





This is to certify that the

thesis entitled

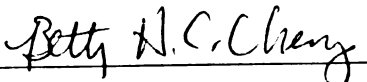
OBJECT-ORIENTED DESIGN OF EMBEDDED SYSTEMS WITH  
TRANSLATION TO VHDL

presented by

Gretel Van Lente Coombs

has been accepted towards fulfillment  
of the requirements for

Master's degree in Computer Science

  
Major professor

Date 5/13/98

**LIBRARY**  
**Michigan State**  
**University**

**PLACE IN RETURN BOX**  
to remove this checkout from your record.  
**TO AVOID FINES** return on or before date due.

DATE DUE	DATE DUE	DATE DUE
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

OBJECT-ORIENTED DESIGN OF EMBEDDED SYSTEMS  
WITH TRANSLATION TO VHDL

By

*Gretel Van Lente Coombs*

A THESIS

Submitted to  
Michigan State University  
in partial fulfillment of the requirements  
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1998

ABSTRACT

OBJECT-ORIENTED DESIGN OF EMBEDDED SYSTEMS WITH  
TRANSLATION TO VHDL

By

*Gretel Van Lente Coombs*

As embedded systems become more complex, there is an increasing demand for development techniques and tools to manage the complexity. Experiences from the software engineering industry have shown that diagram-based modeling techniques are useful in providing a means to represent abstract concepts of a system that can eventually be refined, in a stepwise fashion, to obtain more detailed design descriptions. Currently, a hardware description language is the most abstract representation commonly used to model embedded systems. The potential disadvantage of using a hardware description language at the beginning of the design process is that implementation decisions can be introduced before requirements are understood clearly.

The objective of this thesis is twofold. First, the thesis introduces an object-oriented modeling framework for requirements analysis and design of embedded systems, including a stepwise refinement process. The second objective is to provide a mapping between the graphical, object-oriented models and VHDL. Given the wide-range of tool support, VHDL can then be used to check the consistency of the diagrams and can be used to simulate the behavior of the system to ensure a clear understanding of requirements prior to introducing implementation details.

© Copyright 1998 by Gretel Van Lente Coombs  
All Rights Reserved

To My Sweet Boys, Derrick Van and Seth Daniel.

## ACKNOWLEDGMENTS

I would like to acknowledge the love and support of a heavenly Father, who has sustained me through these years of challenges.

Much heartfelt thanks are due to Dr. Betty H.C. Cheng for all of her wisdom, patience, and direction these last few years. I am very thankful for her advisory style of motivation by encouragement. My hope is that this thesis lives up to the high expectations that she sets for herself and her students.

Special thanks are extended to my committee members, Dr. Diane Rover and Dr. Anthony S. Wojcik. Their diverse backgrounds helped strengthen this thesis' contributions.

I'd also like to acknowledge the assistance I received from Mats Heimdahl and Kurt Stirewalt on various aspects of this thesis.

I never could have accomplished this degree without the continual and constant support of my loving parents, Dale and Ann Van Lente. I want to thank them for their belief in me and their steady encouragement of my goals and dreams.



# TABLE OF CONTENTS

<b>LIST OF FIGURES</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>4</b>
2.1 VHDL Overview . . . . .	4
2.2 OMT Overview . . . . .	11
2.2.1 Object Model . . . . .	12
2.2.2 Dynamic Model . . . . .	15
2.2.3 Functional Model . . . . .	19
<b>3 OMT Analysis and Modeling of Embedded Systems.</b>	<b>21</b>
3.1 Automobile Door System . . . . .	22
3.2 High-level Object Model . . . . .	23
3.3 System-level Dynamic Model . . . . .	28
3.4 System-level Functional Model . . . . .	29
3.5 Further Refinement of the Models. . . . .	33
3.5.1 Consistency between Models . . . . .	34
3.5.2 Associations . . . . .	35
3.5.3 Refinement of Dynamic and Functional Models . . . . .	35
<b>4 OMT to VHDL Translation Process</b>	<b>39</b>
4.1 Flattening the Dynamic Model . . . . .	40
4.2 Rules for deriving VHDL from OMT . . . . .	41
4.3 VHDL Translation Process . . . . .	45
<b>5 VHDL Analysis</b>	<b>52</b>
5.1 VHDL Simulation . . . . .	53
5.2 Other Analysis Tools . . . . .	55
5.3 Analysis of Automobile Door System . . . . .	55
5.3.1 Static Analysis . . . . .	56
5.3.2 Dynamic Analysis . . . . .	56
<b>6 Related Work</b>	<b>59</b>
6.1 The Formalization of OMT models . . . . .	59
6.2 Object-oriented design of VHDL modules . . . . .	60
6.3 Generation of VHDL from Graphical Models . . . . .	60

<b>7 Conclusions and Future Work</b>	<b>62</b>
<b>APPENDICES</b>	<b>65</b>
<b>A Automotive Door System: OMT Models</b>	<b>65</b>
<b>B Automotive Door System: VHDL Entity/Architectures</b>	<b>72</b>

## LIST OF FIGURES

2.1 Entity template. . . . .	5
2.2 VHDL architecture. . . . .	5
2.3 A simple 2 input multiplexor. . . . .	5
2.4 VHDL mux entity. . . . .	6
2.5 VHDL behavioral architecture for mux. . . . .	6
2.6 Components of a multiplexor. . . . .	8
2.7 Mux's structural architecture. . . . .	8
2.8 Example of a VHDL package. . . . .	9
2.9 Example of a VHDL procedure. . . . .	9
2.10 Classes are shown as boxes. . . . .	12
2.11 Associations are lines between classes. . . . .	13
2.12 A triangle uses generalization into superclasses and inheritance by subclasses. . . . .	13
2.13 Diamonds are used to show aggregation. . . . .	14
2.14 Line annotation for association multiplicities. . . . .	14
2.15 The notation for the state diagrams within the dynamic model. . . . .	15
2.16 The notation for showing hierarchical states within a dynamic model. . . . .	17
2.17 The notation for showing concurrency within a dynamic model. . . . .	18
2.18 Notation used in the OMT functional model. . . . .	19
3.1 The door system's high-level object model. . . . .	24
3.2 Object Model showing multiple controllers. . . . .	25
3.3 Object Model showing actuator refinement. . . . .	26
3.4 Object model showing an object that is both sensor and actuator. . . . .	26
3.5 General System-level dynamic model. . . . .	28
3.6 System-level dynamic model for door system. . . . .	30
3.7 System-level functional model of an embedded system. . . . .	31
3.8 System-level functional model of an automotive door system. . . . .	32
3.9 Refined functional model of an automotive door system. . . . .	33
3.10 Refined object model for door system . . . . .	36
3.11 Refinement of the <i>Door Locking Mechanism Control</i> state. . . . .	38
4.1 A View of the OMT to VHDL translation process. . . . .	40
4.2 Hierarchical state chart H and state charts for its super states. . . . .	42
4.3 Flattened state chart F, equivalent to state chart H. . . . .	42
4.4 Main entity and architecture templates for Automotive Door System. . . . .	49
4.5 Ports added to Door_System entity from the Functional Model and the number of instances from the Object model. . . . .	49

4.6	Input port type information added. . . . .	49
4.7	Component information added to the declarative section of the structural architecture. . . . .	50
4.8	Behavior architecture for a controller. . . . .	51
5.1	Simplified diagram of VHDL simulation cycle [10]. . . . .	54
A.1	Object model for door system . . . . .	66
A.2	System-level Dynamic Model . . . . .	67
A.3	Dynamic Model for Window Rolling Down State . . . . .	68
A.4	Dynamic Model for Window Rolling Up State . . . . .	68
A.5	Dynamic Model for Door Lock State . . . . .	69
A.6	System-Level Functional Model . . . . .	70
A.7	Functional Model refined by controller objects . . . . .	71
B.1	A view of the entities and architectures in the Automotive Door System. . . . .	73

# Chapter 1

## Introduction

As embedded systems become more complex, there is an increasing demand for development techniques and tools to manage the complexity. Experiences from the software engineering industry have shown that diagram-based modeling techniques are useful in providing a means to represent abstract concepts of a system that can eventually be refined, in a stepwise fashion, to obtain more detailed design descriptions. In contrast, embedded systems are commonly developed with design languages, such as VHDL [24], VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. The potential disadvantage of using VHDL at the beginning of the design process is that it forces developers to make numerous implementation decisions before a clear understanding of requirements has been obtained. Therefore it increases the potential to include errors or inconsistencies in the requirements that will then be propagated to the implementation and fabrication stages, potentially 10 to 100 times more costly than errors introduced during implementation [16, 17].

Object-oriented design decomposes a system into abstractions based on real world objects. This decomposition and abstraction process facilitates the creation of understandable, maintainable designs. The Object Modeling Technique (OMT) [23] is an

object-oriented design method used widely in industry. OMT displays a system from three complementary perspectives: structural, behavioral, and services and data flow captured by the object, dynamic, and functional models, respectively. These three graphical views enable a visual reduction of the system's inherent complexity, thus promoting design understanding and validation.

