Automated Analysis of Unified Modeling Language (UML) Specifications

By

Meyer C. Tanuan

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2001

© Meyer C. Tanuan 2001

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Unified Modeling Language (UML) is a standard language adopted by the Object Management Group (OMG) for writing object-oriented (OO) descriptions of software systems. UML allows the analyst to add class-level and system-level constraints. However, UML does not describe how to check the correctness of these constraints.

Recent studies have shown that Symbolic Model Checking can effectively verify large software specifications. In this thesis, we investigate how to use model checking to verify constraints of UML specifications. We describe the process of specifying, translating and verifying UML specifications for an elevator example. We use the Cadence Symbolic Model Verifier (SMV) to verify the system properties. We demonstrate how to write a UML specification that can be easily translated to SMV. We propose a set of rules and guidelines to translate UML specifications to SMV, and then use these to translate a non-trivial UML elevator specification to SMV. We look at errors detected throughout the specification, translation and verification process, to see how well they reveal errors, ambiguities and omissions in the user requirements.

Acknowledgements

I would like to thank my supervisor, Dr. Joanne M. Atlee, for her valuable comments and insightful suggestions. I am very grateful for her encouragement and guidance throughout this work. To Professor Dan Berry, I thank him for pointing out that some of the errors revealed during model checking were requirements-level errors. I also thank my thesis reader, Dr. Nancy Day, who provided a thorough review of my thesis. I highly appreciate her detailed and helpful comments. Finally, I thank Doug Guderian for acting as our domain expert for the elevator case study.

To my wife,

Marrieta,

and sons,

Mayer and

Marko

Contents

1 Introduction 1

1.1 Model Checking 2

1.2 Computer-Aided Verification of UML Models 3

1.3 UML to SMV Model Checking Process 6

1.4 Main Contribution 9

2 Modeling Reactive Systems Using UML 11

2.1 Requirements Analysis 12

2.1.1 Three-floor Elevator System Problem Statement 12

2.1.2 System Properties 14

2.2 Defining Object Structure using UML 15

2.2.1 UML Class Diagram 15

2.2.2 UML Object Model 21

2.2.3 UML Constraints 23

2.3 Defining Object Behavior using UML 25

2.3.1 UML State Machine Semantics 25

2.3.2 Example Scenarios of Elevator State Machine 30

2.3.3 Inherited State Diagrams 33

2.3.4 UML Operations 35

3 Translating UML to SMV 39

3.1 Translation Overview 40

3.2 Translating the Elevator Example 43

3.2.1 Translating the Button Class 43

3.2.2 Translating the RequestBn Subclass 48

3.2.3 Translating the External_Event Class 52

3.2.4 Translating Timer Class 55

3.2.5 Translating Door Class 58

3.2.6 Translating the Elevator Class 60

3.2.7 Putting it all together: Main SMV Module 65

3.3 Translation Summary 66

4 Verifying System Properties Using SMV 68

4.1 CTL Overview 68

4.2 Verifying the Elevator System Properties 70

4.2.1 Safety Property for Elevator 70

4.2.2 Liveness Properties for Elevator and Hall Buttons 70

4.2.3 Additional System Properties 72

4.3 Model Checking Partial Specifications 74

5 Effectiveness of Model Checking 77

5.1 Time Spent per Activity 78

5.2 Requirements-level Errors 81

5.3 UML Model Style Defects 84

5.4 Object-oriented Modeling Errors 86

5.5 SMV Errors 88

5.6 Summary 90

6 Conclusions and Future Work 92

Appendix A: UML Notation Summary 94

A1. UML Class Diagram Notation 94

A2. UML State Diagram Notation 97

Appendix B: Elevator UML Specification 99

Appendix C: Elevator SMV Program 107

Appendix D. Additional Elevator Features 121

References 122

List of Figures

Figure 1: Finite state model checking 2

Figure 2: UML to SMV model checking process 7

Figure 3: Two-passenger scenario sequence diagram 13

Figure 4: Three-floor elevator main class diagram in UML 16

Figure 5: Three-floor elevator object model in UML 22

Figure 6: System properties shown as UML constraints 23

Figure 7: Elevator state diagram 26

Figure 8: Door state diagram 28

Figure 9: Button state diagram and RequestBn state diagram 30

Figure 10: SMV module program structure 41

Figure 11: Button SMV module 44

Figure 12 : RequestBn SMV module 49

Figure 13 : External_Event SMV module 52

Figure 14: Timer state diagram 55

Figure 15: SET Event SMV module 56

Figure 16: Partial listing of Timer SMV module 57

Figure 17: Partial listing of Door SMV module 58

Figure 18: Partial listing of Elevator SMV module 62

Figure 19: Partial listing of `main` SMV module 65

Figure 20: Main SMV module for Timer class unit verification 75

List of Tables

Table 1: getNextDest operation modeled as a decision table 36

Table 2: Macros for going to floor 1 with direction up 37

Table 3: SMV program structure / rules mapping 66

Table 4: Naming convention for SMV modules, variables and macros 67

Table 5: Requirements-level and OO modeling defects summary 80

1 Introduction

Object oriented (OO) modeling started in the late 1980s, modeling the concepts of object oriented programming at a higher level of abstraction. Several OO methodologists, spearheaded by G. Booch, J. Rumbaugh, and I. Jacobson, joined efforts to combine the best practices of various OO modeling approaches into a single method called the Unified Modeling Language (UML) [UML99]. On November 17, 1997, the Object Management Group (OMG) adopted the UML as a standard for specifying, visualizing, constructing and documenting artifacts of software systems. Since then, UML has gained popularity for providing software practitioners with a set of standard symbols and semantics to effectively communicate and produce predictable, repeatable results. The popularity of UML is evidenced in the number of books, conferences, seminars, reports, papers and software tools available in the commercial market today.

UML has become the most popular method for OO modeling. Most UML specifications are descriptive and easy to understand. However, UML does not provide a standard approach to verify correctness properties of the model. In this thesis, we translate UML object-oriented software models to Symbolic Model Verifier (SMV) [McMi93] models in order to verify the correctness of the UML model.

UML has a very rich set of diagrams, notations and well-formedness rules. However, it does not provide guidelines for writing clear and unambiguous object-oriented specifications that are amenable to model checking. In this thesis, we demonstrate that the use of Class Diagram, Object Model and State Diagrams are sufficient to translate into SMV for model checking. We introduce a set of guidelines in constructing well-formed unambiguous UML specifications that makes translation to SMV easy. We use UML constraints to describe the system properties informally. We provide a set of rules to translate the UML specifications including the UML constraints to SMV. We apply these guidelines to build a UML model for a three-floor elevator and apply these rules to translate the UML model to SMV. To complete the case study, we use SMV to perform model checking automatically.

Our aim is to make systematic model checking available for software developers who use UML to specify reactive systems. The guidelines and examples of translation from UML to SMV described in this thesis are aimed at helping software practitioners translate UML models to SMV easily. This approach has the potential to add model checking capability to commercial UML

tools such as Rational Rose (by Rational) and Rhapsody (by i-Logix). We view model checking as complementary to current simulation and testing capabilities of commercial UML tools.

1.1 Model Checking

Software verification techniques can be classified as *verification-based* or *refutation-based*. Theorem provers use the *verification approach* to find a proof for a given property, while model checkers use a *refutation approach* to refute the correctness property by finding a counter-example. With the aid of deductive verification using axioms and proof rules, theorem proving has the advantage of effectively verifying infinite state models. However, it requires a lot of expertise in Formal Methods. If and when a failed proof occurs, the theorem prover does not present a counter-example.

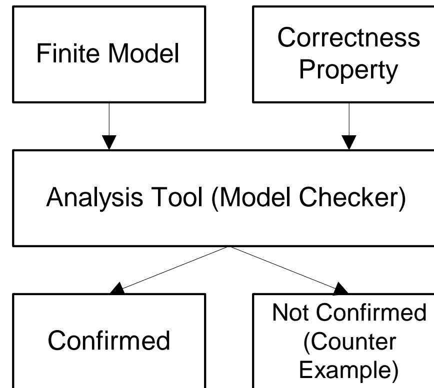


Figure 1: Finite state model checking

Figure 1 shows the process of finite-state model checking where a finite model is checked against a correctness property [CES86]. Both the finite model and the correctness property are expressed in the language of the analysis tool called a model checker. The model checker performs an exhaustive check and responds with confirmed or "not confirmed" with a counter-example. If a counter-example is found, either the finite model was incomplete or wrong, or the expected property was incorrectly specified or the property does not hold in the model specified. In this thesis, we selected model checking for automated analysis because it is more cost-effective to check the software system. Unlike theorem proving, model checking requires less expertise in the use of formal methods. As such, refutation-based tools such as model checkers are more promising for software practitioners.

Recent papers on automated software analysis [GlWe98] [JaRi00] describe how using an essential model (i.e., an abstract model of an important requirement or behavior) helps to make

model checking more practical. Model checking key properties of an essential model is known to incur much lower cost than verifying a complete model [Berr99]. In this thesis, we apply model checking on the essential model of the elevator case study.

1.2 Computer-Aided Verification of UML Models

Computer-aided verification of UML Models is currently an active area of research. Some researchers suggested verifying UML specifications using a theorem prover. Kamuller and Helke [KaHe00] translate UML object and state diagrams into Isabelle/HOL, and use the Isabelle interactive theorem prover to detect inconsistencies and check well-formedness. Guaspari and Naydich [GuNa98] describe how to model a UML specification consisting of state diagrams for a restricted class diagram (e.g., no inheritance) in PVS [ORS92], also an interactive theorem prover. Muthiayen [Muth00] proposes real-time extensions to UML and demonstrates how models written in this RTUML notation can be formalized as PVS axioms and theories and how safety and liveness properties can be verified using the PVS theorem prover. Using PVS, one can formally prove that the model is consistent with a property. However, the use of PVS for software verification requires more expertise in Formal Methods because writing theorems and proofs requires mathematical background and reasoning.

There are papers that described how to convert a UML model into a Promela specification [LiPa99][Bose99][McCh01]. Promela is the input language of the SPIN model checker [Holz97]. The SPIN software verification system, developed by AT&T, is based on an interleaving model of concurrency where only one component of the system's state is allowed to change at a time. The interleaving semantics supports a reduction algorithm based upon exploiting symmetries in the order of execution known as *partial order reduction* [Pele94]. This makes SPIN suitable for concurrent processes with uncoupled events such as network protocols. For most reactive systems where events are tightly coupled in distinct processes, SPIN's partial order reduction technique cannot be put to good use.

In contrast, tools that use synchronous models such as COSPAN [Kurs96] (by AT&T) and Symbolic Model Verifier (SMV) allow any number of components to execute at the same time. The SMV tool uses Binary Decision Diagrams (BDDs) to symbolically represent state information rather than explicitly representing each state. Since SMV represents the sets of states and transition relations as boolean formulas, model checking is based on formula manipulation and simplification. In this thesis, we decided to use SMV because it is well suited for reactive

systems and process control systems where components are tightly connected via synchronization messages and internal events.

Studies have shown that symbolic model checking can be successfully applied to large software systems [SrAt96][CAB+98]. Our work builds on previous work that uses SMV to perform automated verification. However, we differ by translating from UML models (instead of CoRE/SCR, RMSL, and Statechart models) to SMV models.

McUmber and Cheng [McCh01] investigated formalization of UML diagrams using formal target languages. Specifically, they demonstrated a framework for translating a subset of UML state diagrams and UML class diagram into Promela/SPIN based on a homomorphic mapping between UML and Promela metamodels. Our work is consistent with their homomorphic mapping approach because we preserve UML's class structure when translating from UML to SMV. Our work differs in that we are translating UML models into SMV and not UML models to Promela. We provide specific rules on how to map UML classes and attributes to SMV's modules and variables.

Our work also differs from the previous translations from UML to PVS and to Promela because the other translations concentrate on the UML state machine only. They use the state diagram for defining internal behavior of objects and translate the state diagram to a formal language, but not in the context of the class diagram. They provide no guidance in how to instantiate the state models associated with the various classes from their class diagram. In contrast, we examine and translate the major model elements of a UML class diagram. We discuss how to model and translate different types of associations including aggregation, composition and generalization (superclass / subclass) relationships. We also investigate how to translate operations and attributes defined in a class. For UML state diagrams, we show how to specify state models that adhere to the object-oriented principles and propose translation rules for events with arguments. Only a subset of UML's notation (i.e., the most common and important modeling elements) is considered in our case study.

In this thesis, we describe how to verify properties of UML specifications using SMV. We expect our guidelines for specifying a well-formed UML specification to be applicable to other non-SMV model checkers. Thus we have separated the UML modeling step from the translation and verification steps, to make it possible to use a different model checker to verify the same UML

specification. Using this approach, software practitioners can concentrate on UML modeling and use the SMV tool as a separate component to perform model checking.

1.3 UML to SMV Model Checking Process

In practice, the translation from UML to SMV is iterative. We specify the user requirements using UML and translate to SMV. Then, we perform model checking using SMV. UML models are updated and retranslated based on the result of the SMV model checking. The iteration is complete when all properties to be checked are satisfied (i.e., no counter-examples or no errors are detected). The complete UML to SMV model checking process is illustrated in Figure 2, where we use basic flowchart notation to describe the process. The UML to SMV model checking process consists of five major steps:

Step 1. Attempt to specify the user requirements

This is the requirements elicitation step, in which the stakeholders of the software system discover, reveal, articulate and understand their requirements. For simplicity, we assume that the outcome of this step is the problem statement and system properties that must hold[1]. The following steps help in specifying and refining the problem statement.

Step 2. Model the requirements in UML

Using all the relevant information provided by the requirements elicitation step, we build a UML class diagram, UML object model and for each class, a UML state diagram. The UML class diagram defines the system's object structure including inter-object associations such as composition, aggregation and generalization. We use the UML object model to represent the maximal instantiation of the class diagram. The object model is annotated with UML constraints that specify desired system properties such as "elevator never moves with its doors open". A UML state diagram defines the behavior of all objects of a specific class. We build the state diagrams that conform to well-formed rules to ensure consistent and intuitive translation to SMV for model checking. Guidelines and detailed examples are discussed in Chapter 2.

Step 3. Translate the UML model to SMV

In this step, we translate the classes and their state machines into SMV modules. Then, we translate the objects defined in the object model as instantiations of SMV modules. Details of

---

[1] In practice, the outcome of this step would include objectives, anticipated benefits, strategic and future considerations, constraints and assumptions, security, audit and control aspects of the software system.

translation rules are discussed in Chapter 3. We also translate the UML constraints defined in the UML object model into Computational Tree Logic (CTL) formulas, which is the property language for SMV.

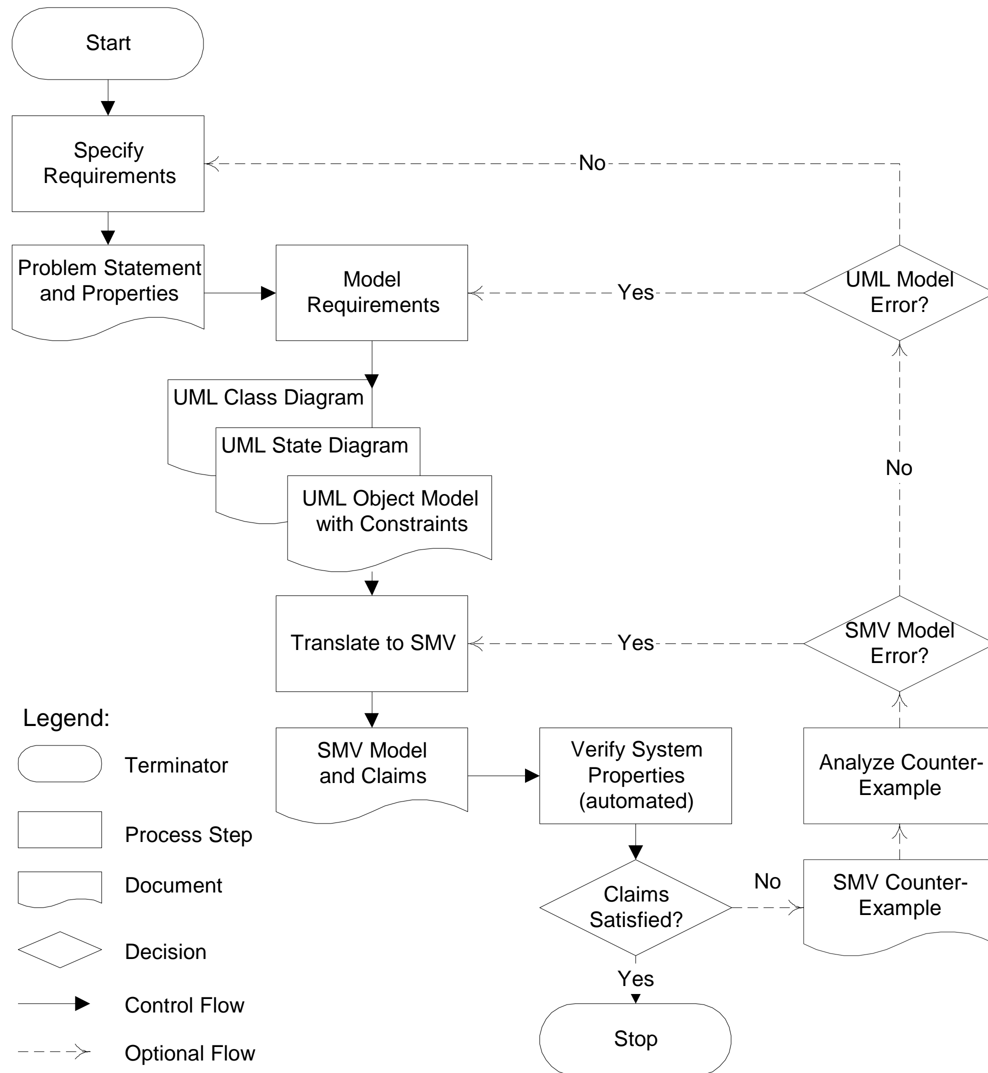


Figure 2: UML to SMV model checking process

Step 4. Verify the system properties using SMV

Verifying the CTL expressions using SMV is an automated step. SMV checks for syntax errors and cyclic dependencies. The SMV software tool builds a representation of the system state space and checks the CTL expressions with respect to the SMV model. There are two possible outcomes:

1. The system property is true.

2. The system property is false. In this case, SMV typically shows a counter-example to substantiate its claim.

For very large SMV models, the SMV tool will run for hours and may run out of memory and abort execution. The Cadence SMV used in this thesis controls the size of the BDDs created and manipulated during state exploration. However, it is still possible that the Cadence SMV tool will exhaust all memory available and abort execution.

In addition to checking the systems properties defined in the user requirements, the developer can check additional CTL expressions, such as test conditions, liveness or progress properties, guarantee conditions and valid value range conditions. Details of these additional system properties will be discussed in Chapter 4.

Step 5. Analyze the counter-example

If the outcome of the model checking is true, we have increased confidence that the UML model accurately represents the system requirements defined by the system properties. If the outcome of the model checking is false, a counter-example shows a sequence of states that leads to state where the property is violated. If the counter-example reveals an error, then fixing the error *may* mean changing the requirements or changing the UML model. It may also mean that there is an error in the translation process (i.e., UML model to SMV model or system property to CTL formula). Alternatively, we may intentionally construct claims that we expect to be false, in which case a counter-example is a desirable outcome because the counter-example provides a detailed trace of a specific execution path. This trace can be useful for understanding the execution steps of SMV. The types of errors we encountered in the elevator case study and the corrective actions taken are discussed in Chapter 5.

Step 6. Iterate Steps 1 through 5 until no more problems are found

Using model checking as an automated analysis tool allows us to verify a partial UML model very early in the UML modeling phase. Constraints are specified on an object or a group of objects (on the links) on the UML object model. To verify if a UML constraint is true, one can identify and translate only the objects associated with the constraint and proceed to model check a small portion of the software system.

Thus, new objects and state diagrams can be added in an incremental fashion. This can increase the probability of getting an answer from the SMV tool especially if the final UML model is large. Once a specific object or group of objects has been verified to be correct, we have more confidence and can add more objects for translation and verification. Verifying partial specifications will be discussed in Chapter 4.

1.4 Main Contribution

The main contributions of this thesis are as follows:

1. We propose UML extensions and notation conventions that ease the translation from UML model to SMV input. The main extensions to the UML class diagram notation include: adding a special External_Event class to centralize all the events generated by the environment; adding input events and input objects stereotypes to the class name section of a class; adding a fourth section of a class to define output events; and adding array indices to role names to distinguish multiple instances of the same class. In this thesis, we treat UML operations as mathematical functions. We do not support dynamic use of constructor and destructor operations. We associate a UML state diagram with a single class. We propose changes to the UML state transition by adding an optional source object to distinguish the source of the event trigger; adding an array index to the source or destination object of the events; and introducing an optional quantification formula over multiple source objects to specify multiple transitions in a more compact manner.
2. We develop the SMV program structure (i.e., a template) and rules for translating UML class diagrams and UML state diagrams to SMV. These rules address UML's *composition* and *generalization* constructs and semantics found in UML class diagrams. They also address UML's *timer*, *concurrent states* and *multiple-event processing* semantics found in UML state diagrams.
3. We demonstrate how our bottom-up approach in translating and verifying the UML model allowed us to perform unit verification early in the specification process. This approach helps us detect and correct errors for each object before analyzing the entire specification.
4. We demonstrate how model checking a UML specification can help to iterate and refine a requirements document. We observe how model checking detected errors that were not detected when writing the user requirement or specifying the UML models.

The primary goal of the proposed translation rules and guidelines is to make both UML and SMV specifications easy for software practitioners to understand. We evaluate our approach in model checking UML models by the ease of translation and the effectiveness of detecting errors in the original UML specification. We also discuss how the proper use of object-oriented specifications can improve the readability of the UML and SMV models. Although it is possible to automate the proposed UML-to-SMV translation rules, this is not part of the thesis.

2 Modeling Reactive Systems Using UML

Throughout this chapter, we provide guidelines for writing UML specifications that are amenable to model checking. We provide modeling guidelines for a subset of UML class diagram notation and UML state diagram notation. These guidelines improve readability of the specifications. They also eliminate ambiguity by using clear and precise UML notation. The guidelines were developed based on the elevator example and eliminate many of the common pitfalls in object-oriented specifications.

This chapter provides a case study on how to use the Unified Modeling Language (UML) to model reactive systems[2]. UML provides a set of notations and diagrams to model object-oriented structure and complex behavior of reactive systems. First, we describe the three-floor elevator system and the system properties to be checked. In Section 2.2, we use UML's class diagram to show the object-oriented structure of the software system. In Section 2.3, we use UML's state diagram to describe how each of system's objects reacts to external events. We use the methodology described by Douglass [Doug98] to construct our UML specifications. In Appendix A, we provide a summary of the UML notation used in our case study.

[2] A reactive system is a long running system that interacts continuously with its environment. It is designed not to terminate.

2.1 Requirements Analysis

In this section, we present the problem statement and system properties for a three-floor elevator that we use as a case study. The problem statement is an informal description of the system to-be-developed and its basic requirements.

2.1.1 Three-floor Elevator System Problem Statement

A software system controls an **elevator** for a building with **three floors**. The first and third floors each have *one* **elevator request button**, located outside the elevator, used to request that the elevator to come to that floor. The second floor has *two* **elevator request buttons**, one for requesting service to a higher floor and one for requesting service to a lower floor. This means that when both the up and down second floor hall buttons are pressed and the elevator arrives to pick up a passenger in one direction, the backlight of the hall button for the other direction will remain lit. Elevator request buttons are also known as **hall buttons**. Pressing a **hall button** will keep the elevator door open, if the elevator is on the same floor as the pressed button and the door is currently open.

Inside the elevator, there are *three* **floor request buttons** used to request to go to a specific floor. The elevator also has an **open door button** and a **close door button**. The **open door button** opens the doors when the elevator is not moving. Each time the **open door button** is pressed, the time period that the door will remain open shall be reset to five seconds. The **close door button** closes the doors immediately even if the timeout period of five seconds is not yet reached. To ensure fairness to all service requests, a maximum of three (3) consecutive open door requests is allowed. Collectively, the **floor request buttons**, **open door button** and **close door button** are called **elevator buttons**.

The elevator has a sliding door called the **inner door** of the elevator. Each floor has a sliding door called the **outer door** of the elevator. When the elevator arrives at the floor, the **inner door** and the **outer door** of the elevator simultaneously open using a mechanical device. The **doors** close within five seconds after the doors open.

Each time an **elevator request button** or **floor request button** is pressed, the request is placed in a **pending queue**. Once pressed, a request button is lit to indicate that a request is in the **pending queue**. Pressing a request button that is already in the pending queue has no additional effect. The

elevator has a **floor sensor** that determines when the elevator is at a floor. When the elevator arrives to handle an elevator- or floor-request, the light on the request button is no longer lit.

The elevator system shall respond to an elevator request whenever it is idle or already going in the requested direction. This means that if a passenger from the first floor requests to go to the third floor, a passenger with an **elevator *up* request** from the second floor will be picked up along the way. An **elevator *down* request** on the second floor will not be picked up until the elevator changes direction from up to down.

Two-Passenger Scenario

In order to understand the requirement that the elevator responds to new requests for travel in the same direction as its current destination, we present a two-passenger scenario using a UML sequence diagram.

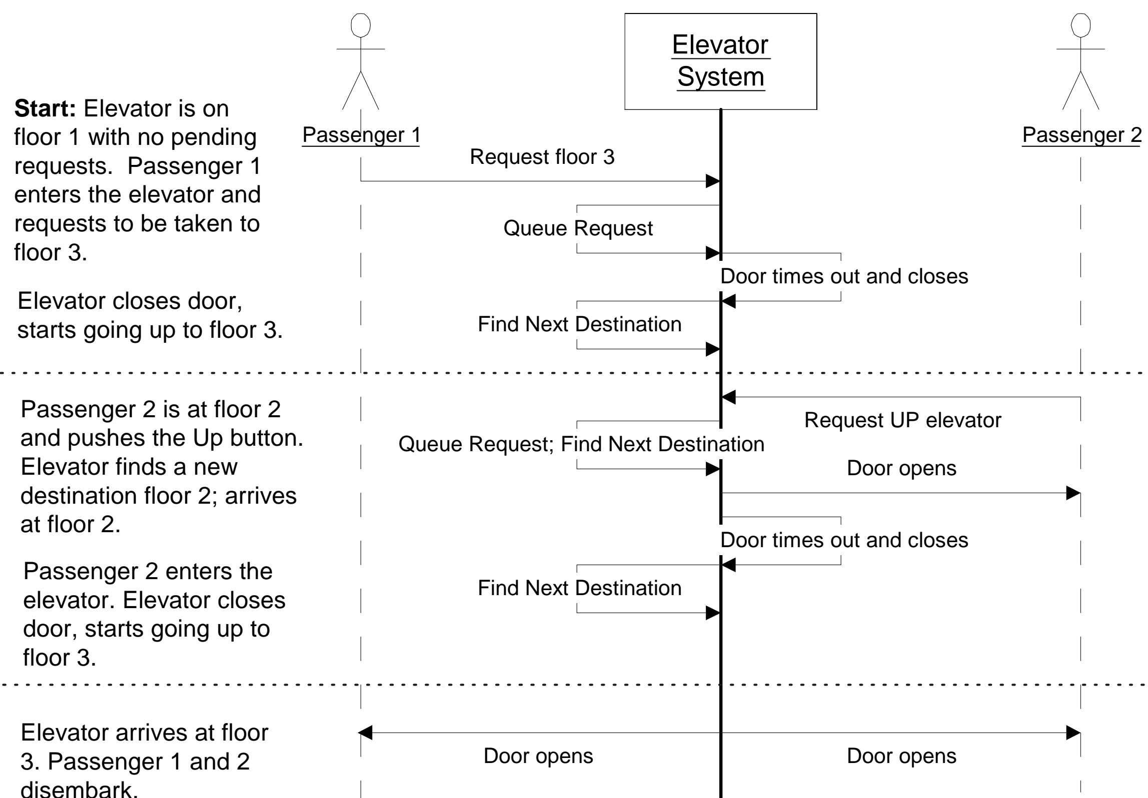


Figure 3: Two-passenger scenario sequence diagram

Figure 3 shows how the requests are sent to the elevator system asynchronously. Each request is stored in the pending queue. This allows the elevator to find the next destination (floor) each time the door closes or when new requests are received by the elevator while it is moving.

2.1.2 System Properties

The elevator should have the following system properties:

- Requests to use the elevator are eventually serviced; requests to be delivered to a particular floor are eventually serviced.
- The elevator never moves with its doors open.
- The elevator doors are not held open indefinitely.

2.2 Defining Object Structure using UML

In this section, we describe the object structure of our elevator case study. We will introduce the guidelines in writing UML class diagrams, object models and UML constraints as we construct the UML model for the elevator example. We used Visio's UML template for creating and maintaining our UML diagrams. To implement some of our proposed changes, we added extensions to the original UML template.

2.2.1 UML Class Diagram

The main class diagram shows an object-oriented decomposition of the problem description. When we specify the UML class diagram, we only show objects that form part of the software system. To avoid clutter, classes that represent environmental devices are kept to a minimum. Our primary goal is to produce successful software requirement specifications. As such, we build object models that have sufficient details to verify all the system properties defined by the domain expert, and no more.

Identifying Classes

A *class* captures the common structure and common behavior of a set of objects. An *object* is an *instance* of a class. It can contain both *data* and *functionality*. Data are represented as *attributes* in a class. Functionality is represented as *operations* that describe computational behavior.

When modeling and analyzing reactive systems, we are interested in the control flow of the software system. Therefore, we are most interested in identifying the *active objects* of the system. *Active objects* can autonomously perform actions, coordinate the activities of *component* objects, and generate events to other objects. In our elevator example, the active objects are the building, elevator, the door and the buttons.

Environmental devices such as sensors and actuators that are not controlled by the software system are shown in our main class diagram with the <<environment>> stereotype[3]. We introduce the *environment* stereotype to define a new form of class that defines classes that are

---

[3] A *stereotype* is an extension mechanism to UML, allowing analysts to add new semantics to the UML notation.

external to the software system. In our elevator case study, the **External_Event** and **Engine** classes are environment classes.

