MODELING AND SCHEDULING ANALYSIS OF REAL-TIME SYSTEMS

BY

DAYANG NORHAYATI ABANG JAWAWI

HABIBAH BINTI ISMAIL

YAVUZ SELIM SENGOZ

UNIVERSITI TEKNOLOGI MALAYSIA

CHAPTER 3

SCHEDULABILITY ANALYSIS MODELING

3.1 Real-Time Modeling Profiles

Unified Modeling Language (UML) is the best standard of Object-Oriented (OO) method language for all software and it is widely used due to its extensibility mechanism, stereotypes, tag values and various profiles. Another important feature of UML is its object paradigm that supports coping with software complexities effectively and these features allow adaption of UML to support a specific analysis. A common notation must provide the following rules:

i. A language that used for communicating design decisions (very difficult to infer from code)
ii. Rich Semantics to capture important strategic and tactical decisions.
iii. In a form that concrete enough for humans to work with and for tools to manipulate.

UML provides a rich visual notation to support the analysis and design activities (Booch, 1996). There are some UML profiles that attempted to solve RTS' complexities and additionally, most of them include some specific features to fit UML into the real time design.

3.2 UML Real-Time (UML-RT)

This UML-RT was derived from ROOM and it is used to capture ROOM modelling concept by using UML extension (Selic et al., 1994). Basically, the designer can design the complex and large RTS in UML-RT, particularly event-driven and distributed systems. This UML-RT developed to handles the system structurally and behaviourally. Unfortunately, even though UML-RT claimed that it was designed to model the real-time design, it does not support timing requirement model (Gherbi and Khendek, 2006). Rational Rose Real-Time is a design modelling tool that supports UML-RT, and this tool allows an automatic code generation during the design process.

3.2.1 Structural Aspect of UML-RT

Structural aspects consist of capsules, port and connector. The capsules (correspond to the actor in ROOM) referred to an active object and it can encapsulate other capsules within its structure, while capsules that placed within the contained capsule is called sub capsules. Capsules are the fundamental element of UML-RT models. As mentioned, they are active objects since they interact with their environment via their ports; therefore a capsule may have a state machine that shows the active object's behaviour. The capsule responsibility is receiving the signals, once a capsule receives the signal from its end-port, it will be executed in the state machine. Besides, capsules possess its own control of thread and the process of receiving signals is executed in the thread.

Port is an object that sends and receives messages to and from capsule instances. It could be named based on three classifications; visibility, connector type and termination. According to the visibility, a port can be located on the boundary of the capsule that called a *public* port. This type of port can be seen from outside and inside of the capsule. Moreover, public port is used to connect with other capsules. And the *protected* port is only located in the capsule and it cannot

be seen from outside of the capsule. Ports are owned by the capsules instance; thus once a capsule instance is destroyed, its ports also would be destroyed altogether and vice versa. In capsule instance, each port has an individual identity apart from its owning capsule instance. Moreover, port is an interface of a capsule that establishes interaction between other capsules and the environment (Selic and Rumbaugh, 1998). With respect to the interaction, the message is sent and received through ports by using two types of signals that are output and input signal. Output signal is used for sending signal, while input signal is used for receiving signal.

A protocol class specifies signals that are to be sent or received via ports. In other words, protocol class is similar as a contract between capsules to provide a good communication. In addition, the protocol has two roles; base or conjugate roles, in which used to define the two communicating ports based on its incoming or outgoing signal. With regard to the protocol roles, base and conjugate roles are opposed to each other, for example, when two capsules communicate with each other thorough their ports, one port will plays the base role by using incoming signals while the another plays the conjugate role by using outgoing signals in the protocol class.

With respect to the type of port, *relay port* passes the messages that received by the container capsule directly to its contained capsules (sub capsules), the relay port is known as wired and public. However, if there is no connection from a *relay port* to the contained capsules, the arriving messages would be lost or undelivered in the system. Another type of port is *end port*, it is the ultimate destination for all messages that sent by other capsules. Thus, those messages are directly delivered to be processed by the contained capsule's state machine. Thereafter, the state machine will execute the events once it arrives at the end-port; however it does not mean that the state machines do not execute the internal messages of the contained capsule. When an internal message reach one of the contained capsule's end-port, those messages can be executed in the state machine as well (Ferreira et al., 2008). Therefore, each end-port has 'event queue' to hold asynchronous messages that are not yet executed.

Last classification of ports is based on connector types, unwired and wired ports. Regarding the wired port, it must use a connector to connect to other ports in order to exchange messages while unwired ports cannot be connected to other ports by using connectors. Moreover, not as wired ports which defined, created and destroyed with capsule instances at design time (Rational Software Corporation, 2003), unwired ports are dynamically controlled at run time. Therefore, unwired ports provide dynamic communication through runtime.

3.2.2 Behavioural Aspect of UML-RT

UML-RT' structural aspect are modelled by using class, structure and collaboration diagrams while the behavioural aspects of UML-RT construct by using statechart and sequence diagram. In UML-RT model, it known that a each capsule has its structure and statechart diagrams. Moreover, statechart diagram is used to model the internal behaviour of a capsule where it uses states, transitions, events and actions. In addition, sequence diagram can be used to show the distinct flows of events in the system. This is because sequence diagram focus more on timing issues and it is used to show the timing requirements of the system. Apart from sequences diagrams, the statechart diagram is mostly used to describe capsule's behaviour. Once the capsule receives signal from the environment or other capsules, its statechart executes some actions that respond to the signal. In this situation, it can be said that while statechart diagram often used to mentions about a capsule's behaviour, the sequence diagram is used to depict the collaboration of capsules that work together to achieve the same purpose. However, UML-RT gave important properties to the state machines due to the development processes of the system that designed by using capsule. The capsule has two diagrams; structure to show its structure and statechart to show its behaviour. It should not be forgotten that, only one sequences diagram can never sufficient to specify the whole system behaviour because it is designed depending on the scenario. Due to this, more than one sequence diagrams must be designed for specifying entire system behaviour as needed.

3.3 UML for Schedulability, Performance and Time (UML-SPT)

UML-SPT profile is for schedulability, performance and time; it was produced by OMG (OMG, 2005). This profile does not create a new technique but it only uses some existing techniques within it in order to capture timeliness, schedulability and so on. This profile represents a framework for modelling of time, resource and performance aspect of real-time systems (Douglass, 2004). UML-SPT supports existing UML semantics and this existing semantic is unchangeable. In addition, UML-SPT offers a set of stereotypes that can indicate which meta-model elements is annotated on a diagram. In the context of stereotypes, it is used to indicate the UML specific meta-model element within the diagrams. For example, in UML there is class, object, state and et cetera; each of them can be shown in a diagram by using their individual special stereotypes.

From the perspective of the end-user, UML-SPT looks like a set of stereotypes and tagged values that used to annotate UML design model by supporting the quantitative information such as schedulability, timing and performance (Gherbi and Khendek, 2007). Therefore, the UML-SPT model can be easily used to enable predictability of the real-time properties in the early phase of design by using the mentioned quantitative information. This profile enables the exchange of quantitative information between different tools as with any UML design tools and any scheduling and performance analysis tools. For example, when any RTS is designed by using UML model, the designer can place a prediction of quantitative information to annotate the model and after the model has been annotated, it can be converted into any convenient scheduling or performance tool to analyse the system.

3.3.1 Structural Aspect of UML-SPT

An UML-SPT profile is divided into three main sub-profile packages to raise understandability and usage (Douglass, 2004). The important point is the sub-profile packages are also composed of other small sub-profile packages, so users

are not overwhelmed with the entire profile packages and other unnecessary parts. The main sub-profiles are the General Resource Modelling (GRM) Framework, Analysis Modelling Package, and infrastructure model as shown in Figure 3.1.

General Resource Model (GRM) framework is a sub-profile that provides resource and quality of service for systems. In addition, GRM concept is similar to client-server model (OMG, 2007) and it includes the basic concept of RTS. Moreover GRM is composed of three sub-profiles:

i. **RTresourceModelling –** provides resource and quality of service. This strongly highlights the differences between static and dynamic analysis on the specification. In this sub-profile, a resource is determined with finite properties such as time availability, capacity, safety, timing and so on.
ii. **RTtimeModelling –** defines text and stereotypes for timing issues of real-time design. In this sub-profile, the designers can specify time values, time constraints and time related concepts by using stereotypes and tagged values.
iii. **RTconcurrencyModelling –** this sub-profile refines the ambiguous notion of concurrency in the core of UML, in order to use it effectively for the concurrent model.

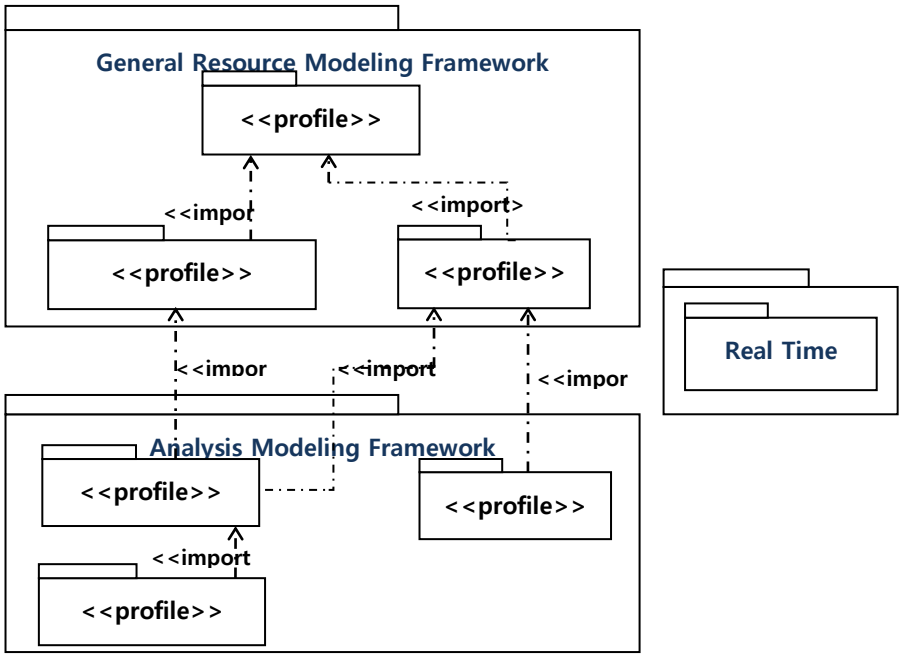

Figure 3.1: UML-SPT structures (OMG, 2005)

Analysis Modelling Package is very useful for SA because this profile provides tags and stereotypes to annotate the most common type of SA. This package is divided into two sub-profiles:

i. ***SAprofiles*** **–** This sub-profile is concerned with annotating a model SA. Table 3.1 shows the most commonly used scheduling stereotype and tagged values.

ii. ***PAprofiles*** **–** Performance analyzing profiles is used to annotate a model for computation of system performance. In this sub-profile, the stereotypes and tags defined can be used for adding quantitative measure of performance analysis. There are differences between SA and PA, whereby in SA a system must ensure whether it is schedulable or not.

Table 3.1: UML-SPT, SAProfile common stereotypes (OMG, 2005)

Stereotype	**UML Model Element**
«SAsituation»	Collaboration, Sequence diagrams
«SAtrigger»	Message, Stimulus
«SAresponse»	Method, Action
«SAaction»	Method, Stimulus
«SAscedulable»	Instance, Object, Node
«SAresource»	Instance, Class, Node
«SAengine»	Object, Class, Node

The structural elements within UML-SPT by depicted using classes in OMD (Object Modeling Diagram), interfaces and so on. In this research, the structural aspects of chosen case study were modelled by using these elements.

3.3.2 Behavioural Aspect of UML-SPT

Each element in the system may have some responsibilities in executing its task that is defined as element's methods. UML behavioural modelling mostly refers to the specification of objects' responsibilities in the system. For example, action is the fundamental behaviour of the object within the UML and it can be

defined in the method of objects. In addition, action execution within the UML might be shown in the statechart and activity diagrams. If the object receives events, transitions are triggered so that the actions can be executed. All steps that trigger the execution of the actions can be shown by using statechart diagrams. The big facility of behavioural aspect is that it depicts not only the single object's behaviour as shown in the object's statechart, but it also describes the behaviour that results from the collaboration of other objects. This is called interactions and UML provides some diagrams to show the interactions between objects such as sequences, timing and collaboration diagrams. Moreover, significant interaction between the system's object along with their time related concepts are commonly shown in sequences diagram. Additionally, when the behavioural aspect of the system design is modelled by using such UML diagrams, the designers use certain other elements such as events, states, transitions unlike structural elements like classes, objects and uses cases.

With respect to the behaviour of structural elements, it is more concerned about how the structural elements act while the system is running. The behaviour of the elements can be analyzed as one by one (statechart) or along with other objects (sequence diagram).

3.4 UML for Modeling and Analysis of Real-Time Embedded systems (UML-MARTE)

The extension of the UML profile called the Modelling and Analysis of Real-Time Embedded profile was introduced by the OMG (2007) to be the future standard for UML modelling of real-time and embedded systems although a number of other modelling standards existed already. The standard also was defined to improve the specification of timing requirements and to prepare models for timing analysis. The UML-MARTE profile provided some new key features such as support non-functional property modelling and added rich time and resource models to the UML.