The objective of this thesis is twofold. First, it proposes an object-oriented modeling framework for requirements analysis and design of embedded systems. This framework enables developers to start with high-level, abstract (graphical) representations of an embedded system and, through a stepwise process, gradually add more details to the graphical models. Wang formalized the OMT notation into commonly used formal specification languages (LOTOS and ACT ONE [2]) that enabled rigorous analysis of the OMT diagrams as well as behavior simulation [4, 27]. This formalization provided a bridge between easy to use, graphical design models and formal methods for software development. Using this work as a foundation, the second objective addresses the systematic development of embedded systems by providing a mapping between the graphical, object-oriented models and VHDL. That is, once a developer creates the graphical models of a system, VHDL can be generated, using automated techniques. Given the wide range of tool support, VHDL can then be used to check the consistency of the diagrams and can be used to simulate the behavior of the system to ensure a clear understanding of requirements.

**Thesis Statement:** *Using an object-oriented analysis and design technique can facilitate a stepwise development process for embedded systems, where the object-oriented models can be used to generate VHDL specifications.*

The following is a summary of the contributions of this research project.

- A process for using an object-oriented modeling technique to describe embedded systems.

- A process for developing VHDL specifications from object-oriented graphical models.
- A refinement process for evolving high-level requirements into design models according to the analysis of VHDL specifications.

The remainder of this thesis is organized as follows. Overviews of VHDL and OMT are given in Chapters 2.1 and 2.2 respectively. A stepwise process for creating OMT models for embedded systems is described in Chapter 3. An automobile door control system example is used to illustrate this process. The OMT to VHDL translation is presented in Chapter 4. Chapter 5 describes analysis that can be applied to the OMT diagrams via the corresponding VHDL specifications. Chapter 6 describes related work. Conclusions and future investigations are briefly discussed in Chapter 7. The complete OMT models and generated VHDL for the automotive doors system and the washing machine system are included in the appendices.

# Chapter 2

## Background

This chapter overviews VHDL and the Object Modeling Technique (OMT).

### 2.1 VHDL Overview

VHDL is a hardware description language used worldwide. It is a design language that was developed to assist in the development, documentation, and exchange of hardware designs. This chapter covers the main components of a VHDL design, including only the elements used in the rest of the paper.

A VHDL design entity consists of two basic parts, an entity and an architecture. The entity defines the porting or the interface of the object to its environment. All communication must be performed through the ports of an entity. Figure 2.1 is a template of the entity form.

In order to simulate an **entity**, a corresponding **architecture** is needed. An **architecture** is considered to be the body of the entity that describes how the design operates. The **architecture** describes the function or behavior performed by the entity and assumes access to all the ports defined in the entity. An entity can have more than one architecture, each realizing the behavior of the system in different



---

```
entity <entity_identifier> is
  port      -- Input and Output ports defined in this section
    (<identifier>:[mode]<type> -- where identifier is name of port
     -- mode is In, Out, or InOut
     -- type is the type of data using the port
    );
end<identifier_entity> ;
```

Figure 2.1: Entity template.

---

ways. These architecture modules have the form shown in Figure 2.2.

---

```
architecture architecture_type of entity_identifier is
  -- any declarations, such as signals or components
begin
  --concurrent statements including block statements, procedures, processes.
  end [Identifier];
```

Figure 2.2: VHDL architecture.

---

For example, a multiplexor, like the one shown in Figure 2.3, can have both an behavioral architecture and a structural architecture. A behavioral architecture shows the behavior of the entity, with no indication of how the design is implemented. One approach is to use a **process** of sequential statements. A structural description shows the various components and how their ports and signals are connected together.

---

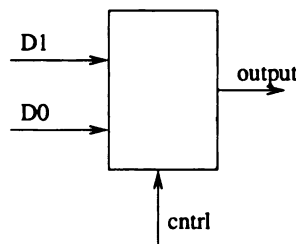


Figure 2.3: A simple 2 input multiplexor.

---

Figure 2.4 is the **entity** statement that defines the multiplexor in Figure 2.3. The

**port** clause is called the interface declaration. D0, D1, and cntrl are all inputs to the multiplexor and they are all of type *bit*. The signal output is defined as a bit wide signal that is associated with an **out** port.

---

```

entity mux is
  port
    ( D0, D1, cntrl : in BIT;
      output : out BIT
    );
end mux;

```

Figure 2.4: VHDL mux entity.

---

An **architecture** that consists of a behavioral description of the functionality of this multiplexor is shown in Figure 2.5. The **process** block allows for sequential statements written in the style of a programming language to be expressed instead of concurrent statements. The behavior of the multiplexor is specified logically to indicate that if 'cntrl' is a 'one', then the output should be signal 'D1', else 'D0' is placed on the outgoing port.

---

```

architecture behavior of mux is
begin
  -- operational description of behavior
  process(D0, D1, cntrl) -- process and sensitivity list
  begin
    -- sequential statements
    IF cntrl = 1 THEN output <= D1;
    ELSE output <= D0;
    end process;
  end behavior;

```

Figure 2.5: VHDL behavioral architecture for mux.

---

The signals listed in parentheses within the **process** statement constitute the *sensitivity list*. If any of these signals change, then the process begins executing, reevaluating its output values. The alternate method to initiate a concurrent pro-

cess is to include a **wait** statement at the end of the **process** block. The possible statements for starting a **process** or other modules include:

- **wait until** (condition);
- **wait for** period\_of\_time;
- **wait on** signal\_a, signal\_b, ...;
- or a combination, such as:

**wait on** control\_a **until**(CLK = '1') **for** 5ms;

A structural description of the multiplexor illustrates the ability to hierarchically combine lower-level entities into more complex designs. Figure 2.6 shows the layout of the multiplexor (mux) comprising other smaller components. The structural architecture found in Figure 2.7 consists of various 'and', 'or', and 'not' gates. These internal gates have their own previously defined entity/architecture pairs. Each of these already-defined entities is included as a **component** in the structural **architecture**. Any internal signals needed are also included in the declarative section of the architecture using a **signal** statement. In the body of the architecture, the actual instances of the components are declared. These instances are named followed by the component type. Then the ports of the component are mapped to the signals of the entity in which it is contained.

For example, in Figure 2.7 an 'and\_gate' is declared as a **component** and then two 'and\_gate' instances are instantiated with their ports mapped to various signals of the mux. These signals include the multiplexor's ports as well as internal signals.

A VHDL **package**, as shown in Figure 2.8, is a collection of declarations. These declarations are accessible by any entity or architecture. As can be seen in Figure 2.8, user-defined types and globally declared signals can be defined within a **package**. A

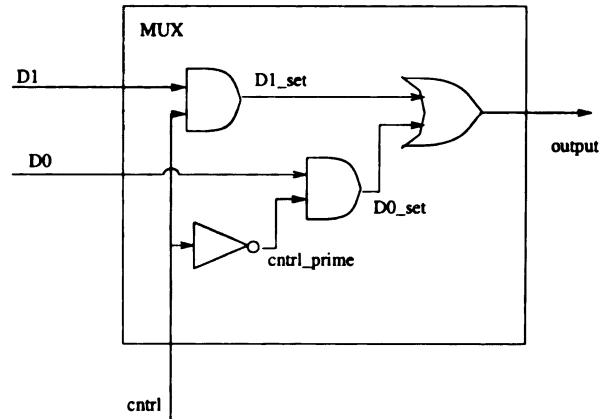


Figure 2.6: Components of a multiplexor.

---

```

architecture structure of mux is
  component and_gate
    port
      (in1, in2 : in BIT;
       output : out BIT);
  end component;
  component or_gate
    port
      (in1, in2 : in BIT;
       output : out BIT);
  end component;
  component not_gate
    port
      (input : in BIT;
       output : out BIT);
  end component;
  sig-
  nal D0_set, D1_set, cntrl_prime : BIT; -- 3 internal signals declared.
begin
  and_1 : and_gate port map (in1=>D0, in2=>cntrl_prime,output=>D0_set);
  and_2 : and_gate port map (in1=>D1, in2=>cntrl, output=>D1_set);
  not_1 : not_gate port map (input=>cntrl, output =>cntrl_prime);
  or_1 : or_gate port map (in1=>D0_set, in2=>D1_set, output=>output);
end structure;

```

Figure 2.7: Mux's structural architecture.

---

package can be referenced by an entity. The phrase ‘`use WORK.package_name.all;`’ preceding a VHDL entity allows all declarations of the package to be seen by that entity and any of its architectures.

---

```

package user_definitions is
  type user_array is array (1 to 12);
  type color is (white, blue, red);
  sig-
  nal ex_signal :Bit:='0'; --makes globally accessible signal
end user_definitions;

use WORK.user_definitions.all; -- Uses the package
entity example is
  (port flag : in color);      -- Access the package
end example;

```

Figure 2.8: Example of a VHDL package.

---

A **procedure** in VHDL is considered a subprogram that may be hierarchically defined inside packages, entities, architectures, processes or other procedures. The example in Figure 2.9 shows the key word **procedure**, followed by a user-defined name and a list of formal parameters. These parameters consists of 4 parts: object class (constant, variable or signal), mode (in, out, or inout), type, and default value.

---

```

procedure calculate
  (constant name1 : in bit_vector;
   variable name2 : in bit := 0;
   variable result : out bit ) is
  -- any needed declarations
begin
  -- any sequential statements
end calculate;

```

Figure 2.9: Example of a VHDL procedure.

---

VHDL supports numerous approaches for representing system design information.

