow or too fast). In such cases, the elevator will be stopped immediately and an alarm signal will be issued. Practical values for the above constraints can be expressed in seconds per floor, for instance the lower limit can be 5.0 seconds/floor and the higher limit can be 3.0 seconds/floor (during continuous movement). It can be considered that floor sensors are available to detect the presence of the elevator by any given floor;

[T5] <Stop Request> If the Stop button on board the elevator is pressed, the moving elevator will stop as soon as possible, in any case no later than STOP_MAX_TIME seconds (e.g., 2.0 seconds). The floor doors will not open if the elevator is not positioned at a floor, and the elevator will remain in this state of emergency stop for STAY_STOPPED_TIME seconds (e.g., 20.0 seconds) unless the Stop button will restart the above timeout from zero. Before the devator resumes its movement, the Stop button will be illuminated for a sequence of several consecutive visual signals (timing requirements for both audio and visual signals are specified by [T7]);

[T6] <Alarm Triggered> If the Alarm button inside the elevator is pressed, then the elevator will stop immediately according to the timing condition STOP_MAX_TIME (from T5) and a continuous, highly audible alarm signal will be issued (T7 gives details on timing characteristics of these signals). In contrast to T5, the elevator will not resume its movement after STAY_STOPPED_TIME seconds, and will stay stopped until authorization for moving is given by a designated staff member and the alarm system is set-off;

[T7] <Signal Timing> The audio alarm will consists of a sequence signals, each of a duration no less than MIN_SIGNAL_DUR (e.g., 1.0 seconds) and no greater than MAX_SIGNAL_DUR (e.g., 2.0 seconds). The separation between signals, SIGNAL_SEPARATION, should be preset to a given value, e.g. 1.0 seconds. The same constants can be used in the case of the visual signals mentioned by [T5];

8.3.3 Coverage of Dasarathy Constraints by the Elevator's Timing Requirements

The timing constraints T1-T8 placed on the Elevator System's behaviour are intended to illustrate the way our proposed approach deals with a variety of temporal requirements. Since, as indicated in Section 5.2, Dasarathy's classification of timing restrictions offers a reference basis for such requirements, it is useful to notice that eight out of nine classes presented in Subsection 5.2.1 are covered in our case study. The correspondence between the elevator's timing constraints [TC1]-[TC7] and the corresponding Dasarathy classes of temporal constraints [DC1]-[DC9] to which they can be related is given in Table 8.I, and shows that all DC classes except [DC4], which is a constraint placed on external stimuli, are covered by at least one of the elevator timing requirements TC. This table, together with the solution of the problem presented in the remaining of this chapter, demonstrates that our proposed approach can deal with a large variety of timing restrictions placed on TCS.

Table 8.I Correspondence between ELS's Timing Requirements and Dasarathy's Constraints

Elevator Timing Constraint	Corresponding Dasarathy Constraints
T1 <Open Floor Door>	DC7 (MaxRR), and DC8 (MinRR)
T2 <Stay Open Floor Door>	DC6 (MinSR), DC7 (MaxRR), and DC8 (MinRR)
T3 <Resume Elevator Movement>	DC7 (MaxRR) and DC8 (MinRR)
T4 <Elevator Speed Constraints>	DC1 (MaxSS) and DC2 (MinSS)
T5 <Stop Request>	DC3 (MaxRS) and DC5 (MaxSR)
T6 <Alarm Triggered>	DC5 (MaxSR)
T7 <Beep Timing>	DC9 (Duration)

8.4 The Modelling Solution

The steps of the “regular” flow of modelling activities described in Chapter 7 are illustrated below using the Elevator case study. The role of each modelling step is highlighted and examples of artefacts obtained in each step are given.

8.4.1 Definition of Use Cases

A single use case diagram is sufficient to describe the externally visible behaviour of the elevator system, as shown in Fig. 8.1. Within this diagram, two use cases are considered, Inside Request and Outside Request, and there are only two actors that interact with the systems, the User, a person that issues a command to the elevator, and the Elevator itself.

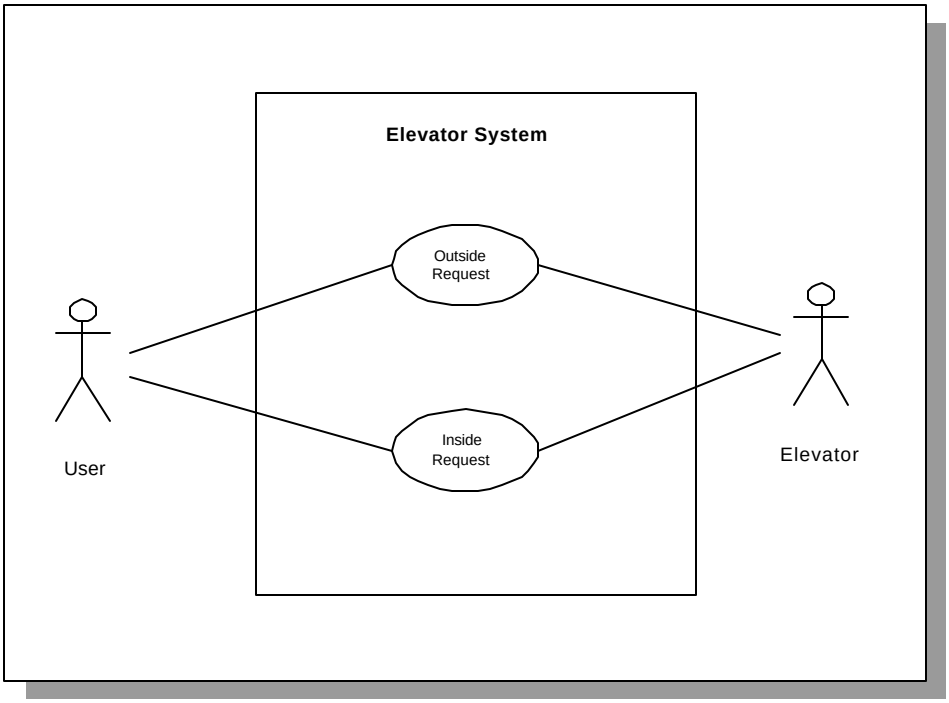

Fig. 8.1 ELS Use Case Diagram

8.4.2 Elaboration of Scenarios

The two use cases represented in Fig. 8.1 are next detailed through the use of scenarios, four such scenarios being presented in Fig. 8.2 to 8.5. As mentioned in Chapter 7, the scenarios can be described in various ways, one of which being to use application-tailored visual descriptions. In our case, a number of graphical symbols are used, as indicated in the legend attached to the figures, allowing a more elaborate description of the system's externally visible behaviour.

While developing the scenarios it has been observed that there is not a clear cut line between the two use cases considered initially, in real-life situations combinations of internal and external requests being issued for the elevator's service. For this reason, the scenarios that follow are "attached" to the two use cases considered in Fig. 8.1 based on the type of the first issued request shown in the scenario. For illustration purposes only a segment of the building in which the elevator operates is considered (levels 2 to 6), sufficient however to describe the most important aspects of the elevator's operation.

While developing the scenarios, a number of rules regarding the functioning of the elevator have been established. To describe them, the notions of direction-changing and direction-keeping requests need be introduced. While a direction-keeping request is simply not a direction-changing request, the latter can be either an internal request (for instance, someone presses the car button number 4 while the elevator is at floor 6 and moving up) or an external request (for instance when the elevator is at floor 3 and moving down a request is issued at floor 5, no matter for what direction). Using these two terms, the "rules of the elevator" can be formulated as:

Rule #1: "Maintain direction as long as possible". This rule means basically that if more service requests exist, the elevator will serve first the direction-keeping requests. If, at a given time, there are only direction-changing requests, than the stipulations of Rule#3 below have to be followed;

Rule#2: “When maintaining the direction, go to the closest floor from which a direction-keeping request has been issued”;

Rule #3: “If direction has to be changed, change direction and then (a) try to apply rules Rules #1 and #2 or (b) if this is not applicable, go to the farthest floor from which a direction-changing request has been issued.” This rule prevents the situation of an “infinite loop” in the elevator’s traveling, as described in the scenario shown in Fig. 8.5.