3.4.1 Structural Aspect of UML-MARTE

The MARTE profile or specification consists of three main packages (OMG, 2007). The profile, as illustrated in Figure 3.2, consists of foundation, design and analysis modules that provide the basic elements.

The first package is the MARTE foundation package which defines the foundational concepts to design and analyse an embedded and real-time system. It provides basic model constructs for non-functional properties, time and time-related concepts, allocation mechanisms and generic resources, including concurrent resources.

The second package is the MARTE design model that offers elements for requirement capturing and the specification, design and implementation phases. The four elements of the design model are the generic component model (GCM), high-level application modelling (HLAM), software resource modelling (SRM) and hardware resource modelling (HRM).

The third package is the MARTE analysis model which defines the specific model abstraction and annotations that could be used by external tools to analyse the described tools. The main analysis package is the generic quantitative analysis modelling (GQAM) package that supports two main sub-profiles for schedulability analysis called the schedulability analysis modelling (SAM) to predict whether a set of software tasks meets its timing constraints in terms of timeliness, and performance analysis called the performance analysis modelling (PAM) to determine if a system with non-deterministic behaviour can provide adequate performance. Table 3.2 shows the most commonly used scheduling stereotypes and tagged values.

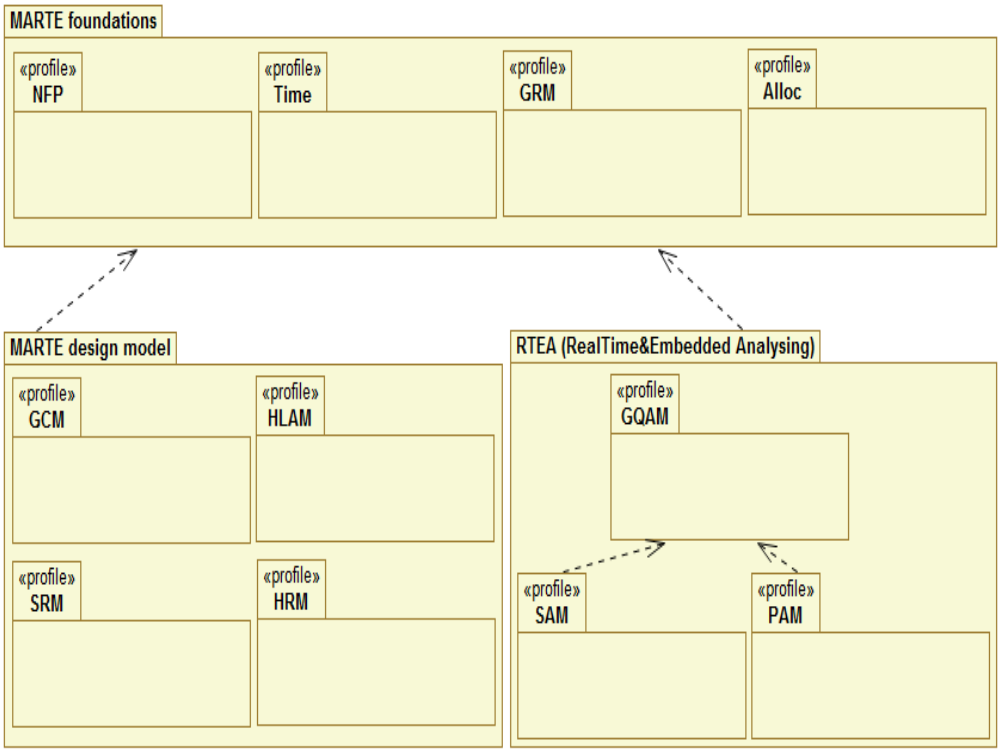

Figure 3.2: UML-MARTE structures (OMG, 2007)

Table 3.2: UML-MARTE, SAM profile common stereotypes (OMG, 2007)

Stereotype	UML Model Element
«saExecHost»	Object, Class, Node
«Scheduler»	Object, Class, Node
«schedulableResource»	Instance, Object, Node
«saEnd2EndFlow»	Activity
«gaWorkloadEvent»	Initial-Node
«gaScenario»	Method, Action
«saStep»	Method, Stimulus
«gaTimingObs»	Method, Action
«gaLatencyObs»	Method, Stimulus

3.4.2 Behavioral Aspect of UML-MARTE

Each element in the system may have some responsibilities in executing its task that is defined as element's methods. UML behavioural modelling mostly refers to the specification of objects' responsibilities in the system. For example, action is the fundamental behaviour of the object within the UML and it can be

defined in the method of objects. In addition, action execution within the UML might be shown in the statechart and activity diagrams. If the object receives events, transitions are triggered so that the actions can be executed. All steps that trigger the execution of the actions can be shown by using statechart diagrams. The big facility of behavioural aspect is that it depicts not only the single object's behaviour as shown in the object's statechart, but it also describes the behaviour that results from the collaboration of other objects. This is called interactions and UML provides some diagrams to show the interactions between objects such as sequences, timing and collaboration diagrams. Moreover, significant interaction between the system's object along with their time related concepts are commonly shown in sequences diagram. Additionally, when the behavioural aspect of the system design is modelled by using such UML diagrams, the designers use certain other elements such as events, states, transitions unlike structural elements like classes, objects and uses cases.

3.5 Design Tool

In this study, Rhapsody, Papyrus and Rational Rose Real-Time (RoseRT) design tools are used to perform the design of real-time systems. These three design tools are related, with RoseRT used for designing UML-RT and UML-SPT, the Rhapsody used for designing UML-SPT and UML-MARTE and the Papyrus tool used only for designing the UML-MARTE model.

3.5.1 Papyrus

Papyrus is an open source modelling tool for UML. This open source tool is based on the Eclipse environment. It supports UML 2.0 and the systems modelling language (SysML) for embedded real-time systems. Papyrus provides an efficient graphical editor for the UML 2.0. Papyrus addresses the two key features expected from a UML 2.0 graphical editor, namely, modelling and profiling. In modelling, Papyrus supports several of the diagrams described in the UML specification. Meanwhile, in profiling, Papyrus adapts the UML to specific the modelling aspects

or the business domain. Moreover, Papyrus offers a graphical editor to create new profiles and to define new modelling concepts (stereotypes). Papyrus codes generation plug-ins for the C++, C and Java languages. Besides using the Papyrus tool, Rhapsody can also be used in combination with the UML-MARTE profile.

3.5.2 Rhapsody

Rhapsody is a tool that provides a visual design platform for the design, analysis and evaluation of the RTS and embedded systems. It is produced by i-Logix Company for providing visual designing environment for designers by using UML (I-Logix, 2004). It uses all basic UML notations and methods. In addition, it supports code generation and animation of diagrams such as statechart and sequence diagram. This feature makes analysis and specification of the system easier and understandable, especially given that animation of diagram is very useful to observe the running of system and detect errors before the implementation phase.

Moreover, this tool supports and provides multi-thread and reactive real time environment. It also provides basic concurrency and synchronization construct such as mutexes, semaphores or timers (Harel and Gery, 1997). In addition, Rhapsody offers GUI (Graphical User Interface) for designers, which makes the detection of defects easier by comparing between a generated diagram and user-defined diagram. Another advantage of Rhapsody is that it provides orthogonal statechart to illustrate the execution of more than one event that might occur concurrently. Finally, Rhapsody generates automatic codes through design in C, C++ or Java (Fischer et al., 2004).

3.5.3 Rational Rose Real-Time (RoseRT)

RoseRT tool was developed by Rational Software Company to model the RTS. Moreover, In this research this tool was selected because it is widely used and supports UML-RT (Kim et al., 2000). It is also associated with Object Time Developer to draw UML diagrams and generates codes from its diagram (Rational

Software Corporation, 2003). The biggest advantage of this tool is it can generate complete and executable programs. In addition, the skeleton code in architectural level is automatically generated and the designer can detect important numbers of defects and fix the defect earlier before entering the implementation phase. RoseRT tool uses UML-RT annotation for design and it also provides message handling and schedule execution between the capsules in the same thread. On the other hand, this tool also has a clock that may be used to fire a state transition at the pointed time. However this tool does not support the real-time constraint because RoseRT does not have any execution deadline (Toivanen, 2004).

3.6 Comparison of Design Tools

The evaluation criteria and comparison results of the tools are presented in Table 3.3. Papyrus and Rhapsody are suitable modelling tools for real-time systems because of the use of modelling language as the modelling notation in both tools, although using different UML profiles.

The Rhapsody tool can support the UML-SPT profile. Meanwhile, the Papyrus tool is the most commonly used tool in combination with the UML-MARTE profile. Moreover, Papyrus can support the UML-MARTE profile through plug-ins but for such operation it still needs to be improved because it has some major limitations, such as missing diagrams and a lack of diagram tools. And, the Rational Rose Real-Time (RoseRT) tool can support UML-RT and UML-SPT as well.

Table 3.3: Comparison of three design tools

Tool / Feature	Papyrus	Rhapsody	Rational Rose Real-Time (RoseRT)
Open source	Yes	No	No
Software license	Eclipse Public License	Commercial	Node-locked and Floating Licence
Programming language used	Java	Java, C and C++	Java, C and C++
Comment	Supports UML 2.0 and SysML	Supports UML 2.0 and SysML	Supports UML 2.0
Diagrams	Class Use Case Sequence Component Deployment Activity Composite	Class Use Case Sequence Component Deployment Activity Collaboration Object Model State chart Structure Panel	Class Structure State chart Sequence
Element tools	Two elements (Palette and UML element)	Three elements (diagram tools, common and free shapes)	Two elements (capsule and port)

CHAPTER 4

REAL-TIME STRUCTURAL AND BEHAVIOURAL

4.1 Elevator System Case Study

Embedded RTS have a set of computer's parts that are controlled (devices and other mechanical parts) and those parts accomplish their task by receiving signals from the controlling system. In addition, the controlling system can receive information messages on the situation of control units. With respect to this issue, Elevator System case study was designed by using real-time UML models.

This case study was chosen based on the complexity of design and the existence of important characteristics of RTS such as timing requirement, concurrent tasks and etcetera. The structure of this case study is complex because it illustrates a building with ten floors and three elevators, and each floor consists of three shelves for elevators. In addition, it must be able to provide transportation in safe critical mission and the elevator has some sensors that contribute to the complexity of this system. There are some risks in using the elevator, as many factors has to be taken into consideration in order to ensure the passenger is transferred between floors safely and smoothly. Furthermore, all signals must be performed in-time since the Elevator System is known as hard RTS. For example, in case of fire or emergency, the elevator should be stopped at the nearest floor and the door should be in 'open' state. Due to this, there many more exceptional situations should be considered throughout the design phase of this system.

4.1.1 The Complexity in the Elevator System Operation

In this section, the Elevator System case study's complex structure along with its relationships and its environments are depicted in detail by using collaboration diagram including the events, tasks and devices that used by the system and also describes the communication of objects between each other as seen in Figure 4.1. Hereafter, the attentions are focused on the tasks of this RTS case study, therefore the timing constraints or requirements of the tasks and actions should be defined.

Thus, at first, all the tasks must be included in collaboration diagram to illustrate the interaction flow of the system. Next, it is necessary to determine the characteristic of tasks whether they are synchronous or asynchronous. At this point, the designer also must decide the environmental devices that interact with the system such as sensors, motor, door and so on and the communication type with the system, since the tasks are described based on the device's characteristics. As shown in Figure 4.1, sensor devices interacted with the control object by using asynchronous communication type, which means arrival sensor task is asynchronous according to the communication type used.