Included here are only the elements of VHDL that will be used in the OMT to VHDL translation process.

## 2.2 OMT Overview

The Object Modeling Technique (OMT)[23] is a modeling technique that facilitates the object-oriented development of a system. The methodology includes steps on how to perform analysis and design, how to develop the models, and how to refine the models as part of the development process.

An object-oriented design approach encourages thinking about the conceptual issues before thinking about the implementation issues. The use of abstraction of objects and the encapsulation of data and operations within an object are key to an object-oriented approach. Information about how an operation is implemented is hidden from the outside world. Only the interfaces to the object are accessible for easy upgrading and changing. An object-oriented approach encourages the reuse of objects in other systems and settings.

OMT modeling develops three orthogonal views of a system, expressed in three different models. These are the *object model*, the *dynamic model*, and the *functional model*. The object model depicts the system based on its static structure and the various real world objects, grouped together into classes. The object model also shows the associations between various classes. This model is important to embedded systems in order to capture the structure of the various components of the system and their relationships. The dynamic model depicts behavioral information and shows the dynamic, changing nature of the system over time. The control aspects of the system are contained in this model, making it the most important model for an embedded control system. The last diagram, the functional model, focuses on the data flow through the system. The data transforms and services of the system are depicted, as well as the environmental actors that input data into the system.

The remainder of this section describes in further detail, each of the OMT models,

including example diagrams.

### 2.2.1 Object Model

The *object model* shows the various classes of objects that make up a system. The static and structural aspects of a system are graphically depicted. The objects of the system, such as the sensors, actuators, and controllers of an embedded system, are grouped into classes of similar objects.

The *object model* in OMT uses a limited set of graphical symbols in specific ways to create a class diagram. A description of the various components and symbols of an object model or class diagram follows.

- A *class* is shown as a box in an object model, as shown in Figure 2.10. This box can be divided into three sections by horizontal lines. The top section contains the name of the class. The middle section has a list of *attributes* associated with this object and the attribute type information. Attributes are data values held by different objects in a class. The last section contains a list of *operations* that are accessible to the outside world that can be performed on this object. These operations can contain any arguments needed by the operation, as well as an optional return data type.

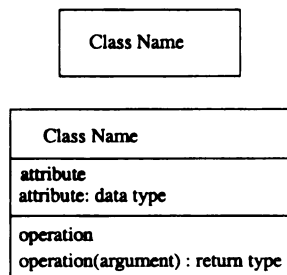


Figure 2.10: Classes are shown as boxes.

---



- An *association* is denoted by a line between two classes. Figure 2.11 shows an association line, labeled with a text description, which is considered the association name.

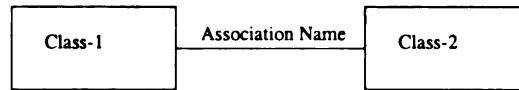


Figure 2.11: Associations are lines between classes.

---

- A *subclass* inherits properties, such as attributes, operations, and associations from its *superclass*. A triangle placed on the association shows this *inheritance* relationship as illustrated in Figure 2.12. A subclass is a more refined version of the superclass constituting an *is-a* relationship. Inheritance and generalizations allow for succinct abstractions, useful both for modeling and for implementation. Inheritance also allows for the reuse of code during implementation.

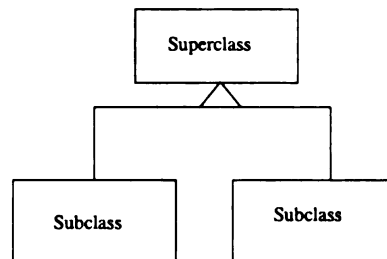


Figure 2.12: A triangle uses generalization into superclasses and inheritance by subclasses.

---

- A diamond denotes an *aggregation*. A composite object is composed of a number of aggregate objects as shown in Figure 2.13. These aggregate objects make up the totality of the composite object.
- Associations can also be annotated by numbers that show the multiplicity of the

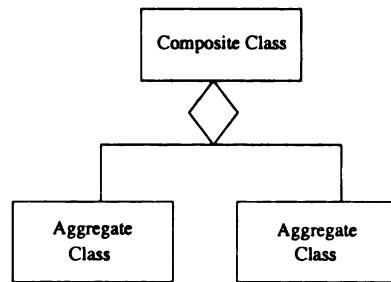


Figure 2.13: Diamonds are used to show aggregation.

---

association. See Figure 2.14 for examples. An undecorated line means exactly one. A filled circle on an endpoint means there are zero or more of these classes. If there is a number or a number with a plus sign then that number illustrates the number of objects of this class.

---

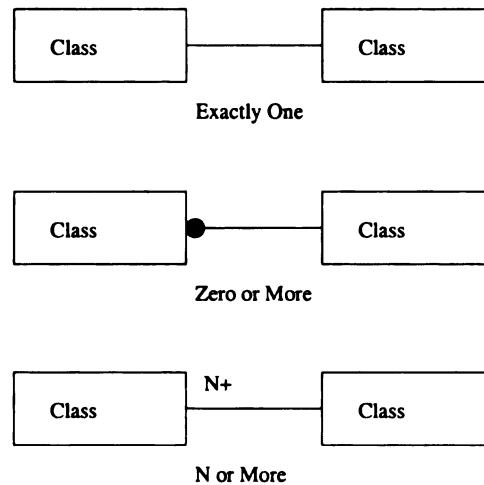


Figure 2.14: Line annotation for association multiplicities.

---

## 2.2.2 Dynamic Model

The *dynamic model* in OMT shows the changes in state that a system or object goes through as time proceeds. The dynamic model is based on Harel's StateCharts [14]. A statechart shows the states of the system or class and the events that cause the transitions to other states. Control information for a system is illustrated clearly in dynamic models. Concurrency in a system is also depicted. A system-level dynamic model is included in OMT development. This is an overall statechart for the whole system. Individual classes also need individual dynamic models. These are necessary for every class that has easily understood state transitions. For example, a button's dynamic model would be relatively trivial, with two states, 'on' or 'off', and two transitions. This class necessarily need to be modeled.

The dynamic model notation is similar to other state diagram notations. The notation used within the states and on the transitions between the states is shown in Figure 2.15. This notation is discussed below, as well as how to represent state refinement and concurrency.

---

### Notation in States and on Transitions

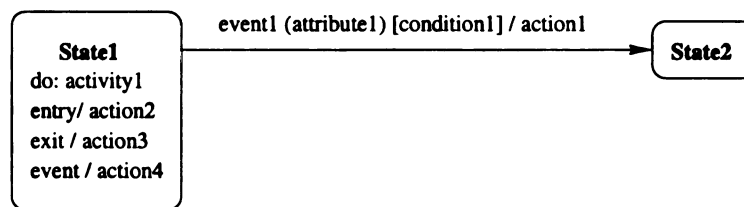


Figure 2.15: The notation for the state diagrams within the dynamic model.

---

- A rounded box represents a *state*. A state is labeled with a state name. A state is considered to be the interval when some attributes of interest are stable and the system is waiting for an event of interest. In OMT, the changing of

attributes that are not of interest do not force a change of state [23]. This property enables the creation of more abstract states that increases clarity.

- *Transitions* are shown as arrows that reflect the movement of a system from one state to the next.
- *Events* cause transitions to other states. The transition arrow is labeled with the event that causes the state transition. An event is something that occurs at a particular moment in time. The time in which an event takes place is instantaneous compared with the granularity of the other things happening in the system.
- *Actions* are considered instantaneous operations. An action is something that can be done at a single moment of time. Actions can be done upon entering a state, when exiting a state, or in response to an event happening while in a state. While a system is transitioning to another state, an action can also be triggered. An action is always preceded by a “/” symbol.
- An *activity* is an operation that takes time to complete. In a dynamic model an activity can be associated with a state. An activity is the operation that is performed while the system is in a state. The syntax for showing the performance of an activity is to place a “do: activity\_name” within the state.
- A filled circle represents the *start state* or starting place for the state diagram.
- A *guard* is a construct placed on a transition. A guard is a condition that must be satisfied or the transition will not be taken to the next state. Conditions or guards evaluate to Boolean values and are placed between square brackets after an event on a transition.

- *Abstractions* in the dynamic model facilitate the reading and understanding of the diagram. Without abstraction, all the states of a system would require inclusion in a flat statechart. This would be potentially very large and difficult to read. The two types of abstractions that are available in an object-oriented design are:

– **Hierarchical states:**

A state may be refined to show substates of greater granularity. The state with more *generalization* is expanded, becoming the superstate that encapsulates other substates. The state generalization example in Figure 2.16 shows a superstate, represented by a large rounded box, and substates, shown as smaller rounded boxes within the superstate. These substates have their own dynamic models that may contain other substates.

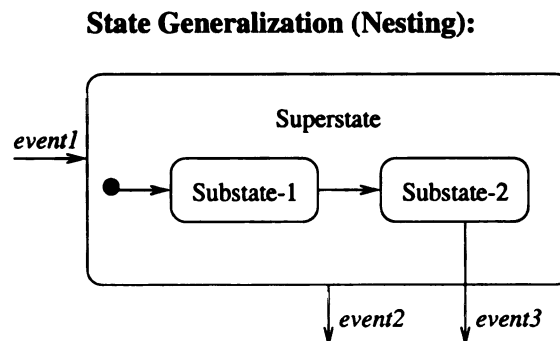


Figure 2.16: The notation for showing hierarchical states within a dynamic model.