The first scenario, presented in Fig.8.2, is a normal instance of the Outside Request use case. Specifically, while the elevator is waiting at floor 6, a request from floor 3 is issued for movement up to floor 5. No other requests are issued while the elevator services this request. The basic behaviour of the elevator is captured in this scenario.

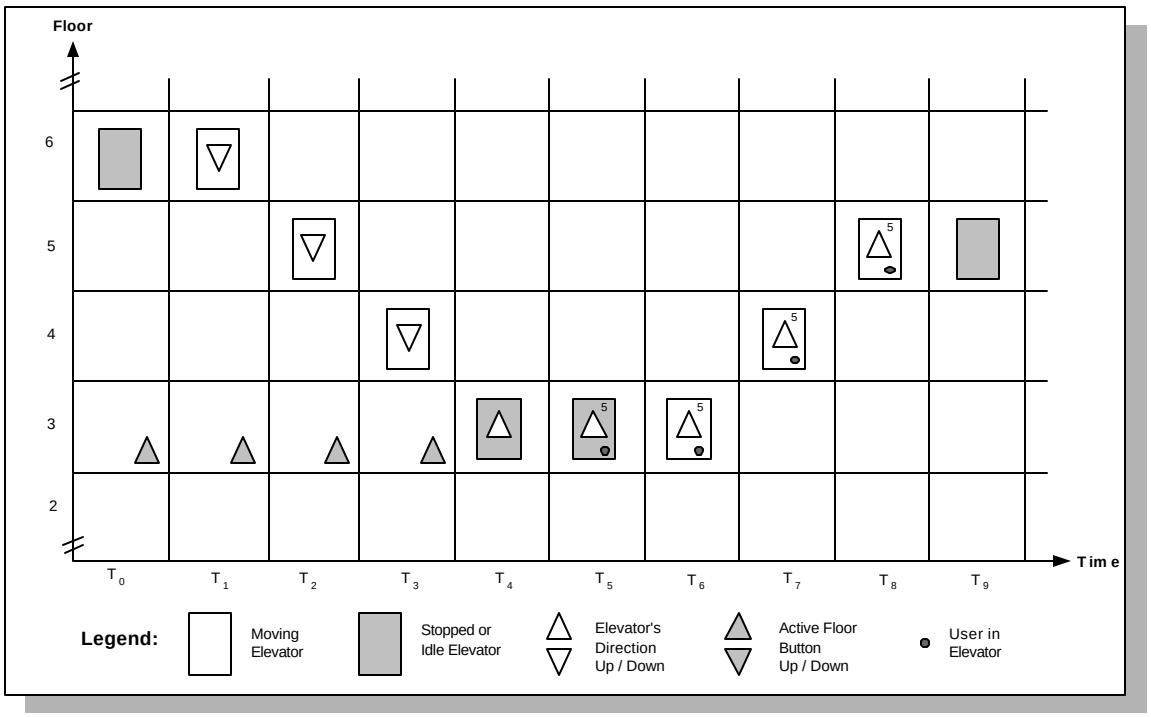

Fig. 8.2 ELS Scenario: Outside Request A

The second scenario, shown in Fig. 8.3, is another instance of the Outside Request use case. Its can be described summarily as “Sorry, but I changed my mind!,” because in it the issuer of the request from floor 3 is no longer taking the elevator after it arrives at the floor. The elevator, not having any other internal or external requests to serve, becomes idle at floor 3.

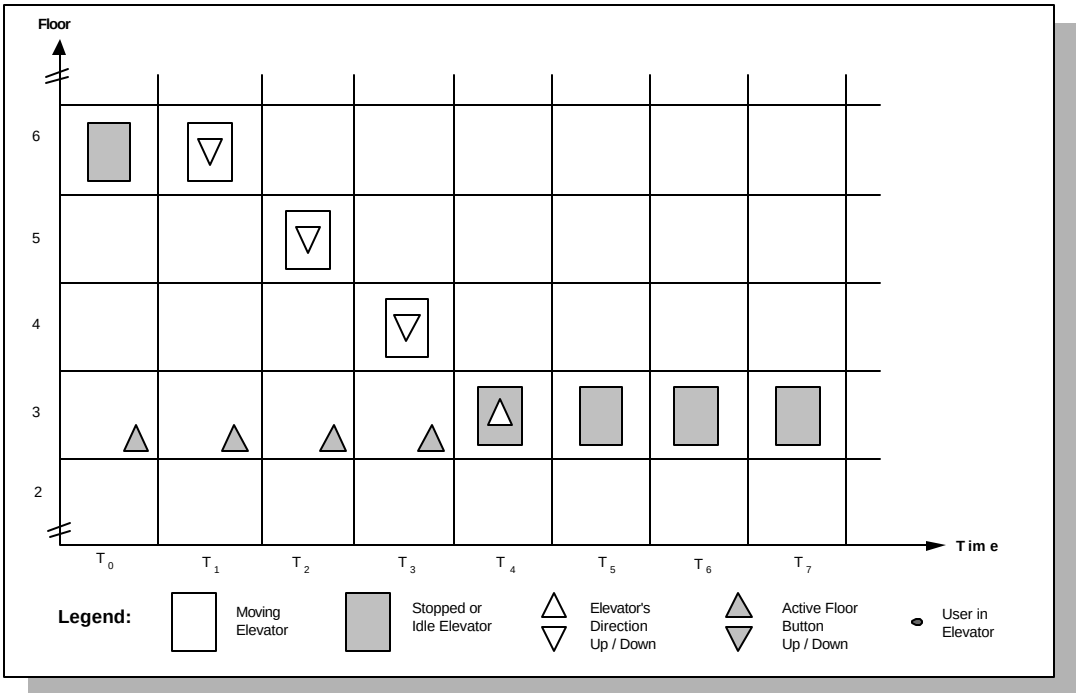

Fig. 8.3 ELS Scenario: Outside Request B

The third scenario, presented in Fig. 8.4, also an instance of the Outside Request use case, describes a situation in which two requests are issued at the same floor for different directions. By applying Rules #1 and #2 of the elevator, the Up request at floor 4 is served before the Down request at the same floor, although the latter was issued first.

The fourth scenario, depicted in Fig. 8.5, is an instance of the Inside Request use case that serves for illustrating the solution for a situation in which a user attempts to use the elevator in a rather mischievous way. The scenario, which can be denoted “You cannot be the exclusive user of the elevator!” shows that a user that repeatedly tries to travel between floors