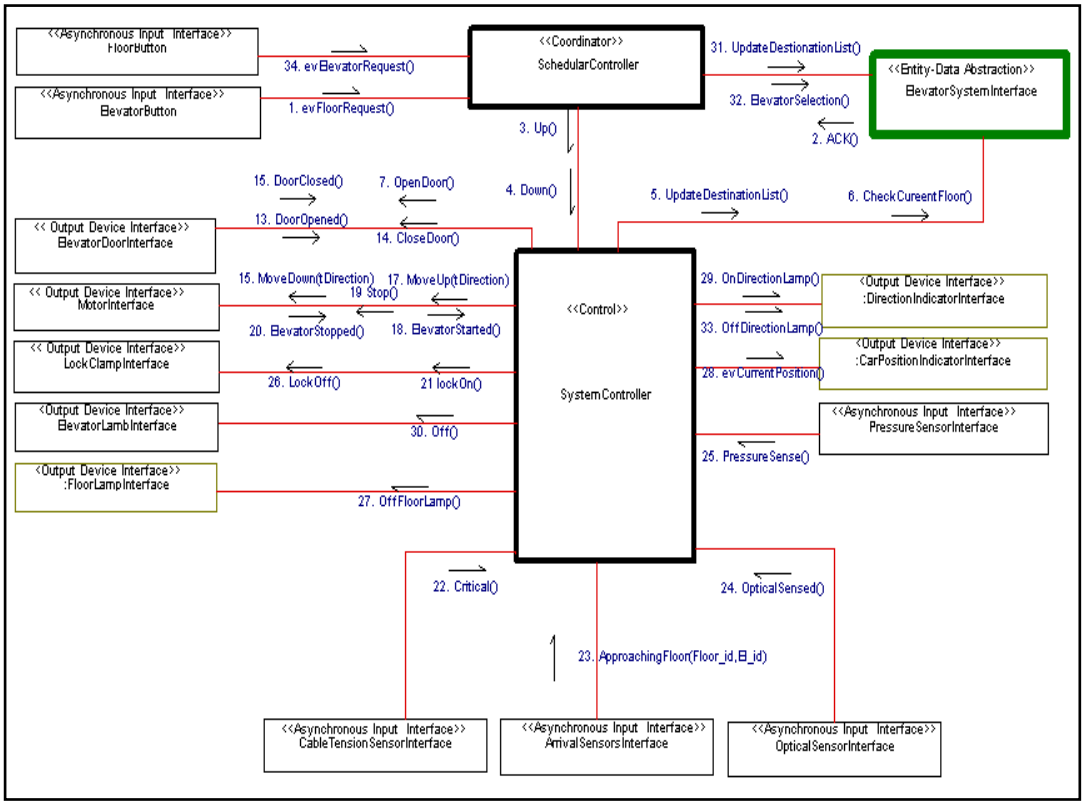

Figure 4.1: Collaboration diagram of Elevator System object in UML-SPT

Devices are part of the RTS and they communicate with the system through Input Output (I/O) device controller interfaces. In other words, CPU has direct communication with I/O device controller interfaces rather than I/O devices themselves (Gomaa, 2000). Generally, two common approaches are used to indicate on how the operating system handles I/O which is:

- *Interrupt driven approach*: when an interrupt result arrives in the CPU, it suspends the current task execution of CPU and invokes an interrupt handler to perform the interruption. When it completes the preempted task, the context is restored and resumes its execution by CPU.
- *Polled driven approach*: the system periodically checks whether any input or output arrives or completes, when a new task arrives as an input or output to the input or output buffer respectively, they are polled and checked periodically for execution.

An *interface* is also an object that has communication with external environment entities. RTS contain external physical devices (sensors, users, motor and so on) or actors that interact with the system and the interface objects are used to interface the devices to the system. In the Elevator System case study, there are some physical devices such as motor, door, arrival, pressure, optical, and cable tension sensors; hence they are interfaced by using external interface objects to the system as shown in Figure 4.1. In addition, interface objects can be categorized as input/output or input and output device interfaces. *Input device interfaces* interface only input devices and input device provides input to the system, while *output device interfaces* interface only output devices and output device merely receives output from the system's controller object. It is significant to determine I/O devices' interfaces characteristics whether it have asynchronous or synchronous device and following is the explanation on the characteristics;

- *Asynchronous device*: generates an interrupt when it produces an input or output data to send or receive to/from the system and asynchronous device interfaces interface these devices to the system and when a real word device sends an interrupt to its interface, an asynchronous device interface task is triggered by that interrupt. For example, in Elevator System case study, the arrival sensor provides external input from a real world sensor device and it is connected to the system by using an input device interface object. This object receives arrival sensor input from the real world sensor device that is depicted in Figure 4.1 as an external input device. After receiving input, the interface object converts the input data to an internal format and sends the *approachingFloor* request to the *SystemController* object. In the system design, arrival sensor device was designed as an asynchronous device and its interface was also depicted as asynchronous input device interface. This interface sends the input to the *SystemController* as asynchronous messages as depicted in Figure 4.1.
- *Synchronous device*: does not generate any interruption; however it sends or receives the data periodically. Apart from the asynchronous device interface task, synchronous device interface task is triggered by a timer

event. Hence, the timer determines the tasks' periods between two consecutive activations.

With respect to the output device interface, these devices also known as passive I/O devices depicted in Figure 4.1 and mentioned as follow:

- *Motor, Door Device Interface*: They interface Motor and Door real world devices to the system and they are passive output objects that merely receive output from the controller.
- *CarPositionIndicatorInterface*: It provides the display of current elevator places for waiting passengers in the floor or in the elevator.
- *DirectionIndicatorInterface*: This device interface works the same as the CarPositionIndicatorInterface, which means that it only shows direction of elevator (up or down) by receiving the information signal from SystemController.
- *ElevatorLampInterface*: Each elevator has 28 buttons; hence once the passengers pressed them, the hardware of the elevator light up the lamps automatically but the software in the Elevator System must switch them off when the elevator arrives at its destination.
- *FloorButtonLampInterface*: Each floor has two buttons for up and down requests and it receives the output from *SystemController* object. These lamps also should be switched off when the designated elevator arrives at its destination.

ElevatorSystemInterface is defined as a data abstraction object in the system as shown in Figure 4.1, because objects have to access this data repository to fulfil their tasks. Due to the open accessed by the objects, this data structure is exposed to some changes. As known, encapsulation is similar to the combination of data and processes within an object and this information hiding concept can be used on the data or process, depending on which information should be invisible or hidden by external entities. Hence, by using data encapsulation structure, one can only access the hidden data by calling the operations that are provided by the corresponding

object and only operations that are defined within the corresponding object are capable of manipulating the data structure. This is also called *data abstraction*. With this, in case if the data structure changed, only objects that contains in the data structure will be affected.

In order to illustrate Elevator System process that evoked by users, use case diagram as depicted in Figure 4.2, use case diagram commonly used to show the interaction of the system with users, in Elevator System case study there are two actors; Passenger and Potential Passenger. Passenger is the actor that evoke the event from inside the elevator box while, Potential Passenger is the actor on the floor and requesting for floor. There are three main actions that can be performed by the user to ping the system to work, which are identified as *Request Elevator*, *Request Floor* and *Press Hold Button* use cases. Detailed descriptions of each use case are given in following Table 4.2, 4.3, and 4.4.

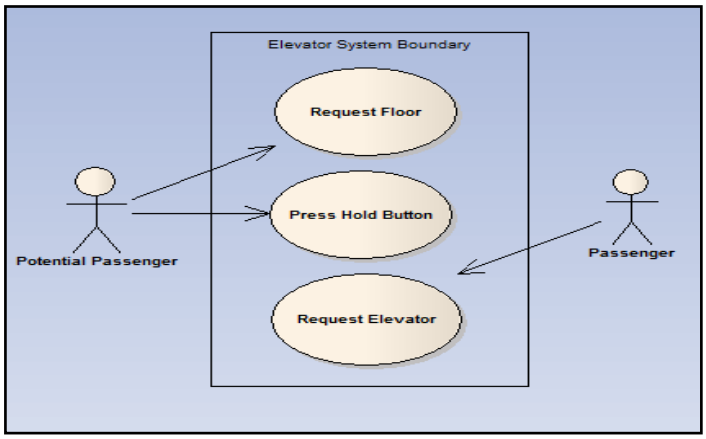

Figure 4.2: *Elevator System* Use Case Diagram

Table 4.2: Description of *Request Elevator* Use Case

Use Case 1	**:**	*Request Elevator*
Summary	**:**	Potential passenger pressed the elevator UP or DOWN request button from the floor.
Dependency		
Actor	**:**	Potential Passenger
Pre-Condition	**:**	Potential passenger pressed the elevator request button from the floor.
Description	**:**	1. Passenger presses elevator request button.

		2. The request from the floor is sent to the scheduler, and the scheduler checks the available elevator based on some condition and sends the selected elevator id and direction information to the controller. 3. The controller adds the request information to the destination list plan and moves the selected elevator to the directed direction. 4. When the elevator reaches the floor, the elevator stopped and the door opened.
Alternatives	**:**	-
Post-Condition	**:**	Elevator arrives as requested.

Table 4.3: Description of *Request Floor* Use Case

Use Case 2	:	Request Floor
Summary	:	Passenger presses desired floor button from the elevator.
Dependency		
Actor	:	Passenger
Pre-Condition	:	Passenger should be inside the elevator
Descriptions	:	1. Passenger presses desired floor button. 2. The desired floor request is sent to the scheduler and the scheduler sends the floor and direction information of request to the controller. 3. The controller adds the requested floor information to the destination list plan and sends the motor signal to move the elevator to the directed direction. 4. Each time the elevator approaches any floor, the arrival sensor sends the floor information and direction of elevator moving to the controller to be updated to the current floor indicator and elevator direction indicator. 5. The requested floor and approached floor are compared until the approached floor is same with the requested floor, if this condition is fulfilled the controller stops the elevator and opens the door.
Alternatives	**:**	Passenger press any other desired floor button
Post-Condition	**:**	Elevator is stopped and the door opened at the desired floor as requested.

Table 4.4: Description of *Press Hold Button* Use Case

Use Case 3	**:**	*Press Hold Button*
Summary	**:**	Passenger presses hold button inside the elevator while the door attempts to close.
Dependency		
Actor	**:**	Passenger

Pre-Condition	**:**	Passenger should be inside the elevator and the elevator must be on stopped state.
Description	**:**	1. Passenger presses hold button when the elevator stops and while the door attempts to close. 2. The controller opens the door and the timer is restarted.
Alternatives	**:**	-
Post-Condition	**:**	The door is opened again.

4.2 Designing Structural Aspect of the Case Study using UML-RT

The Elevator System case study designed using Rational Rose Real-Time (RoseRT) for UML-RT. To make the Elevator System case study design slightly complex and relevant to the concept of RTS, some additional sensors were added as explained detail in Section 4.1.4 such as pressure sensor, optical sensor and cable tension sensor. This addition is needed to construct a structure that is as complex as possible with RTS requirements.

4.2.1 UML-RT Class Diagram

In UML-RT, class diagrams are used to show the structural or static aspect of RTS. In other words, class diagrams represent the relationships between the capsules, protocols and passive classes. Also, class diagrams are used to show the aggregation relationships between the container capsules and their subcapsules (capsule roles) while a structure diagram of a capsule is used to show a container capsule and its sub-capsule's relationships (capsule roles) internally by using connectors. The container capsule uses composition relationship semantics, which is a very strong type of aggregation relationship. It is used to express a 'whole-parts' approach; an example is shown in Figure 4.3, where *Floor* capsule (container capsule-whole) associates with *Direction*, *FloorLamp, FloorButton and CarPosition* capsules (parts) by using composition relationships.

Aggregating the capsules into a new capsule has some advantages. Firstly it makes the complexity of design fairly simplified and understandable. Secondly, when a container is created; all its sub-capsules are created at run time or vice versa. Therefore the designer does not consider creating capsules at run time. Moreover, the container capsule, which may also be called as a top capsule, includes the main method after code generation has generated the code (Toivanen, 2004). Finally, the designer able to design the capsule's internal structure easily using collaboration (structure) diagram; however, the classes do not have such a feature.

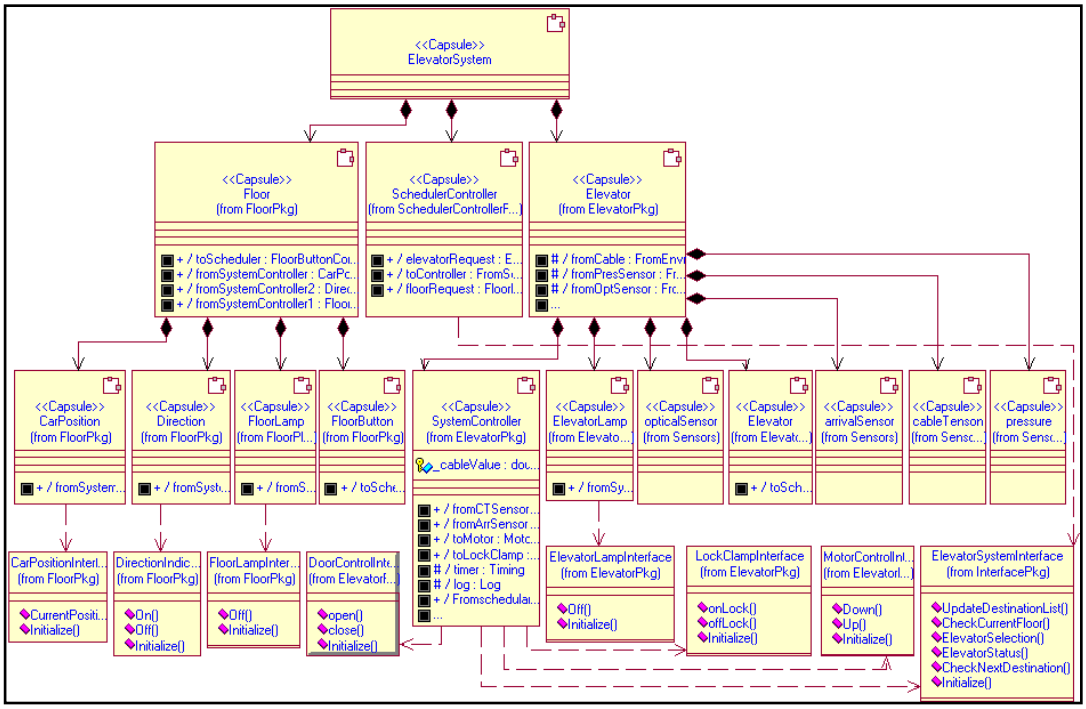

Figure 4.3: *ElevatorSystem* class diagram in UML-RT

For this research purpose, Elevator System case study was designed in UML-RT by using capsules, classes, ports, protocols and distinct diagrams. In the context of the diagrams, there are few class diagrams to depict the capsules, classes and protocols along with their relationships of *ElevatorSystem* in the UML-RT. Figure 4.3 shown *ElevatorSystem* class diagram, this diagram is the main diagram that illustrates the entire capsules and classes that are involved in the Elevator

System case study. As shown in Figure 4.3, there are thirteen (13) capsules and four (4) passive classes (can be recognized by comprising only functions with red prefix box in the Figure 4.3) that are used to depict the whole system but protocol classes are not shown in this diagram to keep the visibility of diagram neat.