– **Concurrent states:**

A dotted line inside a superstate dividing two or more state diagrams is used to show *concurrency*. When the superstate in Figure 2.17 becomes active, because of a transition on event1, then Substate-1 and Substate-3 concurrently become active.

---

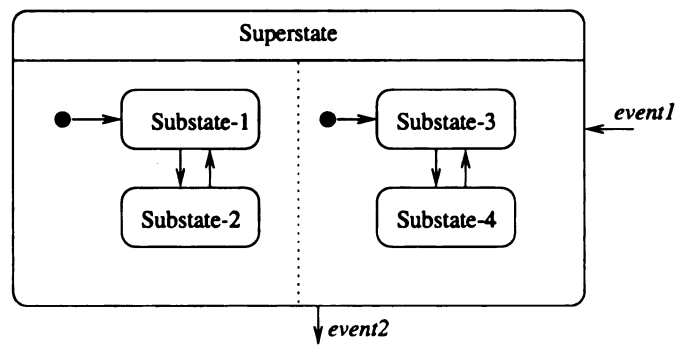
**Concurrent Subdiagrams:**

Figure 2.17: The notation for showing concurrency within a dynamic model.

---

### 2.2.3 Functional Model

Functional models in OMT are data flow diagrams that show the flow of data among various processes that transform the data. These processes can be further decomposed into more detailed data flow graphs. Boxes in the model are considered as actors interacting with the system. In an embedded system, these actors would be various sensors and actuators that are in the environment of the system. Any data stores (e.g. table of initialization values) for the system are also shown on the functional model.

The notations used in the *functional model* are shown in Figure 2.18 and are described below.

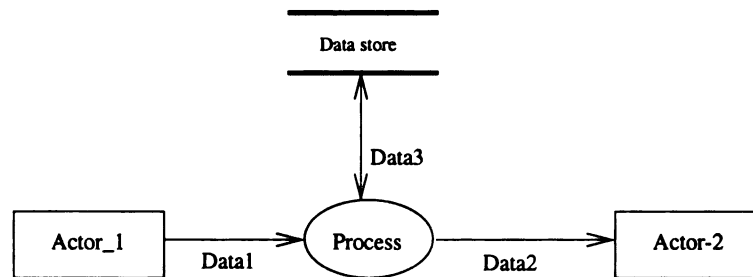


Figure 2.18: Notation used in the OMT functional model.

---

- Ovals represent *processes* that transform input data to form output data. In an embedded system, these transforms are normally quite simple.
- The arrows of the functional model show the flow and direction of data or signals as they move through the various processes of the system. These *dataflows* are labeled with the name of the data.
- The *actors* are shown as rectangles in the functional model. Actors are active agents that drive the flow of the data. Actors actively produce data for the

system, as well as consume data produced by the system.

- The *data stores* of the system are indicated by a pair of parallel horizontal lines. The name of the data store is contained between these lines. Data stores are passive objects that store data for later use.

Functional models are hierarchical in nature. At the most abstract level, level 0, the whole system's functional model is depicted by only one oval representing the entire system's processing. This process can then be refined into subprocesses that show more and more levels of detail. Successive functional models show more details and are numbered 0, 1, 2, etc.

The basics of OMT have been presented in this chapter. The next chapter will specifically show how OMT can be used for embedded systems' modeling. Later in this thesis will be presented an approach for taking the OMT models of embedded systems and translate them into VHDL.



## Chapter 3

# OMT Analysis and Modeling of Embedded Systems.

Humans typically use abstraction to handle complexity. When analyzing a complex system, different views of the system can be abstracted and modeled. With the use of precise notation and an iterative method, abstract models can be systematically refined.

Using OMT, three models are developed for a system: the object model, dynamic model, and functional model. Working from a problem statement, and with feedback from the customer, the requirements and specifications for a new system can be systematically determined. Each of the three models will consist of multiple diagrams, each showing more details at subsequent levels of refinement.

The primary model is the object model that depicts the static aspects of the system, that is, the objects and the relationships between the objects; the notation is similar to entity-relation diagrams. However, because of the nature of embedded systems, the dynamic model is essential for showing the behavior of the system, with state changes showing events and control captured in state diagrams. The functional

models will be relatively simple, because most embedded systems have few data transforms. Data Flow diagrams that depict data flows and data transformations (processes) are used for the functional model. However, the system-level functional model is helpful for visualizing the data and control signals flowing into the processes.

The analysis and modeling processes consists of the following steps:

1. Develop a high-level object model.
2. Develop a system-level dynamic model.
3. Develop a system-level functional model.
4. Verify consistency among the three models.
5. Refine and add details to the object model.
6. Develop a dynamic model for every object, unless it is trivial, and refine.
7. Refine data transformation processes in the functional model.

Details of how these steps can be applied to an application are described in the remainder of this chapter.

### **3.1 Automobile Door System**

An automobile door system is used as an example throughout this thesis. The following is a brief description of the system and its required functionality.

Automobiles can be designed with automatic windows, locks, and a keyless entry fob [28]. This example will design an automobile doors system to handle input from door panels, as well as an optional keyless fob. All doors have window controls, both

up and down. Both front doors have automatic door lock and unlock controls. The driver's door has a window lock and unlock control that disables the other doors' window controls.

The lock button on the fob unit will honk the vehicle's horn once and blink the vehicle's light twice. This notifies the driver that the 'lock' has taken effect. If a door is ajar anywhere in the vehicle, the fob lock is disabled. An 'unlock' by the fob unit unlocks just the driver's door, if the doors are locked. A second 'unlock' from the fob unit will unlock the remaining locks in the vehicle.

The door system should be able to be implemented without the keyless fob and/or without back doors. These different configurations must be handled without redesigning the system.

## 3.2 High-level Object Model

When starting a design process, an object model should be developed first, showing the objects of the system grouped together in classes. This model shows the static structure of the system. The development process begins with the designer looking for the "nouns" within the problem description that represent real world objects. Frequently these objects are pieces of equipment in the system that interact with a controller. These normally can be grouped into two major classes of objects, sensors and actuators. Sensors are any objects that give feedback to the system, such as buttons, knobs, thermometers, etc. Actuators are objects that the system influences, such as indicator lights, motors, and valves. Sometimes there is also an user interface that might be made up of both sensors and actuators, such as an appliance's panel for user input and for indication of cycle selection and sequencing.

In addition to sensor and actuator objects, a controller object is a key element

of the system. This controller object manages the signals that are coming from the environment and makes decisions based on the signals and their orders. Many times the state of the system must be considered before responding to an input. For example, the event of changing the power level on a microwave is handled differently depending on whether the microwave is idle or cooking. The controller object class decides which actuators should be activated and chooses the proper state transitions.

These three major groups of classes, sensors, actuators, and controllers, as well as the possible user interface object should be placed in a high-level object model. The lines between classes in the object model represent associations. An object model for embedded systems has the general form seen in Figure 3.1. The system is a composite object (shown by a diamond on an association) made up of an aggregation of the user-interface, sensor, controller, and actuator classes. The fact that there could be multiple user-interfaces, sensors, and actuators is depicted by a filled circle indicating zero or more of these objects.

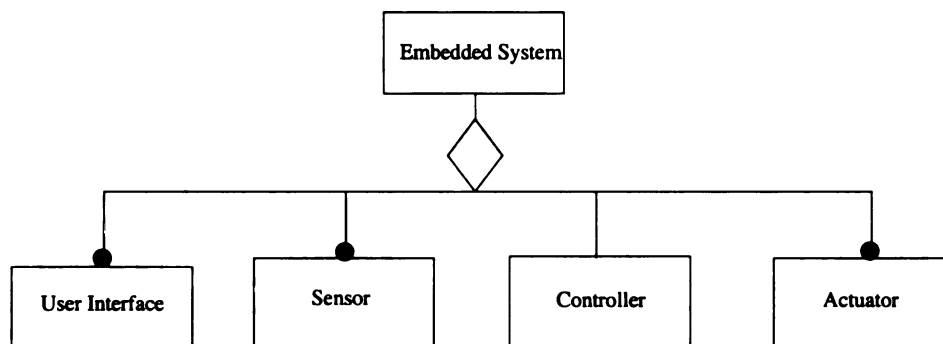


Figure 3.1: The door system's high-level object model.

---

The controller class might also be a composite object, made up of several different controllers that handle different input and output. The amount of coupling or sharing of data and control between these controllers can influence the decision to divide them (if loosely coupled) or to keep them together (if strongly coupled). An example of

multiple controller classes for the automotive door system is seen in Figure 3.2.