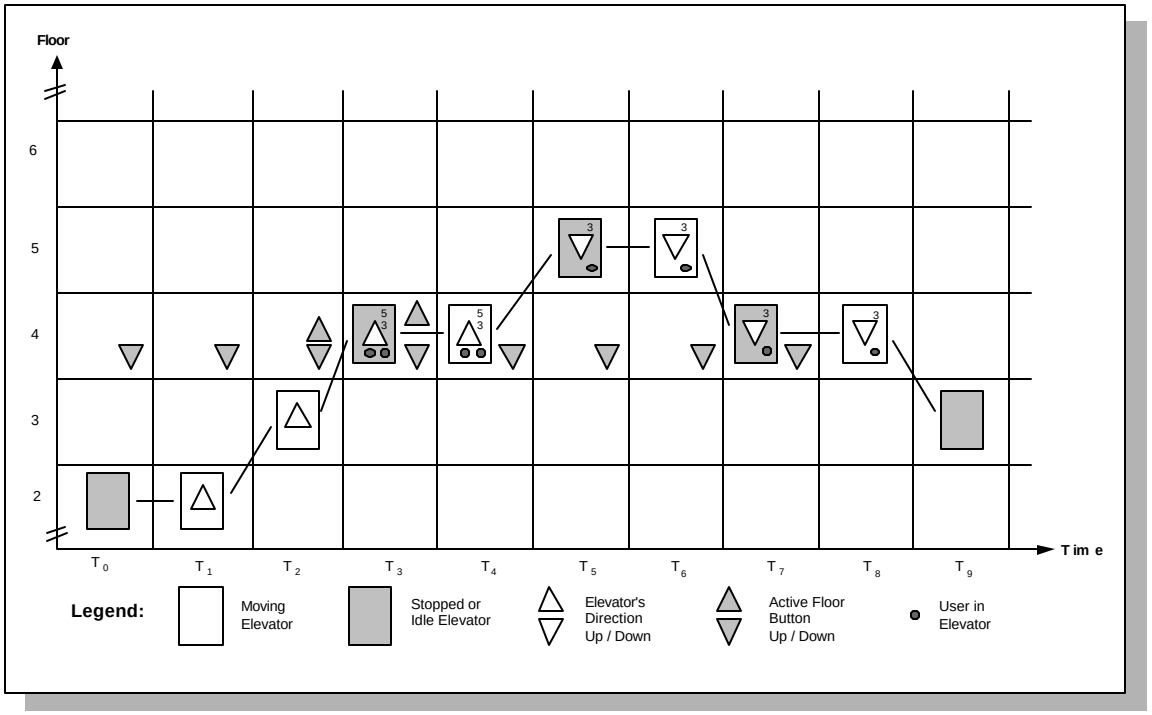

Fig. 8.4 ELS Scenario: Outside Request C

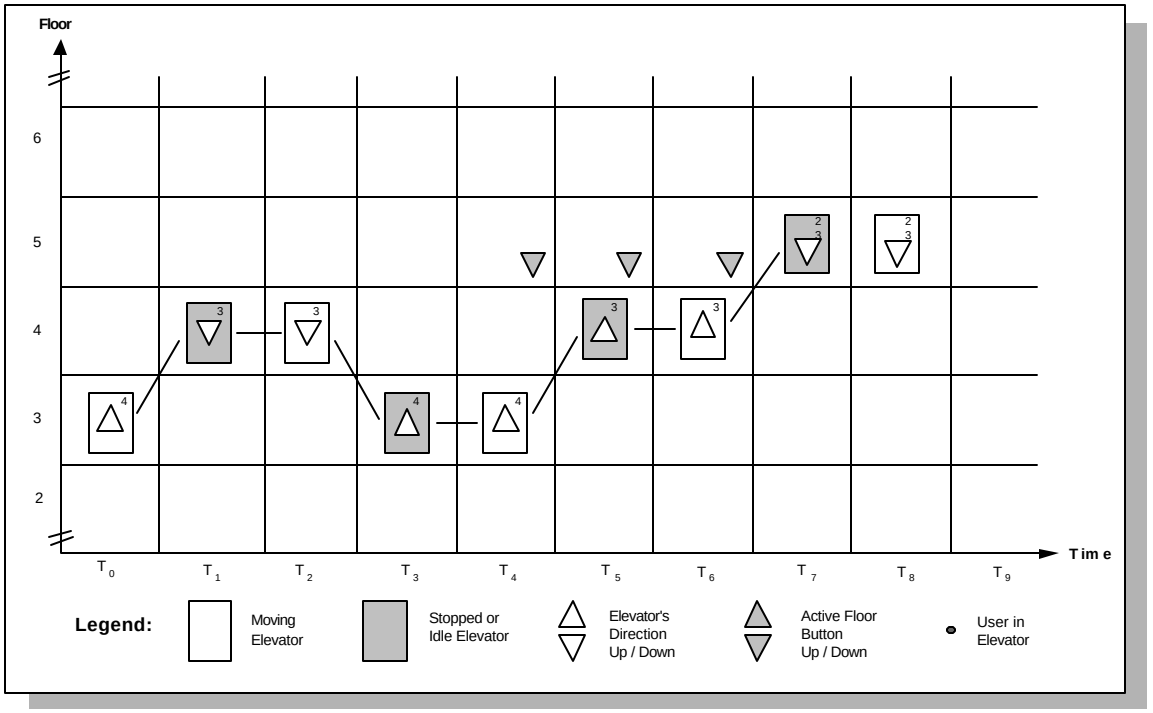

Fig. 8.5 ELS Scenario: Inside Request A

3 and 4 (by pressing car button number 3 when moving from floor 3 to 4 and car button 4 when moving from floor 4 to3) is interrupted in this action when another request is issued at floor 5. This solution is made possible by Rule #3b.

8.4.3 Construction of the Class Diagram

The scenarios shown previously serve not only for establishing a set of rules for the intended behaviour of the elevator, but they also help the construction of the system's a class structure. The class diagram resulted from the information gathered while developing scenarios, as well from the inspection of the problem's requirements is shown in Fig. 8.6. This diagram is given only in terms of component classes and relationships between classes, and provides no details about the contents of the classes (attributes and operations are not specified yet).