In the Elevator System case study, as shown in Figure 4.3, UML-RT has one top system capsule that has the responsibility to initialize the system, align the resource and so on. The top capsule is *ElevatorSystem* capsule and it is incarnated into a main thread. However, each capsule can also be mapped into different physical threads in order to make it concurrent. As seen in Figure 4.3, *ElevatorSystem* capsule consists of *Elevator*, *Floor* and *SchedullerController* subsystem capsules. All subsystem capsules have their own individual tasks to do. In addition, they are also composed of several capsules that are designed to make the system concurrent. Therefore, they can be called as concurrent capsules. For example, *Elevator* subsystem capsule is composed of seven concurrent capsules that are *pressureSensor*, *opticalSensor*, *cableTensionSensors*, *ArrivalSensor*, *ElevatorLamp, ElevatorButtons* and *SystemController.*

The classes designed can be considered as passive class in UML-RT design. They provide the interfaces for devices and needed services through operations. In other words, these classes indicate real world devices such as door, motor, lamp and so on. Moreover as mentioned in the previous discussion, in UML-RT passive class is also preferred for use with data storage while active class (capsule) is used for sending messages (Rational Software Corporation, 2003). Those classes have some operations and the capsules invoke those operations when needed by the system. For instance in *ElevatorSystem*, the classes defined are *DirectionIndicatorInterface*, *FloorLampInterface*, *CarPositionInterface*, *MotorControlInterface*, *DoorControlInterface*, *ElevatorSystemInterface*, *Elevator LampInterface* and *LockClampInterface.* As for *SystemController* capsule, it was designed to control and monitor all elevator operations. It receives signals from sensors that exist in the environment or in the elevator and sends signals to the related capsules. *SchedullerController* capsule is another main capsule that is responsible to check whether there is any available elevator corresponding to the

request from floor button, and to add new destinations into the elevator destination list. In addition, there are other two main container capsules in the systems which are *Elevator* and *Floor* capsules. They were used to ensure that the design of elevator system is well-organized and reasonably understandable.

Protocol class specifies signals that are sent or received via a port for connection between capsule roles. In other words, protocol class is similar to a contract between capsules in order to provide good communication. The protocol has two roles, base or conjugate role, which defines the two communicated ports depending on incoming or outgoing signals. Figure 4.4 illustrates *Elevator* capsule class diagrams that include its relationships with its parts. In this diagram, some protocol classes are used to show the signals that are sent or received between related capsules. As stated previously, the protocol classes are used as a contract between two capsules and that contract involves which signal can be used as incoming or outgoing between capsules. For example, ElevatorLamp capsule and System controller capsule uses *ElevatorLampControl* protocol classes as a contract with each other, which means that *SystemController* sends *offElevatorLamp* signal to the *ElevatorLamp* capsule. At this point, this signal is an outgoing signal for *SystemController* and incoming signal for *ElevatorLamp* capsule.

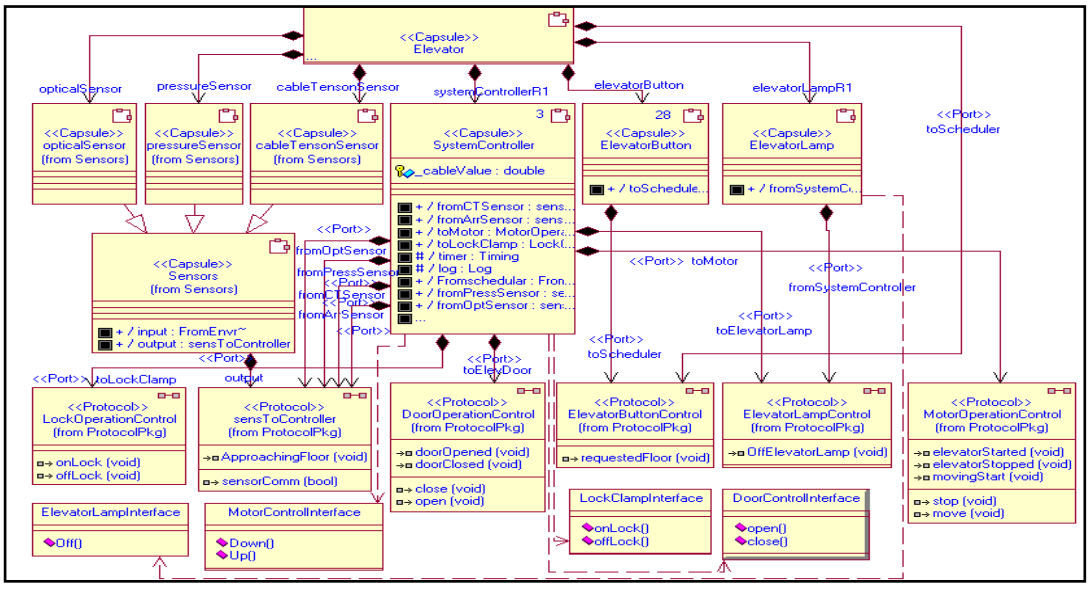

Figure 4.4: *Elevator* capsule class diagrams and relationships

As seen in Figure 4.4, *Elevator* structure consists of seven (7) contained capsules which are *arrivalSensor, elevatorLamp, elevatorButton, SystemController, cableTensionSensor, opticalSensor* and *pressureSensor* capsule instances, as well as eight (8) end ports of which four of them are used to indicate sending information to the sensors from its environment. These types of end ports are protected and wired; hence they can be never seen outside of elevator capsule structure. However these end ports can also appear in the container capsule as part of a parent capsule (*Elevator* capsule). In addition, *Elevator* capsule's state machine uses the protected end ports to communicate with its sub-capsules.

4.2.2 UML-RT Structure Diagram

Figure 4.5 shows the structure diagram of *ElevatorSystem* top capsule. A structure diagram is used to represent a structure of a capsule and its internal relationships between capsule roles. It consists of three fundamental elements such as ports, capsule roles, and connectors. As seen in Figure 4.5, *ElevatorSystem* structure diagram has three capsule roles depending on its composition relationships as depicted in Figure 4.3. Furthermore, each capsule role has some ports to provide connection between capsule roles.

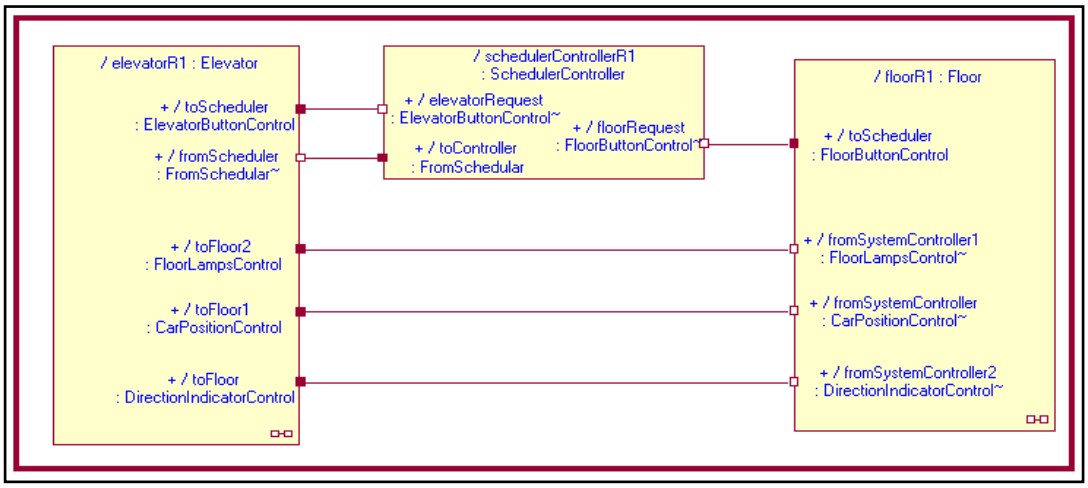

Figure 4.5: Structure diagram of *ElevatorSystem* capsule

As seen in Figure 4.6, ports of the *SystemController* capsule role are indicated by white and red fill in small squares. *SystemController* capsule has fourteen (14) end ports whereby some of them are conjugate or base roles. In that case, the protocol name with tilde (~) suffix identifies the conjugate protocol role as indicated by white-filled square such as *sensToController* and *fromSchedular*. All those identification, whether for base or conjugate roles, is just a view of protocol as it is defined in the system. As noticed earlier, if ports are on the boundary of the capsule, they have public visibility. Thus all public ports of *SystemController* are shown in *Elevator* capsule's structure diagram. Nevertheless protected ports such as timer and log are the service ports that cannot be shown because they are not parts of the capsule.

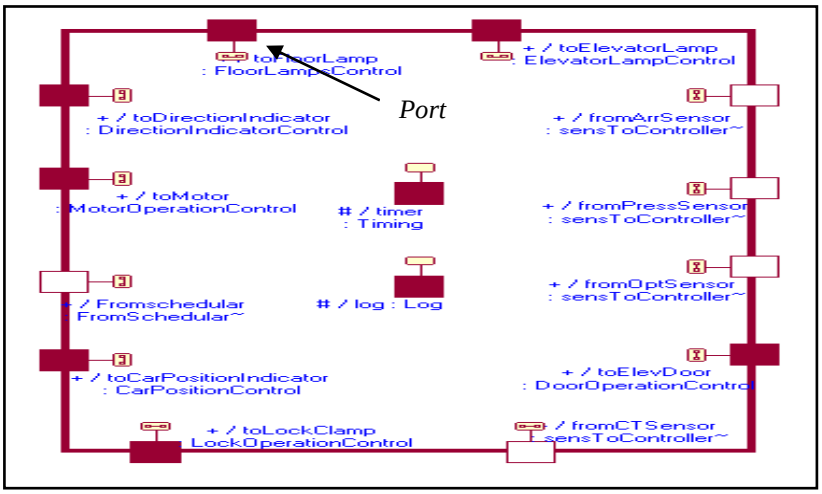

Figure 4.6: Structure diagram of *SystemController* capsule

In Figure 4.7, the important point is that some of the relay ports are conjugated port with filled white square box while others are shown with filled red square box as a base port. For example; when *opticalSensor* capsule receives any signal from optical sensor device via its *input* port, it directly sends the signal to the *sensToController* relay port through its *output* port. On the other hand, when Elevator capsule receives any signal from *SchedulerController* via its *fromSchedular* relay conjugated port, it automatically sends that signal to the *SystemController* capsule's *fromSchedular* port to be executed.

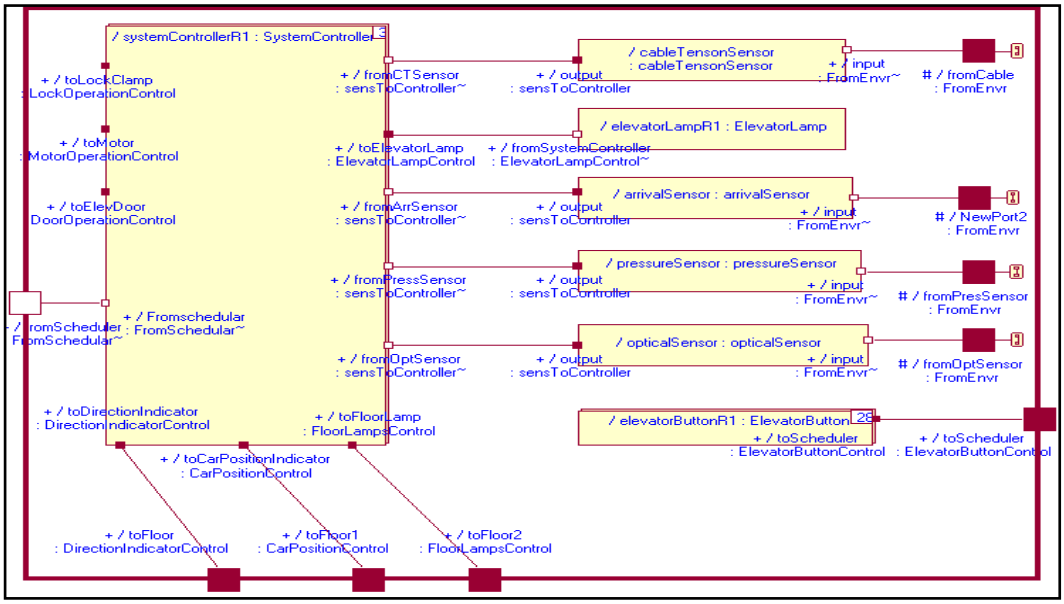

Figure 4.7: *Elevator* capsule structure diagram

UML-RT design model allows designers to determine some common scheduling notation such as absolute, relative deadline, worst case execution time and event priorities (Akhlaki et al., 2006). With respect to the timing issues, timing and log ports are available services provided by run time services (RTS) library in Rational Rose Real-Time tool. Timing ports serve timing facilities to the designer by providing a relative or absolute time (Rational Software Corporation, 2003). If timeout occurs, the capsule instance receives a transition with the timeout signal defined. As far the log port, it provides recording of messages and other logging services but it is not considered in this research. It was used to log system occurrence. As seen in Figure 4.5, *SystemController* capsule has *timer* and *log* ports, with protected visibility and connector type unwired in the structure.