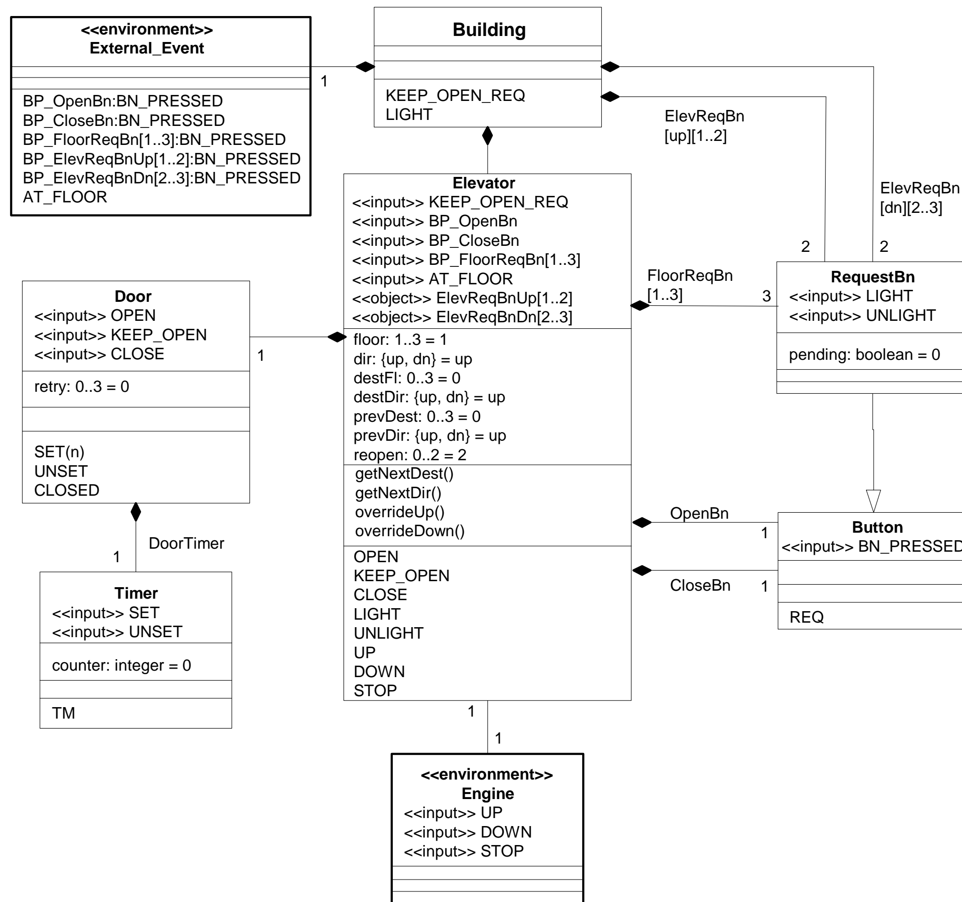


Figure 4: Three-floor elevator main class diagram in UML

Figure 4 shows how the active classes, **Building**, **Elevator**, **Door** and **Button**, are structured as a general solution to the elevator problem. We add an active class **Timer** that generates the timeout event needed by the **Door** class. The **RequestBn** class is a specialization of the **Button** class.

Class attributes refer to the data encapsulated within an object. Attribute definitions must include the data type because SMV will need to know the set of possible values for each attribute. Default values are optional because they can be defined in the state diagram. If both a default value for an attribute is specified and an initial value set in the *state diagram*, then they must agree. Alternatively, default values may be left unspecified (non-deterministic). For example, the **Elevator** class has a *floor* attribute whose value ranges from 1 to 3. The *floor* attribute records which floor the elevator is at. Initially, the elevator is on the first floor. Elevator's attributes also keep track of the elevator's current destination (*destFl*) and previous destination (*prevDest*). The

Elevator class has attributes *dir*, *destDir* and *prevDir* to keep track of the current direction, the destination's direction and the previous destination's direction respectively. The *reopen* attribute of **Elevator** class counts the consecutive requests to reopen the door while the elevator is on the same floor with doors closed. The **Door** class has a *retry* attribute that counts the number of consecutive requests to keep the door open. The **Timer** class has a *counter* attribute, which simulates the count down of the timer.

UML *operations* are the services provided by the class. For example, the **Elevator** class has a *getNextDest* operation that describes the complex rules used to decide the elevator's destination. To avoid clutter, we do not describe the operations of environment classes. Details of UML operations will be discussed in the next section.

Modeling Internal and External Events

Events are the messaging mechanisms that allow a system object to interact with the environment. They also enable the objects within a system to communicate and synchronize. UML does not say anything about how events are modeled in the main class diagram. Since events are critical to reactive systems, we treat events as first class modeling elements in the main class diagram. Modeling events explicitly helps to make the main class diagram more readable and eases its translation to SMV.

The UML notation does not provide a notation to explicitly model the events in the main class diagram. To resolve this problem, we propose several extensions to the UML class diagram notation. We introduce as part of the class name the set of *input events*, which instances of that class can receive and react to. By explicitly defining the input events in the class diagram, we are defining the input interface of the class[4]. The input interface is useful for inspecting the class's state diagram because the state diagram should not generate these input events. An input event serves as an event trigger to enable a transition. Input events are prefixed with <<input>> stereotype. We only declare input events that do not come from an object's class or its component classes because the class has visibility to the events generated by its component classes. For example, **Door** class can access events generated by its own component class **Timer**, so we do not declare the TM (timeout) input event in the **Door** class. In addition to input events, a class

[4] These input events must be consistent with the input events used in the class's UML state diagram.

that needs to access attribute values from other objects that are external to the class are defined as *input objects*. Input objects are prefixed with <<object>> stereotype.

We also introduce in a fourth section of a class declaration the set of *output events,* which instances of that class can generate. By explicitly defining the output events in the class diagram, we are defining the output interface of the class. The output interface is useful for inspecting the class's state diagram because the state diagram should generate these output events. This new *event section* is placed below the name, attribute and operation sections. Although it is possible to use the state diagram to identify the events that the environment and system generate, it is convenient to see them in the main class diagram. Adding a separate event section allows us to check the consistency of event names and prevent conflict. Adding the <<input>> stereotype to the class name and adding a separate event section helps the reader to understand how events are passed among objects without having to simultaneously examine several state models. This improves the readability of the main class diagram.

Input events are classified into internal or external events. *Internal events* are events generated by the classes that belong to the software system. Internal events must correspond to the events raised in the class's state diagram. *External events* are events that are generated by the system's environment. Classifying an event as *internal* or *external* is important during the translation to SMV because the translation will rely on this information to decide how and where to define the variables. We introduce the concept of a special **External_Event** class, which lists all of the external events. These events are distinct from events generated by system classes because we may make assumptions during model checking about when and in what order these events can occur. This allows us to model check the behavior of the system when it is running under certain environmental conditions. In contrast, it would be invalid for us to constrain when system events occur, beyond those constraints defined by the specification. Other than the special **External_Event** class, environment classes such as **Engine** class, are not translated to SMV for model checking.

We can define the events that come from the environment in either a centralized or decentralized manner. We proposed to centralize all environment events in a special **External_Event** class because it makes the design focused on the software system where the software system does not need to know where and how the external events are generated. It makes changes to environment events easier to track because there is only one environment class as opposed to many environment classes. This, in turn, eases the translation to SMV.

The UML state diagram notation does not distinguish between events that are shared or not shared. In our elevator case study, each of the push-button devices that are external to the elevator software system generates a BN_PRESSED event when the button is pressed. Each BN_PRESSED event leads to possibly generating an REQ event of a specific button. As such, we cannot model the push button events from the nine push button devices using the same external event name. We distinguish each of the nine BN_PRESSED external events by declaring nine output events in the event section of the **External_Event** class. These nine output events of **External_Event** class correspond to the multiple instances of **Button** and **RequestBn** classes. We distinguish each of these BN_PRESSED events by explicitly referencing the role names of associations from the **Elevator** class to the **RequestBn** and **Button** classes found in the UML class diagram.

When defining output events, we need to distinguish events that are shared by all of the object instances of the same class. In our elevator case study, the elevator object consumes REQ events, generated by both the OpenBn and CloseBn objects. OpenBn and CloseBn objects instantiates the same Button state diagram where an REQ event is generated whenever a button is pressed. Since the OpenBn and CloseBn state machine do not share the same REQ event, we define the REQ event in the event section of the **Button** class as not shared (i.e., not underlined). Consistent with UML's notation for class-scoped attributes, we would define a class-scoped event where the objects of the same class share the same event name and event value by underlining the event name.

Modeling Generalization Relationships

The **RequestBn** class is modeled as a *specialization or subclass* of a more generic **Button** class: a relationship denoted by a link with triangle arrowhead. This means that the **RequestBn** class inherits all the data and behavior of the **Button** class. All **Button** objects react to a BN_PRESSED event from the environment by generating a corresponding REQ event for the system to process. A **RequestBn** object has the added feature of lighting up when it has issued a REQ event that the system has decided to process, but which the system has not yet serviced. We extend the **RequestBn** class by adding a *pending* attribute to keep track of pending requests.

Identifying Class Associations

An *association* represents a semantic connection between two classes. Associations allow objects to communicate through messages and events. The **Elevator** class has an association to **Engine**

class. This association is required in order for the elevator to send requests to the engine to move the elevator up and down. Next, we discuss two types of associations, namely, composition and aggregation.

We use *composition* relationship when one object physically or conceptually contains another object. The larger class is called the *whole* and the smaller class is the *part* or *component class*. We refer to the *part* class of a composition relationship as the *component object*. In a composition relationship, the *part* class cannot be shared by other *whole* classes. For example, we model the Timer object as a *component object* of the Door object: relationship denoted by a line with a solid diamond arrowhead. This means that the Door object solely owns the attribute values and state values of the Timer object. This is important because other objects cannot share the five-second countdown of Timer object.

An *aggregation* relationship is a weaker form of composition. In an aggregation relationship, the *part* class can be shared by other *whole* classes. We refer to the *part* class as the *aggregated object*. We can convert aggregation relationships by creating a whole class and making aggregated objects as component objects of the newly created whole class. For example, we added the **Building** class and converted the aggregated hall buttons (not shown) of the **Elevator** class into component objects of the **Building** class. The result is shown in Figure 4.

We add *adornments* to the associations to provide more details to our specification. At each end of the association, we define the *multiplicity* or the *number of objects* that the class refers to. For example, one of the associations from Elevator to **RequestBn** class shows that there are three **FloorReqBn** objects associated to one Elevator object. A *role name* represents the purpose or capacity an object plays in an association. In our elevator example, we add role names on each of the component class-end of the composite associations (i.e., the end without the solid diamond arrowhead).

We borrow from Statecharts [Hare87] and RSML [LHHR94] the concept of *parameterization* and apply it to generate distinguishable object instances. Multiple instances of the same class are declared in the class diagram to be an array of instances. To achieve this, we annotate associations with role names plus array indices. For example, the Elevator main UML class diagram (Figure 4) declares the three FloorReqBn objects inside of the elevator, distinguished by index values 1..3 representing the three different floors. The class diagram also declares four ElevReqBn objects outside of the elevator, distinguished by one index value representing the

desired direction of travel and a second index value representing the floor. Array indices are used to distinguish between individual objects, for the purpose of identifying objects, events, and the source of an event or specifying the destination of an event. To ensure that the system can handle simultaneous events, transitions that are triggered by indexed events are quantified, to process all applicable events. We describe how to use array indices when constructing UML state diagrams in the next section.

Naming Conventions

We use naming conventions when defining our class diagrams in order to improve readability and for ease of translation to SMV.

- We specify class names in mixed cases starting with upper case (e.g., Elevator).
- We specify role names in mixed cases starting with upper case (e.g., FloorReqBn).
- We write attribute names in mixed cases starting with lower case (e.g., prevDest).
- We write operation names in mixed cases starting with a lower case. (e.g., getNextDest).
- We write event names all in upper case (e.g., REQ, BN_PRESSED); this will easily distinguish event names from operation names.

Limitations

The SMV model checker requires the state space to be finite. Therefore, we have to explicitly define the number of objects for each class. We assume that there is only one object if the cardinality is not defined. For class attributes with infinite data types (e.g., unbounded integer), we would also need to abstract the set of attribute values. We also expect our object model to be static over the course of execution. We do not support the dynamic use of constructor and destructor operations in our elevator case study.

2.2.2 UML Object Model

Object models are derived from the class diagram. They show *instances* of classes called *objects*. It shows *instances* of associations called *links*. An *object model* emphasizes the specification's object structure where all of the relevant run-time objects are shown. At run-time, each object's state machine runs concurrently with other object's state machine, similar to concurrent regions of a state machine running concurrently.

We construct the object model as a requirement before translation to SMV. In this step, we identify the system boundary in order to translate only those classes that are part of the software system. Only the concrete classes (i.e., classes that are instantiated) and the system-controlled classes (i.e., classes without the <<environment >> stereotype) are included in the object model. In UML, we represent an object with a rectangular box where the distinct object name followed by a colon and the object's class name are underlined. Figure 5 illustrates the run-time configuration of the elevator example that will be used for translation to SMV.

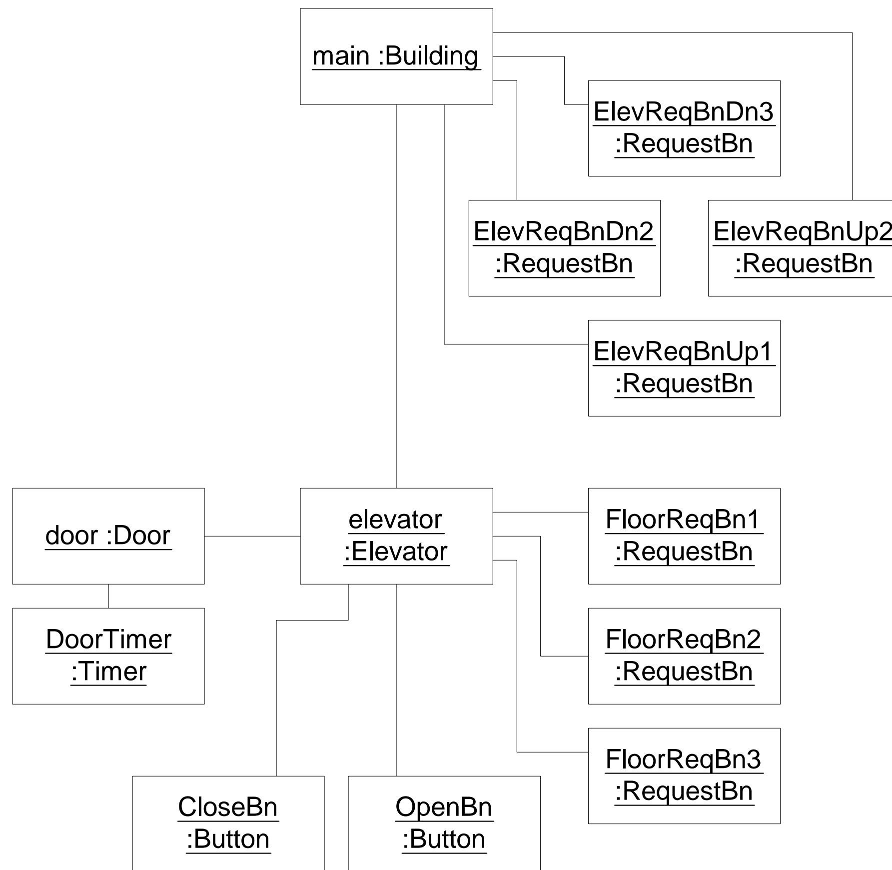


Figure 5: Three-floor elevator object model in UML

We use the following naming conventions to ease translation into SMV. Object names are specified in mixed cases starting with either lower case or upper case. For classes that represent a single object, the object name is the same as the class name starting with lower case. For classes that represent many objects[5], the object name is the role name. For example, ElevReqBnUp1, ElevReqBnUp2, ElevReqBnDn2, and ElevReqBnDn3 are RequestBn objects representing the

[5] Based on the multiplicity of the associations in the class diagram

four hall buttons on the three floors. The object that contains all the other class instances is named **main**.

2.2.3 UML Constraints

In UML, a *constraint* is an expression of some semantic condition that must hold whenever the system is in a steady state. It can be added to classes, class attributes, roles and associations.

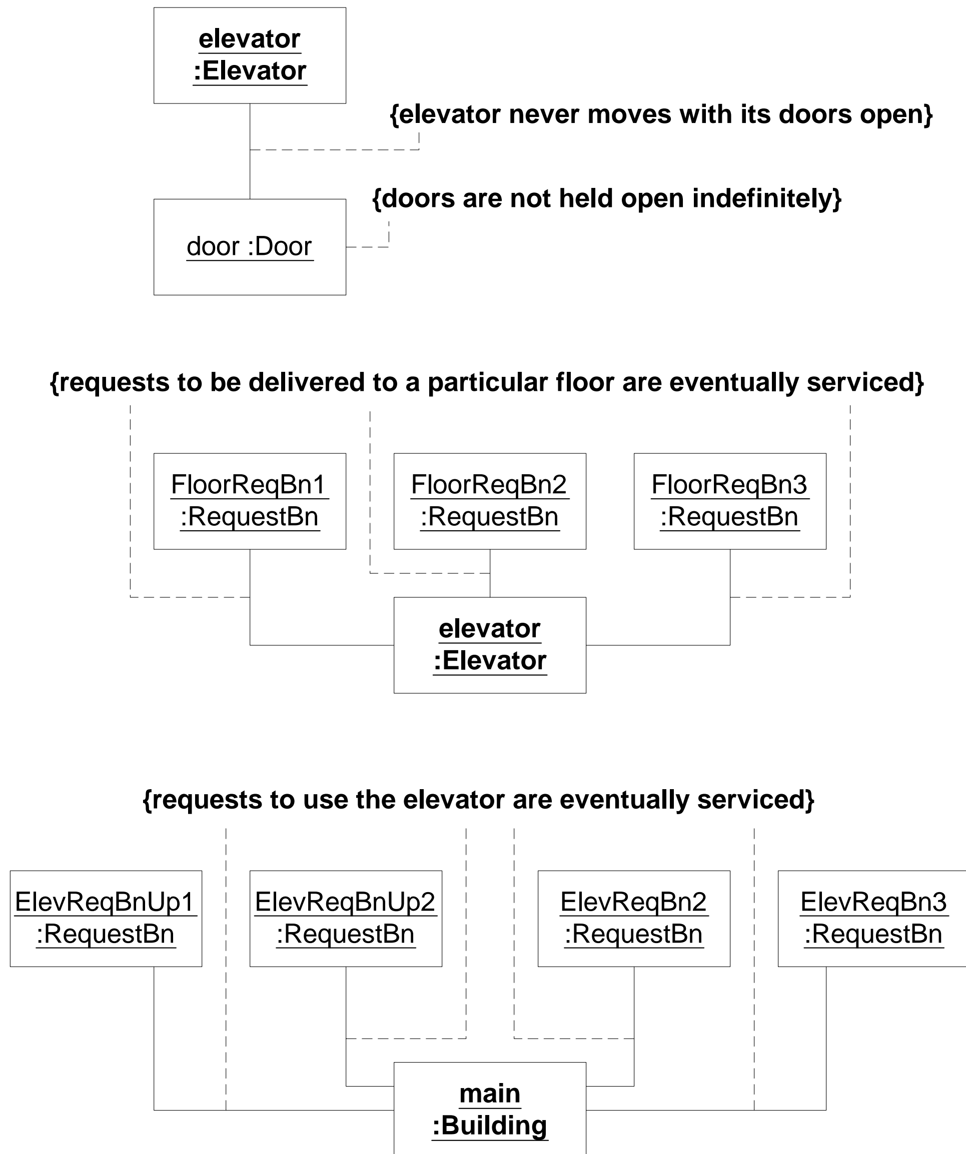


Figure 6: System properties shown as UML constraints

Before the state diagrams and the class attributes are finalized, it is difficult to specify the system properties in terms of object states and object values. Therefore, we write the UML constraints

using the vocabulary found in the problem statement (i.e., without using object states and object values). To avoid clutter, UML constraints are shown separately from the object model, though they are written with respect to the objects and links found in the object model.

Figure 6 shows UML constraints attached to objects and links. The constraint "{doors are not held open indefinitely}" is attached to the door object because only the Door object can achieve this property. The liveness properties "requests are eventually serviced" are related to the links between the request buttons and the elevator that services the requests. The safety constraint "{elevator never moves with its doors open}" is attached to the link between the elevator and the door because it affects both object's behavior. During the translation from UML to SMV, we will rewrite each of these UML constraints in a formal, model-checker-specific language. Details on this translation will be discussed in Chapter 4.

Other UML constraints such as the fairness constraint for environmental variables are defined in the external_Event object (not shown). Environmental constraints are translated as part of the SMV model. Details of translating environmental events are discussed in Chapter 3.

2.3 Defining Object Behavior using UML

In this section, we describe the object behavior of our elevator case study. As we model the elevator example, we will introduce the guidelines in writing UML state diagrams and UML operations.

2.3.1 UML State Machine Semantics

A UML state diagram defines the behavior of UML objects in terms of a state machine. UML semantics allows multiple interpretations of UML state machines. In this thesis, we associate a UML state diagram with a single class. For a class that has many objects, each object is associated with one state machine, which is an instantiation of the state diagram associated with the object's class.

The elevator state machine responds to user requests, input as buttons being pressed. It controls many other classes such as the **Engine** (when and in which direction to move) and the **Door** (when to open or close). The Elevator state diagram shown in Figure 7 is the result of decomposing the state diagram into two orthogonal regions. The first region, New_Requests, decides whether pressing a floor-request button initiates a pending request (it doesn't if the elevator is already at the requested floor and the doors are open). The second region, Operation, services the pending requests and has two substates (i.e., Stop and Moving) to represent whether the elevator is either stopped or moving. When in the state Stop, the elevator controls the opening or closing of the elevator door. When in the state Moving, the elevator controls whether the engine is moving up or moving down. Within each region, there is a *start-state* icon (drawn as a small filled circle) with a transition[6] pointing to the initial state. The initial state configuration for the Elevator state diagram is state Waiting of the New_Requests region and state Closed_Doors within the Stop superstate of the Operation region. The start-state transition of Operation region sets the default values of the elevator attributes floor, destFl, and prevDest to 1, 0, and 0 respectively.

UML state diagrams can be hierarchical. A *superstate* is a state that contains one or more substates. A state without a substate is an *atomic* state. A superstate is either an *and-state*, meaning all its substates are active, or an *or-state*, meaning exactly one of its substate is active

[6] We do not assign names to start-state transitions because they are not needed for translation to SMV.

and there are transitions among its substates. The *root-state* is the top-level state that represents the entire state machine. A state is *active* if the state machine is in that state.

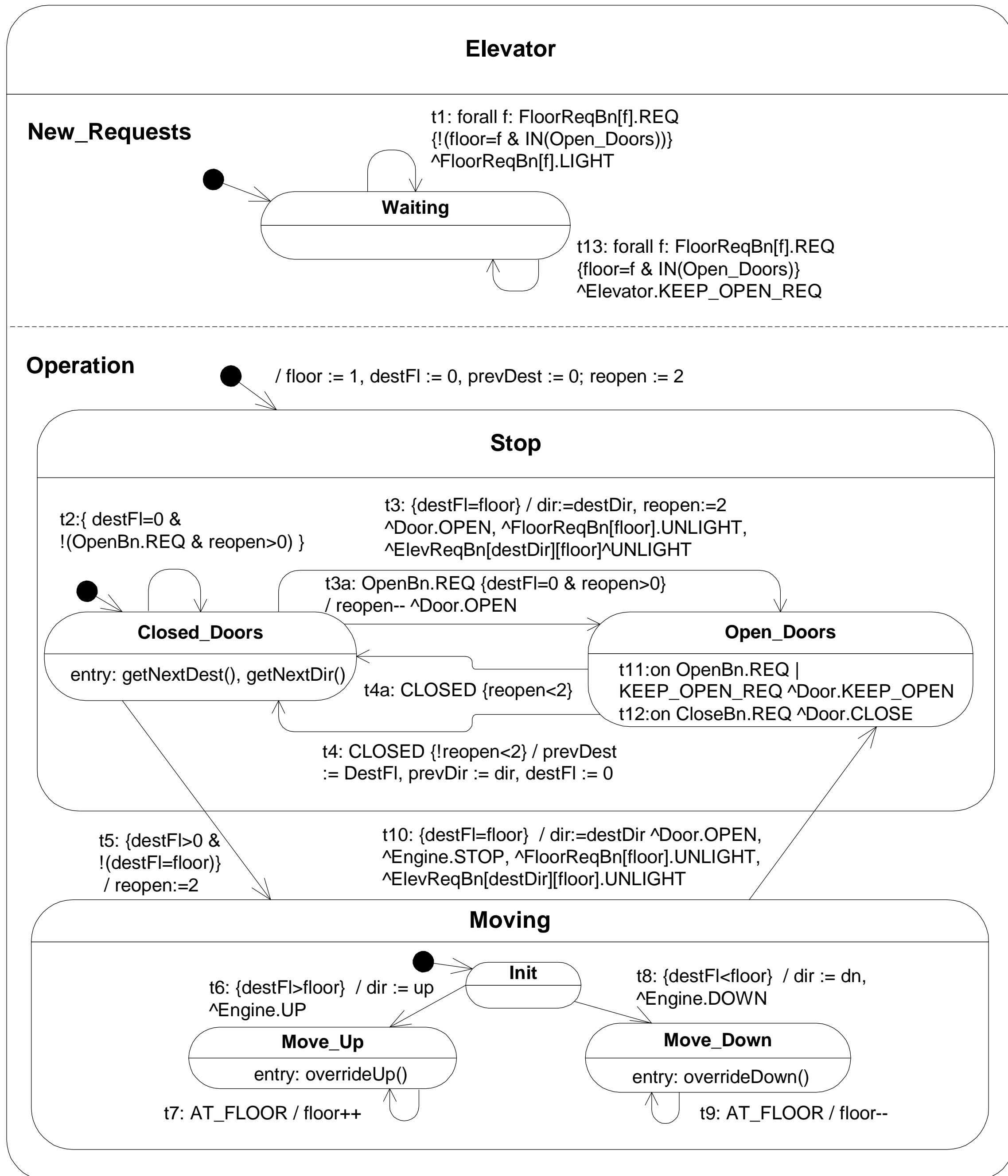


Figure 7: Elevator state diagram

A UML state machine receives *input events* from the system's environment (called *external events*), from other objects within the software system (called *inter-object events*), and from its

own object (called *intra-object events*). Inter-object events and intra-object events are collectively called *internal events* and are defined in event sections of the system-controlled classes. In our Elevator state diagram illustrated in Figure 7, AT_FLOOR is an *external event* that comes from the environment (i.e., **External_Event** class). KEEP_OPEN_REQ is an *inter-object event* that comes from the Building object that resides within the software system. An event is *set* when the event is raised or generated by the state machine.

The state machine reacts to an input event by (possibly) performing an action and by moving to the same state or to another state. This is called a *state transition*. Our syntax for transition events is derived from the syntax used in Statecharts [Hare87], OMT [RBP+91] and UML [UML99]:

label: src[index].event {condition} /action ^ dest[index].event

A state transition is depicted as an arrow originating from a source state to a destination state. It is caused by an *event* and/or a *condition*. An *event-triggered transition* is a transition that is triggered by the occurrence of an event. A *condition-triggered transition* is a transition that is triggered when its guard condition is satisfied. A *conditional event-triggered transition* is a transition that is triggered by an event if its guard condition is satisfied. Finally, an *automatic transition* is a transition that has no event-trigger or guard condition. For example, transition t4a of the Elevator state diagram is a conditional event-triggered transition because it is triggered by CLOSED event and has "reopen < 2" as a guard condition.

We introduce the concept of a transition *label* that serves as a macro for the transition's triggering event. Transition labels provide a compact means for prioritizing one transition over another by specifying that a lower priority transition can only occur if the higher priority transition is not enabled (represented as the negation of the latter transition's label). In our Door state diagram (Figure 8), we use transition priorities to specify that transition t4 (KEEP_OPEN event) has priority over transition t2 (a timeout event) and t3 (CLOSE event) if all these three events happen at the same time[7].

We also introduce designating the source (*src)* of an input event, because the event's source may affect the state machine's reaction to the event. For example, when the elevator receives a FloorReqBn request, it needs to know which floor the button is associated with, in order to decide

[7] Note that while transitions t2, t3, t4 without their priorities may look deterministic since they are triggered by different events, they need not be because multiple events may occur simultaneously. To ensure that the transitions are deterministic, one has to specify transition priorities in the transition events.

whether the request should be serviced. Since this information is encoded in the name of the button sending the event, the source of the event is needed to make this decision[8]. We may also designate the destination (*dest*) of an output event.

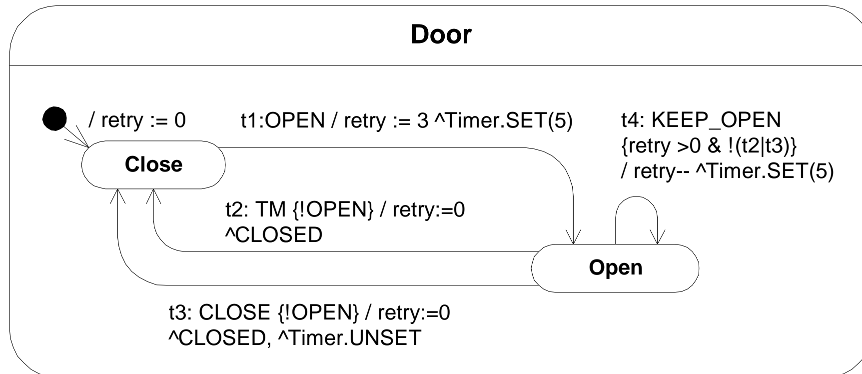


Figure 8: Door state diagram

Sometimes we may refer to a set of objects with object names that differ only in their indexed value. A benefit of introducing indexed objects is that simple tests and/or actions that need to be performed on a set of indexed objects can be expressed as a formula quantified over the objects' indices. They are represented as *src[index]* and *dest[index]* in our transition syntax presented. Note that we use square braces, which are the traditional condition delimiter, to designate an *index* value that distinguishes an object instance from other instances of the same class. As such, we cannot use square brackets to delimit conditions. Instead, we use curly braces to delimit conditions. For example, the transition t1 uses a quantified formula where state Waiting can receive and simultaneously make decisions about any number of FloorReqBn requests. Without using the quantified formula, we would have to declare three transitions (one for each floor).

In UML, event-processing proceeds as a series of execution steps, initiated by an external event. A transition executes in one *execution step*. A transition executes if at the beginning of a step the system's source state is *active*, its (optional) triggering event is set, and its (optional) condition evaluates to true. The effect of the transition execution is visible at the beginning of the next step, at which point all of the transition's variable assignments have completed, its generated events are set, and its destination state is active. If a transition generates events, these events may trigger

---

[8] Note that storing the floor as an attribute of the **RequestBn** class and sending it as a parameter of the REQ event is not an ideal modeling decision, since the floor never changes as the program executes, so it is not really a variable of the class.

other transitions in the next execution step. An event that is set at the beginning of a step expires at the end of that step (whether or not any transition reacts to it). It is only set in the *next step* if some executing transition generates its reoccurrence. For example, transition t10 executes if the elevator is in state Moving and the guard condition {destFl = floor} is satisfied. When enabled, transition t10 sends an OPEN event to Door, sends a STOP event to Engine and sends an UNLIGHT event to the appropriate RequestBn objects.

UML semantics allows actions to be associated with states. *Entry actions* are executed when the state is entered or reentered. Entry actions are prefixed with "entry:" and we do not add a transition label to entry actions because they do not react to events. In our Elevator state diagram (Figure 7), we associated entry actions to state Closed_Doors. Upon entry to state Closed_Doors, the state machine executes getNextDest and getNextDir operations even if the state is entered via a self-transition (e.g., transition t2). "On event" actions are executed within an active state and the state machine stays in the same state after executing an "on event" action. This means that the entry actions are not performed when "on event" actions are executed. The syntax of "on event" actions is similar to the transition syntax discussed earlier. For example, Figure 7 shows transition t11 and t12 as "on event" actions of state Open_Doors and reacts to an event trigger without re-entering the state Open_Doors.

The UML state machine semantics makes no assumptions about the time intervals between event reception, event dispatching and processing. Analysts will sometimes make assumptions about when input events can occur. One such assumption is the *synchrony hypothesis* [BeGo92], in which the system is assumed to be fast enough to respond to one input event from the environment before receiving a subsequent event. Another assumption is the *single-event hypothesis*, in which the system is assumed to be able to distinguish the arrival order of near-simultaneous events and is thus defined to handle one input event at a time. These assumptions reduce the number of cases in which transitions can execute simultaneously. Such assumptions do not appear explicitly in a UML specification. But for model checking, whether or not such assumptions apply must be known, as they affect constraints on when environmental input events can occur. In our elevator example, we assume that these events occur simultaneously. At the beginning of any execution step, there may be many set external and/or internal events. Multiple transitions may execute simultaneously if they are activated by the same event, if they are activated by different but simultaneous events, or if they are enabled by true conditions.