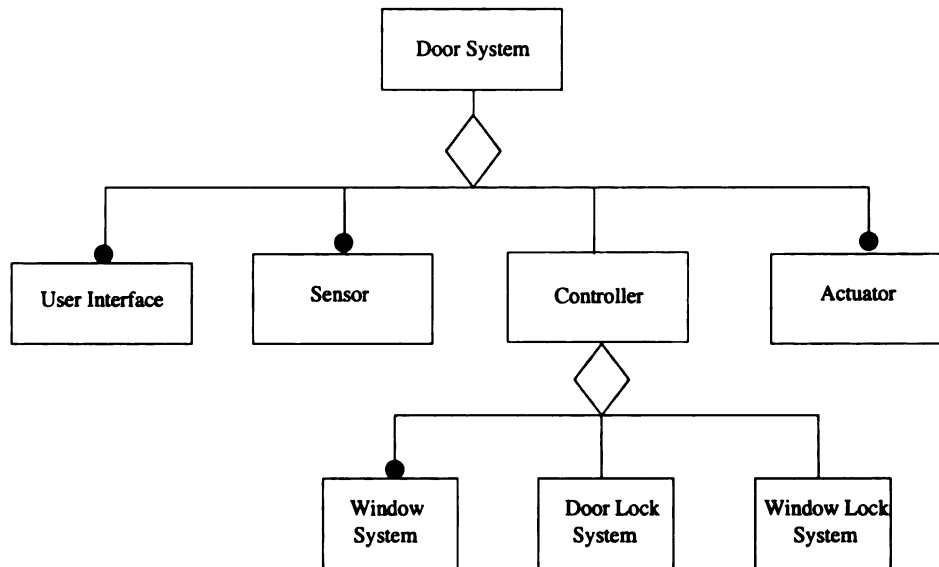


Figure 3.2: Object Model showing multiple controllers.

---

Under the sensors and the actuators, additional classes may be identified. These classes should group together objects that have similar characteristics and properties. Superclasses (indicated by the triangle connector), such as the sensor class, may be refined into other subclasses (that inherit the attributes and operations of the superclass) such as toggles and switches. For the automotive door system in Figure 3.3, actuators are initially identified as the multiple *window motors*, the *door locking mechanism*, the *light flashing actuator*, and the *horn*.

Some real world objects can be both sensors and actuators. For example, a motor can be an actuator because a controller can turn it on or off. However, it can also be a sensor if it sends some internal values to the controller, like motor torque. These objects must be identified and inherit attributes and operations from both sensor and actuator objects. This is true of the window motor in the door system, shown in Figure 3.4.

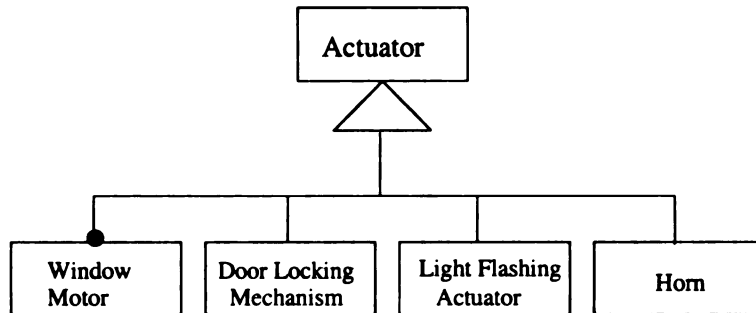


Figure 3.3: Object Model showing actuator refinement.

---

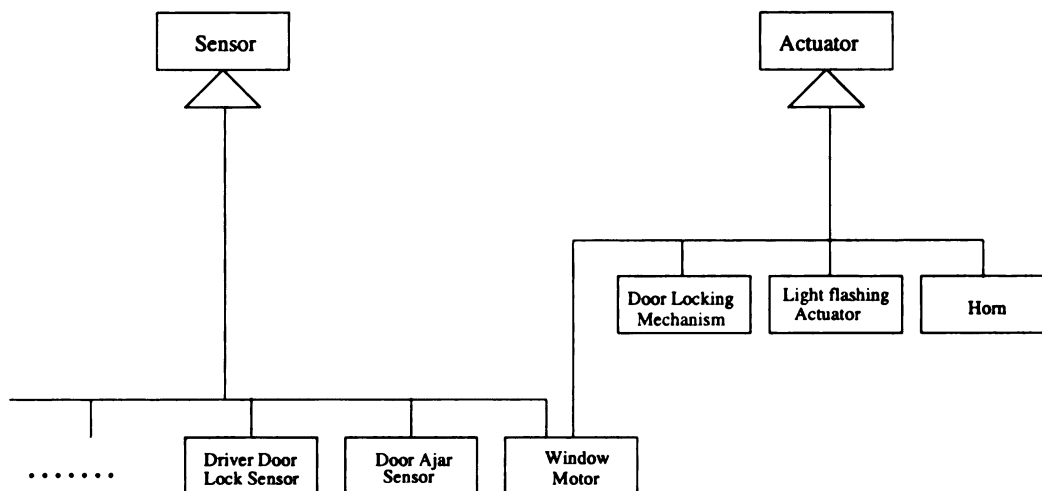


Figure 3.4: Object model showing an object that is both sensor and actuator.

---

An optional user-interface might be shown as an aggregation of any sensors and actuator included on the interface. In the doors system, only sensors, specifically buttons, are included on the door panels. The normal door panel is a back seat panel with only window controls. The front passenger's door panel is a subclass of the normal door panel with door lock buttons added. The driver's door-panel inherits these buttons as well as the window lock and unlock buttons.

The number of instances of each object must be noted on the object model. If there are many buttons, a filled circle will be placed at the endpoint of its association, such as the sensors in Figure 3.2. If there is only one door ajar sensor, then an undecorated line is adequate. If a specific number of an object class is known, it should be noted at the endpoint of the association.

The following are helpful tips on the creation of an object model [6]:

- Understand the problem. Domain research might be necessary.
- Keep the model simple at first and refine the objects later.
- Choose class name carefully.
- Develop a data dictionary containing a written description of all the classes, attributes, and operations.
- Try to have only binary relationships.
- Do not worry about multiplicities on the first iteration.
- Document the reasons behind the model.
- Refine until complete and correct.

### 3.3 System-level Dynamic Model

After an object model has been developed, a system-level dynamic model should be constructed. This dynamic model is a state diagram of the sequence of states that the objects in the system will undergo as they receive different events and stimuli.

The development of a scenario is helpful to identify the actions and the states that a system goes through in the course of its execution. A scenario shows the particular events that happen during one specific execution of a system. For example, a dishwasher scenario would consist of the 'start' button being pushed, cycle information buttons being checked, timers being set, rinse/wash/rinse/dry being executed, indication lights being activated, and the system being shut-off.

Many systems at a high-level have the form found in Figure 3.5 showing an idle state (shown as a rounded rectangle) and a running state. The transitions (arrows) between these states are labeled with the events that move the system from one high-level state to another. The running state is considered an abstract state because it encapsulates several substates within it. Abstract states are expanded gradually in later refinements to show more detail. The starting state called the start state is indicated by a small filled circle pointing to the state in which the state chart starts. For example, the start state of the dynamic model in Figure 3.5 is the idle or off state.

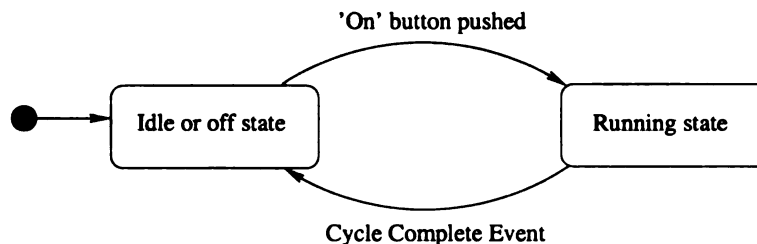


Figure 3.5: General System-level dynamic model.

---

System-level concurrency in the system should be identified. Concurrency refers to



the condition when two different objects can receive events at the same time without interacting [23]. An automobile door system has concurrency because doors should be able to be locked at the same time as multiple windows are moving up and down. Each concurrent controller object should have its own state diagram. This inherent concurrency is shown by the dashed lines and multiple state diagrams in Figure 3.6. Each of the three concurrent sections contains a high-level statechart [14] for the three corresponding controllers. Guarding conditions (shown inside square brackets) are placed on some of the transitions, such as disallowing a transition if the window lock is on. An abstract state can contain entry and exit actions, as well as an activity to be performed while in the state.

While creating a dynamic model, the following tips may be helpful [6].

- Use scenarios to begin the construction of a dynamic model.
- Only create state diagrams for objects that have meaningful behavior.
- Let the application decide on the granularity of events, actions, and activities.
- Use superstate abstraction and nesting of substates within a superstate to improve readability.
- To simplify reading make use of entry and exit actions for multiple transitions.
- Look for possible race conditions, where two different concurrent objects can begin execution, with nondeterministic outcomes for the system.