4.3 Mapping UML-RT into UML-SPT – Structural Elements

In order to design the Elevator System case study using UML-SPT model's elements, significant UML-RT elements have to be mapped into UML-SPT meta-classes. In this research, since UML-SPT supports UML 2.0 meta-model, UML-RT

essential elements are mapped into UML 2.0 meta-model. Table 4.5 shows detailed description of the meta-classes used in UML 2.0.

Table 4.5: Mapping of UML-RT elements into UML-SPT

UML-RT Stereotypes	UML-SPT(UML2.0) Meta-classes
Capsules	Active Class
Port	Port (in structured class) Class (in base UML class)
Protocol	Interface
Class	Class
Signal	Signal / Event
Connector	Association / Link

UML-RT Capsules are mapped into the UML 2.0 as an active class (object) as illustrated in Table 4.5. As known, capsules are fundamental elements of UML-RT. Moreover, they can have operations and attributes like active classes. Both of them also have their own control of thread, and the capsule maps to the active class. Another important point is protocol, which is a special contract between capsules to send and receive the signals as defined. In UML 2.0, the protocol is mapped to interface that provides a contract. However, in the UML-SPT design as shown in Figure 4.8, the interface was used to provide operations to the active classes.

In this research, composite classes were used instead of the structured classes to model the system in UML-SPT design. Thus, some UML 2.0 model's features were not used in UML-SPT design such as interface and port. With respect to the signal, UML 2.0 provides event semantic in order to map. Furthermore in UML-RT, connector provides interconnection between related capsules' ports. In UML 2.0, link element fulfils similar work. As shown in Figure 4.9, it is used to connect between objects in the system and similarly for class in UML-RT, the same entity can be used to map into UML 2.0. As shown in Figures 4.10 and 4.11, classes can be represented as a set of objects that has similar semantics and constraints.

In Rhapsody design tool, Object Model Diagram (OMD) focus mostly on the specification of classes. The name of this diagram leads to misunderstanding of class concept. OMD specifies the static structure of RTS and relationships of the classes in the system as a class diagram (I-Logix, 2004). Moreover, this diagram can also be used as an object diagram to specify objects, and its relationships. As mentioned previously in UML-RT (RoseRT tool), the container capsule is designed in UML-SPT (Rhapsody tool) as a composite class. Which is a container class that consists of instance of other classes called *parts*. After creating all classes used by system, the relevant classes can be placed into corresponding container class by drag and drop. At this time, Rhapsody design tool automatically creates a composition relationship in the background. In this sense, reducing of complexity occur by hiding those relationships. For this reason, it makes RTS design less complex, understandable and neat for designer.

4.4 Designing Structural Aspect of the Case Study using UML-SPT

ElevatorSystem was designed by using UML-SPT constructs based on UML 2.0 concepts (OMG, 2005) (OMG, 2007) along with the same requirements for design of UML-RT. During the design phase, the construction of UML 2.0 model was used such as objects, classes, associations and so on. As seen in Figure 4.8 that shows UML-SPT design diagrams, *ElevatorSystem* has two composite classes; container classes that cover other objects and relationships within it such as *Elevator* and *Floor* composite classes. *Elevator* composite class has the ten (10) contained objects instances that are called *parts* in the design such as buttons, sensors, doors, motor, and lock clamps (Figure 4.9). This shows quite a similarity to the UML-RT top capsule (container capsule). As such, in UML-RT and UML-SPT, the creation of the composite class results in the creation of all of its parts and vice versa.

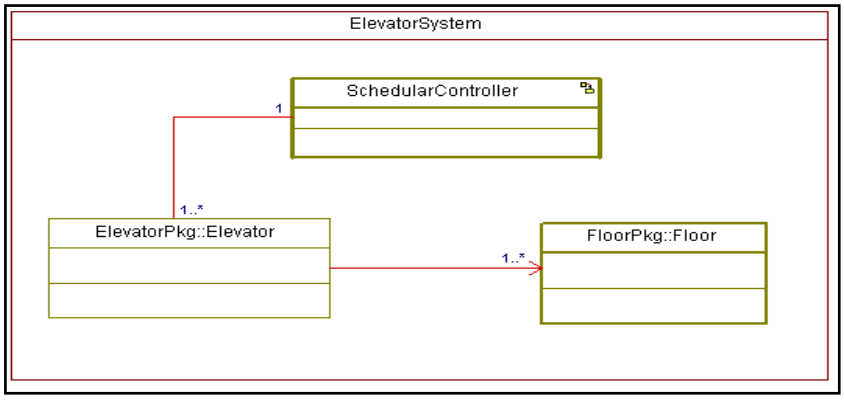

Figure 4.8: *ElevatorSystem* class diagram in UML-SPT

Essentially, in the UML-SPT class diagram as shown in Figure 4.9, *ElevatorSystem* class has composite relationships with all its parts and however, these relationships were hidden by using drag and drop features. As a result, from the comparison on *ElevatorSystem* class diagram of UML-RT (refer to Figure 4.3) and UML-SPT diagrams (refer to Figure 4.9), UML-SPT seems neat and less complex. Also, the design has a passive object that is *itsElevatorSystemInterface* instance that provides a service to the active objects. A passive object in the system acts as a server since they offer services to the causal objects.

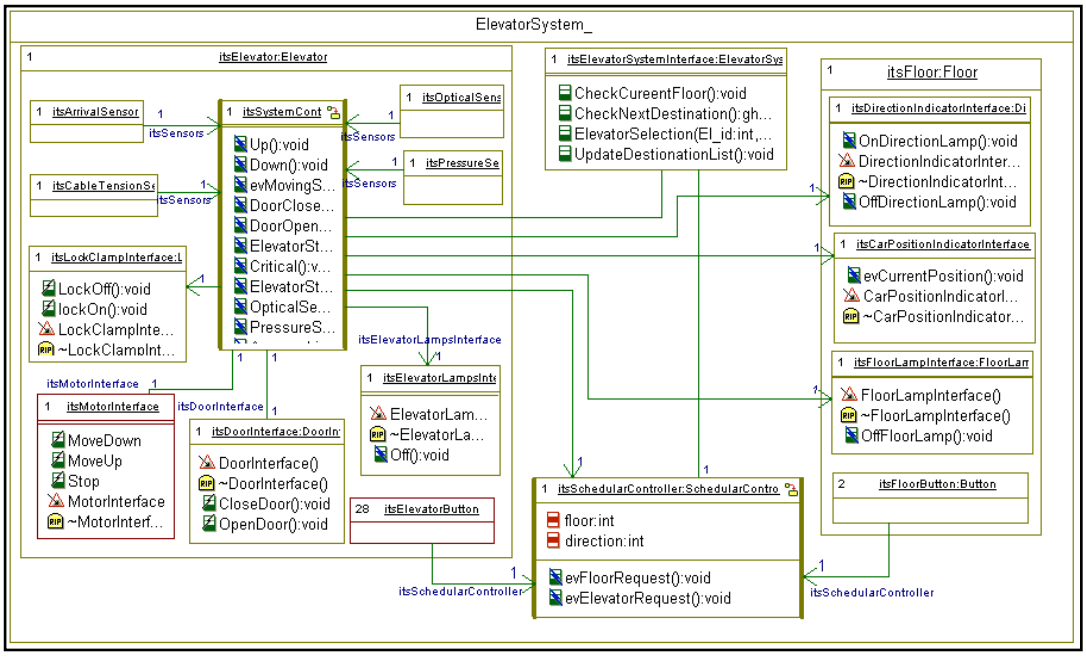

Figure 4.9: *ElevatorSystem* class object model diagram in UML-SPT

The Elevator System works similarly with the UML-RT design model. Figure 4.10 shows another object model diagram (OMD) of *Elevator* object designed in UML-SPT and all relationships and directions between object instances were shown explicitly in this diagram by using arrow labels as an instance of association. In addition in this diagram, functions and events are shown in compartments of the object instances. For example, *itsElevatorDoor* object is instance of the *ElevatorDoor* class and has four (4) operations that are *Door()*, *~Door()*, *open()* and *close()*, with two events received that are *evOpen* and *evClose.*

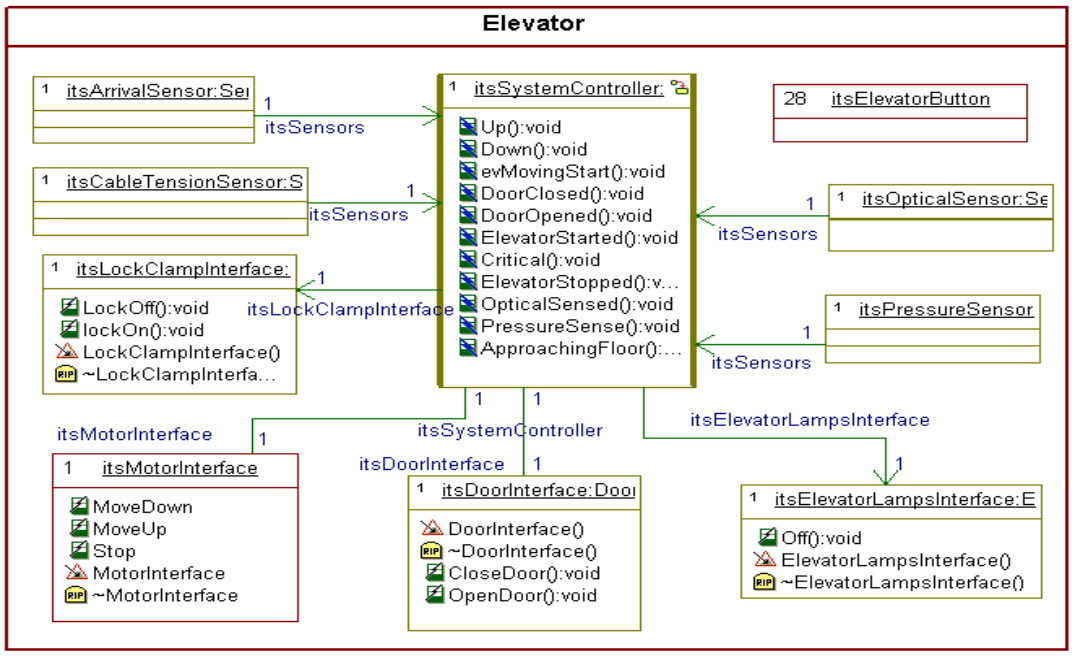

Figure 4.10: Elevator object OMD in UML-SPT

4.5 Designing of Statechart Diagram (UML-RT and UML-SPT)

Statechart diagram illustrates a reactive system that runs, and this diagram consists of several elements such as states, transitions, actions, events. A *state* represents a phase of object's behaviour and it is drawn as a rectangle. In other words, state is a condition of an object (Douglass, 2001). Occasionally a state may also have a very complex behaviour. In this case, nested states or otherwise called substates are used to make complex behaviour easier. These substates are enclosed

in a composite state; the illustration is shown in UML-SPT and UML-RT statechart diagrams as depicted in Figure 4.11 and in Figure 4.12 respectively.

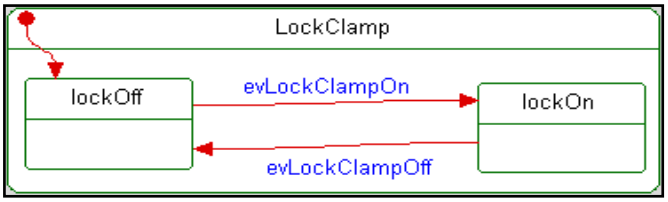

Figure 4.11: *LockClamp* Statechart Diagram and its substates in UML-SPT

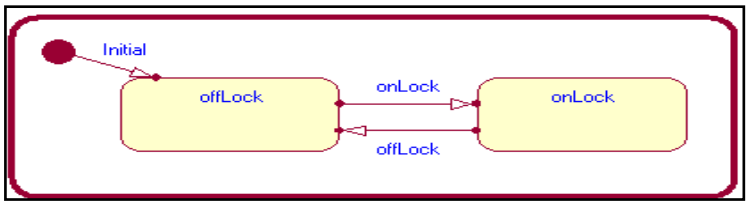

Figure 4.12: *LockClamp* Statechart Diagram and its substates in UML-RT

Transition is a path of process from one state to another. The path can be determined as the relationship between the source and destination states. A transition can be associated with just one single message. In other case, it might consist of three main elements that are events, guards and actions but these features are optional. The transition common form can be depicted in UML-SPT statechart diagram as shown in Figure 4.13.