2.3.2 Example Scenarios of Elevator State Machine

In this section, we demonstrate the execution semantics of our UML state machine.

Two-passenger Scenario

To help understand the elevator state diagram shown in Figure 7, we will trace the steps that make up a sequence of transitions in the two-passenger scenario presented in the problem statement (see Figure 3 of Section 2.1.1). This example exhibits many of the subtleties of UML's concurrent state, nested state and state transitions. We describe this scenario in terms of execution steps as described in our UML state machine definition discussed in the previous section.

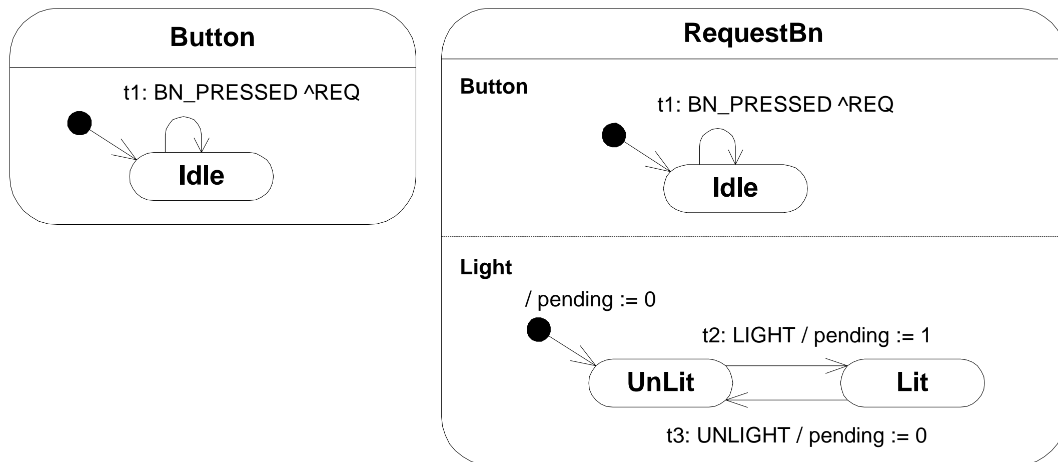


Figure 9: Button state diagram and RequestBn state diagram

Start of Scenario: Passenger 1 is inside the elevator, which is on the first floor with its doors open. This is represented as state Waiting in the New_Requests region and state Open_Doors in the Operation region. This scenario assumes that the elevator is waiting on the first floor in response to a previous floor 1 request. Thus, prevDest has value 1. There are no pending requests. In Figure 9 (on the right), the scenario starts in state Unlit of the Light region of all RequestBn objects (i.e., the three elevator buttons and the four hall buttons). Thus, attribute destFl has value 0. When the door opened, the DoorTimer object started a timer. The timer has counted down to three right before step 1 starts.

Step 1. Environmental Event: Passenger 1 presses the Floor 3 elevator button (i.e., presses a physical push button device for FloorReqBn3 object).

Step 2. External_Event object generates output event BP_FloorReqBn3.

Step 3. In this execution step, both REQ and TM events are broadcast.

a. The state machine for the floor request button FloorReqBn3 (an instance of the RequestBn state diagram shown in Figure 9) responds to the BP_FloorReqBn3 event (of type BN_PRESSED) with a self-transition t1, thereby broadcasting event REQ.

b. Since the door's timer counted down to zero, an internal timeout (TM) event is broadcast by the DoorTimer object.

Step 4. In this execution step, both REQ and TM events are processed.

a. Transition t1 of the Elevator state machine (Figure 7) reacts to the FloorReqBn3.REQ event by sending a LIGHT event to the FloorReqBn3 object.

b. The door object reacts to the TM event by moving from state Open to state Close (transition t2 of Door state diagram shown in Figure 8). The door object broadcasts a CLOSED event.

Step 5. In this execution step, both LIGHT and CLOSED event are processed.

a. FloorReqBn3 responds to the LIGHT event by moving (via transition t2) from state UnLit to state Lit. During this transition, the *pending* attribute of FloorReqBn3 is set to 1 (true).

b. In Figure 7, the elevator's transition t4 responds to the CLOSED event by moving from state Open_Doors to state Closed_Doors.

Step 6. On entry to state Closed_Doors, the elevator calls the *getNextDest* operation. *GetNextDest* is a system-controlled operation that computes the next destination. It finds that FloorReqBn3 has a pending request (i.e., *pending* attribute has value of 1). The *destFl* attribute is updated to 3.

Step 7. After the *getNextDest* operation is performed in Step 6, the elevator's transition t5 executes because the guard condition {destFl > 0 and !(destFl = floor)} becomes true. This transition causes a state transition from Stop superstate to Moving superstate of the Operation region.

Step 8. Of the two transitions leaving state Init in Moving superstate, the guard condition {destFl > floor} of transition t6 is satisfied, so the elevator now transitions to state Move_Up and sends

an UP event to the Engine object. The elevator will stay in state Move_Up until an AT_FLOOR event[9] occurs.

Steps 1 to 8 show that a transition executes in one step and external events are processed in the next step. A step can process an external event, an internal event (i.e., inter-object event or intra-object event), or a combination of multiple external events and internal events. We assume equal priority of external and internal events. Thus, the analyst has to assign the transition priorities explicitly. Moreover, Steps 3 to 5 show that multiple events can be processed in the same step.

Modeling a Single Event Generated by Multiple Objects

Multiple objects may generate the same event. We call such an event a *multiple-source event*. UML does not provide guidance in declaring the source of the event that triggers a transition. Whenever it matters, we add the object name of the source as the prefix of the event separated by a dot ("."). For example, REQ event from the OpenBn object will be declared as "OpenBn.REQ". There are two possible ways to refer to a multiple-source event as a *trigger* in a state transition:

- *Unspecified Source*: When the source object of the event is unspecified, we translate this event to mean, "If the state machine receives the event from *any* one of the objects".
- *Fully Specified Source*: When the source of the object is crucial in determining whether the state transition should be enabled, we explicitly write the condition showing the source of the event. For example, transition t1 of Elevator state diagram (Figure 7) has different enabling conditions, depending on the source of the REQ event:

```
t1: forall f: FloorReqBn[f].REQ {!(floor=f & IN(Open_Doors)}
   ^FloorReqBn[f].LIGHT
```

The floor is crucial to setting the light of a specific floor request button because we want to set the light only on the floor request button that has been pressed.

---

[9] Generating the *AT_FLOOR* external event is dependent on the environment. In our model, we do not constrain the external events. UML state diagram makes no assumption about the time that it takes for the engine to reach the destination floor.

Modeling Events with Parameters

In UML, events may pass data as parameters. In the Door state diagram (Figure 8), transition t1 and t4 generate event SET with a parameter 5. Parameters may refer to local attributes or the current state of an object. Parameters of the latter type avoid declaring attributes that store active states. This departs from the UML standard that does not say anything about passing an active state as an event parameter.

2.3.3 Inherited State Diagrams

UML does not explicitly require that the state diagram of a subclass adhere to a specific inheritance relationship with its parent class. But the Liskov Substitutability Principle (LSP) [Lisk88] states that a subclass may extend the state-model behavior of its parent class, but must still be substitutable for its parent class. Following the LSP, we aim to reuse the state diagram of base classes when constructing state diagram of subclasses. In this thesis, we restrict the inheritance relationship between a subclass and its parent class using the *strict inheritance model* [McGr93]. Following the strict inheritance model allows reuse of the base class's state diagram when constructing the subclass's state diagram. The strict inheritance model follows two requirements: (1) a method invariant, where (a) pre-conditions of a subclass's method may only be weakened relative to the base class's method, and (b) post-conditions of a subclass's method may only be strengthened relative to the base class's method; and (2) a class invariant, where the base class's specification is a subset of the subclass's specification. This approach allows complex state diagrams to be built with less effort. It also eases translation to SMV by including the SMV specification of the base class as a subset of the SMV specification of the new subclass. Using the strict inheritance model allows a subclass to be extended in the following ways[10]:

- *New attributes can be added and new values can be added to existing attributes.* Existing actions found in transitions or state activity section can be modified to set newly added attributes provided the base class's observable behavior does not change with respect to the inherited input events. For example, we add a new pending attribute to the **RequestBn** subclass.

---

[10] The implications of following the strict inheritance model (e.g., a child class cannot delete a state of its parent class) can be found in [McGr93]. For simplicity, we do not include multiple inheritance in our UML models.

- *A new concurrent region can be added to an existing state diagram.* The region introduces new sub-states and transitions between these sub-states. The transitions can be triggered by new or existing input events, can modify new or existing attributes, and can generate new or existing output events. For example, we add a new Light region to the RequestBn state diagram.

- *New states and transitions can be added to an existing state machine if the new transitions from existing states to new states are triggered by only new events* [Muth00]. Adding new states while retaining all the existing states means that the post-conditions of the base class's methods are not violated. Adding new transitions to and from the new states must be triggered by new events because the existing behavior of the base class relative to an existing event must be preserved.

- *An atomic state can be refined to be a superstate containing two or more substates.* The refined state can take advantage of additional information represented by the new substates. This means that we have a more detailed specification and we can construct new system properties (for verification or model checking) that refers to the new substates. For example, we could create a new subclass (not shown) that inherits RequestBn's state diagram where the state Lit is refined to contain substates Low_Level and High_Level to indicate the light level when the button light is on.

- *Input events can be generalized and output events can be specialized.* Existing transitions can be triggered by more general input events and can generate more specific output events. By generalizing input events, a subclass can be substituted for its parent and be sure to accept all of the input its parent would have accepted (it can accept more, but if it is being treated as its parent, it won't be sent more). By specializing output events, the subclass continues to generate output events that its environment will accept (since the environment would accept anything that the parent outputs, and the subclass's output is just a specialization of this).

- *Existing transitions can be modified provided the existing observable behavior is strengthened.* This is achieved by decomposing the existing transition into multiple transitions by modifying the guard conditions.

To ensure substitutability, inherited transitions must have priority over subclass transitions so that the class behaves like the parent class if both inherited and new transitions are simultaneously enabled (e.g., by simultaneous input events).

Figure 9 shows the difference between the state diagram of **RequestBn** class (on the right) and that of its **Button** superclass (on the left). All Button objects react to a BN_PRESSED event from the environment by generating a corresponding REQ event for the system to process. **RequestBn** inherits this behavior shown as Button region. A RequestBn object has the added feature of lighting up when it has issued an REQ event that the system has decided to process. To achieve this, we add a concurrent region (i.e., Light region) that reacts to a LIGHT event generated by Building and Elevator objects. We added substates UnLit and Lit, transitions t2 and t3, and actions to set the *pending* attribute when the button is lit. All the changes and additions to RequestBn state diagram adhere to the strict inheritance model, which means we can substitute RequestBn's state diagram for Button's state diagram, and the original behavior of **Button** class is preserved. If and when the inherited state diagram does not adhere to the strict inheritance model, the translation to SMV is not straightforward.

2.3.4 UML Operations

UML Operations[11] are the services provided by the class. To avoid synchronization issues while the UML operation is not completed, we assume that an operation executes in one step. Thus, we model our UML operations as functions. In this thesis, we are interested in using operations to define complex rules that change values within an object. Complex rules are those that cannot be expressed by the use of simple assignment instructions. UML operations may be specified using signatures, pre-conditions, post-conditions and invariants. In the elevator example, the *getNextDest* operation uses the values of the current floor (floor attribute), current direction (dir attribute), previous destination (prevDest attribute), previous direction (prevDir attribute) and pending requests (pending attribute) to calculate the next destination (destFl attribute). Because the *getNextDest* operation has many cases, we use decision tables to graphically separate the different cases and show relationship between input values and post-condition results. For simple functions, we use assignment statements instead of decision tables. We use the SMV logical operators syntax for writing expressions.

Table 1 shows the decision table for finding the next destination. The decision table is divided into two parts by a double solid line. The upper part describes the cases in terms of states and

[11] *Operation* is a conceptual construct, while *method* is an implementation construct. In this thesis, we consistently use *operation* as an action (service) that can be performed by the object.

condition values, while the last row shows the corresponding assignment to the class attribute listed in the leftmost cell (i.e., *destFl*).

The first column in the upper part shows the states of the state machine using the IN operator. The IN operator evaluates to true if the state specified in the argument list is active. The rest of the columns in the upper part show the conditions that must be satisfied in order to set the class attribute to the value shown in the bottom cell of each column respectively. Each cell on the upper part may contain predicates (e.g. value comparison on attributes), functions (e.g., use of IN operator) or macros (e.g., up_f1_t1_1 of Table 2). We use "!" to stand for the logical operator NOT, "&" for logical AND, "|" for logical OR. These logical operators are used in the SMV language and makes translation to SMV straightforward.

States	Condition 0	Condition 1	Condition 2	Condition 3
IN(Elevator.UP)	!(f1_pending | f2_pending_up | f2_pending_dn | f3_pending)	up_f1_t1_1 | up_f1_t1_n | up_f2_t1 | up_f3_t1	up_f1_t2_sd | up_f1_t2_od | up_f2_t2_1 | up_f2_t2_n | up_f2_t2_od | up_f3_t2_sd | up_f3_t2_od	up_f1_t3 | up_f2_t3 | up_f3_t3_1 up_f3_t3_n
IN(Elevator. DOWN)	!(f1_pending | f2_pending_up | f2_pending_dn | f3_pending)	dn_f1_t1_1 | dn_f1_t1_n | dn_f2_t1 | dn_f3_t1	dn_f3_t2_sd | dn_f3_t2_od | dn_f2_t2_1 | dn_f2_t2_n | dn_f2_t2_od | dn_f1_t2_sd | dn_f1_t2_od	dn_f1_t3 | dn_f2_t3 | dn_f3_t3_1 dn_f3_t3_n
destFl	**0**	**1**	**2**	**3**

Table 1: getNextDest operation modeled as a decision table

We use functions and macros to reduce the size of the decision tables. The *getNextDest* operation decision table (Table 1) uses the macros defined in Table 2 in AND-table format. The leftmost column shows a list of predicates or macros. Each column, except the leftmost one, defines a macro (named in the column header) in terms of the *conjunction* of predicates or macro values. True values are marked as T (true), negations are marked as F (false), and a dot "." means that the predicate or macro has no effect. To avoid circular definition of macros, we mark a cell with an "X" to indicate that the macro is not allowed to be a part of the condition.

Table 2 shows the macros used in the first row and "Condition 1" column of getNextDest operation's decision table (Table 1). These macros represent all the possible outcome of choosing the next destination when the elevator is going up (row 1) and the destination results to 1 (third column). We first define four "pending request" macros used in defining macros for Table 2:

```
f1_pending = ElevReqBnUp1.pending | FloorReqBn1.pending
f2_pending_up = ElevReqBnUp2.pending | FloorReqBn2.pending
f2_pending_dn = ElevReqBnDn2.pending | FloorReqBn2.pending
f3_pending = ElevReqBnDn3.pending | FloorReqBn3.pending
```

	up_f1_t1_1	**up_f1_t1_n**	**up_f2_t1**	**up_f3_t1**
currFloor = 1	T	T	.	.
currFloor = 2	.	.	T	.
currFloor = 3	.	.	.	T
prevDest = 1	F	T	.	.
f1_pending	T	T	T	T
f2_pending_up	.	F	.	.
f2_pending_dn	.	F	.	.
f3_pending	.	F	.	.
up_f2_t2_1	.	.	F	.
up_f2_t2_n	.	.	F	.
up_f2_t3	.	.	F	.
up_f2_t2_od	.	.	F	.
up_f3_t3_1	.	.	.	F
up_f3_t3_n	.	.	.	F
up_f3_t2_od	.	.	.	F

Table 2: Macros for going to floor 1 with direction up

Similar tables defining macros used for the rest of the macros defined in the first row (going up) of Table 1 are shown in Appendix B. The macro names represent the elevator's current direction, what floor the elevator is at, and what the next destination will be. For example, up_f1_t1_1 macro means that *dir* is up (up), *floor* is 1 (f1) and the next destination is going to floor 1 (t1). We distinguish between the first time and the second time the elevator needs to go to the same floor by appending the _1 and _n to up_f1_t1 macro. We do this to ensure progress by giving priority to the second and later request from the same floor and direction, if no other requests are pending. Thus, up_f1_t1_1 represents the elevator on the first floor going to the first floor for the first time. Up_f1_t1_n represents the same floor for the second time and up.

To ensure progress, getNextDest operation was designed to find the nearest pending request in the current and opposite directions. Table 2 implicitly shows a round-robin approach for handling priority of requests depending on the current direction, current floor, previous direction and previous destination. We add _sd and _od to macro names to represent "same direction" and

“opposite direction” respectively. For example, Table 2 shows macro up_f2_t2_od that refers to servicing a request on the second floor in the opposite direction (i.e., service buttons where direction is down).

To avoid any conflict in the results when evaluating the decision table, the combinations of state column and condition columns must be *mutually exclusive*. Unlike pseudo-coding where the result is dependent on the order or sequence the conditions are specified, the decision table conditions must be constructed in such as way that the result is the same for any order or any sequence of evaluation. Using decision tables and AND-tables to present the complex logic helps to detect overlap of conditions and omitted conditions.

Summary

In this chapter, we have discussed the UML class diagram and the UML state diagram notations and concepts and the extensions of UML notations that are used in modeling reactive systems with complex behavior. We have discussed the state machine semantics and complex UML operations that will be used for model checking. We have shown the use of object-oriented techniques to partition the elevator system into real-world objects with clearly defined responsibilities. This object-oriented decomposition makes the UML specification easier to understand and easier to translate to SMV.

3 Translating UML to SMV

In order to verify the UML models specified in the previous chapter, we translate the UML main class diagram, its instantiated object model and the corresponding UML state diagrams to SMV. This chapter proceeds as follows. First, we give a brief introduction to SMV. Then, we incrementally show how pieces of the elevator UML specification are translated using our proposed rules and guidelines for translation. The entire translation of the elevator example is provided in Appendix C. We finish the chapter with a summary of the proposed rules and guidelines for translating UML to SMV.

3.1 Translation Overview

There are a number of principles we tried to adhere to when translating UML specifications into SMV.

- Where possible, we preserve the object-oriented properties of the specification, including class structure and hierarchical state structures. We confine the SMV representation of the class attributes, operations and state diagrams to their respective sub-sections in the SMV program structure. This makes it easier to read and modify the SMV logic model.
- The translated SMV model should be readable and the translation from UML to SMV should be traceable. This helps to maintain a large SMV model where multiple objects interact. In the absence of automated translation programs or when using semi-automatic translation programs, traceability helps pinpoint the cause of errors and counter-examples.
- The SMV variable names should have a one-to-one correspondence to specification names (e.g., SMV variable ev_REQ corresponds to REQ event in Button class). This feature aids in tracing counter-examples in a SMV specification.
- The rules and guidelines for translation should be straightforward, to aid manual translation and possibly help to automate the translation.

In this section, we summarize those features of an SMV specification that are relevant to our translation from UML to SMV. We will explain additional features in the subsequent sections as we translate our UML specification of the elevator. A complete definition of the SMV syntax and semantics can be found in [McMi93].

First, we present our *template* for translated SMV specifications (see Figure 10). The SMV program may consist of several *modules*, including a mandatory **`main`** *module* that has no parameters. A *module* is defined as a set of variables and assignments that specify how variables are updated. Each module, with the exception of the **`main`** module, can be instantiated many times. To preserve the object-oriented properties of the UML specification, we translate each UML class as a module in SMV. The **`main`** module represents the entire system. It is also where system properties are defined, in the module's SPEC section. SMV modules provide modularity and reusability of SMV programs. Modules interact with one another through the main module's variables and through the modules' input parameters. Variables passed as input parameters into modules are passed by reference.

Each module, including the main module, consists of the following sections: VAR, DEFINE and ASSIGN. Variables are declared in the VAR section of the SMV program. They can be of type boolean, enumeration type, integer range, user-defined module (i.e., instantiated module), or array of scalar types. These variables must have a finite range of values, so that they can be encoded internally as a collection of boolean variables. Numeric values one (1) and zero (0) represent boolean values true and false, respectively.

```
-- **************************************************************
-- *                        MODULE1                             *
-- **************************************************************
MODULE Module1(parm_1, .., parm_n)

-- ****************** VARIABLES SECTION ************************
VAR

-- ***************** SYMBOL DEFINITION SECTION ******************
DEFINE

-- ****************** ASSIGNMENT SECTION ***********************
ASSIGN

-- ******************* End of Module1  *************************
```

Figure 10: SMV module program structure

Each SMV module can directly reference variables that are declared in other modules. Because our SMV programs represent translated UML models, we allow a module to reference only those variables that are declared as part of a *composite* module. A *composite* module can reference its components' variables, but the reverse is prohibited. For instance, the **Elevator** class (i.e., a composite object) includes a **Door** class. Thus, the elevator object can reference the door object and its variables, but the door object cannot directly reference the elevator object's variables. This preserves the object-oriented principle of encapsulation and information hiding. It also promotes a layered architecture and supports reusability of modules.

The DEFINE section specifies macro definitions. A *macro* is a name of a more complex expression, which can be used in place of the expression in SPEC and ASSIGN sections. Macros can help to localize changes to complex conditions and can be reused in various parts of the program.

The ASSIGN section specifies initial and succeeding values of the module's variables. Variables are initialized in **`init`** statements and are updated in **`next`** statements. Next statements can *be non-deterministic,* in which case, the variable's value is selected at random. *Non-determinism* is useful to describe models where design decisions are deferred. Similarly, if a variable is not

updated by any **`next`** statements, then SMV can non-deterministically choose any legal value of that variable. This is typically used to model environment variables whose values are not controlled by the system.

In this thesis, we assume that the UML state diagrams do not have *conflicting* transitions (i.e., simultaneously enabled transitions from the same source-state). Otherwise, the translated state machine will behave differently because SMV assumes a default ordering when evaluating conditional statements that may be overlapping, while the UML model does not say anything on how to resolve the conflicting transitions.

For each of these SMV sections, we propose a list of subsections and corresponding naming conventions. These are added to our proposed *template* for readability, traceability and consistency of SMV models. In the next section, we will map all the UML classes and state diagrams into the SMV template's sections and subsections.

3.2 Translating the Elevator Example

In this section, we incrementally present how to translate a UML model into SMV, starting with the lowest-level classes[12] (e.g., **Button** and **Timer**). Then, we continue with the composite classes (e.g., **Door** and **Elevator**), ending with the top-level composite class (i.e., **Building**) which represents the entire system. By taking a bottom-up approach, a module is translated before it is used to declare instantiations of the module.

We translate each class into a distinct SMV module as follows. The class name is used to derive the SMV module name. Class attributes and states are declared as variables in the VAR section. Active states, the state transitions' enabling conditions, the class's output events and complex class operation conditions are defined as macros in the DEFINE section. Finally, changes in class attribute values and state changes are described in the ASSIGN section. Because our aim is to describe the concepts, steps and conventions used to translate UML classes and state diagrams into SMV, we limit our discussion to those classes in the elevator example that exhibit our proposals. The complete SMV program for the elevator example is given in Appendix C.

3.2.1 Translating the Button Class

In this section, we demonstrate how we translate the **Button** class, which is a superclass of **RequestBn** (see Figure 4 of Section 2.2.1)[13]. The SMV translation of the Button state diagram (Figure 9, Section 2.3.2) is shown in Figure 11. We show the line numbers and the translation rule names beside each SMV program statement in order to aid our discussion. The line numbers and rule names are not part of the resultant SMV program.

Line 4 of Figure 11 declares the Button SMV module using the MODULE keyword followed by the Button class name. The general translation rule is

---

[12] Lowest-level classes do not have component classes.

[13] In the elevator example, we do not have an example of an abstract class. However, we recommend translating abstract classes before concrete classes are translated. Even though abstract classes cannot be directly instantiated, we translate them into SMV because a translated abstract class will be reused (with or without modification) when translating a concrete subclass. This helps to ensure that we use a consistent naming convention for all objects that inherit the abstract class's properties.

Rule M1: Declaring Module Names

Description: Modules are named after class names in the UML main class diagram. Blank characters found in the class names are replaced by underscores (i.e., "_").

```
MODULE class_name()
```

Line	Rule	Program
1		`-- ************************************************************`
2		`-- *                     MODULE Button                          *`
3		`-- ************************************************************`
4	M1,M2	`MODULE Button(inp_BN_PRESSED)`
5		
6		`-- ***************** VARIABLES SECTION ***********************`
7		`VAR`
8		` -- *** STATE VARIABLES SUBSECTION ***`
9	S1a	` st_Button: { Idle };`
10		
11		`-- **************** SYMBOL DEFINITION SECTION ****************`
12		`DEFINE`
13		` -- *** ACTIVE STATE MACROS SUBSECTION ***`
14	S2a	` in_Button := 1;`
15	S2b	` in_Idle := in_Button & st_Button = Idle;`
16		
17		` -- *** TRANSITION MACROS SUBSECTION ***`
18	T1	` t1 := in_Idle & inp_BN_PRESSED;`
19		
20		` -- *** GENERATED EVENTS MACROS SUBSECTION ***`
21	E1a	` ev_REQ := t1;`
22		
23		`-- ***************** ASSIGNMENT SECTION **********************`
24		`ASSIGN`
25		` -- *** STATE VARIABLES SUBSECTION ***`
26	S3	` init(st_Button) := Idle;`
27	S4	` next(st_Button) := case`
28		`   t1 : Idle;`
29		`   1: st_Button;`
30		`  esac;`
31		`-- ********************* end of Button ***********************`

Figure 11: Button SMV module

In the UML main class diagram, the **Button** class has an input event BN_PRESSED. We translate input events as module input parameters. By convention, we prefix with "inp_" each of the class's input events and add it as a formal parameter to the module name (Line 4). The general translation rule is

Rule M2: Declaring Module Input Parameters Using Event Names

```
-- For each input event input_event_1 .. input_event_n declared in the class
name section of the class:

MODULE class_name(inp_input_event_1 , .. , inp_input_event_n)
```

To see how our UML event processing definition can be translated to SMV, we need to describe the UML to SMV mapping. The SMV model checker uses a *round-based execution model*. This execution model starts with an initialization round followed by a sequence of update rounds. In each update round, the system executes all the enabled assignment statements in parallel. One SMV *round* is mapped to one UML *step*.

The rest of the SMV translation refers to the Button's state diagram. In Figure 9, the Button state diagram has an explicit root-state, which is an or-state with a single substate called Idle. Or-states are translated as an enumeration of its substates. Atomic states are not translated into SMV variables; they only appear as values of their superstate variable. If the root-state is not explicitly shown in the UML state diagram, we name root-states after their class names (expressed in initial uppercase). Thus, Line 9 of the SMV translation declares the root-state variable st_Button as an enumeration of the Button's substates. The general translation rule is

Rule S1: Declaring the States of Module's State Machine

S1a: Or-states

Description: We declare an or-state as an enumeration of its substates. For implied root-states, we use the class name as the state name. An explicit "Undefined" value could be added to the enumeration of substates.

```
-- For all substates s1, .. ,sn of the or_state:

VAR
 -- *** STATE VARIABLE SUBSECTION ***
 st_or_state : { s1, .. , sn, [Undefined] };
```

For each state, we add a macro to describe the conditions that make that state active[14] (Lines 14 and 15). The substate of an or-state is active if its parent is active and its parent's value is equal to the substate. The root-state of an object is always active. By convention, each macro is named after its corresponding state name prefixed by "in_". The general translation rules are

[14] This translation rule is inspired by Chan's translation from RMSL to SMV [CAB+98]. There are other approaches to represent the hierarchy of states (e.g., using a Boolean variable to represent an atomic state and then derive the and- and or-states from these atomic states [Day93]).

Rule S2: Defining Active State Macros

S2a: Root-states

```
-- For each Class_Name defined in the VAR section:
DEFINE
 -- *** ACTIVE STATE MACROS SUBSECTION ***
 in_Class_Name := 1;
```

S2b: States that Belong to Or-states

```
-- For each substate that belongs to or_state:
DEFINE
 -- *** ACTIVE STATE MACROS SUBSECTION ***
 in_substate := in_or_state & st_or_state = substate;
```

To ease translation of each state transition and its effects, we define a transition macro that specifies the enabling conditions of the transition[15]. Each state transition macro is named after the UML transition name[16]. Since we are only concerned with deterministic state machines, we simply use a macro to specify whether the transition is enabled. For example, Line 18 defines the transition macro t1 as the conjunction of the source-state (i.e., in_Idle) and the event trigger inp_BN_PRESSED. Named transitions are defined in the "transition macros" subsection of the DEFINE section. The general translation rule is (note that square bracket delimit optional components of macro):

Rule T1: Defining Enabled Transitions

```
-- For each named transition named_trans with source-state source, trigger
event trig, guard condition cond:

DEFINE
 -- *** TRANSITION MACROS SUBSECTION ***
 named_trans := in_source [ & trig ] [& cond ];
```

Note: ***trig*** may be an external event, inter-object event or intra-object event. ***cond*** may be any expression, using attributes and states as predicates, that evaluates to true or false.

Each *generated event macro* is defined in terms of the set of transitions that generate a specific class's output event. In the Button state diagram (Figure 9), an event REQ is generated when transition t1 is executed. It is convenient to represent an event as a boolean because an event is either generated, indicated by boolean value true (1) or not generated, indicated by a boolean value false (0). A generated event macro evaluates to true whenever a transition that generates the

---

[15] In this context, transitions exclude the initial-state transition where the source-state is the initial pseudo-state. By convention, we do not assign names to any initial-state transition.

[16] Recall that we attach label names to transitions in our UML state diagrams.

event executes. Otherwise, the generated event macro evaluates to false. We prefix with "ev_" each of the output events defined in the event section of a class in the UML class diagram and add it in the "generated event macros" subsection of the DEFINE section[17]. Line 21 shows that if the transition macro t1 is executed, ev_REQ is generated (i.e., evaluates to 1). The general translation rule is

Rule E1: Specifying Output Events Without Parameters

Description: An output event that has no parameters evaluates to true (one) whenever a transition that generates it executes.

```
-- For each named transitions tran_1, .. , tran_n that generates the event
output_event:

DEFINE
 -- *** GENERATED EVENT MACROS SUBSECTION ***
 ev_output_event := tran_1 | .. | tran_n;
```

Turning now to the ASSIGN section, we formulate the *transition relations* that specify how variables are updated during execution. In deterministic state machines, the initial state of an or-state must be specified. This is shown in the UML state diagram as a start-state with a transition to the initial state. Line 26 uses the SMV **`init`** statement to initialize Button's root-state to Idle.

Rule S3: Initializing State of an Or-state

```
-- For each or_state with an init_state:
ASSIGN
 -- *** STATE VARIABLES SUBSECTION ***
 init(st_or_state) := init_state;  -- if the initial value is defined
 init(st_or_state) := Undefined;   -- if the initial value is not defined
```

A transition in a UML state machine causes the current active state to change to the transition's destination state. To model state transitions, we use the **`next`** statement to update the variable value, and use the **`case`** statement to specify a conditional update. In SMV, a case statement has a form similar to the case statement of programming languages such as C where each branch consists of an enabling condition and a set of statements that are executed if the enabling condition evaluates to true[18]. If the last branch of a case statement has an unconditionally true

---

[17] Instead of specifying output events as macros, it is possible to translate output events as boolean variables. However, this approach would unnecessarily increase the state space of the SMV program.