### **3.4 System-level Functional Model**

After the object model has been constructed, a system-level functional model can be developed. This development can happen concurrently with the construction of the

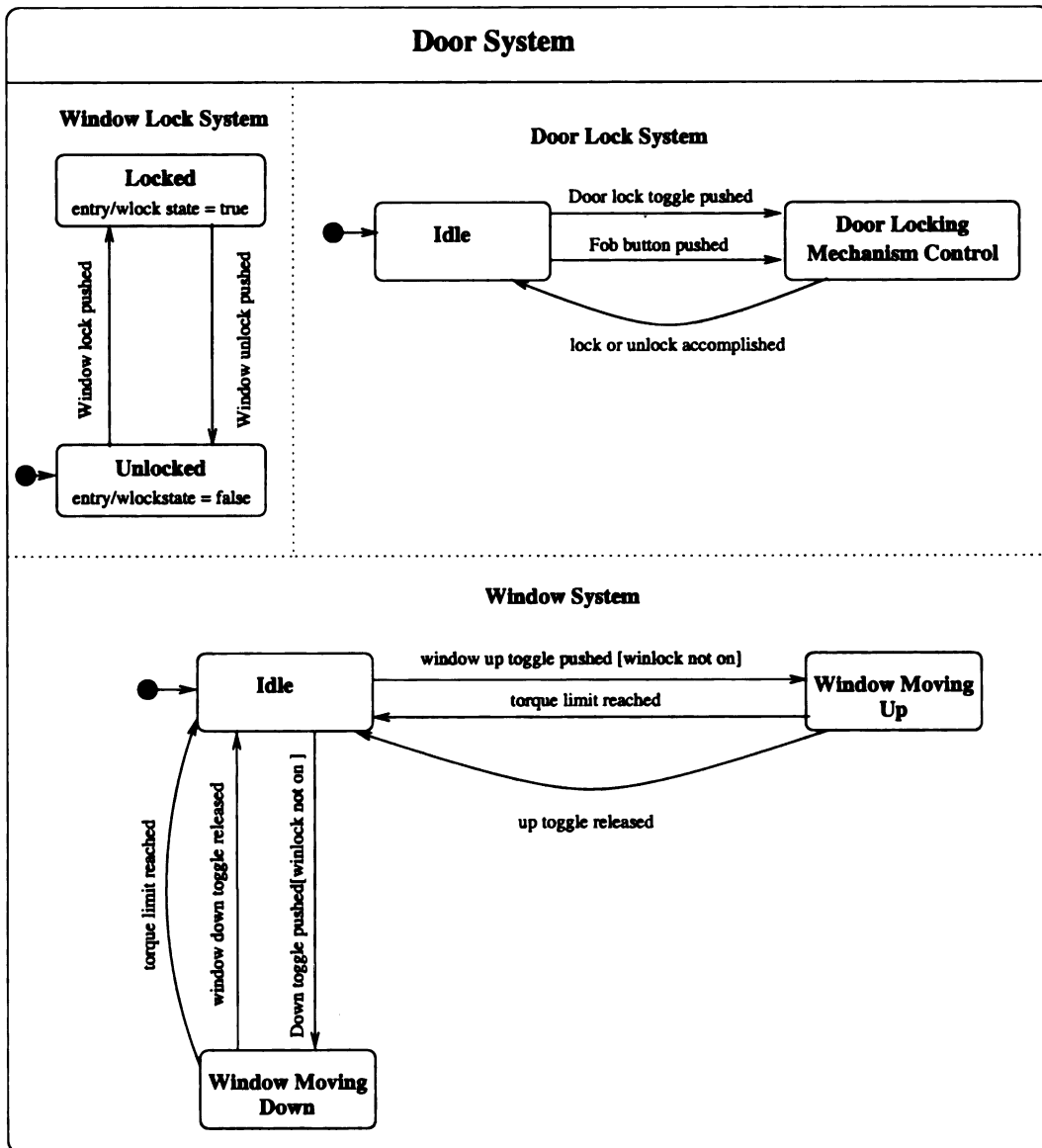


Figure 3.6: System-level dynamic model for door system.

dynamic model. The data flows (represented by labeled arrows) of the system are captured in these functional models, also called data flow diagrams. The services of the system are also displayed as processes (drawn as ovals).

Development starts with one single process oval in a diagram. The actors (represented by rectangles) of the system will be the environmental objects, that is the sensors and actuators of the system. A sample of a typical system-level functional model for an embedded system is shown in Figure 3.7.

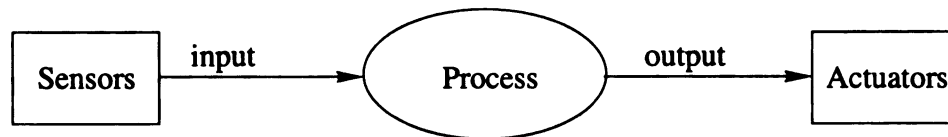


Figure 3.7: System-level functional model of an embedded system.

---

Next, refine the sensors and actuators and show all the specific data or signals that are flowing within the diagram. This data will often be control signals, but in the functional model this information is thought of as data. This diagram, showing all the sensors and actuators and data, is called the system-level functional model. An actual system-level functional model for an automotive door controller is shown in Figure 3.8.

The next refinement of the functional model should correspond to the controllers that were identified in the object model. For example, in the automotive door system, three classes of controllers were identified, as shown in Figure 3.10. These controllers, the *window system controller*, the *door lock controller* and the *window lock controller* should each have corresponding processes in the functional model. The sensors will provide input to one or more of these subprocesses and the actuators will receive output from one or more of these processes. It is possible for internal data to flow from one process to another, such as the window lock state data from the window

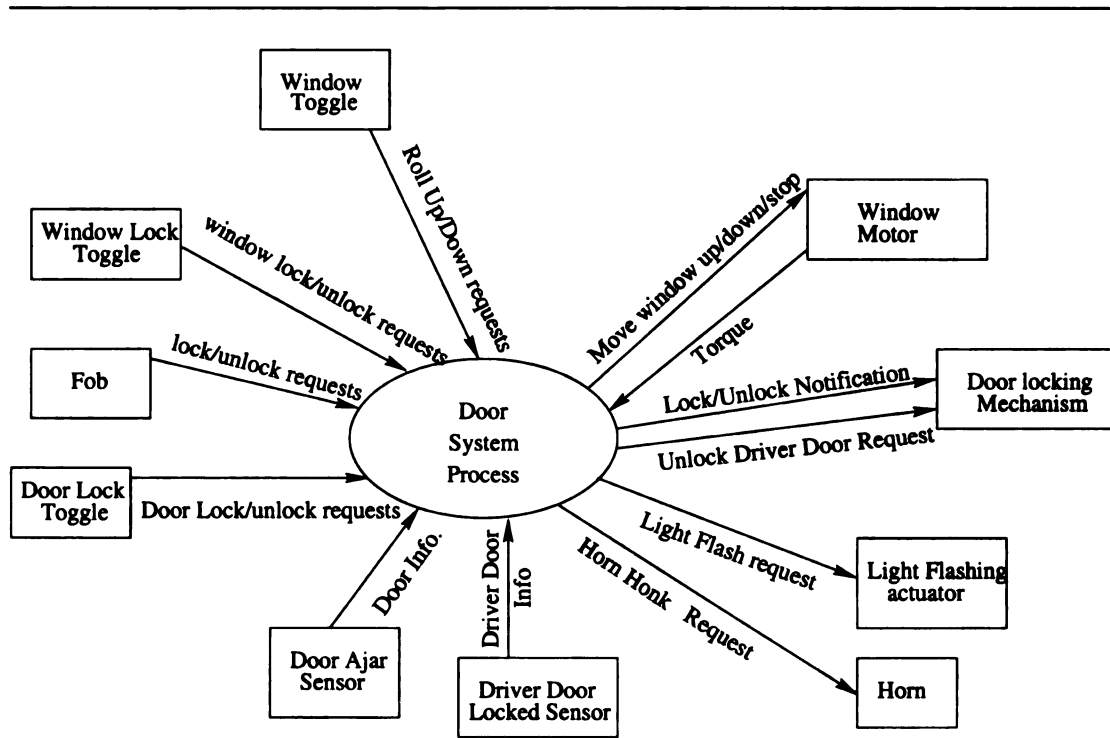


Figure 3.8: System-level functional model of an automotive door system.

---

lock process to the window controller process in Figure 3.9.