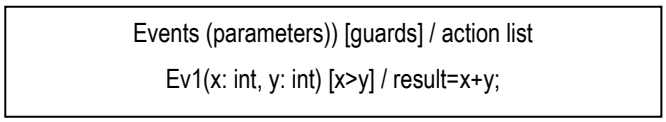

Figure 4.13: Transition, guards and actions

Message is used as a form of communication between objects. When two objects need to communicate with each other, messages are passed from one to another object. *Action* is an executable simple computation (Selic et al., 1994). In other words, action can also be considered as the sending of messages to the

receiver object as the requirements of sender objects. Action is also known as basic unit of executable functionality. An execution of action aids to change from current system state to another. As soon as it is taken, it cannot be interrupted by another event even though the coming events have higher priority.

Once an *event* is sent to the receiver, a *transition* is triggered. The events can be signal or call events. Once the destination object receives the events, the transition taken causes the object's condition (state) to change. Figure 4.14 shows that, in UML-SPT, when *motor* object receives the event (evMove with direction parameter), the transition that has a similar name with evMove is triggered and after the evaluation of whether the motor must go to up or down depending on direction parameter, states change from stationary state to the up or down substates. While motor is working after 2 seconds, it sends event, *evAtFloor*, to the elevator object in order to check whether or not the motor has stopped.

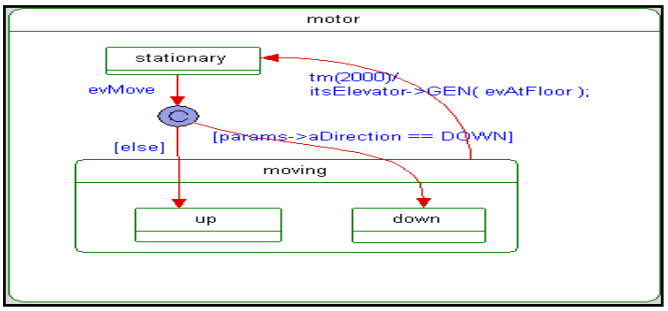

Figure 4.14: *Motor* object State Diagram in UML-SPT

The same situation was designed by using UML-RT model (RoseRT) as seen in Figure 4.15.

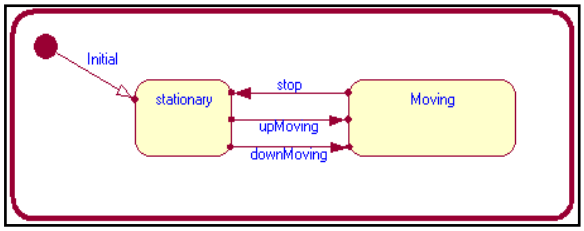

Figure 4.15: *Motor* object State Diagram in UML-RT

i. Orthogonal State

Orthogonal State is a composite state and covered substate. It is also considered that each state of region is an or-state. When the object receives an event (OMG, 2005) each region state becomes active. Moreover, they receive a copy of that event and process the same event freely (Douglass, 2004). This concurrency illustrates a logical concurrency model. When an event is received, more than one transition can be fired based on RTC assumption that the state machine selects only one task to be processed at one point in time. The task is processed till completed, before another task is taken. As RTC steps are applied on the whole state machine, it can also be applied to each region in orthogonal state but this approach is costly and difficult (OMG, 2007). In Elevator System case study, System Controller Statechart Diagram is an UML-SPT orthogonal state diagram as shown in Figure 4.16.

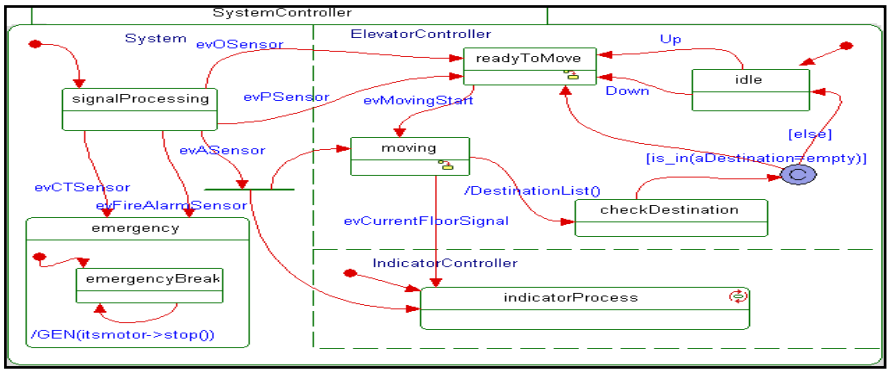

Figure 4.16: Elevator *SystemController* Statechart Diagram in UML-SPT

As shown in Figure 4.16, main state consists of orthogonal states that are elevator controller and indicator controller substates along with or-states that are *emergency* and *signalProcessing* states. For instance, when object receives *up* event, the transaction is directed to the *readyToMove* state that is also a composite state. Besides, as previously mentioned, UML-RT was derived from ROOM model. According to ROOM model, events occurrence is interpreted by two ways in FSM that are the Mealy and Moore approaches. It has been said that they do not support orthogonal state or and-state. As a result, in UML-RT it can be shown that each

region's state must be combined together as a specific state. However, this way can probably increase the number of states. The following Figure 4.17 shows Elevator *SystemController* Statechart in UML-RT model.

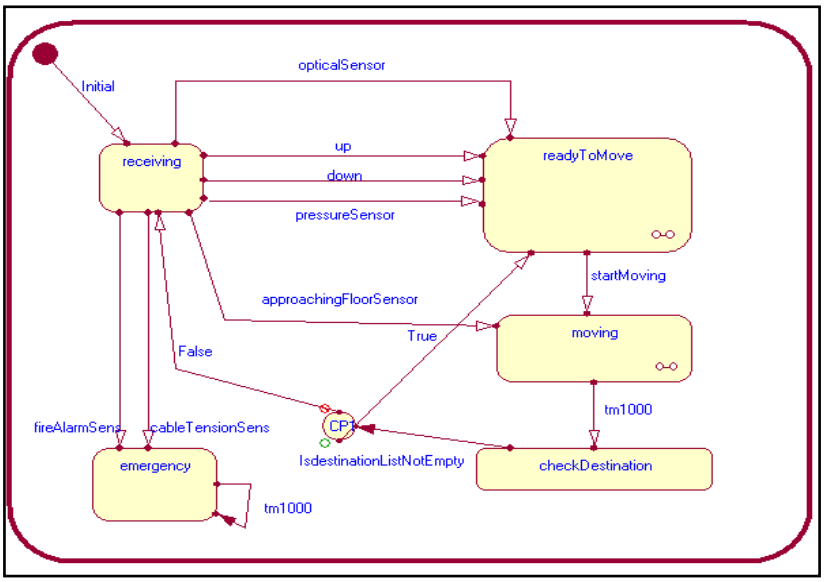

Figure 4.17: Elevator *SystemController* statechart diagram in UML-RT

As seen in Figure 4.18, when this state takes the up transition (named same as the event name), substates of *doorOperation* state is entered by default. While the statemachine executes the activities and if *evOsensor* (received from optical sensor) or *evPSensor* (received from pressure sensor) is received by the object at any time, the substates of *doorOperation* state that is the *initial* state is entered and it is responsible for restarting the door open timer again, once both sensors sends these signals as shown in Figure 4.18.

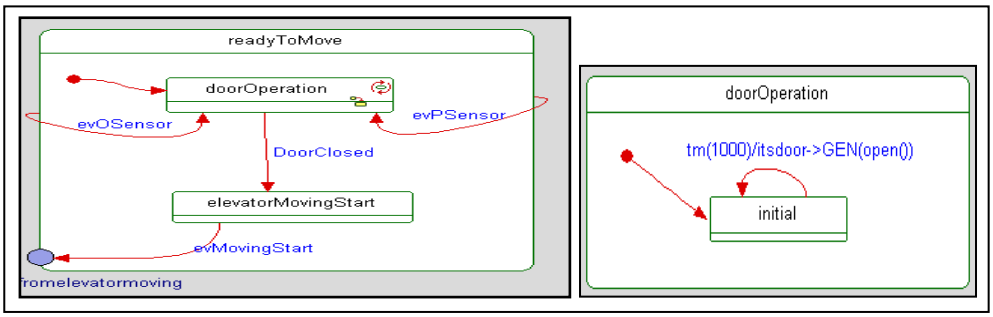

Figure 4.18: UML-SPT state diagram of *readyToMove* and *doorOperation* state

The same situation was designed by using UML-RT model (RoseRT) as seen in Figure 4.19.

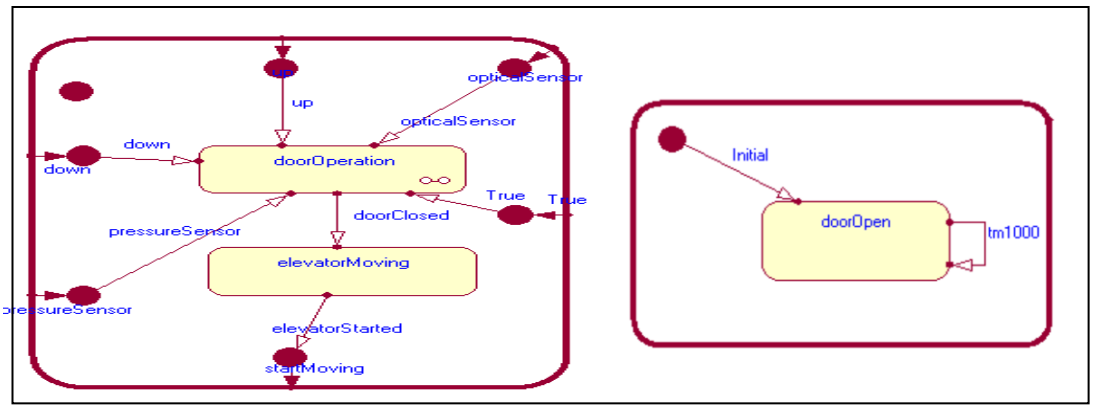

Figure 4.19: UML-RT statechart of *readyToMove* and *doorOperation* states

ii. Fork and Join

Fork used in orthogonal state to enter two states from one single transition. It can be used to bypass the transition to the desired state in the orthogonal state. However, it cannot be used in UML-RT because UML-RT does not support any concurrency occurrence in statechart. Moreover, UML-SPT and UML have some differences because of the use of *fork* construct. For instance, UML allows branch transition to execute its own action but UML-SPT (Rhapsody) does not allow this feature (I-Logix, 2004).

In contrast with fork, *join* construct is used to aggregate two similar transitions into one single transition in orthogonal state. Moreover, there are some differences between UML models in using join construct; the UML models referred here are UML and UML-SPT (Rhapsody 6.0). Each transition is not allowed to take an individual action in UML-SPT to enter into a *join* construct. In Elevator System case study, *join* construct can be shown in *SystemController* composite state in UML-SPT as shown in Figure 4.16. For example, if it was assumed that pressure and optical sensors send the same signal to controller, the join construct

can be used to join two sensor signals into one signal as they would execute the same action, resulting in the opening of the door.

In Elevator System case study, there is the arrival sensor that sends *evASensor* event with three parameters that are elevator id (el_id), level of floor (floorId) and direction of an elevator (direction) to SystemController. Once *evASensor* is received by *SystemController,* it is separated to two branches of transitions to affect the orthogonal state region. As shown in Figure 5.6, *evASensor* event directly affects the moving state and indicator processing state by using fork construct. In moving state, *evASensor* event is used to check whether or not an elevator reached the destination floor; in indicator processing state, *evASensor* event triggers transition to execute showing the current direction, floor and which elevator at the same time. In Figure 4.17, the *indicatorProcess* state is not used in UML-RT model; however the information that includes the elevator current location is sent to the *Elevator* indicator by an action in the appropriate state. *SystemController* object processing is almost the same as UML-SPT model after the object receives any signals.

As for *SchedullerController* state diagram, by designing Rhapsody 6.0 as seen in Figure 4.20, it receives two events *evFloorRequest* inside elevators and *evElevatorRequest* from floor and never changes its state's situation as it only executes events. In addition, when *SchedullerController* receives *evElevatorRequest* event from floor buttons with parameters it follows the steps below:

a) Extracts event and gets parameters to select corresponding elevator
b) Evokes *ElevatorSelection*() operation with parameter from *ElevatorSystem* Interface class to select elevator
c) Sends up or down event related to request to the *SystemController*
d) Evokes *ElevatorStatus*() operation with selected elevator id in order to check whether elevator is idle or not
e) Evokes *UpdateDestinationList*() operation to add floor to be visited next

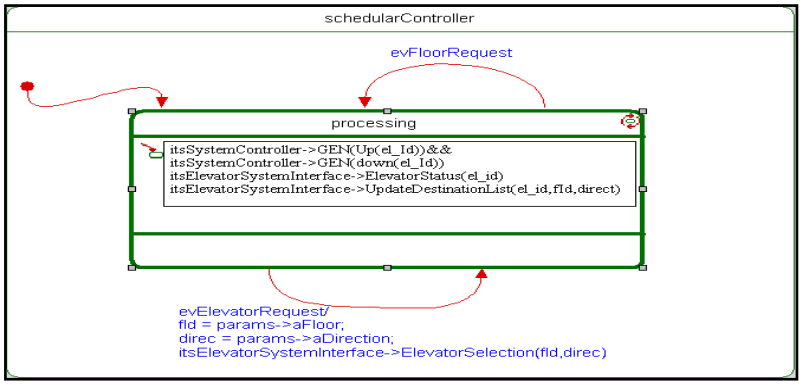

Figure 4.20: *SchedullerController* state diagram in UML-SPT

Figure 4.21 also shows *SchedullerController* statechart in UML-RT but the same steps have been placed in the active state entry actions area. As mentioned earlier, the codes were not considered during design as they were implemented merely to show how *SchedullerController* handles these events.