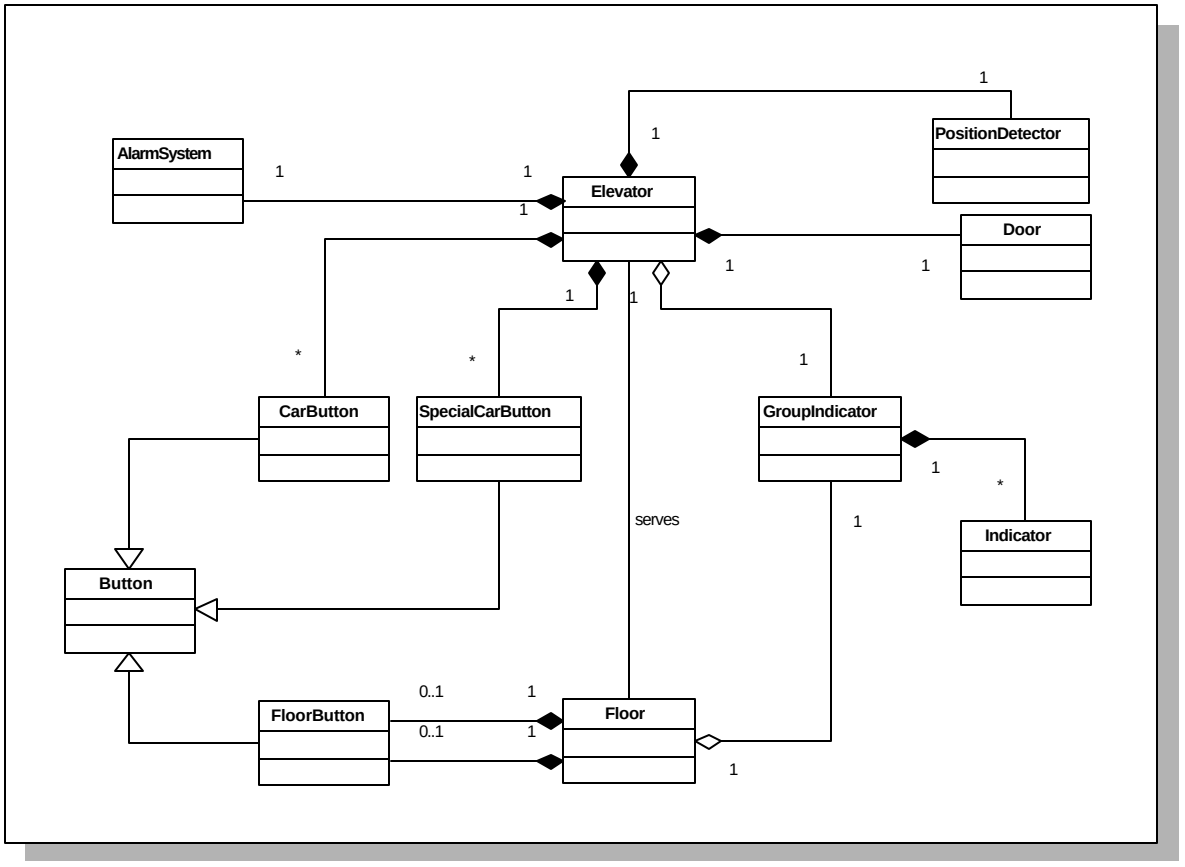

Fig. 8.6 ELS Class Diagram: Initial Structure

In this class diagram the obvious components of the elevator system are represented: the elevator itself, the floors, the buttons, and the indicators. In order to distribute the functionality of the elevator, the classes AlarmSystem, responsible with issuing audio and visual signals in exceptional situations, and PositionDetector, intended to take care of monitoring the position and the speed of the elevator, are also included. Since conceptually both the floor indicators and the indicators present in the elevator have always the same state (they show the same thing, the floor at which the elevator is currently at), a single class GroupIndicator has been introduced. In addition, buttons are modelled by several classes, based on their specific use (regular car buttons, used for accepting internal requests from the user, special car buttons such as Alarm, Stop, Open Door and Close Door, and floor buttons, which indicate the direction of external requests). Only a class Floor has been included in the diagram to keep the graphical representation simple, although two classes TopFloor and BottomFloor from which a MiddleFloor inherits would more accurately describe the floors in an object-oriented way (see [Dong97] solution in this respect).

8.4.4 Specification of Sequence Diagrams

Additional insight into the elevator system is obtained by developing sequence diagrams. In particular, while trying to assign responsibilities to the objects of the classes the internal behaviour of the system becomes more clear. Fig. 8.7 shows a sequence diagram that corresponds to the scenario depicted in Fig. 8.2 (Outside Request A Scenario). In fact, the sequence diagram describes only half of this scenario, yet the analysis of the elevator's behaviour indicates that the key design principles of the Elevator System are captured in this diagram. Specifically, the diagram describes both the situation in which the elevator is checking its requests at a floor ("idle" or "stopped"), and the situation in which the elevator is moving towards a destination floor ("target floor").

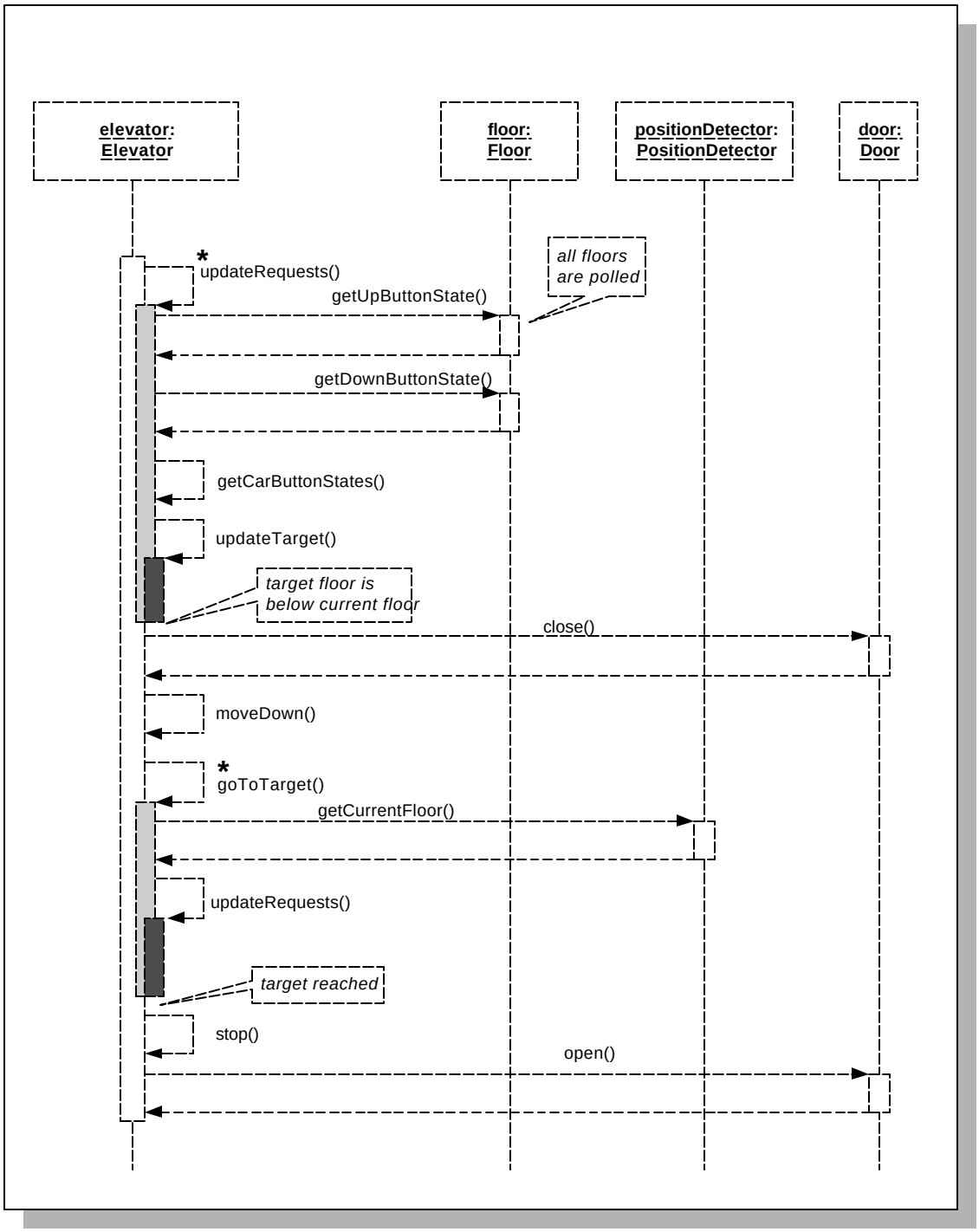

Fig. 8.7 ELS Sequence Diagram: Outside Request A

The design solution considered, apparent in this sequence diagram, is to continuously poll both the floor buttons and the internal car buttons in order to permanently maintain a list of existing requests (the repeated operation updateRequests in the diagram). When the need for a movement is detected by the idle (or stopped) elevator, the target floor is set (updateTarget operation) and the elevator starts moving but not before closing its door (to be precise, if the elevator is idle, the door is already closed). While moving towards the target the list of requests need be continuously updated (within the goToTarget operation, which is labelled "repeated" to indicate the repetitive nature of its major two components: getCurrentFloor and UpdateRequests, the latter perfoming the same task as in the idle or stopped situations). The sequence diagram also stresses the central role the Elevator class has in the system.

8.4.5 Elaboration of Class Compounds

With a better insight into the system's structure and behaviour, the class compounds can be next detailed. Two class compounds are described in this Subsection in terms of both class specification (CLS) and state diagram associated with the class (CLSTD). One class is very simple (the Button class), while the second is quite complex, bearing the responsibility of much of the work done by the system (the Elevator class). The Button class compound, with its two components shown in Fig. 8.8 (the class specification) and 8.9 (the state diagram) is included here because in this particular application other classes present in the class diagram have a behaviour similar to that of the Button (these classes are Indicator and Door).

The Elevator class includes the operations deemed necessary in the sequence diagram drawn in the previous modelling step and has also its attributes specified. These attributes include the state descriptor for the objects of the class (the state attribute), an attribute for denoting the elevator's current direction, another for keeping information about the elevator's current floor, and three attributes for the groups of possible requests (internal requests, external requests for up movement, and external requests for down movement).