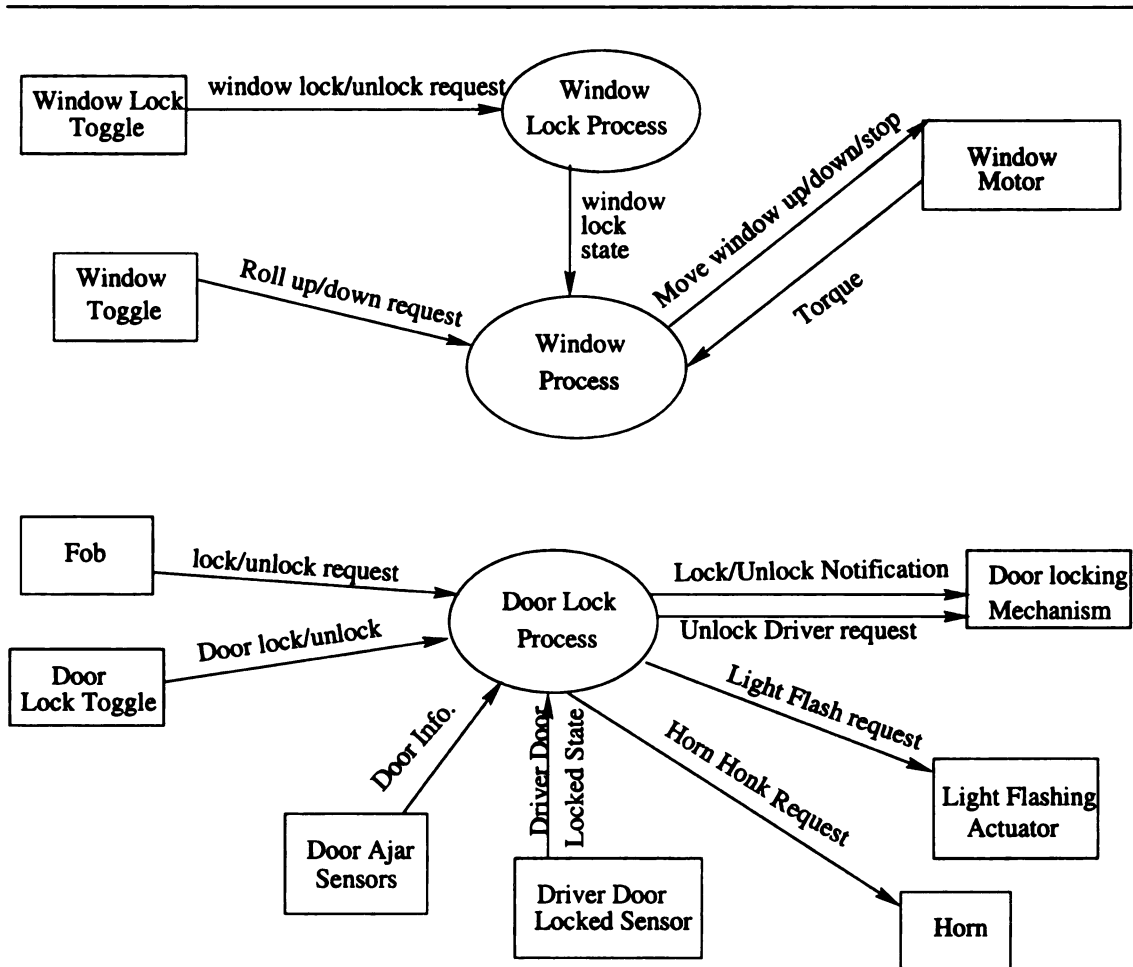


Figure 3.9: Refined functional model of an automotive door system.

### 3.5 Further Refinement of the Models.

The modeling process continues with iterative refinements of all three models. Careful review of the models will enable the detection of incompleteness or inconsistencies between the three models that can be rectified before proceeding to the next iteration. This includes additions to the object model to reflect actions on the dynamic model and the data flows in the functional model. More detailed associations can also be

added to the object model in order to show which objects interact.

### 3.5.1 Consistency between Models

Any changes necessitated by the development of the functional and the dynamic models should be reflected in a refinement of the object model. Specific things to review include:

- Each controller on the object model must have a process on the functional model as well as a state diagram within a dynamic model.
- All the actors in the functional model should be identified as sensors or actuators in the object model and vice versa.
- The data that is flowing from the actors in the system-level functional model should be added as attributes or operations to the object model.
  - Sensors from the object model that are observers should have data reflected in the functional model, shown as attributes with Boolean data type in those sensors' class objects. For example, in the door system, the *door ajar sensor* has the attribute of type 'Boolean' associated with its state, 'door ajar.'
  - Sensors that are sending objects will have an operation relating to the data. If no parameters are sent to the controller, then this data will be control data and will be of type 'bit.' For example, if the window toggle on the door panel has the *up()* operation, this relates to a specific data flow in the functional model.
  - Actuators are receiving objects. They reflect the data of the functional model in their operations. For example the automobile horn object will

have a *honk()* operation.

- Data going from the processes in the functional model to the actors should be reflected in the operations of the controllers. For example, the window lock system can '*lock()*' or '*unlock()*'. These are associated with the window locking mechanisms '*lock()*' and '*unlock()*'.

A refined object model showing attributes and operations for the automotive door system can be found in Figure 3.10. The key refinements include identification of specific operations done by the sensors, actuators, and controllers and the addition of attributes of objects which sense state information.

### 3.5.2 Associations

Associations need to be added to the object model to show the communication interaction between objects. All sensors inherit the association with the controller class labeled 'sends information to' on the object model. Similarly, the processor has an association to the actuator superclass that is inherited by all the actuators. Any other specific associations should be added, such as the association between the *window system* and the *window lock system* labeled 'gets state from.'

### 3.5.3 Refinement of Dynamic and Functional Models

The dynamic model should undergo several levels of refinement. These levels would show the details of the state changes for each object. Refinement is necessary for all controller objects. Each controller object will have its own dynamic model. The system-level dynamic model will contain some abstracted superstates that need to be refined into another dynamic model. Transitions between these states will also be labeled with events that trigger state changes. The actions that the controller





object triggers need to be specified on the transitions or upon entry and exit of the states. Activities done while in each state must also be refined. An illustration of state refinement within a dynamic model can be seen in the doors system. The state *locked* within the door lock system from the system-level dynamic model (Figure 3.6), is refined in Figure 3.11. The one superstate on the system-level model is expanded to show three substates and the actions that are related to each.

The functional model might also need further refinement. At this point, each controller should have a corresponding process bubble on a level-one functional model (as shown in Figure 3.9 for the doors system.) If there are calculations or other algorithmic-type processing of data, further refinement of the process bubbles for each process will be needed. These refined functional models will correspond to actions and activities that are shown on the dynamic models. Many embedded systems have minimal data processing, so further refinement at this point might not be necessary.

This chapter has presented the concepts and principles used to model an embedded system with OMT. This OMT model now can be translated into VHDL. A translation technique is presented in the following chapter that can do this translation. Once the system has been translated to VHDL, various tools can be used to analyze the embedded system.

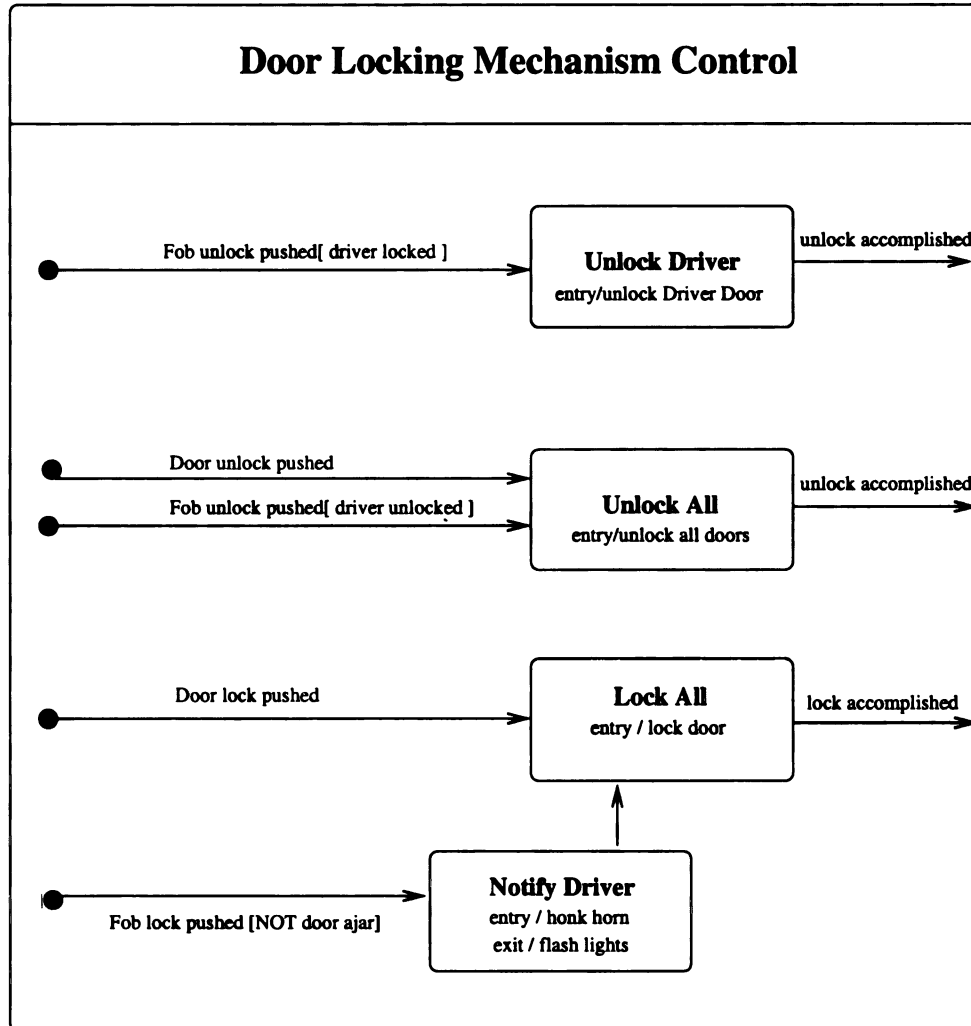


Figure 3.11: Refinement of the *Door Locking Mechanism Control* state.

# Chapter 4

## OMT to VHDL Translation

### Process

This section presents a group of rules for deriving VHDL specifications from OMT diagrams, developed using the process presented in the previous chapter. A VHDL [22] design entity consists of two basic parts, an entity and an architecture. The *entity* defines the porting or the interface of the object to the rest of the world. In order to simulate an entity, a corresponding *architecture* is needed that is considered to be the body of the entity. The architecture describes the function or behavior performed by the entity and assumes access to all the ports defined in the entity. Accordingly, the rules are decomposed into two groups that respectively describe how specific parts of OMT diagrams can be used to generate both the structural representation of OMT in VHDL as well as the behavior representation as shown in Figure 4.1. The rules are followed by a design process that uses the rules, but also indicates where developer input is needed to fill in portions of the VHDL specifications. Before the rules and process are presented, a brief description of how to transform a hierarchical dynamic model into a flattened state chart is given.