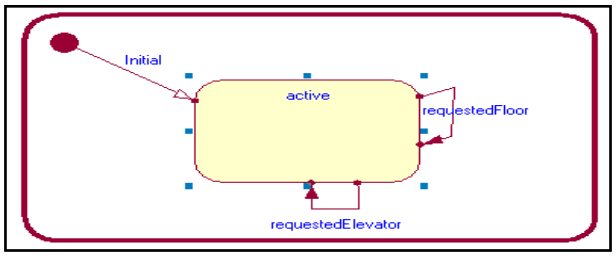

Figure 4.21: *SchedullerController* statechart diagram in UML-RT

4.5.1 Comparison of Statechart Diagram Design

In this section, the comparison criteria were chosen by referring some literature and based on experience of evaluating the case study. The closest researches were done by Glinz (2002), Harel and Gery (1997) and Douglass (2001) discussed on the behavioural aspect of UML by choosing statechart diagrams. According to those papers, the diagram's features can be used as criteria to show its ability regarding to non-functional requirements of RTS such as concurrency and timeliness.

The discussion on both UML design model features in statechart diagram is concluded in this section. As shown in Table 4.6 below, the main purpose in this research is to design non-functional requirements of RTS completely by using a suitable UML design model such as concurrency, and timeliness.

In real time, there are a lot of possibilities for the tasks to occur at the same time. This is called concurrency. Since real-time systems' structures are very complex and activities happen simultaneously, concurrency issues must be taken into account in order to provide reliable real-time systems that provide timely response. In UML models, concurrency criterion is shown in statechart by using and-state (orthogonal state), fork and join elements. As mentioned earlier, the object can be in one state when it receives events from the environment at a time based on RTC assumption, called as state as or-state but under some circumstance especially in real- time system, the object must be more than one state at the same time to process all activities simultaneously. Furthermore, an orthogonal state is separated by dash-line to depict concurrency and each separated part is called region as shown in Figure 5.4. In addition, each region operates the activities independently like other regions if they are concurrent (Douglass, 2004).

In Table 4.6, it is seen that UML-RT statechart does not allow the designer to design concurrent events. However, in UML-SPT, concurrency can be drawn easily by using *and-states* (orthogonal). Moreover, fork and join constructs make complex and large designs of real-time systems easier and neater. As a result, it can be concluded that UML-SPT can be used to design complex and large real-time systems.

With respect to the timing issues, UML-SPT is more focused on timeliness rather than UML-RT. As mentioned previously, the RTTime modelling sub-package provides some stereotypes to capture timing issues. By using this profile, time values and constraints can be expressed in the UML-SPT diagrams such as <<RTtimer>>, <<RTdelay>>, <<RTclock>> and so on. However, UML-RT provides only timing services in order to define absolute and relative time of messages.

Table 4.6: Comparison of UML-RT and UML-SPT features (statechart)

Statechart Features	**UML-SPT Statechart (Rhapsody)**	**UML-RT Statechart (Rational Rose RT)**
Concurrency	Supported	Not Supported
• ***And-State(Orthogonal)***	Supported	Not Supported
• ***Fork & Join***	Supported	Not Supported
Timeliness	Support all timing service	Support only defining relative or absolute deadline

4.6 Designing of Sequence Diagram (UML-RT and UML-SPT)

Sequences diagram is the most important type of the object interaction diagram and it allows the designers to analyze how the object instances do their job within the structure of the system, and to see clearly the events that used by the objects based on specific scenario. In addition, sequences diagram represents messages exchanging between instances of the objects that are defined in the system. Moreover, it is used more than communication or timing diagram because it shows the sequences of messages in the diagram (Booch et al., 2005). Thus, it is easy to recognize which message will be the next message. In addition, it can clearly analyse the interaction of the objects by depicting messages that were sent and received by each other over time in the sequence diagram. Moreover, this diagram consists of some basic elements that are frames, regions, object's roles, lifelines, arrows (for showing message's type synchronous or asynchronous), messages and calls (for showing operations).

Classifier's roles (instance) are fundamental elements that are depicted by vertical line (lifeline) labelled with instance of class name in the both profiles' sequences diagrams. For example, some instances of class names of UML-RT are shown in Figure 5.12 such as *Elevator*, *SystemController*. These roles may be the parts of a composite, structured class, component or subsystem. Furthermore, the dashed vertical lines that descend from the object box represent the existence of the object at a particular time. When an object is created or destroyed, then its lifeline starts or stops at the appropriate point.

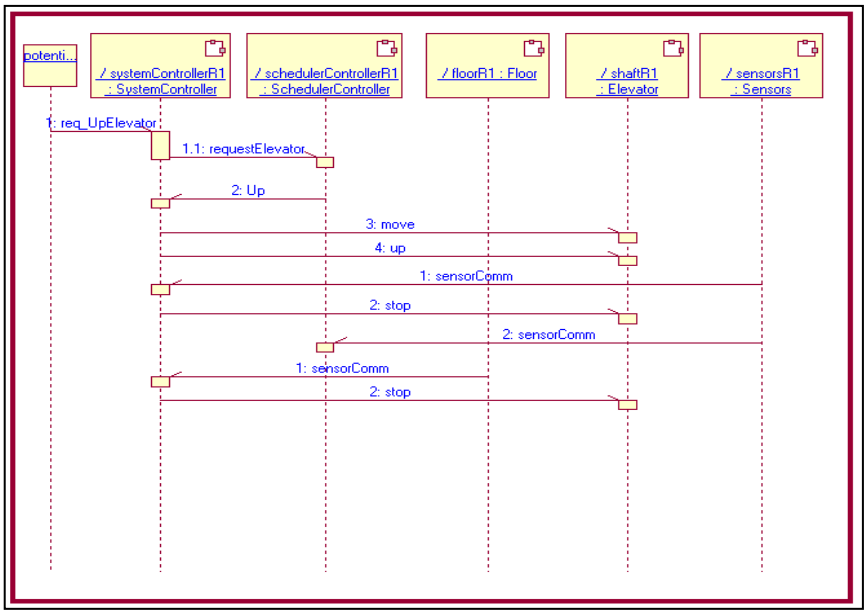

Figure 5.12: Instances of classes in UML-RT sequence diagram

In addition, *messages* are depicted between instance's lifelines by using arrows. Such arrows with filled arrowheads are used to represent synchronous call messages while open arrowheads are used to represent asynchronous messages. With respect to the call messages, replies to these messages are shown in dash line arrows with open arrowheads. In addition, replying messages can be shown as a return value such as true or ACK. The sequence diagram as shown in Figure 5.13 represents the sequences of events when the potential passenger presses the floor button in order to request an elevator to move up to the floors above. This diagram was designed using UML-SPT's sequence diagram (Rhapsody design tool) to show processing of request of elevator from floor. As shown in Figure 5.13, when potential passenger presses the floor button ('up' button), the button sends *evElevatorRequest* event with two parameters that include floor number and direction, named as *aFloor* and *aDirection* respectively to *SchedullerController* instance to be performed. The e*vElevatorRequest* event is an asynchronous message and is drawn on slant between two lifelines in Rhapsody tool while an asynchronous event such as *RequestedElevator*.

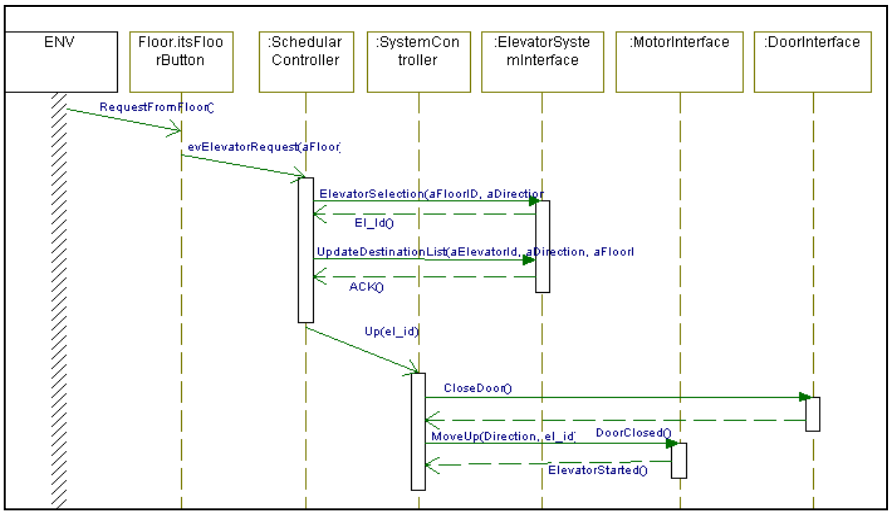

Figure 5.13: Sequence diagram for *Request Elevator* scenario in UML-SPT

Similarly, e*vElevatorRequest* can be drawn as horizontal line with half open arrowhead in UML-RT's sequence diagram (Rational Rose Real Time tool) as shown in Figure 5.14.

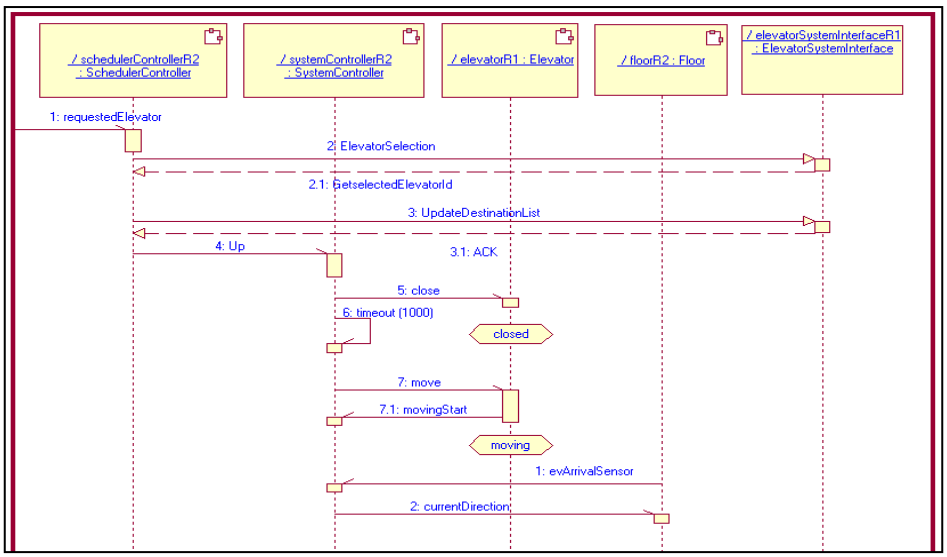

Figure 5.14: Sequence diagram for *Request Elevator* scenario in UML-RT

Another type of message is *call message* that is used to invoke a specific operation on an object such as *ElevatorSelection ()* operation as seen in Figure 5.13. In addition, this specific operation is invoked from *ElevatorSystemInterface* class that provides the operations for other objects or system as a server. There is a slight difference in representing an operation between design tools, where RoseRT

uses an arrow without filled arrowheads, while Rhapsody uses an arrow with filled arrowheads. This difference can be observed in UML-SPT (refer to Figure 5.13) and in UML-RT (refer to Figure 5.14). Regarding the action of invoking an operation, results of the operation can be shown in the sequence diagram by using dash line arrows. For example, after the execution of *ElevatorSelection ()* operation, the particular elevator is chosen to move to the requested floor and the selected elevator 'Id' is sent as a result to *SystemController*. This case is shown in both diagrams with dash line arrows.

Regarding to *fragment* feature, sequence diagram displays only one flow of events in the system; hence it may be necessary to create more than one sequence diagrams in order to show the entire events of the system. However, UML 2.0 has several new features for sequences diagrams that may have many interaction fragments such as loops, parallels, alternatives, branches, options and so on. The following information introduces certain significant interaction fragments briefly:

- *References* – indicated with 'ref' on sequence diagram and express reference to the other sequence diagrams.
- *Alternative* – indicated with 'alt' on sequence diagram and supports alternative fragments where only one branch is taken, depending on evaluation of guard.
- *Parallel* – indicated with 'par' and defines concurrent or parallel regions that occur in sequence diagram.
- *Loop* - indicates the number of iterations.

In this research, Rhapsody 6.0 tool was used to show this UML-SPT's semantics however, Rhapsody 6.0 has some limitations in using these operators, as it provides only the *ref* operator as shown in Figure 5.15.