The CLSTD of the Elevator class compound, shown in Fig. 8.11, is constructed based on the following states:

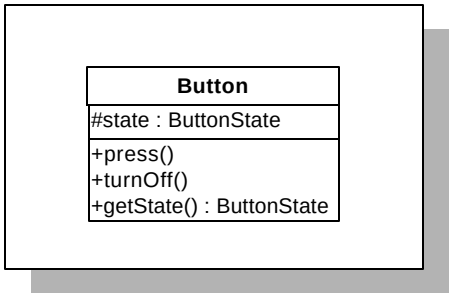

Fig. 8.8 ELS Class Button

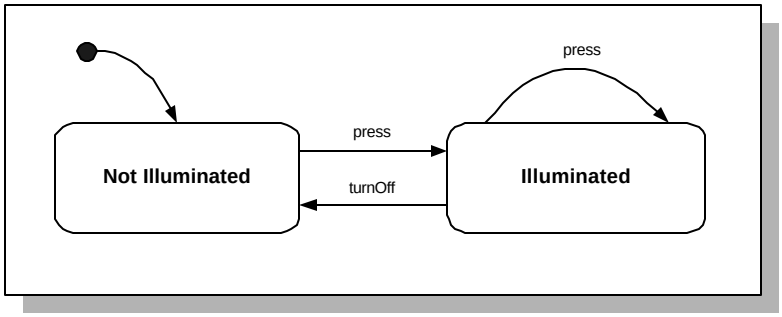

Fig. 8.9 ELS State Diagram for the Button Class

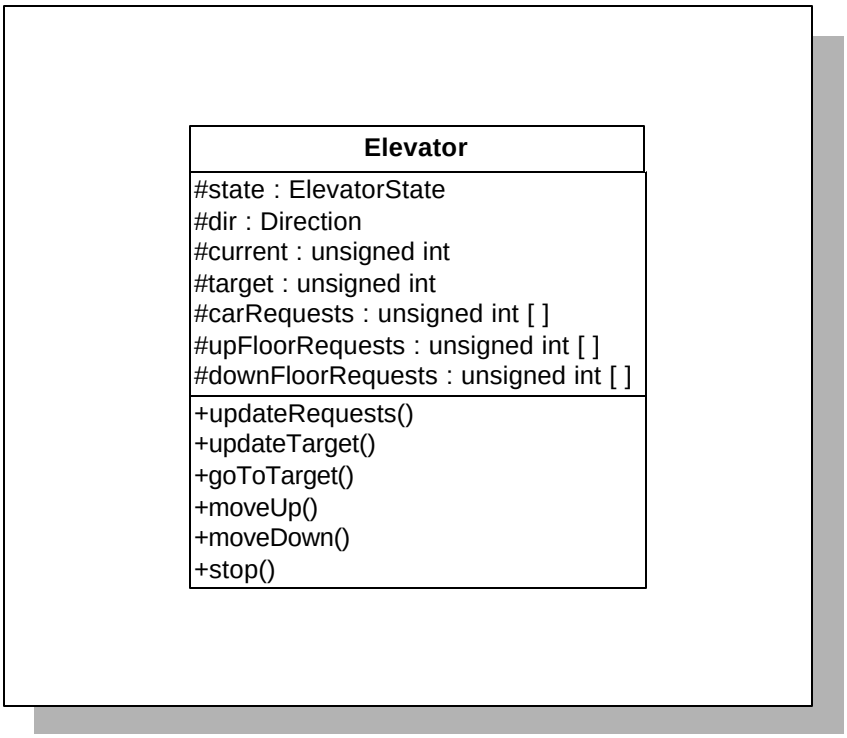

Fig. 8.10 ELS Class Elevator

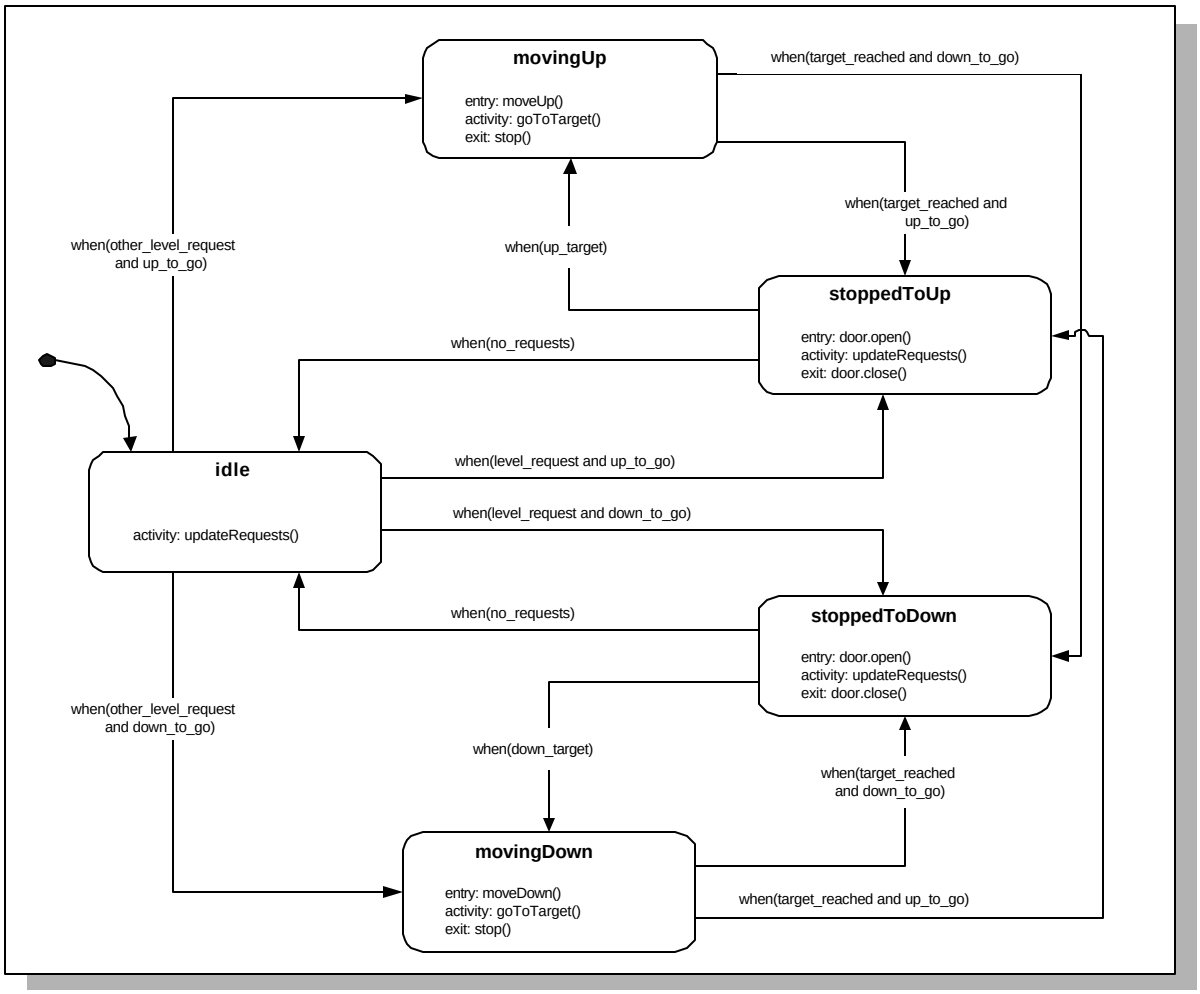

Fig. 8.11 ELS State Diagram for the Elevator Class

- idle, describing the situation in which the elevator has no requests, neither internal nor external (to be more precise, the elevator enters this state after stopping at a floor, waiting for a period of time, and still not having requests);
- movingUp, which is the state of the elevator moving upwards to its current target floor;
- movingDown, same as above, but for the opposite direction;
- stoppedToUp, which denotes the state in which the elevator is stopped at a floor and either (a) has pending requests, the analysis of which indicating that the next movement of the elevator is an up movement, or (b) has no pending requests but it

has just completed its last service coming from a lower floor and has not yet entered yet the idle state;

- stoppedToDown, same as above, but with either (a) a down "next direction to take," or (b) a last moving direction "down."