[18] In a case statement, SMV executes the statements of the first branch whose condition that evaluates to true. If the condition of more than one branch evaluate to true, only the statements of the first satisfied

enabling condition (i.e., numeric value 1), then the branch acts as an "else" clause and specifies default assignment. We use transition macros to specify the enabling conditions. Our translation guidelines suggest that all case statements have default clauses. Lines 27 to 30 assign the next state of the Button root-state (i.e., st_Button) using a case statement. If transition t1 is enabled, the Idle state becomes active (i.e., self-transition). Otherwise, Button's root-state stays in its current state. The general translation rule is

Rule S4: Updating Values of Or-States

```
-- For each or-state with substates substate_1, .. , substate_n and for each
substate_i with incoming transitions tran_i1, .. , tran_ik :

ASSIGN
 -- *** STATE VARIABLES SUBSECTION ***
 next(st_or_state) := case
   tran_11 | .. | tran_1k : substate_1;
   ..
   tran_i1 | .. | tran_ik : substate_i;
   ..
   tran_n1 | .. | tran_nk : substate_n;
   1: st_or_state;
  esac;
```

Note: If a transition causes the or-state to be left in an undefined state, we set the `st_or_state` to "Undefined".

This completes our translation of the Button SMV module.

3.2.2 Translating the RequestBn Subclass

We now discuss how we translate the **RequestBn** state diagram, illustrated in Figure 9 (on the right), into SMV. **RequestBn** is a subclass of **Button**, which means that it inherits **Button**'s behavior and most of its SMV translation. Because SMV does not have a built-in inheritance mechanism, we have to manually implement inheritance by copying all of the inherited variables and behavior into the subclass module. The SMV statements copied from Button SMV module are marked as Rule G1, which is generalized as follows:

Rule G1: Translating a Subclass

Description: When translating a subclass, we copy the contents of the superclass's module.

---

condition are executed. If all the conditions are false, then no statements are executed. Details of SMV syntax can be found in [McMi99].

Line	Rule	Program
1		-- **
2		-- * MODULE RequestBn *
3		-- **
4	M1,M2	**MODULE** RequestBn(inp_BN_PRESSED, inp_LIGHT, inp_UNLIGHT)
5		
6		-- ****************** VARIABLES SECTION ************************
7		**VAR**
8		-- *** STATE VARIABLES SUBSECTION ***
9	G1	st_Button: { Idle };
10	S1a	st_Light: { UnLit, Lit };
11		
12		-- *** CLASS ATTRIBUTES SUBSECTION ***
13	C1	pending : boolean;
14		
15		-- ***************** SYMBOL DEFINITION SECTION ******************
16		**DEFINE**
17		-- *** ACTIVE STATE MACROS SUBSECTION ***
18	G1	in_Button := 1;
19	S2c	in_Light := 1;
20		
21	G1	in_Idle := in_Button & st_Button = Idle;
22	S2b	in_UnLit := in_Light & st_Light = UnLit;
23	S2b	in_Lit := in_Light & st_Light = Lit;
24		
25		-- *** TRANSITION MACROS SUBSECTION ***
26	G1	t1 := in_Idle & inp_BN_PRESSED;
27	T1	t2 := in_UnLit & inp_LIGHT;
28	T1	t3 := in_Lit & inp_UNLIGHT;
29		
30		-- *** GENERATED EVENTS MACROS SUBSECTION ***
31	G1	ev_REQ := t1;
32		
33		-- ****************** ASSIGNMENT SECTION ***********************
34		**ASSIGN**
35		-- *** STATE VARIABLES SUBSECTION ***
36	G1	init(st_Button) := Idle;
37	G1	next(st_Button) := case
38		t1 : Idle;
39		1: st_Button;
40		esac;
41		
42	S3	init(st_Light) := UnLit;
43	S4	next(st_Light) := case
44		t2 : Lit;
45		t3 : UnLit;
46		1: st_Light;
47		esac;
48		
49		-- *** CLASS ATTRIBUTES SUBSECTION ***
50	C2	init(pending) := 0;
51	C3	next(pending) := case
52		t2 : 1;
53		t3 : 0;
54		1: pending;
55		esac;
56		-- ********************** end of RequestBn *********************

Figure 12 : RequestBn SMV module

Using Rule M1, we define a new module name RequestBn (Line 4 of Figure 12). Subclass-specific variables and behavior are added as new SMV statements. This does not violate the Liskov Substitution Principle (LSP) because it extends the behavior of the superclass. The module declaration (Line 4) is extended to include the two new input events LIGHT and UNLIGHT. We also added new state macros and new transition macros using previously described translation rules.

Note that in Line 18 the definition of st_Button did not change. This follows the LSP [Lisk88], in that we extend the parent (base) class instead of deleting or replacing the properties of the parent (base) class. Whenever an and-state is active, all of its substates are active, so it is not necessary to declare an SMV variable to keep track of RequestBn's substates. The general rule is

Rule S1: Declaring the States of Module's State Machine

S1b: And-states

Description: We do not declare the substates of an and-state.

The concurrent region of an and-state is active if and only if the parent-state is active. Thus, in Line 19, the active state macro for Light region defines the region as active whenever the parent-state of Light region is active. Since the RequestBn root-state is always active, we assign in_Light macro a value of 1. The general rule is

Rule S2: Defining Active State Macros

S2c: States That Belong to And-states

```
-- For each and-state and_state, where parent_state is the parent state of
and_state:

DEFINE
 -- *** ACTIVE STATE MACROS SUBSECTION ***
 in_and_state := 1;                       -- if parent state is a root-state
 in_and_state := in_parent_state;         -- if parent state is not a root-state
```

Using Rules S3 and S4, we assign the initial and next state of Light region (Lines 42-47). **RequestBn** class's *pending* attribute is modeled as an SMV variable in the "class attributes" subsection of the VAR section, as shown in Line 13. By convention, we do not add any prefix to

each of the class attributes. The variable's type is the same as defined in the **RequestBn** class, which means that attributes are restricted to SMV data types[19].

Rule C1: Declaring Class Attributes

```
-- For each class attribute attrib in the class being translated:
VAR
 -- *** CLASS ATTRIBUTES SUBSECTION ***
 attrib : boolean;  -- if data type is boolean
 attrib : i_1..i_n; -- if data type is integer range from i_1..i_n
 attrib : {e1, .., en};    -- if data type is enumerated with values e1, .., en
 attrib : array 1..n of scalar;  -- if data type is an array of scalars
```

We initialize a class attribute to its default value set in the UML class diagram or to the value assigned in the initial state transition in the UML state diagram. If the default is not defined, we omit an initial assignment and let SMV explore all possible initial values. We initialize variables using the **`init`** statement of SMV in the "class attributes" subsection of the ASSIGN section. Line 50 initializes attribute *pending* to zero, as defined in the **RequestBn** subclass (Figure 4 of Section 2.2.1). The general translation rule is

Rule C2: Initializing Class Attributes

```
-- For each class attribute attrib declared in the VAR section of the SMV
module:

ASSIGN
 -- *** CLASS ATTRIBUTES SUBSECTION ***
 init(attrib) := default_attrib_value;  -- from class diagram
 init(attrib) := initial_state_value;   -- from initial state
```

Class attributes in UML model can be assigned new values: a) in actions performed by a transition within the object, b) in operations performed within the object, c) or in actions or operations performed by the parent object. If an attribute's value changes as a result of a transition action, we use the transition's macro as the enabling condition for the assignment. For example, Lines 51-55 update the *pending* attribute of the **RequestBn** subclass. An attribute keeps its current value if a transition does not affect its value (Line 56). The general translation rule is

---

[19] We restrict our UML classes to define finite (scalar) values in the class attributes. Complex attributes must be modeled as an association (link) to another class.

Rule C3: Updating Values of Class Attributes

Rule C3a: Actions performed when a transition is enabled within the object.

```
-- For each transition tran_1, .. , tran_n that changes the value of attribute
attrib to expression_1, .. , expression_n as described in transition actions:

ASSIGN
 -- *** CLASS ATTRIBUTES SUBSECTION ***
 next(attrib) := case
   tran_1 : expression_1;
   ..
   tran_n : expression_n;
   1: attrib;
  esac;
```

Note: `tran_1 .. tran_n` could be an incoming transition that may trigger actions defined in the initial pseudo-state transition. This covers the case where actions on initial transitions are initialized whenever an incoming transition becomes true.

This completes our translation of the RequestBn SMV module.

3.2.3 Translating the External_Event Class

We translate the **External_Event** class (Figure 4 of Section 2.2.1) into a separate SMV module (Figure 13) that declares and constrains the environmental events.

Line	Rule	Program
1		`-- ******************************************************************`
2		`-- *                     MODULE External_Event                      *`
3		`-- ******************************************************************`
4	M1	`MODULE External_Event()`
5		
6		`-- ****************** VARIABLES SECTION *************************`
7		`VAR`
8		` -- *** ENVIRONMENT VARIABLES SUBSECTION ***`
9	X1	` env_AT_FLOOR : boolean;`
10	X1	` env_BP_OpenBn : boolean;`
11	X1	` env_BP_CloseBn: boolean;`
12	X1	` env_BP_FloorReqBn  : array 1..3 of boolean;`
13	X1	` env_BP_ElevReqBnUp : array 1..2 of boolean;`
14	X1	` env_BP_ElevReqBnDn : array 2..3 of boolean;`
15		
16		`-- ********************* FAIRNESS SECTION ***********************`
17		`FAIRNESS`
18	X2	`  env_AT_FLOOR`
19		
20		`-- ******************** end of External_Event *******************`

Figure 13 : External_Event SMV module

We define a distinct SMV variable for each distinct environmental event, so that SMV can distinguish among the events and so that multiple environmental events can occur simultaneously. For example, each of the declarations in Lines 10-14 defines a unique

BN_PRESSED event. We declare each simple environmental event (i.e., one without parameters) as boolean. A boolean value of 1 represents the event being generated. For multiple environmental events that share the same name and differ only in the index value (e.g., BP_FloorReqBn[1..3]), we declare them as an array of boolean variables. By convention, we prefix with "env_" each of External_Event class's output events. The general translation rule is

Rule X1: Declaring External Events

Description: An external event of name **`external_event`** that has no parameters is declared as a boolean variable named **`env_external_event`** in the "environment variables" subsection of the VAR section.

```
VAR
 -- *** ENVIRONMENT VARIABLE SUBSECTION ***
 env_external_event : boolean;                -- for external_event
 env_external_event : array j..k of boolean;  -- for external_event[j..k]
```

In this module, we specify any constraints or assumptions about the values of environmental variables and events. By default, SMV searches all possible ways, in which the environmental events can occur, including extreme cases where they occur continuously or they do not occur at all. In order to force a given environmental event to occur regularly, we use a *fairness constraint*. A fairness constraint restricts the attention of the SMV model checker to only those execution paths along which an environmental event will regularly occur. We add each of the environmental events that need to regularly occur to the FAIRNESS section of the SMV module. One of the system properties states that the elevator will eventually service a pending request. This property initially failed because the environment is not forced to regularly generate the AT_FLOOR event, which meant there is an execution path in which the elevator is stuck in state Moving, indefinitely waiting to reach the next floor. To correct this, we forced the AT_FLOOR event to regularly occur by adding env_AT_FLOOR (Line 18) to the FAIRNESS section of the External_Event SMV module. The general translation rule for adding a fairness constraint is

Rule X2: Adding Fairness Constraints

Description: Events that are assumed to occur regularly must be added to the FAIRNESS section.

```
-- For external events external_event_1 ,.., external_event_k that need to
occur regularly:

FAIRNESS
 env_external_event_1
..
FAIRNESS
 env_external_event_k
```

In order to constrain SMV from changing a set of variables at the same time, we use the TRANS declaration to specify a constraint on the system's transition relations. We now describe two ways of using TRANS conditions.

In some reactive systems, the environment may be constrained by physical devices outside the control of the software system. For example, a physical device generates the AT_FLOOR event when a floor is reached. To ensure that there is sufficient time between AT_FLOOR events to update the system's variables, we add a constraint on variable values that prevents the event from re-occurring if the system is executing elevator module's transition t10 in response to the event's previous occurrence. We add the TRANS condition in the SMV module where the variables of the TRANS proposition can be accessed. Since the Building class can access variables from both External_Event and Elevator objects, we add the TRANS condition in the **Building** class's SMV module **`main`** (Line 52 of Figure 18).

```
TRANS
 !(external_Event.env_AT_FLOOR & elevator.t10)
```

If we wanted to assume that a passenger could not push all the elevator buttons at the same time, it would be expressed as the following constraint in the External_Event SMV module.

```
TRANS
 !( env_BP_FloorReqBn[1] & env_BP_FloorReqBn[2] & env_BP_FloorReqBn[3] &
env_BP_ElevReqBnUp[1] & env_BP_ElevReqBnUp[2] & env_BP_ElevReqBnDn[2] &
env_BP_ElevReqBnDn[3] )
```

The general translation rule is

Rule X3: Adding TRANS Conditions

```
-- For conditions condition_1 ,.., condition_k that constrains the environment
variables:

TRANS
 condition_1
..
TRANS
 condition_k
```

Using the TRANS condition is a convenient way of constraining the environment without specifying the exact behavior of the environment. Most of the time, we do not expect the environmental events to be modeled as part of a UML state diagram. However, if the behavior of the environment is constrained, the "environment variables" subsection in the ASSIGN section can be used to simulate the generation of the environmental events.

3.2.4 Translating Timer Class

In reactive systems, timing requirements are essential to the proper behavior of the system. In UML, it is convenient to use a *timeout* event, denoted by *tm*(interval), without explicitly specifying timer objects. Since SMV does not have primitives to support a *timeout* event, we specified a Timer class with SET, UNSET and TM events in our UML model.

The Timer class (Figure 4) has a *counter* attribute that represents the timeout interval. The Timer state diagram (Figure 14) starts in state Idle and waits until it is activated by a SET event in transition t1. While in state Count_Down, the *counter* attribute is decremented in each execution step until it reaches 0 (transition t2), at which point the timer transitions to state Idle and the timeout event TM is broadcast (transition t3). If the timer receives a UNSET event while it is counting down, the *counter* is set to 0 and the timer transitions to state Idle (transition t4). This timer can also be reset while it is counting down (transition t5).

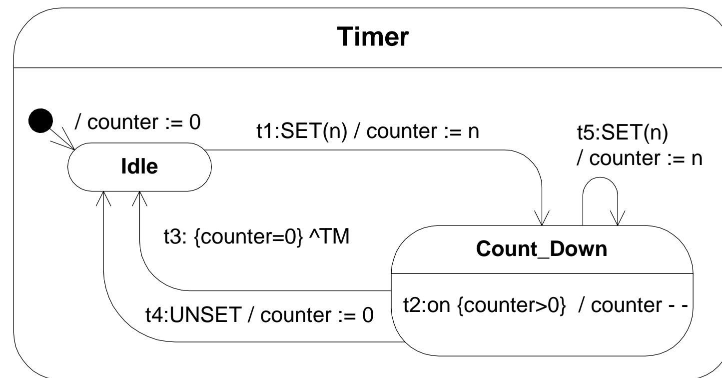


Figure 14: Timer state diagram

Events that have parameters need to be modeled as SMV modules. One can think of such events as classes that have attributes. The SET event with parameter n is modeled as a structured data type where the parameter n is declared as one of the individual data attributes. Using Rules M1 and C1, the SET event with parameter *n* is translated into an SMV module (without any input parameters and no assignments) as shown in Figure 15.

Line	Rule	Program
1		`-- ****************************************************************`
2		`-- *                     MODULE SET                                 *`
3		`-- ****************************************************************`
4	M1,**E2**	`**MODULE** SET()`
5		
6		`-- ***************** VARIABLES SECTION ************************`
7		**`VAR`**
8		`  -- *** CLASS ATTRIBUTES SUBSECTION ***`
9	**E2**	`  raised : boolean;`
10	**E2**	`  n : {5};`
11		
12		`-- *********************** End of SET *************************`

Figure 15: SET Event SMV module

By convention, we add a special *raised* attribute (Line 9 of Figure 15) which indicates if the event occurred (i.e., evaluates to true). Line 10 declares the value of parameter n to be 5 all the time. If the SET event is used with different valued parameters, we can declare n to accept a bigger range of appropriate values. The general translation rule is

Rule E2: Declaring Output Events with Parameters

```
-- For each output event output_event, with parameters parm_1,.., parm_k:

MODULE output_event()

VAR
 -- *** CLASS ATTRIBUTES SUBSECTION ***
 raised : boolean;
 parm_1 : data_type;
 ..
 parm_k : data_type;
```

In the previous subsections, we have translated the **Button** class, **RequestBn** subclass and **External_Event** class by showing the complete SMV module for each of the classes. Starting from this subsection, all further SMV translations are not completely shown. We show a partial listing and highlight the lines of specification pertaining to the new translation rules being defined.

Using Rules M1 and M2, we define a new SMV module named `Timer` with input events inp_SET and inp_UNSET (Line 4 of Figure 16).

Line	Rule	Program
1		`-- ****************************************************************`
2		`-- *                        MODULE Timer                         *`
3		`-- ****************************************************************`
4	M1,M2	**`MODULE`** `Timer(inp_SET, inp_UNSET)`
5		
6		`-- ****************** VARIABLES SECTION ************************`
7		**`VAR`**
..		
11		`-- *** CLASS ATTRIBUTES SUBSECTION ***`
12	C1	`counter: 0..5;`
13		
14		`-- ***************** SYMBOL DEFINITION SECTION *****************`
15		**`DEFINE`**
..		
21		`-- *** TRANSITION MACROS SUBSECTION ***`
22	**T2**	`t1 := in_Idle &` **`inp_SET.raised`**`;`
23	T1	`t2 := in_Count_Down & counter > 0;`
24	T1	`t3 := in_Count_Down & counter = 0;`
25	T1	`t4 := in_Count_Down & inp_UNSET;`
26	**T2**	`t5 := in_Count_Down &` **`inp_SET.raised`**`;`
..		
31		`-- ****************** ASSIGNMENT SECTION ***********************`
32		**`ASSIGN`**
..		
41		`-- *** CLASS ATTRIBUTES SUBSECTION ***`
42	C2	`init(counter) := 0;`
43	C3	`next(counter) := case`
44		`  t1 | t5 :` **`inp_SET.n`**`;`
45		`  t2 : counter - 1;`
46		`  t4 : 0;`
47		`  1: counter;`
48		` esac;`
49		`-- ****************** End of Timer *************************`

Figure 16: Partial listing of Timer SMV module

When translating a class attribute, whose data type has no finite boundary value, we choose the smallest integer-range that the system requires[20]. We found that in any of the state diagrams that generate the SET event, the largest value of parameter *n* used was 5. Thus, we declare counter attribute with integer range from 0..5 (see Line 12 of Figure 16).

The Timer state diagram (Figure 14) shows that transition t1 and t5 are triggered by a SET event with a parameter n. The parameter n is needed to set the number of time units required for counting down. To access a structured data type, we specify the variable name followed by a dot, followed by the attribute. Lines 22 and 26 of Timer SMV module (Figure 16) show how the

[20] Instead of defining the smallest integer range, it is possible to define an arbitrarily larger integer range. However, this approach would unnecessarily increase the state space of the SMV program.

special *raised* attribute can be accessed when we define transition macros t1 and t5. The general translation rule is

Rule T2: Defining Transitions with Event Triggers that Carries Parameters

```
-- For each named transition named_trans with source-state source, triggered by
an event with parameters trig:

DEFINE
 -- *** TRANSITION MACROS SUBSECTION ***
 named_trans := in_source [ & trig.raised ];
```

To complete the translation of transition t1 and t5, Line 44 of Figure 16 shows how we assign the value of SET event's parameter n (i.e., inp_SET.n) to the *counter* attribute of the Timer SMV module.

3.2.5 Translating Door Class

Line	Rule	Program
1		`-- **************************************************************`
2		`-- *                   MODULE Door                              *`
3		`-- **************************************************************`
4	M1	`MODULE Door(inp_OPEN, inp_KEEP_OPEN, inp_CLOSE)`
5		
6		`-- ****************** VARIABLES SECTION *************************`
7		`VAR`
8		` -- *** STATE VARIABLES SUBSECTION ***`
9	S1a	` st_Door: {Close, Open};`
10		
11		` -- *** CLASS ATTRIBUTES SUBSECTION ***`
12	C1	` retry : 0..3;`
13		
14		` -- *** GENERATED EVENTS SUBSECTION ***`
15	**E3**	` ev_SET : SET;`
16		
17		` -- *** COMPONENT OBJECTS SUBSECTION ***`
18	**L1**	` DoorTimer: Timer(ev_SET, ev_UNSET);`
19		
20		`-- ***************** SYMBOL DEFINITION SECTION ******************`
21		`DEFINE`
..		
27		` -- *** TRANSITION MACROS SUBSECTION ***`
28	T1	` t1 := in_Close & inp_OPEN > 0;`
29	T1	` t2 := in_Open & DoorTimer.ev_TM = 1 & !(inp_OPEN > 0);`
30	T1	` t3 := in_Open & inp_CLOSE & !(inp_OPEN > 0);`
31	T1	` t4 := in_Open & inp_KEEP_OPEN & retry > 0 & !(t2 | t3);`
..		
37		` -- ***************** ASSIGNMENT SECTION **********************`
38		`ASSIGN`
..		
47		` -- *** GENERATED EVENTS SUBSECTION ***`
48	E4	` ev_SET.raised := t1 | t4;`
49	E4	` ev_SET.n := 5;`
..		
60		`-- ***************** End of Door *********************`

Figure 17: Partial listing of Door SMV module

The partial listing of the SMV translation for the Door state diagram (Figure 8 of Section 2.3.1) is shown in Figure 17. Note that not all input events used in the Door state diagram illustrated in Figure 8, are translated as input parameters for the Door SMV module. Events generated by a module's component objects (e.g., DoorTimer's ev_TM event in Line 29 of Figure 17) are not translated as input parameters for the module because a composite object can directly access the variables of its component modules.

An event with parameters is prefixed with "ev_" and declared in the "generated events" subsection of the VAR section. In Line 15 of Figure 17, we instantiated the variable ev_SET with the user-defined SET event SMV module. The general translation rule is

Rule E3: Instantiating Events with Parameters

Description: An event that has parameters is declared as a user-defined SMV module.

```
-- For each event output_event with a an SMV module output_event_module:

VAR
 -- *** GENERATED EVENT SUBSECTION ***
 ev_output_event : output_event_module;
```

Line 48 shows that if either transition macro t1 or t4 is executed, ev_SET.*raised* is generated (i.e., evaluates to 1). The parameter *n* of SET event is fixed to five time units (Line 49 of Figure 17), since that is the only parameter value of SET used in the UML model. The general translation rule is

Rule E4: Assigning Values to Events with Parameters

```
-- For each named transitions tran_1, .. , tran_n that generates the event
output_event, with parameters parm_1,.., parm_k and corresponding expressions
expression_1,.., expression_k:

ASSIGN
 -- *** GENERATED EVENT SUBSECTION ***
 ev_output_event.raised := tran_1 | .. | tran_n;
 ev_output_event.parm_1 := expression_1;
 ..
 ev_output_event.parm_k := expression_k;
```

We now describe how we translate UML *composition* relationships into SMV. In UML, a *component object* belongs to one and only one *composite object.* Line 18 of Figure 17 declares

the DoorTimer (component object) as an instantiation of the Timer SMV module[21]. The DoorTimer's parameters, which serve as the Timer's input events, are ev_SET and ev_UNSET events that are generated by the Door SMV module. We declare the component objects in the "component objects" subsection of the VAR section. By convention, we use the object name from the UML object model as the *component variable name*. The general translation rule is

Rule L1: Instantiating a Single Component Class

```
-- For each component object name component_part that is an instance of user-
defined module Component_Module with actual input parameters var_1, .., var_n:

VAR
 -- *** COMPONENT OBJECTS SUBSECTION ***
 component_part : Component_Module(var_1, .., var_n);
```

3.2.6 Translating the Elevator Class

In this section, we first discuss variations of declaring module parameters. Then, we discuss translation rules for some variations of declaring component objects. A partial listing of the SMV translation of the Elevator state diagram (Figure 7 of Section 2.3.1) is shown in Figure 18.

Our translation rules for UML semantics do not allow objects to directly access attribute values from non-component objects. In our elevator case study, Elevator class's getNextDest operation needs to access the *pending* attributes of ElevReqBn objects, which are not accessible via a component object. The UML main class diagram (Figure 4 of Section 2.2.1) shows that the hall buttons (i.e., ElevReqBn objects) are components of the **Building** class. In order for the **Elevator** class to access the *pending* attribute of each of the hall buttons, hall button objects are passed from the Building SMV module to the Elevator SMV module as input objects. Line 4 of Figure 18 shows how the four hall buttons (ElevReqBnUp1, ElevReqBnUp2, ElevReqBnDn2, ElevReqBnDn3) are declared as input parameters of the Elevator SMV module. By convention, we prefix with "inp_" the object name and add it as a formal parameter to the module name. The general translation rule is

---

[21] Recall that the Timer SMV module can only be set with a maximum of five time units. If the maximum value of parameter *n* is changed from five to a larger number, the Timer SMV module's *counter* attribute declaration will need a larger integer range.

Rule M3: Declaring Module Input Parameters For External Attributes

```
-- For non-component objects object_name_1 .. object_name_n whose attributes
are referenced in an external class class_name:

MODULE class_name(inp_object_name_1, .. , inp_object_name_n)
```

Line	Rule	Program
1		`-- ****************************************************************`
2		`-- *                        MODULE Elevator                        *`
3		`-- ****************************************************************`
4	M1,M2 ,M3	**MODULE** `Elevator(inp_KEEP_OPEN_REQ, inp_BP_OpenBn, inp_BP_CloseBn, inp_BP_FloorReqBn1, inp_BP_FloorReqBn2, inp_BP_FloorReqBn3, inp_AT_FLOOR, inp_ElevReqBnUp1, inp_ElevReqBnUp2, inp_ElevReqBnDn2, inp_ElevReqBnDn3)`
5		
6		`-- ****************** VARIABLES SECTION ************************`
7		**VAR**
8		`-- *** STATE VARIABLES SUBSECTION ***`
..		
12	S1a	`st_Moving : { Init, Move_Up, Move_Down,` **Undefined** `};`
..		
23		`-- *** COMPONENT OBJECTS SUBSECTION ***`
..		
26	L2	`FloorReqBn1 : RequestBn(inp_BP_FloorReqBn1, ev_LIGHT_FloorReqBn[1], ev_UNLIGHT_FloorReqBn[1]);`
27	L2	`FloorReqBn2 : RequestBn(inp_BP_FloorReqBn2, ev_LIGHT_FloorReqBn[2], ev_UNLIGHT_FloorReqBn[2]);`
28	L2	`FloorReqBn3 : RequestBn(inp_BP_FloorReqBn3, ev_LIGHT_FloorReqBn[3], ev_UNLIGHT_FloorReqBn[3]);`
29		`door: Door(ev_OPEN, ev_KEEP_OPEN, ev_CLOSE);`
30		
31		`-- ***************** SYMBOL DEFINITION SECTION *****************`
32		**DEFINE**
33		`-- *** ACTIVE STATE MACROS SUBSECTION ***`
..		
49		`-- *** TRANSITION MACROS SUBSECTION ***`
50	O1	`t1_FloorReqBn1 := FloorReqBn1.ev_REQ & !(in_Open_Doors & floor=1);`
51	O1	`t1_FloorReqBn2 := FloorReqBn2.ev_REQ & !(in_Open_Doors & floor=2);`
52	O1	`t1_FloorReqBn3 := FloorReqBn3.ev_REQ & !(in_Open_Doors & floor=3);`
53	O1	`t1 := in_Waiting & (t1_FloorReqBn1 | t1_FloorReqBn2 | t1_FloorReqBn3);`
..		
97		`-- *** OPERATION MACROS SUBSECTION ***`
98		
99		`-- *** getNextDest and getNextDir operations ***`
100	O2	`m_f1_pending := (inp_ElevReqBnUp1.pending | FloorReqBn1.pending);`
101	O2	`m_f2_pending_up := (inp_ElevReqBnUp2.pending | FloorReqBn2.pending);`
102	O2	`m_f2_pending_dn := (inp_ElevReqBnDn2.pending | FloorReqBn2.pending);`
103	O2	`m_f3_pending := (inp_ElevReqBnDn3.pending | FloorReqBn3.pending);`
..		
159		`-- ****************** ASSIGNMENT SECTION ***********************`
160		**ASSIGN**
161		
162		`-- *** STATE VARIABLES SUBSECTION ***`
..		
184	S3	`init(st_Moving) :=` **Undefined**`;`
185	S4	`next(st_Moving) := case`

```
 186            t5 : Init;
 187            t6 | t7 : Move_Up;
 188            t8 | t9 : Move_Down;
 189            t10 : Undefined;
 190            1 : st_Moving;
 191           esac;
 192
 193          -- *** CLASS ATTRIBUTES SUBSECTION ***
  ..
 210      C2   init(destFl) := 0;
 211     C3b   next(destFl) := case
 212            t4 : 0;
  ..
 221            -- ****************  going up ***************
 222            (t2) & (m_up_f1_t1_1 | m_up_f1_t1_n | m_up_f2_t1 | m_up_f3_t1):
         1;
 223            (t2) & (m_up_f1_t2_sd | m_up_f1_t2_od | m_up_f2_t2_1 |
         m_up_f2_t2_n | m_up_f2_t2_od | m_up_f3_t2_od | m_up_f3_t2_sd): 2;
 224            (t2) & (m_up_f1_t3 | m_up_f2_t3 | m_up_f3_t3_1 | m_up_f3_t3_n ):
         3;
  ..
 279         -- ****************** End of Elevator **********************
```

Figure 18: Partial listing of Elevator SMV module

The Elevator object contains multiple instances of component RequestBn objects. When instantiating multiple component classes such as FloorReqBn[1..3], we declare each object separately because the actual parameters of each object are distinct (Lines 26 to 28 of Figure 18). If the component objects have the same input parameters, we can specify the *component variables* as an array of instances of a user-defined SMV module (no example shown). The general translation rule is

Rule L2: Instantiating Multiple Component Classes

```
-- For k instantiations of user-defined module Component_Module object with
link_role_name and actual input parameters var_1, .., var_n:

VAR
 -- *** COMPONENT OBJECTS SUBSECTION ***
 -- if at least one actual input parameter is not shared by all objects:
 link_role_name_1: Component_Module(var_1, .., var_n);
 ..
 link_role_name_k: Component_Module(var_1, .., var_n);

 -- if every actual input parameters is shared by all objects:
 link_role_name: array 1..k of Component_Module(var_1, .., var_n);
```

UML transitions with the same source and destination states and whose event names differ only in their source object can sometimes be expressed as a single transition with a compact transition label. Such a transition label consists of a quantified formula over multiple source objects. We translate a quantified formula into a series of SMV macros, with each macro referring to a single instantiation of the formula. For example, Elevator state diagram's transition t1 has a quantified

event that we expand into three logical transitions, one for each possible event source (Lines 50-52). Line 53 defines a macro that combines the three sub-macros. The general translation rule is

Rule O1: Adding Transition Macros to Expand Quantified Formulas

```
-- For each transition tran with quantified transition label of the form
"Forall x: source[x].trigger_event {condition(x)}" where x ranges from n to m:

DEFINE
 -- *** TRANSITION MACROS SUBSECTION ***
 tran_source_n := source_n.trigger_event & cond_n;
 ..
 tran_source_m := source_m.trigger_event & cond_m;
```

We now turn to translating UML operations, which is straightforward if the UML operations are defined as assignment statements as discussed in Section 2.3.4. Such UML operations are modeled as SMV macros, and we prefix each macro with "m_" and add it to the "operation macros subsection" of the DEFINE section. For example, Lines 100-103 were translated directly from the getNextDest operation macros described in Section 2.3.4. We note that these conditions are mutually exclusive. Therefore, the order of the conditions in the *DestFl* case statement (Lines 210-224) does not affect the desired behavior. The general translation rule is

Rule O2: Adding SMV Macros to Simplify Complex Macros

```
-- For each UML macro name macro_name with its corresponding macro condition
cond defined in the UML operation

DEFINE
 -- *** OPERATION MACROS SUBSECTION ***
 m_macro_name := cond;
```

In Section 2.3.4, our UML operations are modeled such that it only takes one execution step to assign the next value of a class attribute. As such, each UML operation is completed after one execution step. Any new events can immediately run after the UML operation is completed. A complex UML operation where multiple conditions could change the class attribute's value, is shown as a decision table. To convert such complex UML operations into SMV, we specify the next value of the class attribute using **`next`** and **`case`** statements. Different cases define different branches in the case statement. For example, Line 222 shows that the next value of *destFl* attribute will be assigned with a numeric value 1. The assignment happens if transition t2 is enabled and at least one of the macros m_up_f1_t1_1, m_up_f1_t1_n, m_up_f2_t1, m_up_f3_t1 is true. The general translation rule is:

Rule C3: Updating Values of Class Attributes

Rule C3b: UML operation performed after a transition is executed.

```
-- For each transition tran_1, .. , tran_n and a set of conditions condition_1,
.. , condition_n that changes the value of attribute attrib to expression_1, ..
, expression_n as described in transition actions:


ASSIGN
 -- *** CLASS ATTRIBUTES SUBSECTION ***
 next(attrib) := case
   tran_1 & condition_1 : expression_1;
   ..
   tran_n & condition_n : expression_n;
   1: attrib;
  esac;
```

When translating the Elevator state diagram into SMV, we found that some state variables (i.e., state Stop and state Moving) do not have a defined value in some configurations. For example, if the system is not in the state Moving, st_Moving is undefined. Using Rule S1a, we add an explicit “Undefined” value in st_Moving’s enumerated values (Line 12 of Figure 18). Using Rule S3, we set the initial value of st_Moving to Undefined (Line 184 of Figure 18) because state Moving is initially undefined since the elevator starts in state Stop. Lines 184 – 191 show under what circumstances the state Moving is entered and how state Moving becomes “undefined” when transition t10 is executed. The complete Elevator SMV module is shown in Appendix C.