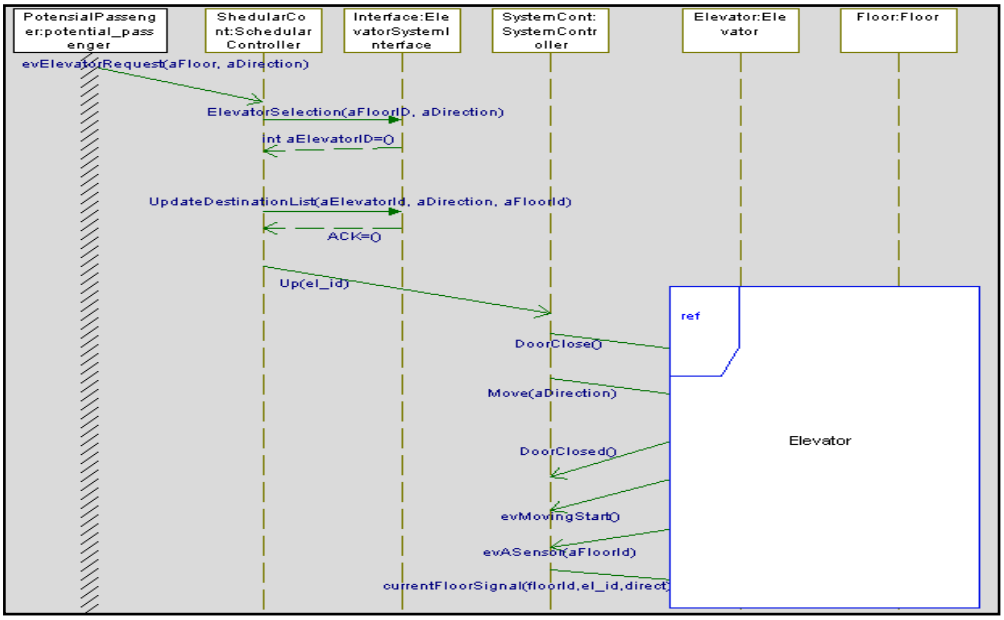

Figure 5.15: Sequence diagram for *Request Elevator* and *Stop at Floor* use cases scenario in UML-SPT

Therefore, this tool only supports ‘reference’ interaction fragment as shown in Figure 5.15 and inside the ‘reference’ interaction fragment is shown in Figure 5.16.

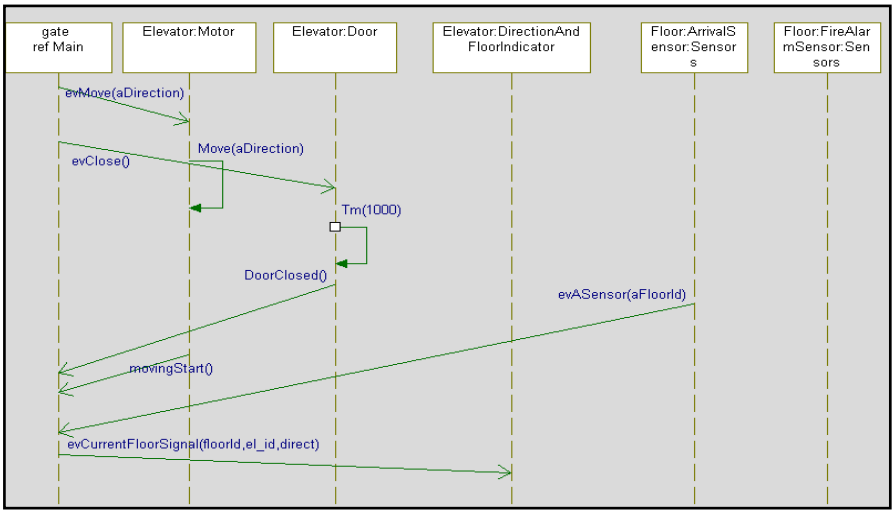

Figure 5.16: Inside view of *Elevator reference* fragment

UML-RT (RoseRT tool) is based on UML 1.4 semantics for sequence diagrams while, UML-SPT (Rhapsody 6.0) is based on UML 2.0 core. In this case, there are some differences in semantics between two models; it means UML 2.0

has been extended at some point, such as for capturing several scenarios in sequences diagrams (par, ref, loop, alt operator and so on). UML-RT (RoseRT tool) does not support these operators at all because it uses UML 1.4 semantics hence, the fragments cannot be used, and the same scenario is depicted in UML-RT sequence diagram as shown in Figure 5.17. In addition, RoseRT tool supports "*Local action*" that indicates an important action or operation that is fulfilled by object instances at that time. As seen in Figure 5.17, log operation local action was used to show how and when *Elevator* object stores data in the system. However, Rhapsody 6.0 does not have this feature.

Co-regions are used to indicate a set of undefined ordering events sent or received by objects' instances. Moreover, they are shown with a thick black horizontal line on the instance's lifelines as shown in Figure 5.17. In the Elevator System case study, there are some undefined ordering messages from sensors that send events at any time when they sense something from their environment. For example, it was previously mentioned in the system that Elevator door has two sensors that are pressure and optical sensors between the doors; when doors attempt to close, if any obstacle detected by the sensors between the doors this would evoke them to send a signal to the *SystemController*. These cases were shown in UML-RT sequence diagram by using co-region concept however, Rhapsody 6.0 tool does not support this feature.

Another useful feature of the RoseRT tool (UML-RT) is that it allows designers to describe states as statechart diagrams through the object instance's lifelines. By using this feature, sequence and statechart diagrams can be combined in one diagram, where states can be located by using rounded rectangles on the lifelines in sequence diagram, as shown in Figure 5.17.

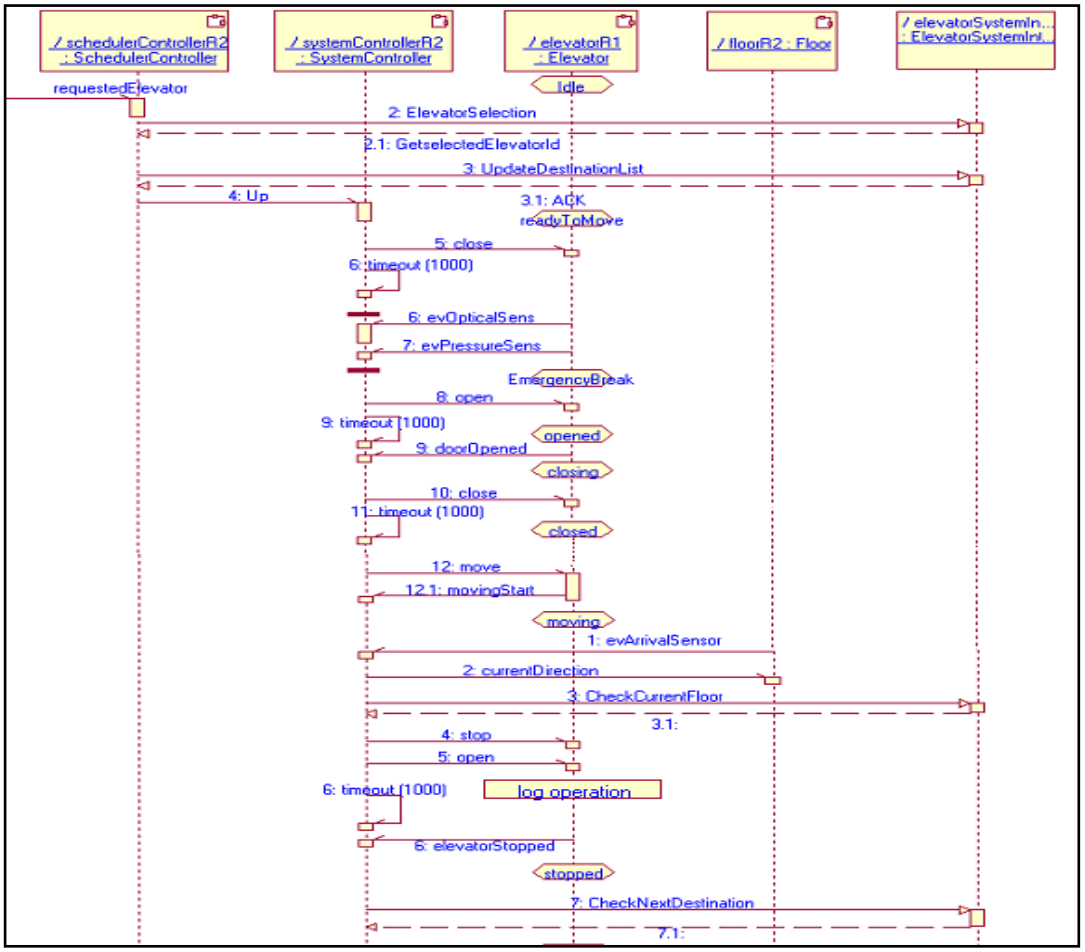

Figure 5.17: Sequence diagram for *Request Elevator* and *Stop at Floor* use cases scenario in UML-RT

Consequently, according to the experience of designing Elevator System sequence diagrams, some difficulties and easiness were experienced. With respect to the concurrency and timing issue, UML-SPT sequence diagram is very useful for real-time designers because it supplies non-functional constraints as shown in Figure 5.18. UML-SPT seems like an annotation model for RTS designers, whereby it defines a set of stereotypes and tags to model schedulability, time and performance for RTS. These features are discussed in detail in the next chapter. However, UML-RT does not support these features at all. This is the biggest disadvantage of using UML-RT for designers.

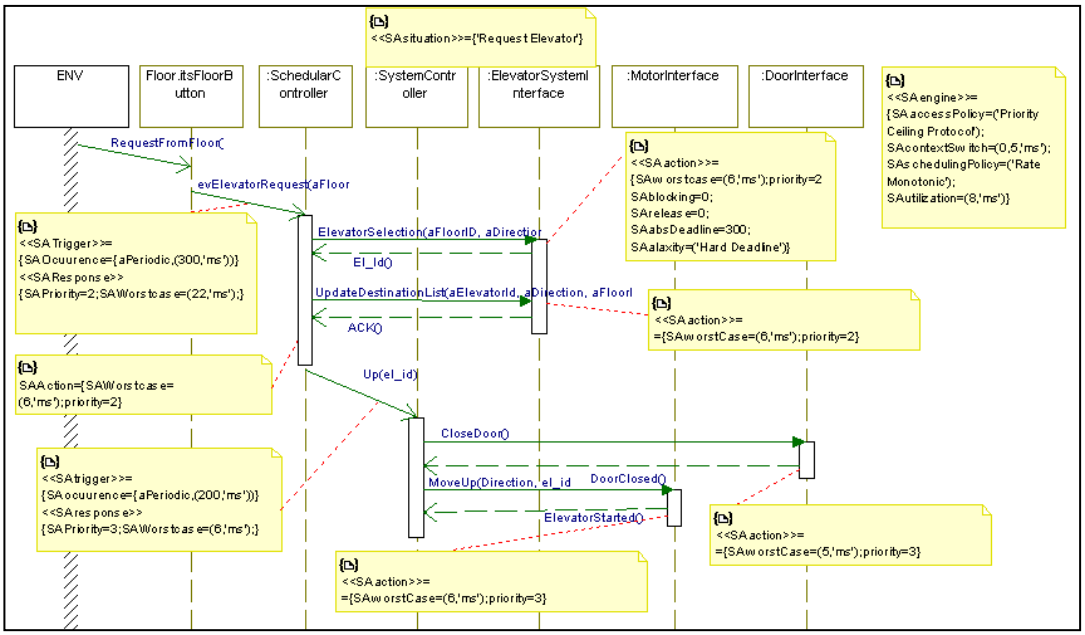

Figure 5.18: UML-SPT annotations in sequence diagram

4.6.1 Comparison for Sequence Diagram Design

As mentioned earlier, this research aim is to compare two UML design models based on real-time requirements and it is more focused on its non-functional requirements such concurrency and timeliness.

In this sense, *concurrency* criterion is very important for RTS and it can be shown in sequence diagrams by using *parallel interaction fragment* (par). However, based on what has been implemented during this phase, Rhapsody 6.0 and RoseRT tool unfortunately does not support parallel flows in the sequence diagram.

Another important criterion is the model must be able to *annotate the non-functional requirements* of the real-time system. UML-SPT has a huge number of annotations to annotate time constraint and concurrency issues. These requirements can be shown by using text or notes in UML-RT. And finally is the *timing issues*, it is a significant issue in RTS. For this purpose, UML-SPT package provides many

stereotypes and annotation to capture the timeliness requirement. However, UML-RT does not support any timeliness requirement in sequence diagram at all.

As shown in Table 4.7, UML-SPT is a suitable framework to model time and concurrency issues by providing stereotypes, tagged values and constraints that can be used to annotate corresponding UML diagrams to show quantitative values, which UML-RT does not support.

Table 4.7: Comparison of UML sequences diagrams

Criteria	**UML-SPT**	**UML-RT**
Concurrency	- Supports concurrency	- Supports concurrency
Non-functional Requirements annotations	- Provides more annotations to indicate non-functional requirement in its diagrams	- Does not have annotations to indicate non-functional requirement. ** Text can be used as alternative.*
Timing issues	- RTTime subpackage support to specify timing constraints. - Directly focuses on timeliness, timing annotations can be shown.	- There is only timing port to define absolute and relative time. - Only timeout can be shown