The Alarm and Stop Elevator situations (triggered by special inside car requests) correspond to two abnormal states of the elevator that for simplicity have been omitted from the diagram. In the diagram, specific change conditions lead to the transition of the elevator from state to state. Detailing of these conditions is best done in Z++, as shown later in the chapter.

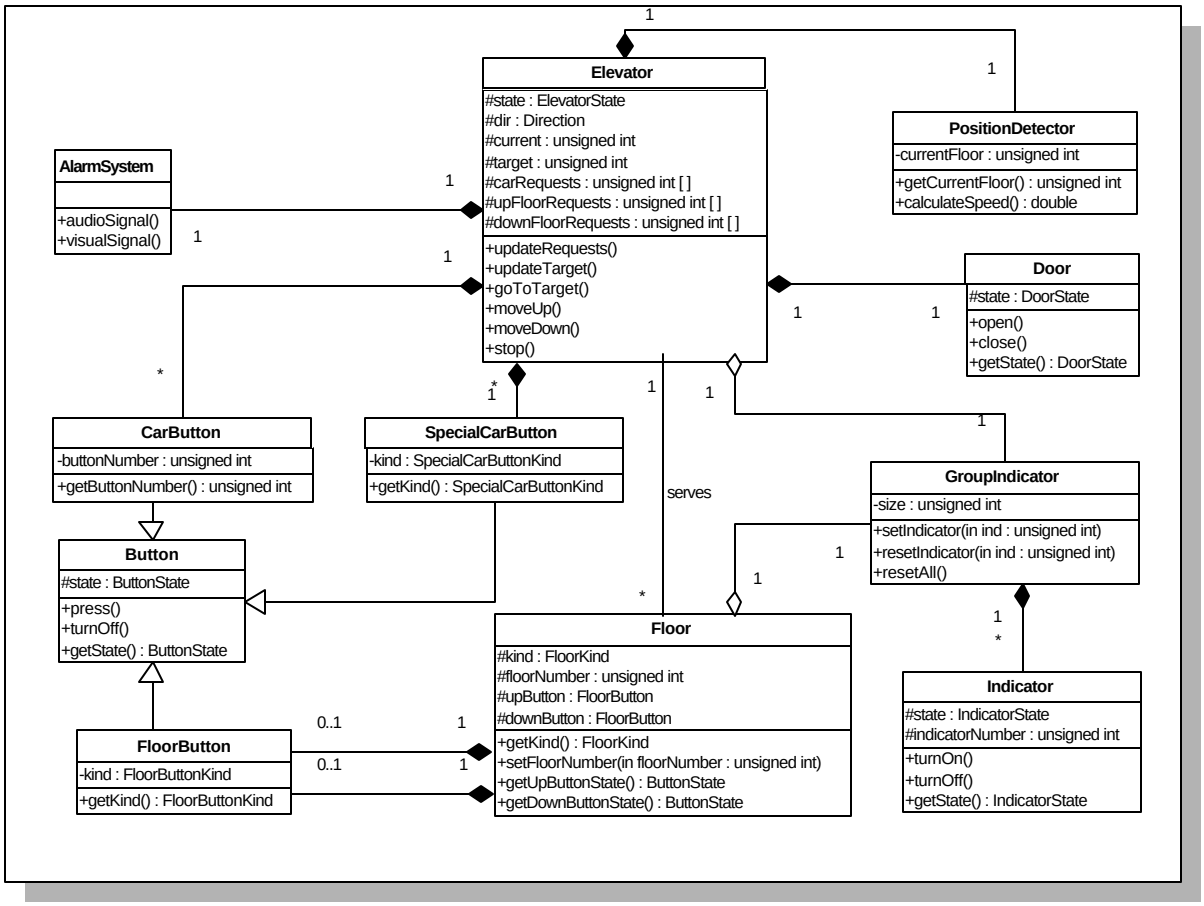

Fig. 8.12 ELS Class Diagram with Attributes and Operations Attached to Classes

With all the classes of the initial class diagram shown in Fig. 8.6 detailed in terms of attributes and operations the resulting class diagram is the one presented in Fig. 8.12.

8.4.6 Formalisation through the AFCD and the AFSD

Having the classes of the class diagram specified in detail in terms of attributes and operations and having the state diagram also specified for some classes, it is possible to apply the automated translation processes from UML to Z++ described in Chapter 6.

First, the class diagram is translated to Z++ by applying the AFCD algorithm detailed in Section 6.3, the result being shown in Fig. 8.13 (the text file generated by the Java program presented in Appendix B has been manually edited with Z specific symbols, the generation of such symbols directly from the program being one of the intended near future enhancements of the AFCD's implementation).

Next, the state diagrams associated with the classes are formalised via the AFSD algorithm presented in Section 6.4. In the case of the Button class the result, shown in Fig. 8.14, reflects the simplicity of the state diagram (it has been included here primarily for showing the groups of predicates generated in the HISTORY clause of the Z++ class), while in the case of the Elevator class presented in Fig. 8.15 it reflects the complexity of both the class' structure and behaviour. Because the Elevator state diagram is specified using transitions triggered by changed events, numerous internal operations have been created. The quite arid nature of these operations (that need be further processed by the human specifier, at least in what regards their proper renaming) has prompted us to add comments for them in the Elevator Z++ class.

In both the case of the complete Z++ specification shown in Fig. 8.13 and of the detailed Z++ class Elevator shown in Fig. 8.15, the intervention of the human formaliser after the application of the AFCD and AFSD algorithms is necessary, as described in more detail in the next Subsection.

```
[ELEVATORSTATE, DOORSTATE, INDICATORSTATE, SPECIALCARBUTTONKIND,
BUTTONSTATE, FLOORBUTTONKIND, FLOORKIND, DIRECTION]
-----------------------------------------------------------------------
CLASS System
PUBLICS
TYPES
FUNCTIONS
OWNS
   theServesDescriptor : ServesDescriptor;
RETURNS
OPERATIONS
INVARIANT
ACTIONS
HISTORY
END CLASS
// -----------------------------------------------------------------------
CLASS Elevator
PUBLICS
   setTarget, updateRequests, moveUp, moveDown, stop, alarm
TYPES
FUNCTIONS
OWNS
   state : ELEVATORSTATE;
   dir : DIRECTION;
   current : ℕ;
   target : ℕ;
   carRequests : seq(ℕ);
   upFloorRequests : seq(ℕ);
   downFloorRequests : seq(ℕ);
   door : Door;
   carButtons : ℙCarButton;
   specialCarButtons : ℙSpecialCarButton;
   groupIndicator : GroupIndicator;
   positionDetector : PositionDetector;
   alarmSystem : AlarmSystem;
RETURNS
OPERATIONS
   updateRequests :  → ;
   updateTarget :  → ;
   goToTarget :  → ;
   moveUp :  → ;
   moveDown :  → ;
   stop :  → ;
INVARIANT
ACTIONS
   setTarget   ==> ;
   updateRequests   ==> ;
   moveUp   ==> ;
   moveDown   ==> ;
   stop   ==> ;
   alarm   ==> ;
HISTORY
END CLASS
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD

```
// ---------------------------------------------------------------------
CLASS AlarmSystem
PUBLICS
   audioSignal, visualSignal
TYPES
FUNCTIONS
OWNS
RETURNS
OPERATIONS
   audioSignal :  → ;
   visualSignal :  → ;
INVARIANT
ACTIONS
   audioSignal   ==> ;
   visualSignal   ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS PositionDetector
PUBLICS
   getCurrentFloor, calculateSpeed
TYPES
FUNCTIONS
OWNS
   currentFloor : ℕ;
RETURNS
OPERATIONS
   getCurrentFloor :  → ℕ;
   calculateSpeed :  → ℝ;
INVARIANT
ACTIONS
   getCurrentFloor  result! ==> ;
   calculateSpeed  result! ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS Door
PUBLICS
   open, close, getState
TYPES
FUNCTIONS
OWNS
   state : DOORSTATE;
RETURNS
OPERATIONS
   open :  → ;
   close :  → ;
   getState :  → DOORSTATE;
INVARIANT
ACTIONS
   open   ==> ;
   close   ==> ;
   getState  result! ==> ;
HISTORY
END CLASS
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```
// ---------------------------------------------------------------------
CLASS CarButton
PUBLICS
   getButtonNumber
TYPES
FUNCTIONS
OWNS
   buttonNumber : ℕ;
RETURNS
OPERATIONS
   getButtonNumber :  → ℕ;
INVARIANT
ACTIONS
   getButtonNumber  result! ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS SpecialCarButton
PUBLICS
   getKind
TYPES
FUNCTIONS
OWNS
   kind : SPECIALCARBUTTONKIND;
RETURNS
OPERATIONS
   getKind :  → SPECIALCARBUTTONKIND;
INVARIANT
ACTIONS
   getKind  result! ==>
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS Button
PUBLICS
   press, turnOff, getStatus
TYPES
FUNCTIONS
OWNS
   state : BUTTONSTATE;
RETURNS
OPERATIONS
   press :  → ;
   turnOff :  → ;
   getStatus :  → BUTTONSTATE;
INVARIANT
ACTIONS
   press   ==> ;
   turnOff   ==> ;
   getStatus  result! ==> ;
HISTORY
END CLASS
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```
// ---------------------------------------------------------------------
CLASS FloorButton
PUBLICS
   getKind
TYPES
FUNCTIONS
OWNS
   kind : FLOORBUTTONKIND;
RETURNS
OPERATIONS
   getKind :  → FLOORBUTTONKIND;
INVARIANT
ACTIONS
   getKind  result! ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS Floor
PUBLICS
   getKind, setFloorNumber, getUpButtonState, getDownButtonState
TYPES
FUNCTIONS
OWNS
   kind : FLOORKIND
   floorNumber : ℕ;
   upButton : FloorButton;
   downButton : FloorButton;
   groupIndicator : GroupIndicator;
RETURNS
OPERATIONS
   getKind :  → FLOORKIND;
   setFloorNumber : ℕ → ;
   getUpButtonState :  → BUTTONSTATE;
   getDownButtonState :  → BUTTONSTATE;
INVARIANT
ACTIONS
   getKind  result! ==> ;
   setFloorNumber floorNumber?  ==> ;
   getUpButtonState  result! ==> ;
   getDownButtonState  result! ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS GroupIndicator
PUBLICS
   setIndicator, resetIndicator, resetAll
TYPES
FUNCTIONS
OWNS
   size : ℕ;
   indicators : ℙIndicator;
RETURNS
OPERATIONS
   setIndicator : ℕ → ℕ;
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```
   resetIndicator : ℕ → ;
   resetAll :  → ;
INVARIANT
ACTIONS
   setIndicator ind? result! ==> ;
   resetIndicator floorNumber?  ==> ;
   resetAll   ==>
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS Indicator
PUBLICS
   turnOn, turnOff, getState
TYPES
FUNCTIONS
OWNS
   state : INDICATORSTATE;
   indicatorNumber : ℕ;
RETURNS
OPERATIONS
   turnOn :  → ;
   turnOff :  → ;
   getState :  → INDICATORSTATE;
INVARIANT
ACTIONS
   turnOn   ==> ;
   turnOff   ==> ;
   getState  result! ==> ;
HISTORY
END CLASS
// ---------------------------------------------------------------------
CLASS ServesDescriptor
PUBLICS
TYPES
FUNCTIONS
OWNS
   instancesofElevator : ℙElevator;
   instancesofFloor : ℙFloor;
   servesInstances : Floor ⇸ Elevator;
RETURNS
OPERATIONS
INVARIANT
   dom servesInstances = instancesofFloor
   ran servesInstances = instancesofElevator
ACTIONS
HISTORY
END CLASS
HPositionDetector ≙ PositionDetector \ [currentFloor]
HDoor ≙ Door \ [state]
HCarButton ≙ CarButton \ [buttonNumber]
HSpecialCarButton ≙ SpecialCarButton \ [kind]
HFloorButton ≙ FloorButton \ [kind]
HGroupIndicator ≙ GroupIndicator \ [size]
HIndicator ≙ Indicator \ [state, indicatorNumber]
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```
CLASS  Button
PUBLICS

   press, turnOff

TYPES

   ButtonState ::= notilluminated | illuminated

FUNCTIONS
OWNS

   state : ButtonState

RETURNS
OPERATIONS

   press: → ;
   turnOff: → ;

INVARIANT
ACTIONS

   init ==> state' = notilluminated;
   press ==> state' = illuminated;
   turnOff ==> state' = notilluminated;

HISTORY

   // mutual exclusion properties

   mutex({press, turnOff}) ∧ self_mutex ({press, turnOff}) ∧

   // permission predicates

    (press ⇒ state = notilluminated ∨ state = illuminated) ∧
    (turnOff ⇒ state = illuminated) ∧

   // definition of transition effects

    (press ⇒ ○(state = illuminated)) ∧
    (turnOff ⇒ ○(state = notilluminated)) ∧

   // reachability properties

    (state = notilluminated ⇒press) ∧
    (state = illuminated ⇒press ∨ turnOff)

END CLASS
```

Fig. 8.14 ELS Z++ Class Button Updated by the AFSD

```
CLASS  Elevator
PUBLICS

   press, turnOff

TYPES

    ElevatorState ::= idle | movingup | movingdown | stoppedtoup |
                          stoppedtodown

FUNCTIONS
OWNS

   state : ElevatorState;
   dir : Direction;
   current : ℕ;
   target : ℕ;
   carRequests : seq(ℕ);
   upFloorRequests : seq(ℕ);
   downFloorRequests : seq(ℕ);
   door : Door;
   carButtons : ℙCarButton;
   specialCarButtons : ℙSpecialCarButton;
   groupIndicator : GroupIndicator;
   positionDetector : PositionDetector;
   alarmSystem : AlarmSystem;

RETURNS
OPERATIONS

   updateRequests :  → ;
   updateTarget :  → ;
   goToTarget :  → ;
   moveUp :  → ;
   moveDown :  → ;
   stop :  → ;
  *t1 : → ;                      // condition triggered operations
  *t2 : → ;                      // describing the transitions of
  *t3 : → ;                      // the Elevator state diagram
  *t4 : → ;                      // shown in Fig. 8.12
  *t5 : → ;
  *t6 : → ;
  *t7 : → ;
  *t8 : → ;
  *t9 : → ;
```

Fig. 8.15 ELS Z++ Class Elevator Updated by the AFSD (continued from previous page)

```
INVARIANT
ACTIONS

// [...]
// definitions of Elevator operations as in Fig.8.12

*t1 ==> state' = movingup;       // idle to movingup
*t2 ==> state' = movingdown;     // idle to movingdown
*t3 ==> stop;                    // movingup or movingdown to stoppedtoup
        state' = stoppedtoup;
*t4 ==> stop;                    // movingup/movingdown to stoppedtodown
        state' = stoppedtodown;
*t5 ==> door.close;              // stoppedtoup to movingup
        state' = movingup;
*t6 ==> door.close;              // stoppedtodown to movingdown
        state' = movingdown;
*t7 ==> state' = stoppedtoup;    // idle to stoppedtoup
*t8 ==> state'= stoppedtodown;   // idle to stoppedtodown
*t9 ==> state'= idle;            // stoppedtoup/stoppedtodown to idle