3.2.7 Putting it all together: Main SMV Module

After all the UML state diagrams are translated, we declare our **`main`** SMV module, which represents the entire system. In our elevator example, the **Building** class is translated into the **main** SMV module (Figure 19). This module instantiates all the other top-level objects (i.e., External_Event, Elevator and ElevReqBn objects) found in the main class diagram (Lines 9, 15-20). By instantiating the hall buttons in the **`main`** SMV module, we allow more than one elevator to share the hall buttons (e.g., ElevReqBnUp1). The **`main`** SMV module also specifies all of the CTL formulas in the SPEC section to be checked against the SMV model. The rest of the translated **`main`** SMV module is presented in Appendix C.

```
Line  Rule  Program
  1         -- ***************************************************************
  2         -- *                          MAIN MODULE                          *
  3         -- ***************************************************************
  4    M1   MODULE main
  5
  6         -- ****************** VARIABLES SECTION *************************
  7         VAR
  8          -- *** ENVIRONMENT VARIABLE SUBSECTION ***
  9    L1    external_Event : External_Event;
 10
 11          -- *** STATE VARIABLES SUBSECTION ***
 12          st_Building : { Idle };
 13
 14          -- *** COMPONENT OBJECTS SUBSECTION ***
 15    L1   elevator : Elevator(ev_KEEP_OPEN_REQ,
            external_Event.env_BP_OpenBn, external_Event.env_BP_CloseBn,
            external_Event.env_BP_FloorReqBn[1],
            external_Event.env_BP_FloorReqBn[2],
            external_Event.env_BP_FloorReqBn[3], external_Event.env_AT_FLOOR,
            ElevReqBnUp1, ElevReqBnUp2, ElevReqBnDn2, ElevReqBnDn3);
 16
 17    L2    ElevReqBnUp1 : RequestBn(external_Event.env_BP_ElevReqBnUp[1],
            ev_LIGHT_ElevReqBnUp[1], elevator.ev_UNLIGHT_ElevReqBnUp[1]);
 18    L2    ElevReqBnUp2 : RequestBn(external_Event.env_BP_ElevReqBnUp[2],
            ev_LIGHT_ElevReqBnUp[2], elevator.ev_UNLIGHT_ElevReqBnUp[2]);
 19    L2    ElevReqBnDn2 : RequestBn(external_Event.env_BP_ElevReqBnDn[2],
            ev_LIGHT_ElevReqBnDn[2], elevator.ev_UNLIGHT_ElevReqBnDn[2]);
 20    L2    ElevReqBnDn3 : RequestBn(external_Event.env_BP_ElevReqBnDn[3],
            ev_LIGHT_ElevReqBnDn[3], elevator.ev_UNLIGHT_ElevReqBnDn[3]);
 21
 22         -- ***************** SYMBOL DEFINITION SECTION ******************
 23         DEFINE
 ..
 63         -- ********* SYSTEM PROPERTIES (CTL FORMULAS) ******************
 ..
179         -- ****************** End of Main Module ***********************
```

Figure 19: Partial listing of `main` SMV module

3.3 Translation Summary

By incrementally translating using a bottom-up approach we construct an object-oriented SMV program structure with subsections that aid in the readability of SMV program. Using the bottom-up approach, we can perform unit verification after each class is translated. We summarize in Table 3 the guidelines and rules presented in this chapter and their mappings to SMV subsections.

SMV Section / Subsection	Rules
MODULE Section	Rule M1: Declaring Module Names Rule M2: Declaring Module Input Parameters Using Input Events Rule M3: Declaring Module Input Parameters for External Attributes Rule E2: Declaring Output Events with Parameters
VAR Section	G1: Translating a Subclass
ENVIRONMENT VARIABLES	Rule X1: Declaring External Events
STATE VARIABLES	Rule S1: Declaring States of Module's State Machine Rule S5: Translating Initial States with More than One Outgoing Transitions
CLASS ATTRIBUTES	Rule C1: Declaring Class Attributes
GENERATED EVENTS	Rule E3: Instantiating Events with Parameters
COMPONENT OBJECTS	Rule L1: Instantiating a Single Component Class Rule L2: Instantiating Multiple Component Classes
DEFINE Section	G1: Translating a Subclass
ACTIVE STATE MACROS	Rule S2: Defining Active State Macros
TRANSITION MACROS	Rule T1: Defining Enabled Transitions Rule T2: Defining Transitions with Event Triggers that Carries Parameters Rule O1: Adding Transition Macros to Expand Quantified Formulas
GENERATED EVENTS MACROS	Rule E1: Specifying Output Events without parameters
OPERATIONS MACROS	Rule O2: Adding SMV Macros to Simplify Complex Macros
ASSIGN Section	G1: Translating a Subclass
ENVIRONMENT VARIABLES	
STATE VARIABLES	Rule S3: Initializing State of an Or-State Rule S4: Updating Values of Or-States
CLASS ATTRIBUTES	Rule C2: Initializing Class Attributes Rule C3: Updating Values of Class Attributes
GENERATED EVENTS	Rule E4: Assigning Values to Events with Parameters
FAIRNESS Section	Rule X2: Adding Fairness Constraints
TRANS Section	Rule X3: Adding TRANS Declarations

Table 3: SMV program structure / rules mapping

Table 4 shows the naming conventions used to declare module names, variables and macros in SMV. The prefix aids in the readability of the resultant SMV program. The naming conventions help the analyst analyze counter-examples for large systems.

	Prefix	Example
Module		
Module name	no prefix	*RequestBn:* module name for RequestBn subclass
Module parameters	inp_EVENT_NAME	*inp_BN_PRESSED*: *input parameter* of RequestBn module
VAR Section		
External events	env_EVENT_NAME	*env_BP_OpenBn*: output event in External_Event
States with or-states as children	st_STATE_NAME	*st_Operation*: substate of Elevator state diagram
Events generated by the module	ev_EVENT_NAME	*ev_REQ*: event generated by Button module
Completion events for sequential processing	comp_OPN_NAME or comp_TRANSITION	*comp_getNextDest:* completion event for getNextDest operation
Class attributes	no prefix	*prevDest:* attribute of Elevator module
Component object variables	no prefix	*openBn: instance* of Button module
DEFINE Section		
Active state macros	in_STATE_NAME	*in_Moving*: macro where state Moving of Elevator module is active.
Labeled transitions	no prefix	*t4*: the transition from state Open_Door to state Close_Door of Elevator module.
Operations Macros	m_MACRO_NAME	*m_f1_pending*: macro where a request from the first floor is pending

Table 4: Naming convention for SMV modules, variables and macros

4 Verifying System Properties Using SMV

In this chapter, we describe some of the system properties that were verified by SMV. We first give an overview of how to write CTL formulas. Then, we translate and verify the elevator's system properties that we defined in the requirement analysis phase. Finally, we discuss how we verify partial specifications within the context of "object-oriented" SMV modules, verifying properties of one or more modules at a time. In our elevator case study, we used a Pentium-II 350Mhz PC with 128 MB of memory. We used McMillan's Cadence Symbolic Model Verifier [McMi99] version November 10, 2000 running on Windows 98 to model check the elevator SMV program presented in Appendix C.

4.1 CTL Overview

In this section, we describe a temporal logic called Computational Tree Logic (CTL) to specify system properties of a state transition system. CTL is a logic for reasoning about state transition systems as they are used in the description of reactive systems. Temporal logic formulas defined over a Kripke model (i.e., sequential system with a set of states connected by transitions) constitute CTL. CTL allows us to verify computation sequences that describe the ordering of events along multiple paths of the state transitions system. In temporal-logic model checking, time is not explicitly mentioned. Rather, we are interested only in the temporal ordering of events: e.g., whether a state or condition *eventually* occurs, *never* occurs, or *always* occurs.

Using SMV, we verify system properties expressed as CTL formulas. CTL is a branching-time temporal logic where statements about all or some paths starting in a state can be verified. Given a finite state machine, an *execution path* is a set of consecutive states that starts from an initial state. Because the future path of a system's execution is unknown, CTL precedes each of the temporal operators with a qualifier over the possible paths. The temporal operators **X**, **U**, **R, F**, and **G** are preceded by path quantifiers **A** (e.g., a property *f* is true in *all* paths) or **E** (e.g., a property *f* is true in *some* paths).

The syntax and semantics for CTL formulas defined in [CGP99] are summarized as follows:

- Every propositional variable is a CTL formula.
- If *f* and *g* are CTL formulas, then so are: **!***f*, *f***&***g*, *f* **|** *g*, *f***->***g*, **AX***f*, **EX***f*, **A**[*f***U***g*], **E**[*f***U***g*], **A**[*f***R***g*], **E**[*f***R***g*], **AF***f*, **EF***f*, **AG***f*, and **EG***f*.

The first order logical operators NOT (!), AND (&), OR (|), and IMPLIES (->) have their usual meanings. **X** is the *nextstate* operator, and formula **AX***f* (**EX***f*) is true in every (some) successor state of s_i in the reachability graph, where s_i is the current state of the CTL formula being evaluated. **U** is the *until* operator, and formula **A**[*f***U***g*] (**E**[*f***U***g*]) is true in state s_i iff along every (some) path emanating from s_i there exists a future state s_j at which *g* holds and *f* is true until state s_j is reached. **R** is the *release* operator, and formula **A**[*f***R***g*] (**E**[*f***R***g*]) is true in state s_i iff along every (some) path emanating from s_i either *g* never holds or there exists a future state s_j at which *g* holds and *f* is true up to and including state s_j. **F** is the *future* operator, and formula **AF***f* (**EF***f*) is true in state s_i iff along every (some) path emanating from s_i there exists a future state in which *f* holds. Finally, **G** is the *global* operator, and formula **AG***f* (**EG***f*) is true in state s_i if *f* holds in every state along every (some) path emanating from s_i. We used the CTL operators **AG**, **EG**, **AF**, **EF**, **AX**, and **EX** to describe the system properties of our elevator example.

4.2 Verifying the Elevator System Properties

As mentioned in Chapter 3, system properties are specified at the end of the **`main`** SMV module with the keyword SPEC. The CTL formulas discussed in this section are all defined in the complete elevator SMV program shown in Appendix C.

4.2.1 Safety Property for Elevator

A *safety* property expresses the condition that something bad should not happen. Typically, a safety property represents the desired system invariant to be checked. For example, we translate the safety constraint “elevator never moves with its doors open”, shown as a UML constraint between door object and elevator object in Figure 6 of Section 2.2.3, into the following CTL formula:

```
SPEC AG ( !(elevator.in_Open_Doors & elevator.in_Moving) )
```

Verifying the CTL formulas using SMV is an automated step. SMV exhaustively searches the entire state space and determines whether there is any state in which the proposition “elevator.in_Open_Doors & elevator.in_Moving” is true. SMV responds with either *true* or *false*. The SMV model-checker determined that the above safety property was true. Response *true* means the SMV model entails the CTL formula and value *false* means the SMV model does not entail the CTL formula.

4.2.2 Liveness Properties for Elevator and Hall Buttons

A *liveness* property expresses the condition that something good should happen. We verify a *liveness* property to check if the system guarantees progress and reaches a desired state. In our elevator example, we want to check that “requests to be delivered to a particular floor are eventually serviced” which represents the three UML constraints attached to the link between floor request buttons and the elevator object as shown in Figure 6 of Section 2.2.3. We also want to verify that “requests to use the elevator are eventually serviced” which represents the four UML constraints attached to the link between the elevator request buttons and the elevator object.

Using the *response* pattern, we formulate the seven liveness properties for elevator and hall buttons shown in Figure 6 of Section 2.2.3 as follows:

```
-- *** tests showing behavior of buttons and opening of door ***
SPEC AG ( external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors &
elevator.floor=1)
 -> AF (elevator.floor = 1 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_FloorReqBn[2] & !(elevator.in_Open_Doors &
elevator.floor=2)
 -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_FloorReqBn[3] & !(elevator.in_Open_Doors &
elevator.floor=3)
 -> AF (elevator.floor = 3 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_ElevReqBnUp[1] & !(elevator.in_Open_Doors &
elevator.floor=1)
 -> AF (elevator.floor = 1 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_ElevReqBnUp[2] & !(elevator.in_Open_Doors &
elevator.floor=2 & elevator.dir=up)
 -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_ElevReqBnDn[2] & !(elevator.in_Open_Doors &
elevator.floor=2 & elevator.dir=dn)
 -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )

SPEC AG ( external_Event.env_BP_ElevReqBnDn[3] & !(elevator.in_Open_Doors &
elevator.floor=3)
 -> AF (elevator.floor = 3 & elevator.in_Open_Doors) )
```

Response pattern is a property specification pattern introduced by [DAC98]. This pattern translates the “a state S must always eventually be followed by a state R within a set of execution paths” as AG(S -> AF(R)) in CTL.

Specifically, the first specification says that if a passenger inside the elevator requested to be delivered to the first floor and the elevator is *not* on the first floor with its doors open at the time he/she made the request, then the elevator will guarantee that this request will eventually be serviced.

Initially, the SMV model-checker determined that the liveness properties were false. While analyzing the counter-examples, we found that the elevator may not reach the destination floor because it could be indefinitely waiting for the environment to generate an AT_FLOOR event. To ensure that the environment regularly generates an AT_FLOOR event, we added AT_FLOOR event in the FAIRNESS section of External_Event’s SMV module. Next, we found that to ensure progress, we had to give priority to repeated requests for the current floor only if there are no pending requests for the other floors. We also found that we need to treat up- or down-button or floor-request button for the current floor as an open-door request, rather than a request to service the current floor. These changes are described in more detail in the next Chapter. Once these three changes were made, all the liveness properties evaluated to true.

Since we use implication to formulate our liveness properties, we have to check that the hypothesis is true. Otherwise, coincidental correctness[22] may occur because if and when the hypothesis is false, the implication holds regardless of whether the consequence is true. Since model checking is exhaustive, the implication is presumably tested in circumstances where the hypothesis is false and when the hypothesis is true. For each liveness property, we introduce the EF test on the hypothesis to ensure that the hypothesis is not always false.

```
-- *** tests the hypothesis to avoid coincidental correctness ***
SPEC EF(external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors &
elevator.floor=1))
SPEC EF (external_Event.env_BP_FloorReqBn[2] & !(elevator.in_Open_Doors &
elevator.floor=2))
SPEC EF (external_Event.env_BP_FloorReqBn[3] & !(elevator.in_Open_Doors &
elevator.floor=3))
SPEC EF (external_Event.env_BP_ElevReqBnUp[1] & !(elevator.in_Open_Doors &
elevator.floor=1))
SPEC EF (external_Event.env_BP_ElevReqBnUp[2] & !(elevator.in_Open_Doors &
elevator.floor=2 & elevator.dir=up))
SPEC EF (external_Event.env_BP_ElevReqBnDn[2] & !(elevator.in_Open_Doors &
elevator.floor=2 & elevator.dir=dn))
SPEC EF (external_Event.env_BP_ElevReqBnDn[3] & !(elevator.in_Open_Doors &
elevator.floor=3))
```

The SMV model-checker determined that these properties were all true. This means that the liveness properties formulated were all valid tests.

4.2.3 Additional System Properties

In addition to the system properties identified during requirements analysis phase, we also checked several other system properties to gain confidence in the correctness of our elevator model.

Some sanity checks are application-independent. For example, we may want to verify that it is always possible to reach a state from any other state such as "the elevator will always eventually stop."

```
SPEC AG( EF(elevator.in_Stop) )
```

A more complicated sanity check tests whether the up and down hall buttons on the second floor represent independent requests to be serviced. Keeping the up hall-button independent from the down hall button on the second floor is important because we expect the potential passenger to

[22] Using the terminology of [Beiz95], we will call this test coincidental correctness.

enter the elevator only if the elevator is traveling in his / her desired direction. For each request button, the button is lit if there is a pending request.

When the elevator arrives at the second floor to pick up potential passengers who want to go to the third floor, only the pending up hall-button backlight is turned off. Using the *response chain* specification pattern [DAC98], we formulate this system property as follows:

```
SPEC AG ( (ElevReqBnUp2.in_Lit & ElevReqBnDn2.in_Lit & elevator.floor = 1 &
elevator.in_Closed_Doors)
 -> AF( (ElevReqBnUp2.in_UnLit & ElevReqBnDn2.in_Lit) &
        AF(ElevReqBnDn2.in_UnLit) ) )
```

This formula states that when both the second floor's up and down hall-buttons have pending requests and the elevator is still on the first floor with its doors closed, eventually the second floor up hall-button is reset but the second floor down hall-button is not. The potential passenger who is waiting outside the elevator on the second floor and who wants to go down to the first floor will not get in the elevator going up and will not have to press the down hall-button again to recall the elevator. To avoid coincidental correctness for this liveness property, we checked the following CTL formula:

```
SPEC EF(ElevReqBnUp2.in_Lit & ElevReqBnDn2.in_Lit & elevator.floor = 1 &
elevator.in_Closed_Doors)
```

4.3 Model Checking Partial Specifications

In this section, we show how to verify the properties of a partial OO specification consisting of a subset of the specification's modules. In software testing, we find that there is a clear distinction between *unit testing* (testing a specific component) and *systems testing* (testing the collaboration between components). In the same manner, we can perform *unit verification* where only some of the objects defined in the Elevator object model (Figure 5 of Section 2.2.2) are verified.

Performing unit verification has the advantage of detecting errors early in the specification process, before the entire software system has been specified. Detecting and correcting errors for each object before analyzing the entire specification simplifies verification of the latter because each object is known to behave as expected. In the *system verification* step, we can focus our attention on the *collaboration of objects*.

Our technique takes advantage of the class structure of UML specifications and uses one or more known classes for unit verification. The class structure makes it easy to perform unit verification because each class is already an abstraction without the unnecessary details. We can perform unit verification on classes that are known and we do not have to wait for classes that are unknown or incomplete.

Ideally, one performs unit verification without changing the individual component's specification, with the exception of the main module where we make some assumptions about the environment of the individual component. We demonstrate how to verify an individual component by verifying the Timer class (which corresponds to the Timer SMV module) from our elevator case study.

In this example, we want to verify the liveness properties of the count down mechanism and the timeout mechanism of the **Timer** class. Since the **Timer** class does not have any component objects, only the Timer SMV module will be needed for unit verification. Instead of declaring the Timer SMV module as a component of Door SMV module, we declare Timer SMV module as a component of the **`main`** SMV module. Moreover, we do not include the External_Event component in the **`main`** SMV module because the Timer module does not directly respond to any environmental events. The **`main`** SMV module for Timer module's unit verification is illustrated in Figure 20.

```
MODULE main
```

```
-- ****************** VARIABLES SECTION *************************
VAR
-- *** GENERATED EVENTS SUBSECTION ***
 ev_SET.raised : boolean;
 ev_UNSET : boolean;

 -- *** COMPONENT OBJECTS SUBSECTION ***
 DoorTimer: Timer(ev_SET, ev_UNSET);

-- ***************** SYMBOL DEFINITION SECTION *****************
DEFINE
 -- *** GENERATED EVENTS MACROS SUBSECTION ***
 ev_SET.n := 5;

-- **************** PROPERTY / CTL FORMULA *********************
SPEC AG ( (DoorTimer.counter=5 & !ev_SET.raised)
       -> AX (DoorTimer.counter = 4) )

SPEC AG ( (DoorTimer.counter=0 & !ev_SET.raised & !ev_UNSET &
DoorTimer.in_Count)
       -> AX (DoorTimer.in_Idle) )
```

Figure 20: Main SMV module for Timer class unit verification

In Figure 20, we declare DoorTimer as a variable in the **`main`** SMV module. We reuse the Timer SMV module (Figure 16 of Section 3.2.4) without any modification. The DoorTimer object requires input parameters (i.e., SET and UNSET events) from its environment. Since modules in the partial specification do not generate the SET and UNSET events, we model the input parameters of DoorTimer as output events of the **`main`** SMV module. We declare these generated events in the "generated events subsection". Because the SET input event has an actual parameter (i.e., numeric 5), we set the parameter n of SET event to 5 in the "generated events macro section". Details of translation rules can be found in Chapter 3.

We specified two properties in the SPEC section. The first property helps to verify that the count down mechanism is working correctly. Literally, it says "If the counter is currently at 5, then it is immediately followed by counter equal to 4 in the next state if no SET event is raised." The second property helps to verify that the timeout mechanism of the DoorTimer is working correctly. This property states that if the counter is zero, no SET event was raised, and no UNSET event was generated, then the timer moves to state Idle. The SMV model-checker determined all the properties were true.

To re-verify the properties in the full specification, the property expressions may need to be modified to reflect the propositions' new context. For example, when we declare DoorTimer to be a component object of the door module, DoorTimer.counter becomes elevator.door.DoorTimer.counter as shown in the updated CTL formulas:

```
SPEC AG ( (elevator.door.DoorTimer.counter=5 & !elevator.door.ev_SET.raised)
       -> AX (elevator.door.DoorTimer.counter = 4) )

SPEC AG ( ( elevator.door.DoorTimer.counter=0 & !elevator.door.ev_SET.raised
       & !elevator.door.ev_UNSET & elevator.door.DoorTimer.in_Count )
       -> AX (elevator.door.DoorTimer.in_Idle) )
```

5 Effectiveness of Model Checking

In this chapter, we discuss the effectiveness of model checking UML specifications. The original intent of this thesis was to demonstrate the feasibility of model checking UML specifications using SMV. When we started fixing the specifications to meet the desired properties, we realized that many of the fixes affected not only the SMV model and the UML specifications, but also the original requirements. The outcome of fixing the specifications is a more accurate, consistent, and complete UML model and a more consistent and complete requirements specification.

Measuring the effectiveness of model checking a UML specification was not part of the original intent of this thesis. However, since we kept records of the UML and SMV specifications as we iterated through the analysis of revised specifications, we decided that we had enough information to informally point out the errors detected, lessons learned and observations made from model checking UML specifications. We found five major types of errors during model checking:

1. *Requirements-level errors*: errors that reveal missing, incomplete, or ambiguous functionality in the problem statement.
2. *Object-oriented modeling errors*: errors that reveal synchronization and coordination problems among the objects in the object-oriented specification.
3. *Translation errors*: errors in the SMV representation of the object-oriented specification.
4. *Environmental constraints*: errors that reveal missing or inaccurate constraints on environmental events and variables.
5. *Temporal logic formula errors*: errors in the logical expressions of the intended desired properties.

In the following, we estimate the number of errors detected for each type of error, but we do not make any formal measurements as to the severity of errors.

This chapter proceeds as follows. First, we review the various steps of our iterative analysis process and the time spent performing each step. Then, we analyze how model checking helped to reveal requirements-level errors, object-oriented modeling errors, and other SMV errors. Finally, we summarize the effectiveness of using model checking to detect requirements-level errors and object-oriented modeling errors.

5.1 Time Spent per Activity

In the elevator case study, we were given an existing specification of the problem statement that had been used in University of Waterloo's CS 746B assignments. We created the initial UML model followed by an iterative process of verifying and revising the UML and SMV models. We now describe the work completed for each activity and discuss the time spent on each activity.

Inspecting and Revising the Problem Statement

The objective of this activity was to find requirements-level errors in the original problem statement. The analyst and the domain expert reviewed the problem statement. This activity includes the time spent on revising the problem statement. A total of 10 hours was spent on this activity.

Since time logging was not part of the original intention of this thesis, we did not officially record the actual time spent on each activity. For each activity, we based our time estimates on the number of times we iterated for that activity multiplied by the estimated number of hours per iteration.

Building the Initial UML Model

The objective of this activity was to specify a UML model using UML class diagram, object model, and UML state diagrams. This activity includes the time spent to understand and clarify the details of the elevator system. A total of 90 hours was spent on this activity.

Translating UML to SMV

The objective of this activity was to find a *homomorphic mapping* from UML to SMV. *Homomorphic mapping* seeks to preserve structural relationships between the UML and SMV models. Each time the UML model is modified, we manually changed the SMV model to reflect the new changes to the UML model. We spent approximately 40 hours to translate UML models to SMV models. When we started the elevator case study, there were no proposed rules and guidelines for translating from UML to SMV as described in Chapter 3. It took more than 20 hours to translate our initial UML specifications and review the initial SMV model. In the succeeding translations, we referred to our rules and guidelines for translating the UML class diagram, object model and UML state diagrams into SMV's input language. The succeeding re-translations (about 10 iterations) due to changes in the UML specifications required less than 20

hours because translation guidelines were available and changes were confined to a specific SMV section or subsection.

Model Checking

The objective of this activity was to find defects in the problem statement, UML models, and SMV model. This activity includes the time spent checking and correcting SMV syntax errors such as SMV cyclic assignment errors, and analyzing SMV counter-examples. Since we were interested in the effectiveness of model checking, we did not include in our time estimates the time it took to learn the SMV input language and model checker. It includes the time to construct the CTL formulas as described in Chapter 4. It does not include the time it took to execute the SMV model checker. In our earlier SMV specifications, it took more than two hours to verify one of the liveness properties.

A total of 100 hours was spent on this activity. Most of the time spent can be attributed to reading and analyzing the counter-examples. When reading counter-examples for large specifications, we found it time consuming to navigate and inspect the values of each SMV variable. Cadence SMV's viewer arranges the SMV variables in alphabetic order, rather than grouping variables by module.

Revising the UML Model and the Problem Statement

The objective of this activity was to modify the UML model and the problem statement (if applicable) in order to satisfy one or more system properties that failed in model checking. This activity includes the effort required to improve the UML modeling guidelines and naming conventions. A total of 160 hours was spent on this activity. The time required for verifying and revising the UML model includes the actual time spent on fixing both requirements-level errors and the actual time spent on fixing UML-related errors.

Table 5 shows how each of the above activities contributed to the total of 400 hours of effort. Table 5 also shows the different types of errors detected per activity.

Activity	**Number of Hours**	Number of **Requirements-level Defects** Found	Number of **Style Defects** (found in revised models only)	Number of **UML Defects** (found in revised models only)
A. Create Initial Model				
Inspecting and Revising the Problem Statement	10	2		
UML Modeling	90	2		
B. Verify and Revise Model				
Translating UML to SMV	40	0	5	4
Model Checking	100	6	0	2
Revising UML Model and Problem Statement	160	1	5	1
Total	**400**	**11**	**10**	**7**

Table 5: Requirements-level and OO modeling defects summary

We divided the activities into two sections. Phase A, **Create Initial Model**, consists of manually inspecting the problem statement and creating the initial UML model. Without model checking, the task of writing the SRS would be considered done once the UML models have been created.

In addition to activities for creating the initial UML model, we added activities that are grouped together in Phase B, **Verify and Revise Model**. During model checking, the time spent on correcting the errors revealed by model checking the desired system properties includes all the time needed:

- to change the problem statement,
- to modify the UML models,
- to translate the new UML model modifications to SMV, and
- to rerun the model checker.

Table 5 shows that we spent 25% of our time (i.e., 100 out of 400 hours) in the "Create Initial Model" phase. It is not surprising that we spent 75% of our time (i.e., 300 out of 400 hours) in correcting and re-verifying the UML specifications because it took about 10 iterations to successfully verify all the system properties.

5.2 Requirements-level Errors

The ultimate goal of model checking UML specifications is to detect errors early in the software development cycle. Based on approximately 400 hours of effort, we found a total of 11 requirements-level errors. Out of the 11 requirements-level errors, four were detected in the "Create Initial Model" phase. The additional seven requirements-level errors were revealed during the "Verify and Revise Model" phase. Without model checking, these 7 of the 11 requirements-level errors would have been discovered later in the software development cycle or not at all. Column 3 of Table 5 shows the breakdown of the number of requirements-level defects found, organized by activity.

Inspecting the Problem Statement

The problem statement was initially presented to a small group of faculty members and students[23]. In this presentation, we found one requirements-level error in which the problem statement failed to mention that the second floor hall buttons must have directions (i.e., one hall button to move up and another hall button to move down). After the problem statement was modified, our domain expert reviewed the problem statement[24] and found one additional requirements-level error. The problem statement failed to mention that the inner and outer doors become mechanically linked when the elevator is at rest at a floor. When the inner door opens, the outer door will automatically open. Only the inner door, which is in the elevator car, can be directly controlled by the software system.

Our domain expert also found additional features that are common to industrial elevators (e.g., warning light to indicate when maximum load has been reached, door sensors that reopens door if obstruction is detected when the door is closing). These additional features were not added to the elevator case study. As a reference for future extensions of the elevator case study, Appendix D lists the additional features identified by our domain expert.

---

[23] University of Waterloo, Department of Computer Science, Software Engineering Talk given on December 17, 1999, attended by Prof. Dan Berry and a few graduate students.

[24] Our domain expert, Doug Guderian (Delta Elevator Co. Ltd.), reviewed a draft copy of the problem statement dated February 13, 2000.

Building the Initial UML Model

In this activity, we found two requirements-level errors as we reviewed the initial UML model with the domain expert. The requirements-level errors found were:

- The **External_Event** class showed only one BN_PRESSED event. This meant that only one BN_PRESSED event could be raised at a time. We later decided that it should be possible to press several buttons simultaneously. The original problem statement was silent about this requirement.
- Other than the floor sensor, which determines which floor the elevator is in, there must be a *slowdown point* for each floor. *Slowdown point* refers to a specified point where the elevator slows down in preparation to stop. The slowdown point of each floor determines if a request made while the elevator is moving can still be serviced. The elevator can't stop to service a request from a specific floor if the elevator has passed the slowdown point of that floor.

Translating UML to SMV

No requirements-level errors were found when translating from UML to SMV.

Model Checking

We found additional requirements-level errors in model checking system properties that were unsatisfied in our initial SMV model. These errors and their corresponding corrective actions were:

(1) Starvation occurs if the user repeatedly presses the open-door button when the elevator doors are open. *Corrective action*: allow only the first three requests to keep the opened doors open, after which the elevator will close its doors within five time units, regardless of subsequent requests to keep the doors open, and will give priority to other pending requests.

(2) Starvation occurs when open-door request is repeatedly made when elevator doors are closed. *Corrective action*: allow only the first two requests to reopen the closed elevator doors, after which the elevator will give priority to other pending requests.

(3) Starvation occurs if we treat as a new request-for-service a button press that corresponds to the elevator's current floor when the elevator doors are open. *Corrective action*: pressing an up- or down-button or floor-request button when the elevator is at the button's floor and the

elevator doors are open is treated as an open-door request (of which only a limited number are recognized and serviced), rather than a request to service that floor.

(4) Starvation can occur when the elevator scheduler always gives priority to the requests for the current floor over requests for other floors. *Corrective action*: give priority to pending requests for the other floors over repeated requests for the current floor.

(5) A passenger needs to know the direction of the elevator (up or down) before the elevator doors open, so that he/she can decide whether to enter the elevator or wait until the elevator returns to pick up passengers traveling in the other direction. *Corrective action*: set the elevator's next direction upon arrival and before the elevator opens its door.

(6) Requirements did not state when the elevator needs to recalculate both the next destination and next destination's direction. If the elevator is traveling up to the second floor to pick up a passenger waiting to travel down, and if a second passenger presses the second-floor up-hall button before the elevator reaches the second floor, the second passenger expects his request to have priority over the pending down-hall button request. *Corrective action*: recalculate the next destination and the next destination's direction right before the elevator reaches its destination floor.

We found that the corrective actions we made in order to satisfy the system properties would be observable in the final system, are changes to the requirements (i.e., problem statement) and not just the specification (i.e., UML model and SMV model).

Revising the UML Model and the Problem Statement

We found one requirements-level error as a result of revising the UML model. Although the primary goal of this activity was to help satisfy the system properties (i.e., for model checking), it also provided an opportunity for the analyst to review the UML model. While revising the UML model, we found that the problem statement did not explicitly mention whether the elevator doors should reopen if the open door button is pressed when the elevator doors are closed and the elevator is not moving.

5.3 UML Model Style Defects

In this section, we discuss the style defects, which consist of missing, ambiguous or mismatched names and trivial defects such as spelling errors and choice of words. These changes might affect the clarity or maintenance of the system but they do not affect correctness of the specifications. Instead of showing the number of times we encountered each type of style defects, we only show the types of style defects found because style defects tend to be global changes (e.g., changing a naming convention for translating environment variables). In the "Verify and Revise Model" phase, we found a total of 10 style defects. Column 4 of Table 5 shows the breakdown of the number of style defects types found in the UML specifications of our elevator case study, organized by activity.

Translating UML to SMV

The types of style defects found when translating from UML to SMV were:

(1) Some of the events used in the state diagram were not defined or were misspelled in the class diagram.

(2) Some transition label names and region names of UML state diagrams were missing.

(3) Some events were qualified with incorrect object names or were not qualified with an object name at all.

(4) Some of the class attribute's default values did not match the corresponding state diagram's initial state transition actions.

(5) Naming of attributes, states, events, operations did not match their corresponding names as used in the real world.

Revising the UML Model and the Problem Statement

The types of style defects found when revising the UML model were:

(1) Some class names, association names, and event names did not conform to the naming conventions.

(2) Some of the low-level objects had direct knowledge of their ancestor objects. Worse, one object manipulated one attributes of its parent. Such low-level objects cannot be reused in

another application domain. To allow the low-level objects to indirectly change their ancestor objects attribute values, we enforced event-passing interfaces among classes.

(3) Some transition labels had *redundant guard conditions*. *Redundant guard conditions* do not add any new information to the event trigger or to an existing condition of a transition.

(4) Some classes, such as the **Floor** class, are not used at all.

(5) Some state diagrams were complex. We simplified the UML state diagram by replacing some parts of the state diagram with a UML operation.

5.4 Object-oriented Modeling Errors

In this section, we discuss object-oriented (OO) modeling defects, such as synchronization and coordination problems among the objects. OO modeling defects also includes correctness defects such as missing events and missing default values in the UML state diagram and missing attributes in the UML class diagram. These defects must be corrected to verify the system properties. In the "Verify and Revise Model" phase, we found a total of seven UML defects. The last column of Table 5 shows the breakdown of the number of UML (i.e., object-oriented) defects found in the revised UML models, organized by activity.

Translating UML to SMV

The correctness defects found in the UML model were:

(1) Some of the class attributes did not have default values. *Corrective action:* assign default values to class attributes that need a default value.

(2) The original Elevator state diagram modeled the elevator's direction (up or down) as a concurrent region. The initial state was in state UP and it changed direction when the next destination was determined and the elevator was about to move. After model checking, we found that the direction may also change right before the elevator doors open. *Corrective action:* model direction of elevator as an attribute (instead of as a state).

(3) The original Elevator state diagram processed a new button-pressed request only when the elevator is not moving. This does not guarantee that all valid requests become pending requests and are eventually serviced. *Corrective action:* add a new concurrent region that processes button-pressed requests in parallel with the elevator's other activities.

(4) The original class diagram showed hall buttons as components of the Elevator class. This did not allow the hall buttons to be shared by other elevators, should we ever want to extend the problem to deal with more than one elevator. *Corrective action:* add a new Building class to contain the hall buttons and the Elevator class; hall buttons can now be shared among multiple elevators.

Model Checking

We found two correctness defects in the UML model as follows:

(1) The original Elevator state diagram generated an "open door" event using a self-transition to the state Open_Doors. If and when the "close door" event happens while the "open door" request occurs, the original Elevator state diagram re-enters the state Open_Doors and at the same time moves to the state Closed_Doors. *Corrective action:* generate the "open door" event using an "on event" transition in order to allow a response to "open door" request without re-entering the state Open_Doors.

(2) The original RequestBn state diagram used the button-pressed event (BN_PRESSED) to set the pending attribute to true and to turn the button's light on. However, there are cases where a BN_PRESSED event does not necessarily result in the light being turned on. *Corrective action:* add a separate LIGHT event generated by both Building class and Elevator class. RequestBn class can now react to LIGHT event separately from BN_PRESSED event.

Revising the UML Model and the Problem Statement

We found one correctness defect in this activity. The original UML model did not have a close door button. *Corrective action:* add a new CloseBn instantiation of the **Button** class.

5.5 SMV Errors

This section briefly presents the SMV errors detected during the "Verify and Revise Model" phase. These SMV errors were corrected in order to verify the system properties. We do not distinguish the severity of these SMV errors.

Translation Errors

We found these SMV model defects when systematically translating the UML model to SMV. These errors were revealed after we found that the SMV model did not match the expected behavior of the UML model. Automating the translation rules will eliminate these translation errors. The translation errors found were:

- The default value of the case statement is wrong. For events, the default is 0 (false); for attributes, the default is the current value.
- The number of SMV module's input parameters did not match the class's input events and class's input attributes.

Environmental Constraints

We added constraints on when environmental events could occur in order to successfully verify the system properties. Without these constraints, the liveness properties were not satisfied because the environment may not regularly generate the events needed to make progress. Moreover, these constraints better model the system's environment. The following environmental constraints were added:

- FAIRNESS constraint for AT_FLOOR event in order to force AT_FLOOR event to regularly occur.
- TRANS condition such that the AT_FLOOR event does not occur while the elevator's transition t10 is executing.

Details of specifying environmental constraints can be found in Section 3.2.3 (Translating the **External_Event** Class).

Temporal Logic Formula Errors

In model checking, it is important to specify a system property accurately because an incorrect formula leads to an incorrect analysis of the desired system behavior. We experienced the following errors when constructing and verifying CTL formulas:

- We found coincidental correctness errors[25] when using the implication operator. We use the CTL operator EF over the hypothesis to ensure that the hypothesis can be true. Details of coincidental correctness errors are discussed in Section 4.2.2 (Liveness Properties for Elevator and Hall Buttons).
- We found logical errors for some CTL formulas where we should have used the logical operator "&" (AND) instead of the implication operator "->" (IMPLY).
- We found that some CTL formulas do not match the intended system property. This happens if and when the SMV model is misunderstood. In some cases where the implication operator was used, we modified the CTL formula by adding a clause in the hypothesis to restrict the circumstance of when property is supposed to hold.

[25] Recall that if the hypothesis is false, the implication holds regardless of whether the consequence is true.

5.6 Summary

This chapter briefly describes how the iterative analysis process (i.e., "Verify and Revise Model" phase) revealed errors that would have otherwise been left in the problem statement and UML models. The entire elevator case study took approximately 400 hours. Initially, approximately 100 hours was spent on inspecting the problem statement and writing the original UML specification. Then, we spent approximately 300 hours iterating the translation to SMV, model checking, and correcting the UML model. In these iterations, we found requirements-level errors, UML-model errors, and SMV errors.

Although only a few classes were defined in the elevator case study, it took about ten iterations before we satisfied all the system properties. These errors were difficult to correct because the state diagram of the Elevator class had to manage the complex coordination and synchronization of events. Separating the "separate concerns" of the Elevator's state diagram was not a trivial exercise. Within the Elevator class's state diagram, we handled multiple concurrent regions, multi-level hierarchical states, multiple events and complex UML operations. In order to find a fix for a single execution path error that is caused by the Elevator SMV module, we had to find a general solution that must handle the complexity of Elevator's state machine.

By investing an additional 300 hours, we found seven additional requirements-level errors in the SMV and UML models. This is consistent with other research studies that also found defects with requirements specification using model checking [SrAt96][CAB+98]. We detected 10 types of style defects and seven correctness defects in the UML specification. The current UML modeling tools such as Rational Rose can detect some of the style defects. However, these UML modeling tools cannot detect correctness defects at all. Our findings suggest that UML modeling is not immune from defects. Model checking revealed many UML errors that would have been left undetected.

This chapter has shown that model checking is an effective tool to detect correctness errors in UML models as well as requirements specifications. It would be interesting to compare the number of hours that it would have taken to detect and correct these defects had the original UML model been used for programming and testing. Using simulation and testing, it is difficult, or even impossible, to detect all the liveness property errors because simulation and testing requires a lot more time to program and execute all the possible test cases as compared to model checking. For safety properties, it is often impossible to detect errors using simulation and testing because

this would require that test coverage is exhaustive which will require a lot of time to complete. Performing exhaustive testing is not cost-effective for large software systems. We argue that model checking at the requirements specification level is cost effective because studies have shown that correcting errors in the requirements analysis phase cost much less than correcting errors at the implementation phase [Boeh81].

6 Conclusions and Future Work

Using a non-trivial three-floor elevator as case study, we have shown how to translate a UML model to SMV while preserving the object-oriented properties of the UML model. Specifically, we provided guidelines on how to translate classes, associations, events (with parameters), and state machines into input to the SMV model checker. With some modifications, these same guidelines can be used to translate UML to other model checkers (e.g., SPIN, COSPAN, etc.). Unlike other case studies where they focus on state machines, we have translated UML models with emphasis on class structuring (associations, generalization, composition, aggregation), event parameters, and class dependencies. Our translation from UML to SMV led to the development of an object-oriented SMV program template that supports class structuring and improves readability of SMV programs.

We have shown that non-trivial properties can be checked using SMV. We have shown a way to translate UML constraints to CTL formulas required for SMV. Errors that would have gone unnoticed during the UML modeling phase were detected using the SMV model checker. This supports the idea that model checking can detect critical errors that are not detected in UML modeling and inspection activities.

Our proposed approach to iterate the analysis of UML and SMV models makes model checking effective in verifying system properties because:

(1) The analysts will continue to use the popular UML notation for specifying class diagrams, state diagrams and object models. No new training for a different notation is required.

(2) The translation from UML to SMV can be done in a systematic fashion. Using our translation rules defined in Chapter 3, it is possible to automate the translation of a subset of the UML notation.

(3) SMV model checker generates a counter-example if and when the system property is not satisfied. The counter-example serves as a debugging tool to pinpoint the error in the system.

In order to improve the readability of the UML and SMV models, we proposed extensions to UML such as how to model events in class diagrams. We also extend UML with the notion of the multiple-source events when modeling state diagrams. Although our UML modeling guidelines and translation rules were effective for our elevator case study, more case studies are needed to validate the translation rules and guidelines proposed in this thesis.

This thesis serves as a starting point for software practitioners who would like to model check UML models. We now summarize some of the future work and extensions.

(1) We need to study more complex case studies to evaluate our ideas. Extending the elevator UML and SMV model will also determine the scalability of model checking large-scale specifications. A list of additional elevator features is shown in Appendix D.

(2) State explosion, an identified problem in model checking needs to be explored further by adding more floors to the case study (i.e., more than 3 floors) and extend the elevator case study to more than one elevator.

(3) UML is a rich language and only a subset of the UML notation was used in this thesis. We need to investigate the use of advanced state diagram notation such as history psuedo-states. We also need to further investigate some UML semantics issues. For example, integrating multiple-step UML operations into the UML models will lead us to synchronization issues with respect to handling events while the UML operation is not completed.

(4) Although we have demonstrated the usefulness of the translation rules and procedures for translating reactive systems, more work needs to be done to apply these rules and guidelines to other domains. Instead of manually translating the UML to SMV, it would be useful to automate (at least partially) the translation to avoid manual translation errors. The ultimate goal would be to add model checking as part of modeling tool support that can be used with commercial tools such as Rational's Rose and I-logix's Rhapsody UML modeling tools.

(5) In this thesis, there were no formal measurements for the effort used to detect the requirements-level and modeling errors. It would be interesting to study how traditional software implementation and testing could reveal these requirements-level and modeling errors. This will help develop a cost-benefit analysis of model checking vs. software testing.

Appendix A: UML Notation Summary

A1. UML Class Diagram Notation

The UML Class Diagram describes the object-oriented structure of the system. A summary of the key notation used in our case study is shown in Figure A-1.

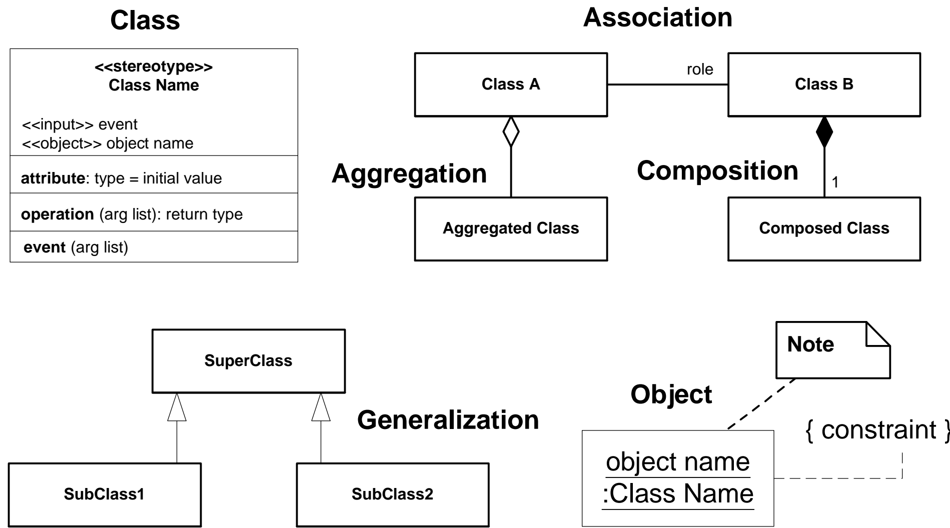


Figure A-1: UML class diagram notation

We summarize the definition of these key concepts and notation as follows:

A ***class*** captures the common structure and common behavior of a set of objects. A ***class*** is an abstraction of real-world items such as persons, places, things, user-interfaces, events or concepts that are relevant to the system. As shown in Figure A-1, we extended the UML class as a four-part box. The class name in the first (top) part is also extended to include the input events and object references. A list of attributes with optional types and values is defined in the second part. A list of operations with optional argument lists and return types is defined in the third part. Finally, we define output events in the fourth (bottom) part.

An ***object*** is an *instance* of a class. It can contain both *data* and *functionality*. Data are represented as ***attributes*** in a class. Functionality is represented as ***operations*** to describe computational behavior. An object with significant temporal behavior can be described in a ***state diagram***. Object names are drawn with the class name after the ":" and underlined.

A ***generalization*** relationship is drawn from a class to another class. It shows that the ***subclass*** shares the structure and behavior defined in one (or more) ***superclass***. The generalization relationship is drawn as a directed solid line with a closed arrowhead pointing to the ***superclass***.

An ***association*** represents a semantic connection between two classes. They are bi-directional. They are the most general of all relationships. An association is drawn as a solid line connecting two classes. To describe the ***association*** relationship with more precision, we add *adornments* such as:

- A ***role name*** represents the purpose or capacity an object plays in an association. Role names are not bi-directional.
- An ***aggregation*** relationship is used when one object physically or conceptually contains another object. The larger class is called the *whole* and the smaller class is the *part* or *component class*. A *part* class can be shared by other *whole* classes. In this thesis, we refer to the *part* class as the ***aggregated object***. They are drawn as an association (solid line) with an unfilled diamond at the *whole* end of the relationship.
- A ***composition*** relationship is a stronger form of aggregation. We refer to the *part* class of a composition relationship as the ***component object***. In a composition relationship, the *part* class cannot be shared by other *whole* classes. They are drawn as an association (solid line) with a filled diamond at the *whole* end of the relationship.
- At each end of the association, we can define the ***multiplicity*** or the *number of objects* that the class refers to. In our example, we assume the multiplicity is one if it is not specified. It is drawn as a number on one of the endpoints of the association.

A ***note*** is a place to hold any amount of text. It is used to present additional information such as assumptions and design decisions that can not be explicitly described using the UML notation. It is drawn using a special note-shaped icon. Notes may be placed anywhere in a diagram. It can be connected or unconnected to a model element.

A ***constraint*** is an expression of some semantic condition that must be preserved while the system is in a steady state. It can be added to classes, class attributes, roles and associations. It is shown as text surrounded by curly braces { }.

A ***stereotype*** is an extension mechanism to UML, allowing analysts to add new semantics to the UML notation. We introduced the <<event>> stereotype to define a new form of class that defines events instead of operations.

Other adornments such as qualified association, navigability and dependency are not used in our case study. A detailed set of UML notation can be found in the OMG UML Specification [UML99].

A2. UML State Diagram Notation

The UML state diagram describes the object-oriented behavior of the system. In this thesis, we use state diagrams to describe the behavior of a single class. Collaboration of classes is modeled as multiple state diagrams communicating via events. The graphical representation of the UML state diagram notation is shown in Figure A-2 and Figure A-3.

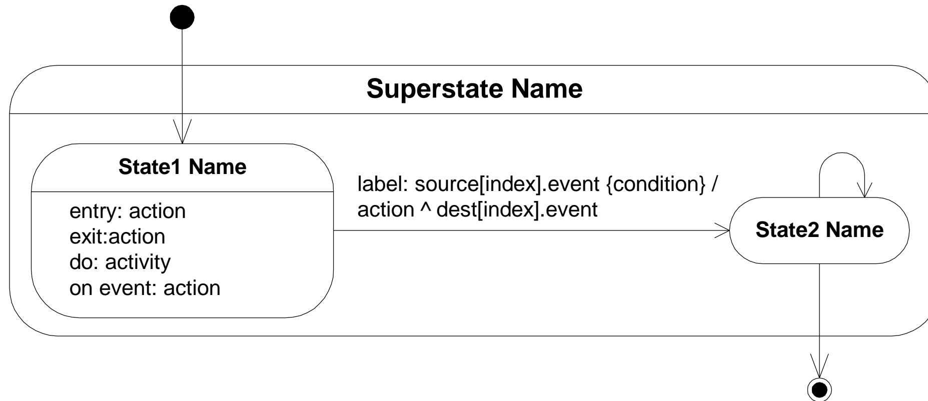


Figure A-2: UML State Diagram Notation

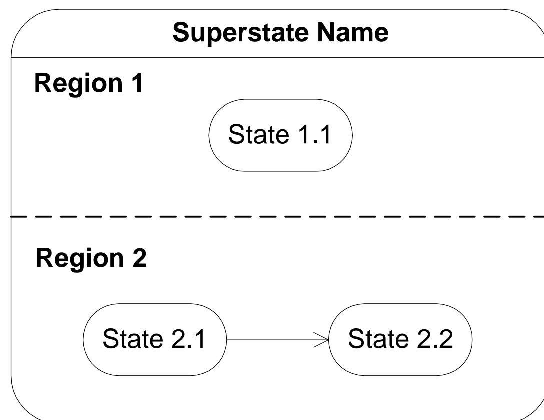


Figure A-3: UML Concurrent States Notation

We summarize the definition of these key concepts and notation as follows:

A ***state diagram*** describes the history of the object's property values of a given class. Rectangles with rounded corners are states. Lines between states are transitions. A state diagram must have a start state, one or more states, zero or more end states, and the state transitions between them. A

state diagram can have nested states where a *superstate* can have one or more *substates* as shown in Figure A-2.

A ***superstate*** can have multiple orthogonal regions where each region runs in parallel to other regions. *Concurrent states* within a state diagram describe states of components that are orthogonal to each other. In Figure A-3, we show orthogonal regions within a superstate as states that are separated by a dotted line. Each region is labeled with a region name.

A state can be *active* or *inactive* during execution. A state becomes *active* as a result of some transition. An *action* is an instruction such as assigning a value to an attribute or sending an event. When a state is entered (i.e., *on entry*), its entry action is executed. Then, the activity (i.e., *on an activity*) is performed. While the state is active, the *on event* action[26] is executed when the event is received. When a state is exited as a result of a transition, the *on exit* action is performed. For our elevator example, we only used the "*on entry*" and "*on event*" actions to show what action are to be carried out when the state is active.

A ***state transition,*** attached from one state to another state or to itself, is a change of state caused by an event. ***Transitions*** may have events with parameters, guard conditions and actions. The syntax for a fully specified state transition is:

```
label: src[index].event {condition} / action ^ dest[index].event
```

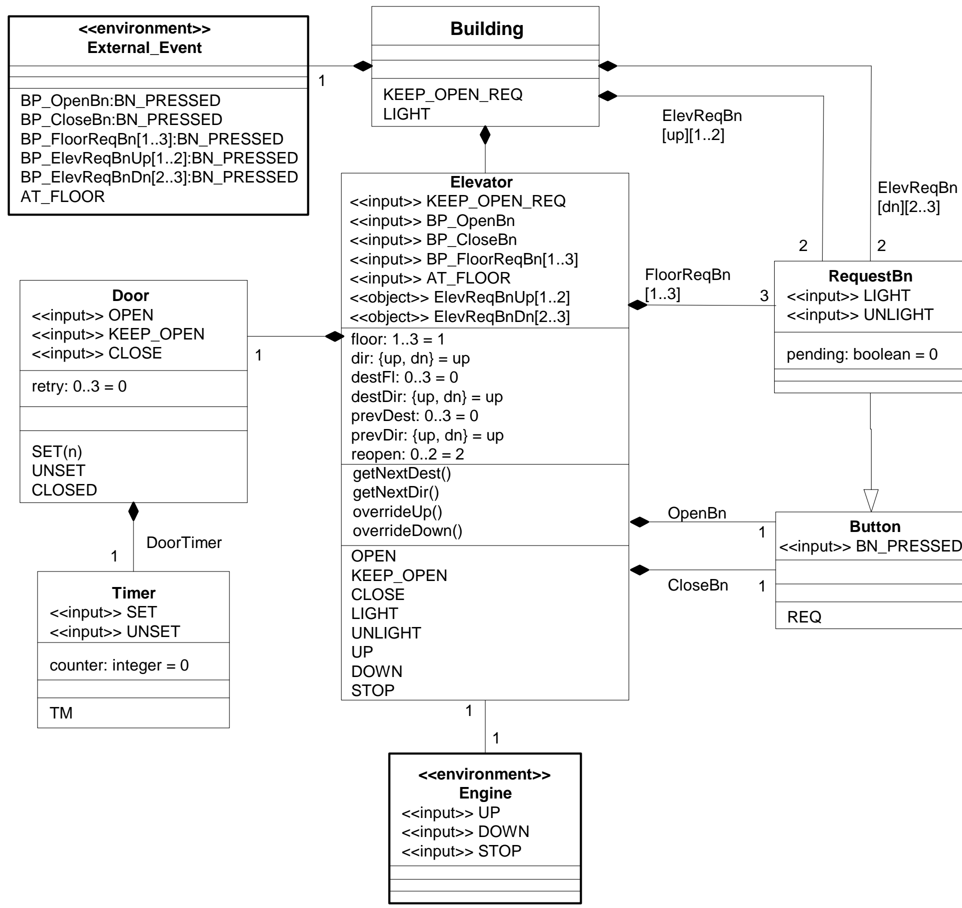
We have extended UML's state transition notation as follows:

- We add a label name to transition labels to ease translation from UML into SMV.
- We can designate events by their source or destination object.
- We can add the index value to the event's source or destination object.
- We use curly brackets to delimit conditions because square brackets are used to designate index values of source and destination objects.

---

[26] On event action is also known as internal transition, which does not exit and re-enter the state it is defined.

In UML, only classes define state diagrams and only objects execute state machines[27] [Doug98]. Additional information on UML state notation can be found in the OMG UML Specification [UML99].



Appendix B: Elevator UML Specification

Figure B-1: Three-floor Elevator Main Class Diagram

[27] We use a *state model* to define a state machine, which defines the behavior of a single class. *State machine* is an instance of a state model and must belong to an object.