HISTORY

// mutual exclusion properties, permission predicates, definition
// of transition effects and reachability properties omitted for
// simplity (examples are available in Fig. 6.30 and 8.14)

// enabling conditions:

(enabled(t1) ≡ (state = idle) ∧ other_level_request_and_up_to_go) ∧
(enabled(t2) ≡ (state = idle) ∧ other_level_request_and_down_to_go) ∧
(enabled(t3) ≡ (state = movingup ∨ state = movingdown) ∧
                target_reached_and_up_to_go) ∧
(enabled(t4) ≡ (state = movingup ∨ state = movingdown) ∧
                target_reached_and_down_to_go) ∧
(enabled(t5) ≡ (state = stoppedtoup) ∧ up_target) ∧
(enabled(t6) ≡ (state = stoppedtodown) ∧ down_target) ∧
(enabled(t7) ≡ (state = idle) ∧ level_request and up_to_go) ∧
(enabled(t8) ≡ (state = idle) ∧ level_request and down_to_go) ∧
(enabled(t9) ≡ (state = stoppedtodown ∨ state = stoppedtoup) ∧ norequests

END CLASS
```

Fig. 8.15 ELS Z++ Class Elevator Updated by the AFSD (continued from the previous page)

8.4.7 Enhancement of the Formal Specification

After the automated translation from UML to Z++ takes place, the results obtained by applying the AFCD and the AFSD must be checked since modifications and additions may be necessary. In the case of the ELS Z++ specification example shown in Fig. 8.13, it can be observed that the "state types" (e.g., ElevatorState) are translated by the AFCD as given sets, although they should be defined as enumerated sets. Also, in the case of the Z++ class Elevator shown in Fig. 8.15, the attributes of array type denoting the internal and external requests of the elevator are translated as seq(ℕ), although a more suitable representation in this particular case is ℙℕ, since these attributes are better modelled as sets.

The work on the formal specification, aimed at its enhancement, encompasses various aspects. In particular, all sorts of constraints on both the structure and the behaviour of the class' instances, as well as the bodies of the operations can be specified. Without entering in too many details, we exemplify this aspect of formalisation by considering the Elevator Z++ class of Fig. 8.15 and by defining more precisely the conditions that trigger the transitions between the elevator's states. Using the modified definitions:

$$carRequests: \mathbb{P}\mathbb{N}$$

$$upFloorRequests: \mathbb{P}\mathbb{N}$$

$$downFloorRequests: \mathbb{P}\mathbb{N}$$

and the equivalences:

$$floorRequests \triangleq upFloorRequests \cup downFloorRequests$$

$$requests \triangleq carRequests \cup floorRequests$$

some of the conditions of the internal transit operations t_K included in the Elevator class can be written as follows (for each condition the transition it triggers is shown on the right-hand side of the formula):

(a) other_level_requests_and_up_to_go ≙ current ∉ floorRequests ∧
∃x ∈ floorRequests • x > current [t_1]

(b) target_reached_and_down_to_go ≙ (current = target) ∧
(dir = down ∧ ∄x ∈ requests • x > target) ∨ (dir = up ∧
∄x ∈ requests • x > target ∧ ∃y ∈ requests • y < target)
[t_4]

(c) up_target ≙ target > current [t_5]

(d) level_request_and_up_to_go ≙ current ∈ upFloorRequests [t_7]

(e) no_requests ≙ requests = ∅ [t_9]

Of course, the above are a very small part of the work needed in the Z++ space, a significant amount of detail being necessary to describe the system in a complete and precise way. In particular, the modelling of the complex Z++ class Elevator is a laborious task, in which the fine interplaying of conditions and operations need be carefully specified. During the enhancement of the formal specification the deformalisation process can also take place, modifications performed in the Z++ space being reflected partially in the UML space.

Attention to temporal properties of the system is also necessary. A detailed, elaborate specification of these properties is possible by writing RTL formulae in the HISTORY clause of the Z++ classes. Taking into consideration the temporal constraints placed in Section 8.2 on the Elevator System, solutions for expressing them in Z++ can involve the following expressions:

(a) For the temporal constraint [T1] the condition for the door to open within a given interval of time after the elevator stops at a floor can be expressed as:

$$\forall i:\mathbb{N}_1 \bullet \exists j:\mathbb{N}_1 \bullet \uparrow(door.open,j) - \downarrow(stop,i \geq OPEN\_MIN\_TIME \wedge \downarrow(door.open,j) - \downarrow(stop,i) \leq OPEN\_MAX\_TIME$$

if the interpretation of the constraint is that the door starts to open and completes this action within the specified time bounds, or as:

$$\forall i{:}\mathbb{N}_1 \bullet \exists j{:}\mathbb{N}_1 \bullet \downarrow(\mathrm{stop},i) = \rightarrow(\mathrm{door.open},j) \\ \wedge\ \mathrm{OPEN\_MIN\_TIME} \leq \mathrm{delay}\ (\mathrm{door.open},j) \leq \mathrm{OPEN\_MAX\_TIME}$$

if the requirement is that the door only starts its opening within the specified time bounds.

(b) For the temporal constraint [T2], the condition for the door to stay open at floor for a specific period of time provided the CloseButton is not pressed during this period of time can be expressed as:

$$\forall i{:}\mathbb{N}_1 \bullet (\nexists j{:}\mathbb{N}_1 \bullet \downarrow(\mathrm{door.open},i) \leq \clubsuit((\mathrm{CloseButton.state} = \mathrm{on}){:=}\mathrm{true},j) \leq \\ \downarrow(\mathrm{door.open},i) + \mathrm{STAY\_OPEN\_NORMAL\_TIME}) \Rightarrow \\ \uparrow(\mathrm{door.close},i+1) = \downarrow(\mathrm{door.open},i) + \mathrm{STAY\_OPEN\_NORMAL\_TIME}$$

(c) For the temporal constraint [T3] the correlation between the closing of the door and the start of the elevator movement (either up or down) can be expressed as:

$$\forall i{:}\mathbb{N}_1 \bullet \exists j{:}\mathbb{N}_1 \bullet \mathrm{CLOSE\_MIN\_TIME} \leq \uparrow(\mathrm{move},i) - \downarrow(\mathrm{door.close},j) \\ \leq \mathrm{CLOSE\_MAX\_TIME}$$

(d) For the temporal constraint [T7] the details regarding the audio and visual signals in case of emergency can be written as:

$$\forall i{:}\mathbb{N}_1 \bullet \mathrm{MIN\_SIGNAL\_DUR} \leq \mathrm{duration}(\mathrm{signal},i) \leq \mathrm{MAX\_SIGNAL\_DUR} \wedge \\ \uparrow(\mathrm{signal},i+1) - \downarrow(\mathrm{signal},i) \leq \mathrm{SIGNAL\_SEPARATION}$$

The constraint [T4] can be modelled using a variable that records the value of now at "new floor" occurrences during the elevator's movement, while conditions [T5] and [T6] can be expressed with predicates similar to those presented above.

Using the specification capabilities of RTL, including the extensions proposed by Lano for its use within the frame of Z++, detailed time-related requirements placed on the system can be rigorously expressed.

8.5 Chapter Summary

In this chapter the modelling approach proposed in the thesis has been exemplified using an Elevator System on which a number of general and temporal constraints have been placed. Examples of artefacts for all the steps of the regular modelling process presented in Chapter 7 have been provided and observations on the role of each step have been included. Examples of applying the AFCD and AFSD algorithms described in Chapter 6 have also been provided, the class diagram of the ELS being translated into a Z++ specification. Remarks on the need for enhancing the formal specification, as well as on the need of precisely describing the temporal aspects of the systems have also been included.