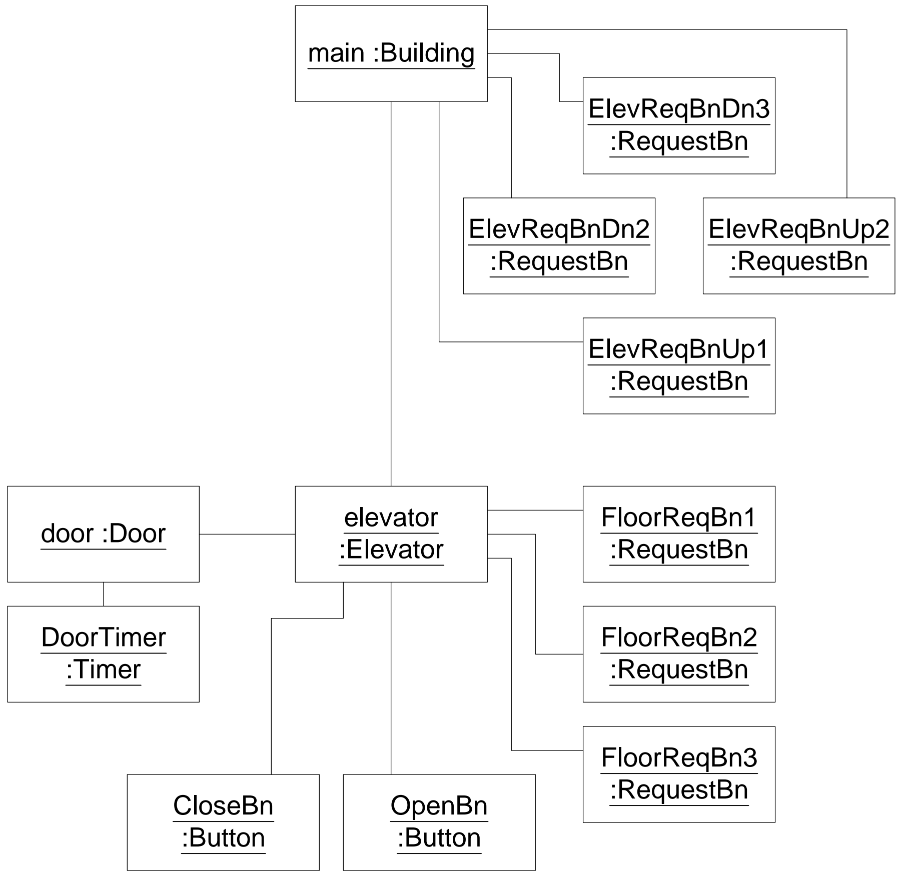


Figure B-2: Three-floor Elevator Object Model

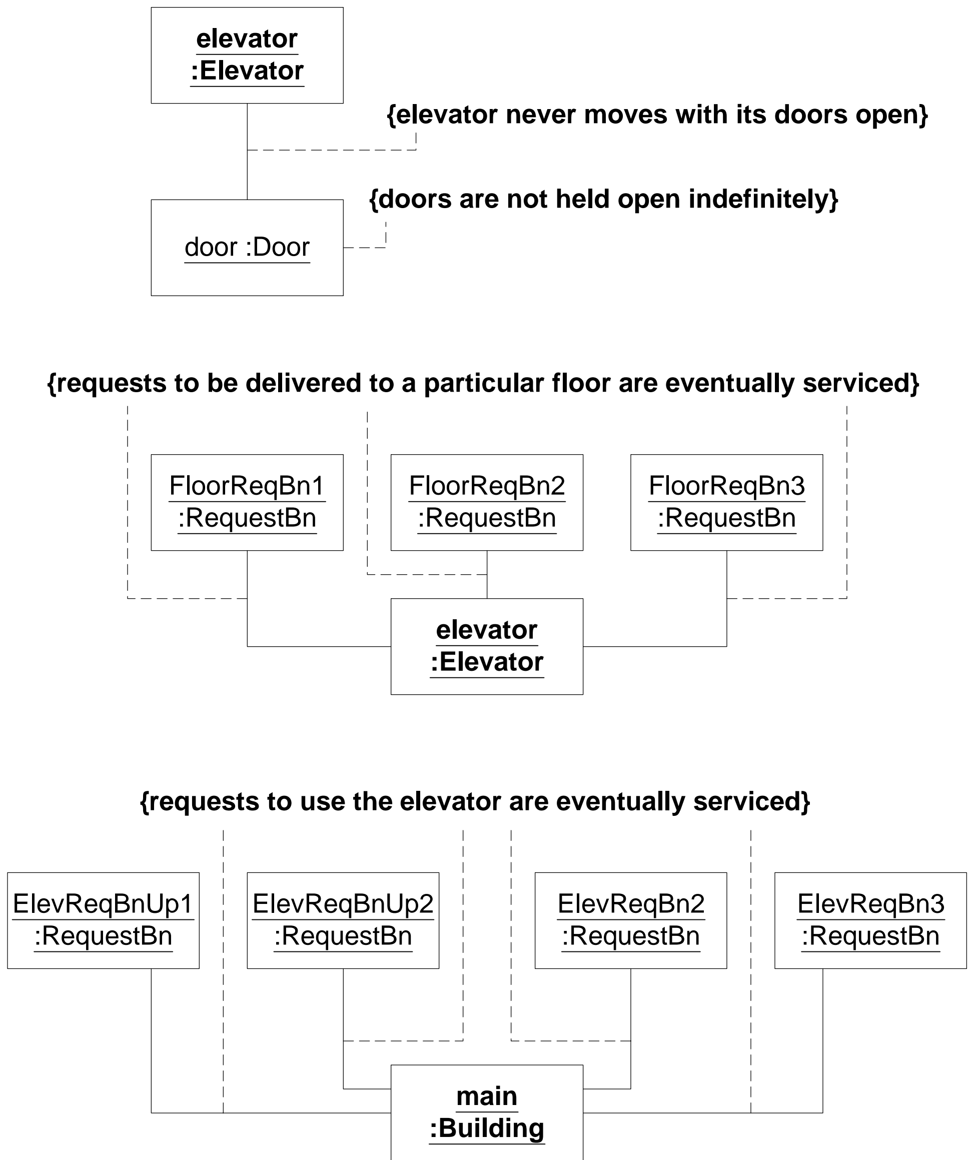


Figure B-3: System Properties shown as UML Constraints

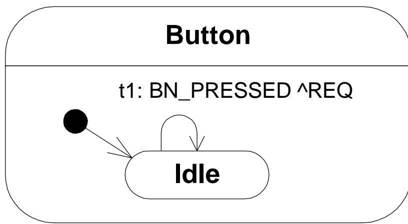


Figure B-4: Button State Diagram

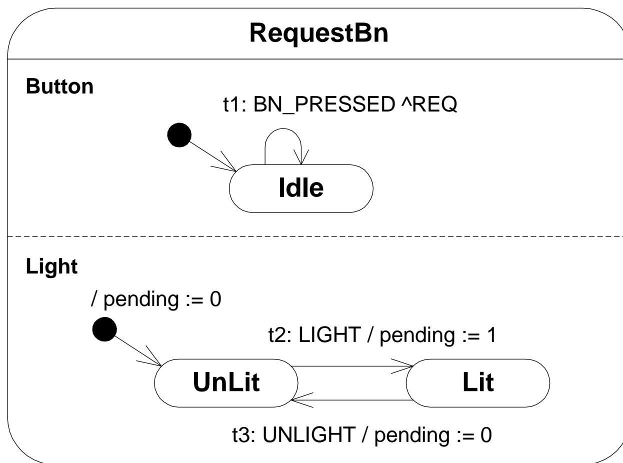


Figure B-5: RequestBn State Diagram for elevator and hall buttons

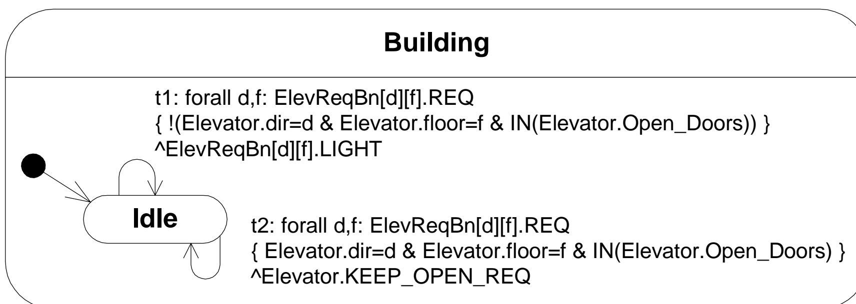


Figure B-6. Building State Diagram

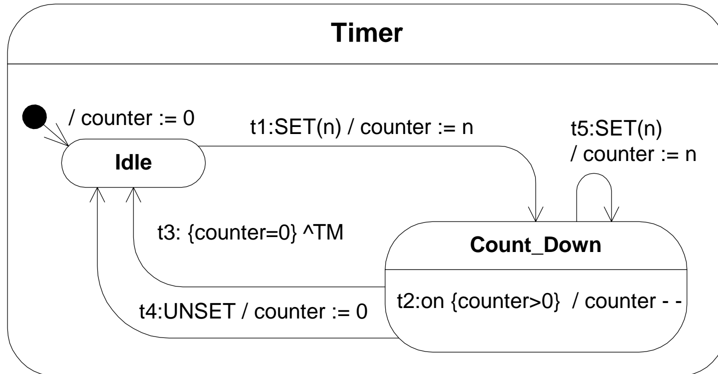


Figure B-7: Timer State Diagram

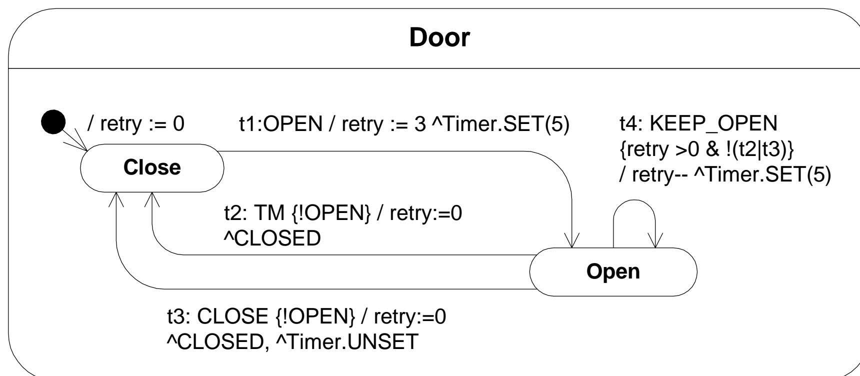


Figure B-8: Door State Diagram

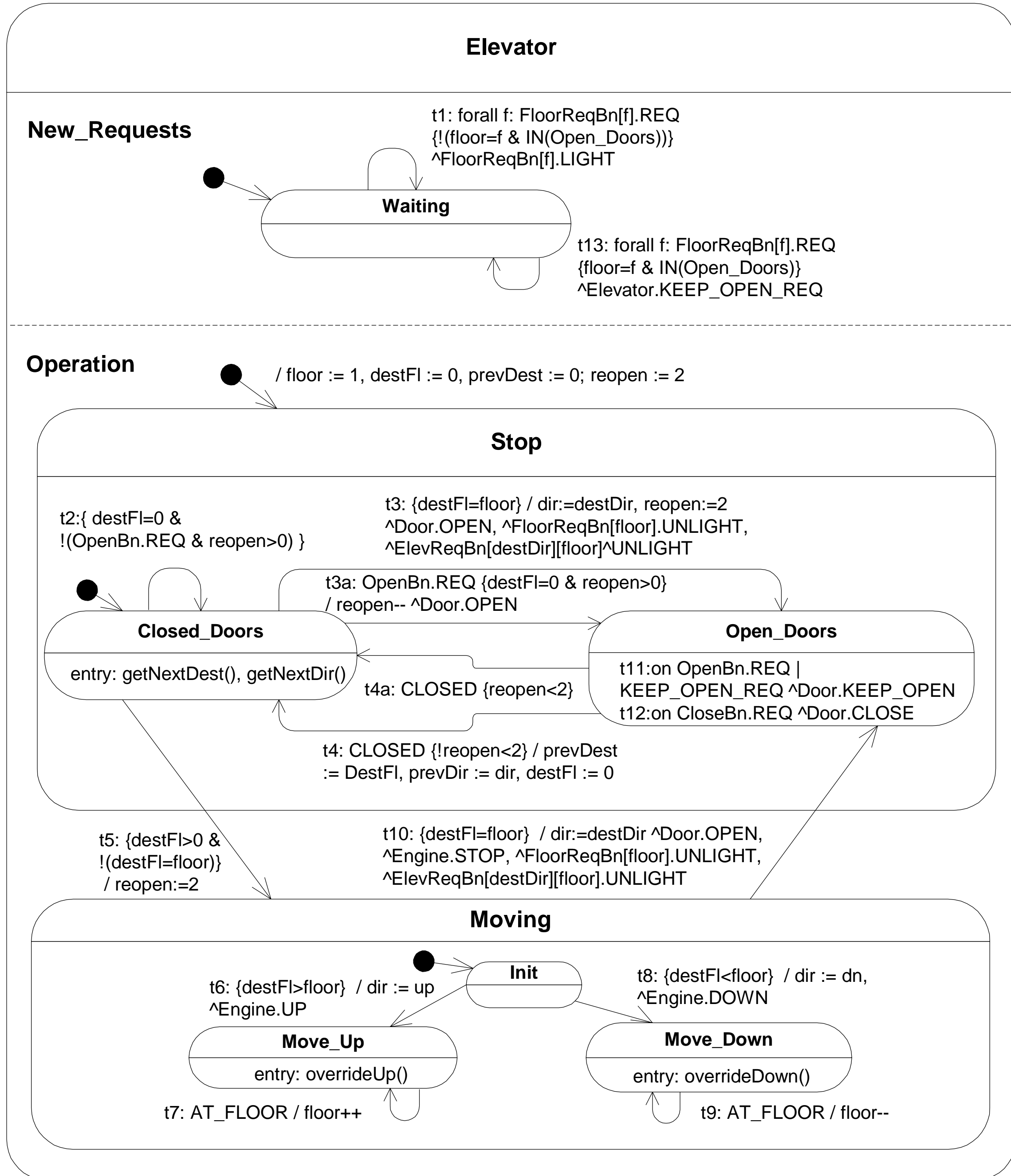


Figure B-9: Elevator State Diagram with Two Concurrent Regions

The getNextDest operation macros (going up only) in decision table format:

States	**Condition 0**	**Condition 1**	**Condition 2**	**Condition 3**
IN(Elevator.UP)	!(f1_pending | f2_pending_up | f2_pending_dn | f3_pending)	up_f1_t1_1 | up_f1_t1_n | up_f2_t1 | up_f3_t1	up_f1_t2_sd | up_f1_t2_od | up_f2_t2_1 | up_f2_t2_n | up_f2_t2_od | up_f3_t2_sd | up_f3_t2_od	up_f1_t3 | up_f2_t3 | up_f3_t3_1 up_f3_t3_n
IN(Elevator. DOWN)	!(f1_pending | f2_pending_up | f2_pending_dn | f3_pending)	dn_f1_t1_1 | dn_f1_t1_n | dn_f2_t1 | dn_f3_t1	dn_f3_t2_sd | dn_f3_t2_od | dn_f2_t2_1 | dn_f2_t2_n | dn_f2_t2_od | dn_f1_t2_sd | dn_f1_t2_od	dn_f1_t3 | dn_f2_t3 | dn_f3_t3_1 dn_f3_t3_n
destFl	**0**	**1**	**2**	**3**

Table B-1: getNext operation modeled as a decision table

```
f1_pending = ElevReqBnUp1.pending | FloorReqBn1.pending
f2_pending_up = ElevReqBnUp2.pending | FloorReqBn2.pending
f2_pending_dn = ElevReqBnDn2.pending | FloorReqBn2.pending
f3_pending = ElevReqBnDn3.pending | FloorReqBn3.pending
```

	up_f1_t1_1	**up_f1_t1_n**	**up_f2_t1**	**up_f3_t1**
currFloor = 1	T	T	.	.
currFloor = 2	.	.	T	.
currFloor = 3	.	.	.	T
prevDest = 1	F	T	.	.
f1_pending	T	T	T	T
f2_pending_up	.	F	.	.
f2_pending_dn	.	F	.	.
f3_pending	.	F	.	.
up_f2_t2_1	.	.	F	.
up_f2_t2_n	.	.	F	.
up_f2_t3	.	.	F	.
up_f2_t2_od	.	.	F	.
up_f3_t3_1	.	.	.	F
up_f3_t3_n	.	.	.	F
up_f3_t2_od	.	.	.	F

Table B-2: Macros for going to floor 1 with direction up

	up_f1_ t2_sd	up_f1_ t2_od	up_f2_ t2_1	up_f2_ t2_n	up_f2_ t2_od	up_f3_ t2_sd	up_f3_ t2_od
currFloor = 1	T	T	.	.	.	.	.
currFloor = 2	.	.	T	T	T	.	.
currFloor = 3	.	.	.	.	.	T	T
prevDest = 2 & prevDir = up	.	.	F	T	.	.	.
f1_pending	.	.	.	F	.	.	.
f2_pending_up	T	.	T	T	.	.	.
f2_pending_dn	.	.	.	.	.	.	T
f3_pending	.	.	.	F	.	.	.
ElevReqBnUp2 .pending	.	.	.	.	.	T	.
ElevReqBnDn2 .pending	.	T	.	F	T	.	.
up_f1_t1_1	F	F	.	.	.	.	.
up_f1_t1_n	F	F	.	.	.	.	.
up_f1_t2_sd	X	F	.	.	.	.	.
up_f1_t3	.	F	.	.	.	.	.
up_f2_t2_1	.	.	X	.	F	.	.
up_f2_t2_n	.	.	.	X	F	.	.
up_f2_t3	.	.	.	.	F	.	.
up_f3_t3_1	.	.	.	.	.	F	F
up_f3_t3_n	.	.	.	.	.	F	F
up_f3_t2_od	.	.	.	.	.	F	X
up_f3_t1	.	.	.	.	.	F	.

Table B-3: Macros for going to floor 2 with direction up

	up_f1_t3	up_f2_t3	up_f3_t3_1	up_f3_t3_n
currFloor = 1	T	.	.	.
currFloor = 2	.	T	.	.
currFloor = 3	.	.	T	T
prevDest = 3	.	.	F	T
f1_pending	.	.	.	F
f2_pending_up	.	.	.	F
f2_pending_dn	.	.	.	F
f3_pending	T	T	T	T
up_f1_t1_1	F	.	.	.
up_f1_t1_n	F	.	.	.
up_f1_t2_sd	F	.	.	.
up_f2_t2_1	.	F	.	.
up_f2_t2_n	.	F	.	.

Table B-4: Macros for going to floor 3 with direction up

Appendix C: Elevator SMV Program

Line	Program

```
     -- SMV Program: elevator.smv
     -- By: Meyer Tanuan
     -- Date: 21-Jul-01
     -- Test Platform: Cadence SMV / Windows 98 / 128 MB

     -- This program matches the complete UML model for 3-level elevator
     -- BDD Nodes Allocated: 845,803
     -- Reachable states: Number of state holding booleans = 37
     -- Number of states reached: 125,760


  1  -- ****************************************************************
  2  -- *                      MAIN MODULE                             *
  3  -- ****************************************************************
  4  MODULE main
  5
  6  -- ****************** VARIABLES SECTION *************************
  7  VAR
  8   -- *** ENVIRONMENT VARIABLE SUBSECTION ***
  9   external_Event : External_Event;
 10
 11   -- *** STATE VARIABLES SUBSECTION ***
 12   st_Building : { Idle };
 13
 14   -- *** COMPONENT OBJECTS SUBSECTION ***
 15   elevator : Elevator(ev_KEEP_OPEN_REQ, external_Event.env_BP_OpenBn,
     external_Event.env_BP_CloseBn, external_Event.env_BP_FloorReqBn[1],
     external_Event.env_BP_FloorReqBn[2],
     external_Event.env_BP_FloorReqBn[3], external_Event.env_AT_FLOOR,
     ElevReqBnUp1, ElevReqBnUp2, ElevReqBnDn2, ElevReqBnDn3);
 16
 17   ElevReqBnUp1 : RequestBn(external_Event.env_BP_ElevReqBnUp[1],
     ev_LIGHT_ElevReqBnUp[1], elevator.ev_UNLIGHT_ElevReqBnUp[1]);
 18   ElevReqBnUp2 : RequestBn(external_Event.env_BP_ElevReqBnUp[2],
     ev_LIGHT_ElevReqBnUp[2], elevator.ev_UNLIGHT_ElevReqBnUp[2]);
 19   ElevReqBnDn2 : RequestBn(external_Event.env_BP_ElevReqBnDn[2],
     ev_LIGHT_ElevReqBnDn[2], elevator.ev_UNLIGHT_ElevReqBnDn[2]);
 20   ElevReqBnDn3 : RequestBn(external_Event.env_BP_ElevReqBnDn[3],
     ev_LIGHT_ElevReqBnDn[3], elevator.ev_UNLIGHT_ElevReqBnDn[3]);
 21
 22  -- ***************** SYMBOL DEFINITION SECTION ******************
 23  DEFINE
 24   -- *** TRANSITION MACROS SUBSECTION ***
 25   m_ElevReqBnUp1 := (elevator.in_Open_Doors & elevator.floor=1 &
     elevator.dir=up);
 26   m_ElevReqBnUp2 := (elevator.in_Open_Doors & elevator.floor=2 &
     elevator.dir=up);
 27   m_ElevReqBnDn2 := (elevator.in_Open_Doors & elevator.floor=2 &
     elevator.dir=dn);
 28   m_ElevReqBnDn3 := (elevator.in_Open_Doors & elevator.floor=3 &
     elevator.dir=dn);
 29
 30   t1_ElevReqBnUp1 := ElevReqBnUp1.ev_REQ & !m_ElevReqBnUp1;
 31   t1_ElevReqBnUp2 := ElevReqBnUp2.ev_REQ & !m_ElevReqBnUp2;
 32   t1_ElevReqBnDn2 := ElevReqBnDn2.ev_REQ & !m_ElevReqBnDn2;
 33   t1_ElevReqBnDn3 := ElevReqBnDn3.ev_REQ & !m_ElevReqBnDn3;
 34   t1 := in_Idle & (t1_ElevReqBnUp1 | t1_ElevReqBnUp2 | t1_ElevReqBnDn2 |
     t1_ElevReqBnDn3);
```

```
35
36   t2_ElevReqBnUp1 := ElevReqBnUp1.ev_REQ & m_ElevReqBnUp1;
37   t2_ElevReqBnUp2 := ElevReqBnUp2.ev_REQ & m_ElevReqBnUp2;
38   t2_ElevReqBnDn2 := ElevReqBnDn2.ev_REQ & m_ElevReqBnDn2;
39   t2_ElevReqBnDn3 := ElevReqBnDn3.ev_REQ & m_ElevReqBnDn3;
40   t2 := in_Idle & (t2_ElevReqBnUp1 | t2_ElevReqBnUp2 | t2_ElevReqBnDn2 |
    t2_ElevReqBnDn3);
41
42   -- *** GENERATED EVENTS MACROS SUBSECTION ***
43   ev_LIGHT_ElevReqBnUp[1] := t1_ElevReqBnUp1;
44   ev_LIGHT_ElevReqBnUp[2] := t1_ElevReqBnUp2;
45   ev_LIGHT_ElevReqBnDn[2] := t1_ElevReqBnDn2;
46   ev_LIGHT_ElevReqBnDn[3] := t1_ElevReqBnDn3;
47   ev_KEEP_OPEN_REQ := t2_ElevReqBnUp1 | t2_ElevReqBnUp2 | t2_ElevReqBnDn2
    | t2_ElevReqBnDn3;
48
49  -- ****************** TRANS CONDITION ************************
50  TRANS
51         !(external_Event.env_AT_FLOOR & elevator.t10)
52
53  -- ****************** ASSIGNMENT SECTION ************************
54  ASSIGN
55
56   -- *** STATE VARIABLES SUBSECTION ***
57   init(st_Building) := Idle;
58   next(st_Building) := case
59     t1 | t2 : Idle;
60     1 : st_Building;
61    esac;
62
63  -- ********* SYSTEM PROPERTIES (CTL FORMULAS) *******************
64
65  -- Test 1: Safety property for elevator (Section 4.2.1)
66  -- It is never the case that the door is open while the elevator is
    Moving
67  SPEC AG ( !(elevator.in_Open_Doors & elevator.in_Moving) )
68
69  -- Test 2: Liveness properties for elevator and hall buttons (Section
    4.2.2)
70  -- *** tests the hypothesis to avoid coincidental correctness ***
71  SPEC EF(external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors &
    elevator.floor=1))
72  SPEC EF (external_Event.env_BP_FloorReqBn[2] & !(elevator.in_Open_Doors
    & elevator.floor=2))
73  SPEC EF (external_Event.env_BP_FloorReqBn[3] & !(elevator.in_Open_Doors
    & elevator.floor=3))
74  SPEC EF (external_Event.env_BP_ElevReqBnUp[1] & !(elevator.in_Open_Doors
    & elevator.floor=1))
75  SPEC EF (external_Event.env_BP_ElevReqBnUp[2] & !(elevator.in_Open_Doors
    & elevator.floor=2 & elevator.dir=up))
76  SPEC EF (external_Event.env_BP_ElevReqBnDn[2] & !(elevator.in_Open_Doors
    & elevator.floor=2 & elevator.dir=dn))
77  SPEC EF (external_Event.env_BP_ElevReqBnDn[3] & !(elevator.in_Open_Doors
    & elevator.floor=3))
78
79  -- *** tests showing behavior of buttons and opening of door ***
80  SPEC AG ( external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors
    & elevator.floor=1)
81         -> AF (elevator.floor = 1 & elevator.in_Open_Doors) )
82  SPEC AG ( external_Event.env_BP_FloorReqBn[2] & !(elevator.in_Open_Doors
    & elevator.floor=2)
83         -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )
84  SPEC AG ( external_Event.env_BP_FloorReqBn[3] & !(elevator.in_Open_Doors
```

```
    & elevator.floor=3)
 85        -> AF (elevator.floor = 3 & elevator.in_Open_Doors) )
 86 SPEC AG ( external_Event.env_BP_ElevReqBnUp[1] &
    !(elevator.in_Open_Doors & elevator.floor=1)
 87        -> AF (elevator.floor = 1 & elevator.in_Open_Doors) )
 88 SPEC AG ( external_Event.env_BP_ElevReqBnUp[2] &
    !(elevator.in_Open_Doors & elevator.floor=2 & elevator.dir=up)
 89        -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )
 90 SPEC AG ( external_Event.env_BP_ElevReqBnDn[2] &
    !(elevator.in_Open_Doors & elevator.floor=2 & elevator.dir=dn)
 91        -> AF (elevator.floor = 2 & elevator.in_Open_Doors) )
 92 SPEC AG ( external_Event.env_BP_ElevReqBnDn[3] &
    !(elevator.in_Open_Doors & elevator.floor=3)
 93        -> AF (elevator.floor = 3 & elevator.in_Open_Doors) )
 94
 95 -- *** additional tests showing behavior of button lights ***
 96 SPEC AG ( external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors
    & elevator.floor=1)
 97        -> AF (elevator.FloorReqBn1.in_Lit &
    AF(elevator.FloorReqBn1.in_UnLit) ) )
 98 SPEC AG ( external_Event.env_BP_FloorReqBn[2] & !(elevator.in_Open_Doors
    & elevator.floor=2)
 99        -> AF (elevator.FloorReqBn2.in_Lit &
    AF(elevator.FloorReqBn2.in_UnLit) ) )
100 SPEC AG ( external_Event.env_BP_FloorReqBn[3] & !(elevator.in_Open_Doors
    & elevator.floor=3)
101        -> AF (elevator.FloorReqBn3.in_Lit &
    AF(elevator.FloorReqBn3.in_UnLit) ) )
102 SPEC AG ( external_Event.env_BP_ElevReqBnUp[1] &
    !(elevator.in_Open_Doors & elevator.floor=1)
103        -> AF (ElevReqBnUp1.in_Lit & AF(ElevReqBnUp1.in_UnLit) ) )
104 SPEC AG ( external_Event.env_BP_ElevReqBnUp[2] &
    !(elevator.in_Open_Doors & elevator.floor=2 & elevator.dir=up)
105        -> AF (ElevReqBnUp2.in_Lit & AF(ElevReqBnUp2.in_UnLit) ) )
106 SPEC AG ( external_Event.env_BP_ElevReqBnDn[2] &
    !(elevator.in_Open_Doors & elevator.floor=2 & elevator.dir=dn)
107        -> AF (ElevReqBnDn2.in_Lit & AF(ElevReqBnDn2.in_UnLit) ) )
108 SPEC AG ( external_Event.env_BP_ElevReqBnDn[3] &
    !(elevator.in_Open_Doors & elevator.floor=3)
109        -> AF (ElevReqBnDn3.in_Lit & AF(ElevReqBnDn3.in_UnLit) ) )
110
111 -- Test 3: Check additional safety property (Section 4.2.3)
112 -- *** From any state, it is possible to go to Stop ***
113 SPEC AG(EF(elevator.in_Stop))
114
115 -- Test 4:  Additional liveness property (Section 4.2.3)
116 -- *** response chain example (up2 & dn2) using button lights ***
117 SPEC EF(ElevReqBnUp2.in_Lit & ElevReqBnDn2.in_Lit & elevator.floor = 1 &
    elevator.in_Closed_Doors)
118 SPEC AG ( (ElevReqBnUp2.in_Lit & ElevReqBnDn2.in_Lit & elevator.floor =
    1 & elevator.in_Closed_Doors)
119        -> AF( (ElevReqBnUp2.in_UnLit & ElevReqBnDn2.in_Lit) &
    AF(ElevReqBnDn2.in_UnLit) ) )
120
121 -- *** weaker property using button pressed events as trigger ***
122 SPEC EF(external_Event.env_BP_ElevReqBnUp[2] &
    external_Event.env_BP_ElevReqBnDn[2] & elevator.floor = 1 &
    elevator.in_Closed_Doors)
123 SPEC AG ( (external_Event.env_BP_ElevReqBnUp[2] &
    external_Event.env_BP_ElevReqBnDn[2] & elevator.floor = 1 &
    elevator.in_Closed_Doors)
124        -> AF( (ElevReqBnUp2.in_UnLit & ElevReqBnDn2.in_Lit) &
    AF(ElevReqBnDn2.in_UnLit) ) )
```

```
125
126  -- *** additional response chain example (up2 & dn3) using button lights
     ***
127  SPEC EF(ElevReqBnUp2.in_Lit & ElevReqBnDn3.in_Lit & elevator.floor = 1 &
     elevator.in_Closed_Doors)
128  SPEC AG ( (ElevReqBnUp2.in_Lit & ElevReqBnDn3.in_Lit & elevator.floor =
     1 & elevator.in_Closed_Doors)
129         -> AF( (ElevReqBnUp2.in_UnLit & ElevReqBnDn3.in_Lit) &
     AF(ElevReqBnDn3.in_UnLit) ) )
130
131  -- *** weaker property using button pressed events as trigger ***
132  SPEC EF(external_Event.env_BP_ElevReqBnUp[2] &
     external_Event.env_BP_ElevReqBnDn[3] & elevator.floor = 1 &
     elevator.in_Closed_Doors)
133  SPEC AG ( (external_Event.env_BP_ElevReqBnUp[2] &
     external_Event.env_BP_ElevReqBnDn[3] & elevator.floor = 1 &
     elevator.in_Closed_Doors)
134         -> AF( (ElevReqBnUp2.in_UnLit & ElevReqBnDn3.in_Lit) &
     AF(ElevReqBnDn3.in_UnLit) ) )
135
136  -- Test 5: Check safety property of system's state variables
137  -- *** prevDest attribute never goes back to zero, once it has been set
     (i.e., > 0)
138  SPEC EF(elevator.prevDest>0)
139  SPEC AG( (elevator.prevDest>0) -> AG(!(elevator.prevDest=0)) )
140
141  -- Test 6: Check additional liveness property
142  -- *** If Request Button is pressed when it is not pending, it
     eventually generates NEW_REQ
143  SPEC EF(external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors &
     elevator.floor=1) & elevator.FloorReqBn1.pending = 0)
144  SPEC AG ( external_Event.env_BP_FloorReqBn[1] & !(elevator.in_Open_Doors
     & elevator.floor=1)
145         & (elevator.FloorReqBn1.pending = 0)
146         -> AF (elevator.FloorReqBn1.ev_REQ) )
147
148  -- Test 7: Check additional liveness property for elevator door
149  -- *** Doors cannot be open indefinitely ***
150  SPEC EF(elevator.door.in_Open)
151  SPEC AG (elevator.door.in_Open -> AF(!elevator.door.in_Open))
152
153  -- Test 8: Quick Test for 2-passenger scenario
154  -- Pick up passenger on f2 or deliver to f2 while going up to f3
155  SPEC EF ((external_Event.env_BP_ElevReqBnUp[2] |
     external_Event.env_BP_FloorReqBn[2]) & elevator.destFl = 3 &
     elevator.floor = 1 & elevator.in_Init)
156  SPEC AG ( ((external_Event.env_BP_ElevReqBnUp[2] |
     external_Event.env_BP_FloorReqBn[2]) & elevator.destFl = 3 &
     elevator.floor = 1 & elevator.in_Init)
157         -> AF( (elevator.floor = 2 & elevator.door.in_Open) &
     AF(elevator.floor = 3 & elevator.door.in_Open) ) )
158
159  -- Test 9: New tests to verify elevator's reopen attribute
160  -- Doors will re-open if open requested and doors are closed within 2
     retries (i.e., reopen has not reached 0)
161  SPEC EF (external_Event.env_BP_OpenBn & elevator.in_Closed_Doors &
     elevator.reopen>01)
162
163  SPEC AG (external_Event.env_BP_OpenBn & elevator.in_Closed_Doors &
     elevator.reopen>0
164         -> AF(!elevator.door.in_Open))
165
166  -- Test 10: Check partial specifications (Section 4.3)
```

```
167  -- *** Test countdown feature: If counter = 5 and timer is not reset,
     then it is immediately followed by counter = 4 ***
168  SPEC EF(elevator.door.DoorTimer.counter=5 &
     !elevator.door.ev_SET.raised)
169  SPEC AG ( (elevator.door.DoorTimer.counter=5 &
     !elevator.door.ev_SET.raised)
170         -> AX (elevator.door.DoorTimer.counter = 4) )
171
172  -- Test 11: Check partial specifications (Section 4.3)
173  -- *** Test timeout feature: If countdown reached 0 (and no reset or
     unset), then it is immediately followed by transition to Idle state
     (i.e., timeout raised) ***
174  SPEC EF(elevator.door.DoorTimer.counter=0 & !elevator.door.ev_SET.raised
     & !elevator.door.ev_UNSET & elevator.door.DoorTimer.in_Count_Down)
175  SPEC AG ( ( elevator.door.DoorTimer.counter=0 &
     !elevator.door.ev_SET.raised
176         & !elevator.door.ev_UNSET & elevator.door.DoorTimer.in_Count_Down
     )
177         -> AX (elevator.door.DoorTimer.in_Idle) )
178
179  -- ****************** End of Main Module ************************
```

```
  1  -- ***************************************************************
  2  -- *                      MODULE Elevator                         *
  3  -- ***************************************************************
  4  MODULE Elevator(inp_KEEP_OPEN_REQ, inp_BP_OpenBn, inp_BP_CloseBn,
     inp_BP_FloorReqBn1, inp_BP_FloorReqBn2, inp_BP_FloorReqBn3,
     inp_AT_FLOOR, inp_ElevReqBnUp1, inp_ElevReqBnUp2, inp_ElevReqBnDn2,
     inp_ElevReqBnDn3)
  5
  6  -- ****************** VARIABLES SECTION ************************
  7  VAR
  8   -- *** STATE VARIABLES SUBSECTION ***
  9   st_New_Requests : { Waiting };
 10   st_Operation : { Stop, Moving };
 11   st_Stop : { Closed_Doors, Open_Doors, Undefined };
 12   st_Moving : { Init, Move_Up, Move_Down, Undefined };
 13
 14   -- *** CLASS ATTRIBUTES SUBSECTION ***
 15   floor : 1..3;
 16   dir : {up, dn};
 17   destFl : 0..3;
 18   destDir : {up, dn};
 19   prevDest : 0..3;
 20   prevDir : {up, dn};
 21   reopen : 0..2;
 22
 23   -- *** COMPONENT OBJECTS SUBSECTION ***
 24   OpenBn: Button(inp_BP_OpenBn);
 25   CloseBn: Button(inp_BP_CloseBn);
 26   FloorReqBn1 : RequestBn(inp_BP_FloorReqBn1, ev_LIGHT_FloorReqBn[1],
     ev_UNLIGHT_FloorReqBn[1]);
 27   FloorReqBn2 : RequestBn(inp_BP_FloorReqBn2, ev_LIGHT_FloorReqBn[2],
     ev_UNLIGHT_FloorReqBn[2]);
 28   FloorReqBn3 : RequestBn(inp_BP_FloorReqBn3, ev_LIGHT_FloorReqBn[3],
     ev_UNLIGHT_FloorReqBn[3]);
 29   door: Door(ev_OPEN, ev_KEEP_OPEN, ev_CLOSE);
 30
 31  -- **************** SYMBOL DEFINITION SECTION *****************
 32  DEFINE
 33   -- *** ACTIVE STATE MACROS SUBSECTION ***
 34   in_Elevator := 1;
```

```
35
36   in_New_Requests := in_Elevator;
37   in_Waiting := in_New_Requests & st_New_Requests = Waiting;
38
39   in_Operation := in_Elevator;
40   in_Stop := in_Operation & st_Operation = Stop;
41   in_Closed_Doors := in_Stop & st_Stop = Closed_Doors;
42   in_Open_Doors := in_Stop & st_Stop = Open_Doors;
43
44   in_Moving := in_Operation & st_Operation = Moving;
45   in_Init := in_Moving & st_Moving = Init;
46   in_Move_Up := in_Moving & st_Moving = Move_Up;
47   in_Move_Down := in_Moving & st_Moving = Move_Down;
48
49   -- *** TRANSITION MACROS SUBSECTION ***
50   t1_FloorReqBn1 := FloorReqBn1.ev_REQ & !(in_Open_Doors & floor=1);
51   t1_FloorReqBn2 := FloorReqBn2.ev_REQ & !(in_Open_Doors & floor=2);
52   t1_FloorReqBn3 := FloorReqBn3.ev_REQ & !(in_Open_Doors & floor=3);
53   t1 := in_Waiting & (t1_FloorReqBn1 | t1_FloorReqBn2 | t1_FloorReqBn3);
54   t2 := in_Closed_Doors & (destFl=0) & !(OpenBn.ev_REQ & reopen>0);
55   t3 := in_Closed_Doors & destFl=floor;
56   t3a := in_Closed_Doors & destFl = 0 & OpenBn.ev_REQ & reopen>0;
57   t4 := in_Open_Doors & door.ev_CLOSED & !reopen<2;
58   t4a := in_Open_Doors & door.ev_CLOSED & reopen<2;
59   t5 := in_Closed_Doors & destFl > 0 & !(destFl = floor);
60   t6 := in_Init & (destFl > floor);
61   t7 := in_Move_Up & (inp_AT_FLOOR);
62   t8 := in_Init & (destFl < floor);
63   t9 := in_Move_Down & (inp_AT_FLOOR);
64   t10 := in_Moving & (destFl = floor);
65   t11 := in_Open_Doors & (inp_KEEP_OPEN_REQ | OpenBn.ev_REQ);
66   t12 := in_Open_Doors & CloseBn.ev_REQ & !t11;
67
68   t3_t10_FloorReqBn1 := (t3 | t10) & destFl=1;
69   t3_t10_FloorReqBn2 := (t3 | t10) & destFl=2;
70   t3_t10_FloorReqBn3 := (t3 | t10) & destFl=3;
71   t3_t10_ElevReqBnUp1 := (t3 | t10) & destFl=1 & destDir=up;
72   t3_t10_ElevReqBnUp2 := (t3 | t10) & destFl=2 & destDir=up;
73   t3_t10_ElevReqBnDn2 := (t3 | t10) & destFl=2 & destDir=dn;
74   t3_t10_ElevReqBnDn3 := (t3 | t10) & destFl=3 & destDir=dn;
75
76   t13_FloorReqBn1 := FloorReqBn1.ev_REQ & (in_Open_Doors & floor=1);
77   t13_FloorReqBn2 := FloorReqBn2.ev_REQ & (in_Open_Doors & floor=2);
78   t13_FloorReqBn3 := FloorReqBn3.ev_REQ & (in_Open_Doors & floor=3);
79   t13 := in_Waiting & (t13_FloorReqBn1 | t13_FloorReqBn2 |
    t13_FloorReqBn3);
80
81   -- *** GENERATED EVENTS MACROS SUBSECTION ***
82   ev_OPEN := t3 | t3a | t10;
83   ev_KEEP_OPEN := t11;
84   ev_CLOSE := t12;
85   ev_LIGHT_FloorReqBn[1] := t1_FloorReqBn1;
86   ev_LIGHT_FloorReqBn[2] := t1_FloorReqBn2;
87   ev_LIGHT_FloorReqBn[3] := t1_FloorReqBn3;
88   ev_UNLIGHT_FloorReqBn[1] := t3_t10_FloorReqBn1;
89   ev_UNLIGHT_FloorReqBn[2] := t3_t10_FloorReqBn2;
90   ev_UNLIGHT_FloorReqBn[3] := t3_t10_FloorReqBn3;
91   ev_UNLIGHT_ElevReqBnUp[1] := t3_t10_ElevReqBnUp1;
92   ev_UNLIGHT_ElevReqBnUp[2] := t3_t10_ElevReqBnUp2;
93   ev_UNLIGHT_ElevReqBnDn[2] := t3_t10_ElevReqBnDn2;
94   ev_UNLIGHT_ElevReqBnDn[3] := t3_t10_ElevReqBnDn3;
95   ev_KEEP_OPEN_REQ := (t13_FloorReqBn1 | t13_FloorReqBn2 |
    t13_FloorReqBn3);
```

```
 96
 97   -- *** OPERATION MACROS SUBSECTION ***
 98
 99   -- *** getNextDest and getNextDir operations ***
100   m_f1_pending := (inp_ElevReqBnUp1.pending | FloorReqBn1.pending);
101   m_f2_pending_up := (inp_ElevReqBnUp2.pending | FloorReqBn2.pending);
102   m_f2_pending_dn := (inp_ElevReqBnDn2.pending | FloorReqBn2.pending);
103   m_f3_pending := (inp_ElevReqBnDn3.pending | FloorReqBn3.pending);
104
105   -- Going Up:
106
107   -- ***** First Floor: By Priority *****
108   m_up_f1_t1_1 := (dir=up) & (floor=1) & !(prevDest=1) & m_f1_pending;
109   m_up_f1_t1_n := (dir=up) & (floor=1) & (prevDest=1) & m_f1_pending &
     !(m_f2_pending_up | m_f2_pending_dn | m_f3_pending);
110   m_up_f1_t2_sd := (dir=up) & (floor=1) & m_f2_pending_up &
     !(m_up_f1_t1_1 | m_up_f1_t1_n);
111   m_up_f1_t3 := (dir=up) & (floor=1) & m_f3_pending & !(m_up_f1_t1_1 |
     m_up_f1_t1_n | m_up_f1_t2_sd);
112   m_up_f1_t2_od := (dir=up) & (floor=1) & inp_ElevReqBnDn2.pending &
     !(m_up_f1_t1_1 | m_up_f1_t1_n | m_up_f1_t2_sd | m_up_f1_t3);
113
114   -- ***** Second Floor: By Priority *****
115   m_up_f2_t2_1 := (dir=up) & (floor=2) & !(prevDest=2 & prevDir = up) &
     m_f2_pending_up;
116   m_up_f2_t2_n := (dir=up) & (floor=2) & (prevDest=2 & prevDir = up) &
     m_f2_pending_up & !(m_f1_pending | m_f3_pending |
     inp_ElevReqBnDn2.pending);
117   m_up_f2_t3 := (dir=up) & (floor=2) & m_f3_pending & !(m_up_f2_t2_1 |
     m_up_f2_t2_n);
118   m_up_f2_t2_od := (dir=up) & (floor=2) & inp_ElevReqBnDn2.pending &
     !(m_up_f2_t2_1 | m_up_f2_t2_n | m_up_f2_t3);
119   m_up_f2_t1 := (dir=up) & (floor=2) & m_f1_pending & !(m_up_f2_t2_1 |
     m_up_f2_t2_n | m_up_f2_t3 | m_up_f2_t2_od);
120
121   -- ***** Third Floor: By Priority *****
122   m_up_f3_t3_1 := (dir=up) & (floor=3) & !(prevDest=3) & m_f3_pending;
123   m_up_f3_t3_n := (dir=up) & (floor=3) & (prevDest=3) & m_f3_pending &
     !(m_f1_pending | m_f2_pending_up | m_f2_pending_dn);
124   m_up_f3_t2_od := (dir=up) & (floor=3) & m_f2_pending_dn &
     !(m_up_f3_t3_1 | m_up_f3_t3_n);
125   m_up_f3_t1 := (dir=up) & (floor=3) & m_f1_pending & !(m_up_f3_t3_1 |
     m_up_f3_t3_n | m_up_f3_t2_od);
126   m_up_f3_t2_sd := (dir=up) & (floor=3) & inp_ElevReqBnUp2.pending &
     !(m_up_f3_t3_1 | m_up_f3_t3_n | m_up_f3_t2_od | m_up_f3_t1);
127
128   -- Going Down:
129
130   -- ***** Third Floor: By Priority *****
131   m_dn_f3_t3_1 := (dir=dn) & (floor=3) & !(prevDest=3) & m_f3_pending;
132   m_dn_f3_t3_n := (dir=dn) & (floor=3) & (prevDest=3) & m_f3_pending &
     !(m_f1_pending | m_f2_pending_up | m_f2_pending_dn);
133   m_dn_f3_t2_sd := (dir=dn) & (floor=3) & m_f2_pending_dn &
     !(m_dn_f3_t3_1 | m_dn_f3_t3_n);
134   m_dn_f3_t1 := (dir=dn) & (floor=3) & m_f1_pending & !(m_dn_f3_t3_1 |
     m_dn_f3_t3_n | m_dn_f3_t2_sd);
135   m_dn_f3_t2_od := (dir=dn) & (floor=3) & inp_ElevReqBnUp2.pending &
     !(m_dn_f3_t3_1 | m_dn_f3_t3_n | m_dn_f3_t2_sd | m_dn_f3_t1);
136
137   -- ***** Second Floor: By Priority *****
138   m_dn_f2_t2_1 := (dir=dn) & (floor=2) & !(prevDest=2 & prevDir = dn) &
     m_f2_pending_dn;
139   m_dn_f2_t2_n := (dir=dn) & (floor=2) & (prevDest=2 & prevDir = dn) &
```

```
    m_f2_pending_dn & !(m_f3_pending | m_f1_pending |
    inp_ElevReqBnUp2.pending);
140  m_dn_f2_t1 := (dir=dn) & (floor=2) & m_f1_pending & !(m_dn_f2_t2_1 |
    m_dn_f2_t2_n);
141  m_dn_f2_t2_od := (dir=dn) & (floor=2) & inp_ElevReqBnUp2.pending &
    !(m_dn_f2_t2_1 | m_dn_f2_t2_n | m_dn_f2_t1);
142  m_dn_f2_t3 := (dir=dn) & (floor=2) & m_f3_pending & !(m_dn_f2_t2_1 |
    m_dn_f2_t2_n | m_dn_f2_t1 | m_dn_f2_t2_od);
143
144  -- ***** First Floor: By Priority *****
145  m_dn_f1_t1_1 := (dir=dn) & (floor=1) & !(prevDest=1) & m_f1_pending;
146  m_dn_f1_t1_n := (dir=dn) & (floor=1) & (prevDest=1) & m_f1_pending &
    !(m_f2_pending_up | m_f2_pending_dn | m_f3_pending);
147  m_dn_f1_t2_od := (dir=dn) & (floor=1) & m_f2_pending_up &
    !(m_dn_f1_t1_1 | m_dn_f1_t1_n);
148  m_dn_f1_t3 := (dir=dn) & (floor=1) & m_f3_pending & !(m_dn_f1_t1_1 |
    m_dn_f1_t1_n | m_dn_f1_t2_od);
149  m_dn_f1_t2_sd := (dir=dn) & (floor=1) & inp_ElevReqBnDn2.pending &
    !(m_dn_f1_t1_1 | m_dn_f1_t1_n | m_dn_f1_t2_od | m_dn_f1_t3);
150
151  -- *** overrideUp and overrideDown operations ***
152  m_overrideUp := in_Move_Up & next(floor)<=2 & !(prevDest=2) & destFl=3
    & m_f2_pending_up;
153  m_overrideDown := in_Move_Down & next(floor)>=2 & !(prevDest=2) &
    destFl=1 & m_f2_pending_dn;
154
155 -- *** override the direction on 2nd floor before open doors ***
156  m_overrideUp_dir := in_Move_Up & next(floor)<=2 & !(prevDest=2) &
    destFl=2 & destDir=dn & inp_ElevReqBnUp2.pending;
157  m_overrideDown_dir := in_Move_Down & next(floor)>=2 & !(prevDest=2) &
    destFl=2 & destDir=up & inp_ElevReqBnDn2.pending;
158
159 -- ****************** ASSIGNMENT SECTION ************************
160 ASSIGN
161
162  -- *** STATE VARIABLES SUBSECTION ***
163  init(st_New_Requests) := Waiting;
164  next(st_New_Requests) := case
165    t1 | t13 : Waiting;
166    1 : st_New_Requests;
167   esac;
168
169  init(st_Operation) := Stop;
170  next(st_Operation) := case
171    t5 : Moving;
172    t10 : Stop;
173    1 : st_Operation;
174   esac;
175
176  init(st_Stop) := Closed_Doors;
177  next(st_Stop) := case
178    t3 | t3a | t10 : Open_Doors;
179    t2 | t4 | t4a : Closed_Doors;
180    t5 : Undefined;
181    1 : st_Stop;
182   esac;
183
184  init(st_Moving) := Undefined;
185  next(st_Moving) := case
186    t5 : Init;
187    t6 | t7 : Move_Up;
188    t8 | t9 : Move_Down;
189    t10 : Undefined;
```

```
190      1 : st_Moving;
191     esac;
192
193    -- *** CLASS ATTRIBUTES SUBSECTION ***
194    init(floor) := 1;
195    next(floor) := case
196      t7 : floor + 1;
197      -- Note: space before the minus operator is required
198      t9 : floor - 1;
199      1 : floor;
200     esac;
201
202    init(dir) := up;
203    next(dir) := case
204      t6 : up;
205      t8 : dn;
206      t3 | t10 : destDir;
207      1: dir;
208     esac;
209
210    init(destFl) := 0;
211    next(destFl) := case
212      t4 : 0;
213
214      -- *** override on 2nd floor ***
215      (t6 | t7) & m_overrideUp : 2;
216      (t8 | t9) & m_overrideDown : 2;
217
218      -- ********* no pending requests ************
219      (t2) & !(m_f1_pending | m_f2_pending_up | m_f2_pending_dn |
   m_f3_pending): 0;
220
221      -- ****************  going up ***************
222      (t2) & (m_up_f1_t1_1 | m_up_f1_t1_n | m_up_f2_t1 | m_up_f3_t1): 1;
223      (t2) & (m_up_f1_t2_sd | m_up_f1_t2_od | m_up_f2_t2_1 | m_up_f2_t2_n |
   m_up_f2_t2_od | m_up_f3_t2_od | m_up_f3_t2_sd): 2;
224      (t2) & (m_up_f1_t3 | m_up_f2_t3 | m_up_f3_t3_1 | m_up_f3_t3_n ): 3;
225
226      -- ****************  going dn  ************
227      (t2) & (m_dn_f3_t3_1 | m_dn_f3_t3_n | m_dn_f2_t3 | m_dn_f1_t3): 3;
228      (t2) & (m_dn_f3_t2_sd | m_dn_f3_t2_od | m_dn_f2_t2_1 | m_dn_f2_t2_n |
   m_dn_f2_t2_od | m_dn_f1_t2_od | m_dn_f1_t2_sd): 2;
229      (t2) & (m_dn_f3_t1 | m_dn_f2_t1 | m_dn_f1_t1_1 | m_dn_f1_t1_n ): 1;
230
231      1: destFl;
232     esac;
233
234    init(destDir) := up;
235    next(destDir) := case
236      -- *** override on 2nd floor ***
237      (t6 | t7) & (m_overrideUp | m_overrideUp_dir) : up;
238      (t8 | t9) & (m_overrideDown | m_overrideDown_dir) : dn;
239
240      -- ********* no pending requests ************
241      (t2) & !(m_f1_pending | m_f2_pending_up | m_f2_pending_dn |
   m_f3_pending): destDir;
242
243      -- ****************  going up ***************
244      -- defaults to up for 1st floor
245      (t2) & (m_up_f1_t1_1 | m_up_f1_t1_n | m_up_f2_t1 | m_up_f3_t1): up;
246      (t2) & (m_up_f1_t2_sd | m_up_f2_t2_1 | m_up_f2_t2_n): up;

247      (t2) & (m_up_f1_t2_od | m_up_f2_t2_od | m_up_f3_t2_sd |
```

```
    m_up_f3_t2_od): dn;
248     -- defaults to dn for 3rd floor
249     (t2) & (m_up_f1_t3 | m_up_f2_t3 | m_up_f3_t3_1 | m_up_f3_t3_n ): dn;
250
251     -- ****************  going dn  ************
252     (t2) & (m_dn_f3_t3_1 | m_dn_f3_t3_n | m_dn_f2_t3 | m_dn_f1_t3): dn;
253     (t2) & (m_dn_f3_t2_sd | m_dn_f2_t2_1 | m_dn_f2_t2_n): dn;
254     (t2) & (m_dn_f3_t2_od | m_dn_f2_t2_od | m_dn_f1_t2_sd |
    m_dn_f1_t2_od): up;
255     (t2) & (m_dn_f3_t1 | m_dn_f2_t1 | m_dn_f1_t1_1 | m_dn_f1_t1_n ): up;
256
257     1: destDir;
258    esac;
259
260  init(prevDest) := 0;
261  next(prevDest) := case
262     t4: destFl;
263     1: prevDest;
264    esac;
265
266  init(prevDir) := up;
267  next(prevDir) := case
268     t4: dir;
269     1: prevDir;
270    esac;
271
272  init(reopen) := 2;
273  next(reopen) := case
274     t3a : reopen - 1;
275     t3 | t5 : 2;
276     1: reopen;
277    esac;
278
279 -- *********************** End of Elevator *********************
```

```
  1 -- **************************************************************
  2 -- *                       MODULE Door                          *
  3 -- **************************************************************
  4 MODULE Door(inp_OPEN, inp_KEEP_OPEN, inp_CLOSE)
  5
  6 -- ****************** VARIABLES SECTION ************************
  7 VAR
  8  -- *** STATE VARIABLES SUBSECTION ***
  9  st_Door: {Close, Open};
 10
 11  -- *** CLASS ATTRIBUTES SUBSECTION ***
 12  retry : 0..3;
 13
 14  -- *** GENERATED EVENTS SUBSECTION ***
 15  ev_SET : SET;
 16
 17  -- *** COMPONENT OBJECTS SUBSECTION ***
 18  DoorTimer: Timer(ev_SET, ev_UNSET);
 19
 20 -- **************** SYMBOL DEFINITION SECTION *****************
 21 DEFINE
 22  -- *** ACTIVE STATE MACROS SUBSECTION ***
 23  in_Door := 1;
 24  in_Close := in_Door & st_Door = Close;
 25  in_Open := in_Door & st_Door = Open;
 26
 27  -- *** TRANSITION MACROS SUBSECTION ***
```

```
28   t1 := in_Close & inp_OPEN > 0;
29   t2 := in_Open & DoorTimer.ev_TM = 1 & !(inp_OPEN > 0);
30   t3 := in_Open & inp_CLOSE & !(inp_OPEN > 0);
31   t4 := in_Open & inp_KEEP_OPEN & retry > 0 & !(t2 | t3);
32
33   -- *** GENERATED EVENTS MACROS SUBSECTION ***
34   ev_UNSET := t3;
35   ev_CLOSED := t2 | t3;
36
37   -- ***************** ASSIGNMENT SECTION ***********************
38  ASSIGN
39   -- *** STATE VARIABLES SUBSECTION ***
40   init(st_Door) := Close;
41   next(st_Door) := case
42     t1 | t4 : Open;
43     t2 | t3 : Close;
44     1: st_Door;
45    esac;
46
47   -- *** GENERATED EVENTS SUBSECTION ***
48   ev_SET.raised := t1 | t4;
49   ev_SET.n := 5;
50
51   -- *** CLASS ATTRIBUTES SUBSECTION ***
52   init(retry) := 0;
53   next(retry) := case
54     t1 : 3;
55     t2 | t3 : 0;
56     t4 : retry - 1;
57     1 : retry;
58    esac;
59
60  -- ****************** End of Door *********************
```

```
 1  -- ****************************************************************
 2  -- *                     MODULE Timer                            *
 3  -- ****************************************************************
 4  MODULE Timer(inp_SET, inp_UNSET)
 5
 6  -- ****************** VARIABLES SECTION ************************
 7  VAR
 8   -- *** STATE VARIABLES SUBSECTION ***
 9   st_Timer: {Idle, Count_Down};
10
11   -- *** CLASS ATTRIBUTES SUBSECTION ***
12   counter: 0..5;
13
14  -- ***************** SYMBOL DEFINITION SECTION *****************
15  DEFINE
16   -- *** ACTIVE STATE MACROS SUBSECTION ***
17   in_Timer := 1;
18   in_Idle := in_Timer & st_Timer = Idle;
19   in_Count_Down := in_Timer & st_Timer = Count_Down;
20
21   -- *** TRANSITION MACROS SUBSECTION ***
22   t1 := in_Idle & inp_SET.raised;
23   t2 := in_Count_Down & counter > 0;
24   t3 := in_Count_Down & counter = 0;
25   t4 := in_Count_Down & inp_UNSET;
26   t5 := in_Count_Down & inp_SET.raised;
27
28   -- *** GENERATED EVENTS MACROS SUBSECTION ***
```

```
29   ev_TM := t3;
30
31  -- ***************** ASSIGNMENT SECTION ***********************
32  ASSIGN
33   -- *** STATE VARIABLES SUBSECTION ***
34   init(st_Timer) := Idle;
35   next(st_Timer) := case
36     t1 | t5 : Count_Down;
37     t3 | t4 : Idle;
38     1: st_Timer;
39    esac;
40
41   -- *** CLASS ATTRIBUTES SUBSECTION ***
42   init(counter) := 0;
43   next(counter) := case
44     t1 | t5 : inp_SET.n;
45     t2 : counter - 1;
46     t4 : 0;
47     1: counter;
48    esac;
49  -- ***************** End of Timer *************************
```

```
 1  -- **************************************************************
 2  -- *                   MODULE RequestBn                         *
 3  -- **************************************************************
 4  MODULE RequestBn(inp_BN_PRESSED, inp_LIGHT, inp_UNLIGHT)
 5
 6  -- ***************** VARIABLES SECTION ************************
 7  VAR
 8   -- *** STATE VARIABLES SUBSECTION ***
 9   st_Button: { Idle };
10   st_Light: { UnLit, Lit };
11
12   -- *** CLASS ATTRIBUTES SUBSECTION ***
13   pending : boolean;
14
15  -- **************** SYMBOL DEFINITION SECTION *****************
16  DEFINE
17   -- *** ACTIVE STATE MACROS SUBSECTION ***
18   in_Button := 1;
19   in_Light := 1;
20
21   in_Idle := in_Button & st_Button = Idle;
22   in_UnLit := in_Light & st_Light = UnLit;
23   in_Lit := in_Light & st_Light = Lit;
24
25   -- *** TRANSITION MACROS SUBSECTION ***
26   t1 := in_Idle & inp_BN_PRESSED;
27   t2 := in_UnLit & inp_LIGHT;
28   t3 := in_Lit & inp_UNLIGHT;
29
30   -- *** GENERATED EVENTS MACROS SUBSECTION ***
31   ev_REQ := t1;
32
33  -- ***************** ASSIGNMENT SECTION ***********************
34  ASSIGN
35   -- *** STATE VARIABLES SUBSECTION ***
36   init(st_Button) := Idle;
37   next(st_Button) := case
38     t1 : Idle;
39     1: st_Button;
40    esac;
```

```
41
42   init(st_Light) := UnLit;
43   next(st_Light) := case
44     t2 : Lit;
45     t3 : UnLit;
46     1: st_Light;
47    esac;
48
49   -- *** CLASS ATTRIBUTES SUBSECTION ***
50   init(pending) := 0;
51   next(pending) := case
52     t2 : 1;
53     t3 : 0;
54     1: pending;
55    esac;
56  -- ********************* end of RequestBn *********************
```

```
 1  -- ****************************************************************
 2  -- *                     MODULE Button                             *
 3  -- ****************************************************************
 4  MODULE Button(inp_BN_PRESSED)
 5
 6  -- ***************** VARIABLES SECTION ***********************
 7  VAR
 8   -- *** STATE VARIABLES SUBSECTION ***
 9   st_Button: { Idle };
10
11  -- **************** SYMBOL DEFINITION SECTION *****************
12  DEFINE
13   -- *** ACTIVE STATE MACROS SUBSECTION ***
14   in_Button := 1;
15   in_Idle := in_Button & st_Button = Idle;
16
17   -- *** TRANSITION MACROS SUBSECTION ***
18   t1 := in_Idle & inp_BN_PRESSED;
19
20   -- *** GENERATED EVENTS MACROS SUBSECTION ***
21   ev_REQ := t1;
22
23  -- ***************** ASSIGNMENT SECTION **********************
24  ASSIGN
25   -- *** STATE VARIABLES SUBSECTION ***
26   init(st_Button) := Idle;
27   next(st_Button) := case
28     t1 : Idle;
29     1: st_Button;
30    esac;
31  -- ********************* end of Button **********************
```

```
 1  -- ****************************************************************
 2  -- *                     MODULE External_Event                     *
 3  -- ****************************************************************
 4  MODULE External_Event()
 5
 6  -- ***************** VARIABLES SECTION ***********************
 7  VAR
 8   -- *** ENVIRONMENT VARIABLE SUBSECTION ***
 9   env_AT_FLOOR : boolean;
10   env_BP_OpenBn : boolean;
11   env_BP_CloseBn: boolean;
12   env_BP_FloorReqBn  : array 1..3 of boolean;
```

```
13   env_BP_ElevReqBnUp : array 1..2 of boolean;
14   env_BP_ElevReqBnDn : array 2..3 of boolean;
15
16  -- ******************** FAIRNESS SECTION **********************
17  FAIRNESS
18    env_AT_FLOOR
19
20  -- ******************* end of External_Event ******************
```

```
 1  -- **************************************************************
 2  -- *                       MODULE SET                            *
 3  -- **************************************************************
 4  MODULE SET()
 5
 6  -- ***************** VARIABLES SECTION ************************
 7  VAR
 8    -- *** CLASS ATTRIBUTES SUBSECTION ***
 9    raised : boolean;
10    n : {5};
11
12  -- *********************** End of SET *************************
```

Appendix D. Additional Elevator Features

Additional Elevator Features:

- Independent service: dedicated service to have continuous service for a specific floor; ignores hall calls; key switch in car (or computer input); door only closes with constant pressure on car call and door close button; If held till door is closed, goes to the floor pressed; If not, reopens door.
- Home landing: If no requests, generate one to home floor (without door opening)
- Car calls take priority over hall calls. This is because the elevator has limited space. This policy brings the passengers to their destination before picking up new passengers.
- A new requirement is to add an indicator (light) in the elevator to warn the passengers if the maximum load (weight) of the elevator is exceeded.
- Repressing the floor button (inside the elevator) that corresponds to the current floor should reopen the door.
- The door closure timer has three different timers for car call (floor buttons), hall call (hall buttons) and reopen (door-open button).

Additional Safety Features:

- The system property where the elevator never moves with its doors open can be satisfied by electrical circuit design. This is called "safety circuit".
- Relevelling: ensure the elevator is at the same floor level; done only with doors open, if in door zone.
- Start protection: If elevator does not leave floor soon enough, abandon moving to the destination.
- Running timer: If motor runs too long, abandon moving to destination.
- Door detector: If obstruction is detected while door is closing, re-open the door.

References

[Berr99] Berry, D. M., “Formal Methods: The Very Idea, Some Thoughts on Why They Work When They Work”, *Electronic Notes in Theoretical Computer Science*, 25, 1999

[BeGo92] Berry, G., Gonthier, G., “The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation,” *Science of Computer Programming*, Vol. 19, No. 2, pp. 87-152, Nov. 1992

[Beiz95] Beizer, B., *Black-Box Testing*: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, 1995

[Boeh81] Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981

[Bose99] Bose, P., “Automated Translation of UML Models of Architectures for Verification and Simulation,“ *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*, IEEE, 1999.

[CAB+98] Chan, W., Anderson, R., Beame, P., Burns, S.;,Modugno, F., Notkin, D., and Reese, J., “Model Checking Large Software Specifications.” *IEEE Transactions on Software Engineering*, Vol. 24, No. 7, July 1998.

[CES86] Clarke, E., Emerson, E., Sisla, A., “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications”, *ACM Transactions on Programming Languages and Systems,* 8(2):244-263, April 1986

[CGP99] Clarke E. M., Jr., Grumberg O., Peled D., *Model Checking*, MIT Press, 1999 pp.30-33

[Day93] Day, N., *A Model Checker for Statecharts*, Technical Report 93-35, Department of Computer Science, UBC, October 1993

[DAC98] Dwyer, M., Avrunin, G., Corbett, J., “Property Specification Patterns for Finite-State Verification”, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, March 1998, pp. 7-15.

[Doug98] Douglass, B. P., *Real-time UML: developing efficient objects for embedded systems*, Addison Wesley, 1998

[GlWe98] Gluch, D., Weinstock, C., *Model-Based Verification: A Technology for Dependable System Upgrade.* CMU/SEI-98-TR-009, Software Engineering Institute, Carnegie Mellon University, September 1998.

[GuNa98] Guaspari, D., Naydich, D., "Flight-Guidance Systems: UML Design, PVS Analysis," Odyssey Research Associates technical report, TM-98-0035, November 30, 1998

[Hare87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems" *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987

[Holz97] Holzmann, G. J., "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997

[JaRi00] Jackson, D., Rinard, M., "The Future of Software Analysis", *The Future of Software Engineering* (editor Anthony Finkelstein), ACM Press June 2000.

[KaHe00] Kammüller F., Helke, S., "Mechanical Analysis of UML State Machines and Class Diagrams." *Workshop on Precise Semantics for the UML, ECOOP 2000*, Cannes, June 2000.

[Kurs96] Kurshan, R. P., "COSPAN", *Proc. 8th Int'l Conf. Computer Aided Verification (CAV96),* New Brunswick, NJ, 1996

[LHHR94] Leveson, N., Heimdahl, M., Hildreth, H., Reese, J., "Requirements Specification for Process-Control Systems" *IEEE Transactions on Software Engineering*, Vol. 20, No. 9, pp. 684-707, September 1994.

[LiPa99] Lilius, J., Paltor, I., *vUML: A Tool for Verifying UML Models*. TUCS Technical Report No. 272, May 1999. ISBN 952-12-0445-1.

[Lisk88] Liskov, B., *Data Abstraction and Hierarchy*. SIGPLAN Notices. May 1988.

[McCh01] McUmber, W., Cheng, B., "A General Framework for Formalizing UML with Formal Languages," *Proc. of IEEE International Conference on Software Engineering* (ICSE01), May 2001, Toronto, Canada

[McGr93] McGregor, J. D., Dyer, D. M., "A Note on Inheritance and State Machines," *Software Engineering Notes*, Vol. 18, No. 4, pp. 61-69. 1993

[McMi93] McMillan, K. L., *Symbolic Model Checking*. Kluwer, 1993

[McMi99] McMillan, K. L., *The SMV Language*. Cadence Berkeley Labs, March, 1999

[Muth00] Muthiayen, D. *Real-Time Reactive System Development - A Formal Approach Based on UML and PVS*. Ph.D. thesis, Department of Computer Science, Concordia University, Montreal, Canada. January, 2000

[ORS92] Owre, S., Rushby, J., Shankar, N. "PVS: A Prototype Verification System," *In Lecture Notes in Artificial Intelligence: Proc. CADE 11*, Vol. 607, pp. 748-752, Saratoga, NY, 1992.

[Pele94] Peled, D., "Combining Partial Order Reductions with On-The-Fly Model-Checking," *Proc. Sixth Int'l Conf. Computer Aided Verification (CAV94),* pp. 377-390, Stanford, CA, 1994

[RBP+91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-oriented Modeling and Design*. Prentice-Hall International, 1991.

Object-oriented Modeling and Design. Prentice-Hall International, 1991.

[SrAt96] Sreemani, T., Atlee, J., "Feasibility of Model Checking Software Requirements: A Case Study." *COMPASS '96, Proc. 11th Annual Conf. Computer Assurance*, pp. 77-88, Gaitherburg, Md., IEEE, June 1996

[UML99] Booch, G., Jacobson, I., Rumbaugh, J., editor. *OMG Unified Modeling Language Specifications Version 1.3*. Rational Corporation, Santa Clara, June 1999