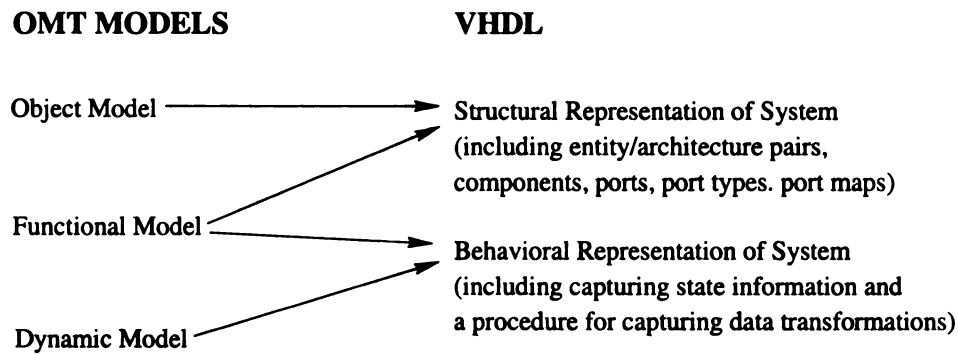


Figure 4.1: A View of the OMT to VHDL translation process.

---

## 4.1 Flattening the Dynamic Model

The dynamic model in OMT has the strength of being constructed hierarchically with substates nested within superstates. This is helpful for initial design, to handle complexity, and to allow for incremental refinement. However, for the translation process the hierarchy must be removed.

Any hierarchical state chart has a corresponding flattened state chart. Any dynamic model in OMT also has a corresponding flattened state chart that shows all its states at the same level. The basic concept is to take the deepest substate and its state chart, which will be flat, and place it in the next higher state chart, removing the superstate. This bottom up approach eventually results in a totally flat state machine.

In Figure 4.2 there is a 3-state dynamic model labeled H with two super states labeled SS1 and SS3. Each of these super states has its own state chart enclosed in a dashed circle. Within SS1 there is a superstate labeled SS1c. A state chart of the substates in SS1c is seen at the bottom of the figure. To change H into F, a flattened state chart, several steps must be taken. First, for each state  $s$  within H, one can compute the set of  $G(s)$  of all the “ground” states that are the deepest substates

within  $s$ . For this example, the ground states would consist of all the states labeled with a single  $S$ . None of these ground states are super states. The set of states in  $F$  is the union of  $G(s)$  for all states  $s$  in  $H$ .

Next, the transitions for the new flattened state chart must be computed. Each transition in a OMT dynamic model is labeled with an event  $e$ . Every transition  $t$  in  $H$  will have a source state, which might be hierarchical, which we will call  $\text{source}(t)$ . Each transition will also have a destination state, also possibly hierarchical, which we will call  $\text{sink}(t)$ . These sources and sinks can be identified by the events on the transitions. The same event and direction will identify the substate within the super state for this transition. The transition set in  $F$  can be computed from the transition set in  $H$  as follows:

For every transition  $t$  in  $H$ , add the transitions:

$$\{ (u,v) \text{ — } u \text{ in } G(\text{source}(t)) \ \& \ v \text{ in } G(\text{sink}(t)) \}$$

to the transition set of  $F$ .

This results in transitions from the deepest “ground” source state within a superstate to the deepest “ground” sink state. The flattened state chart  $F$  for this example is shown in Figure 4.3.

Any concurrency that is present in the dynamic models must not be removed. This concurrency is important to stay in the model representation. VHDL is able to handle this concurrency directly.

## 4.2 Rules for deriving VHDL from OMT

The notations from the three OMT models are translated to various parts of a VHDL specification. The portion of the OMT model that is targeted by a rule is italicized, and the corresponding VHDL component is underlined. The assumption is that

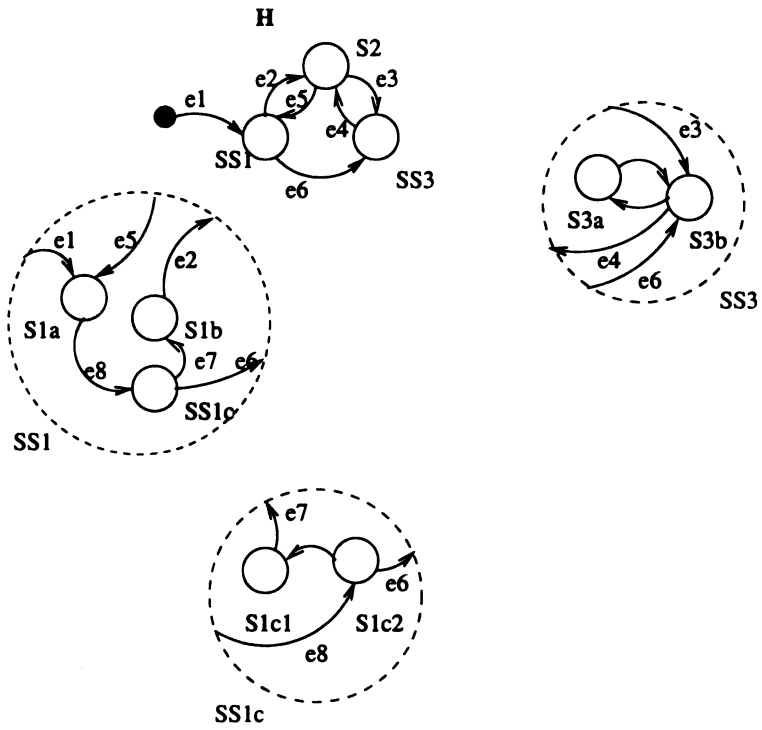


Figure 4.2: Hierarchical state chart **H** and state charts for its super states.

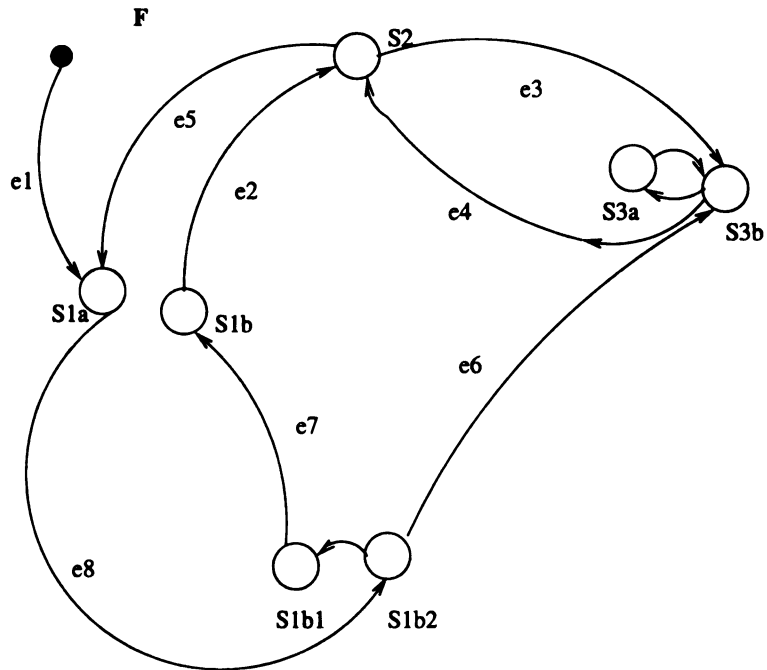


Figure 4.3: Flattened state chart **F**, equivalent to state chart **H**.

high-level object, functional, and dynamic models have been created according to the process presented in Chapter 3.

The rules for creating the **structure** of an embedded system from the various notations of the object and functional model are explained below.

OMT-S1 Every *controller class* in the object model will have a VHDL entity/architecture template created for it.

OMT-S2 *Aggregation* of controller objects (specified with a diamond) translate into VHDL components of the aggregate objects within the composite controller's structural architecture.

OMT-S3 A *data flow* on the functional model becomes a port name for the VHDL entity for a controller. The direction of the data flow will determine if an **in** or **out** port is made.

OMT-S4 An *attribute type* found on a sensor or actuator class object within the object model will become the port types for a port within the VHDL entity for the controller that interfaces with this sensor or actuator.

The rules for converting **behavior** of the system, as shown in the dynamic and functional models, into behavior architectures in VHDL follow. As before, the OMT diagram components are italicized and the generated VHDL component is underlined.

OMT-B1 Every *dynamic model* for a controller needs a behavioral architecture that contains the following items:

- (a) A type declaration called `state_type` that can contain the enumerated values of *all states in the dynamic model*.
- (b) Two variables of type `state_type` called `Present_State` and `State`.
- (c) The *start state* in the dynamic model will be the initial value of both the `Present_State` and the `State` variable.

(d) A process is constructed within the behavioral architecture to force sequential execution of control.

OMT-B2 All *events* that cause transitions from one state to another within the dynamic model will be included in a wait statement or sensitivity list for this architecture's main process. A change to one of these signals will happen when an *event* happens. This event will start the associated process executing.

OMT-B3 Each *state* in dynamic model will be included in a CASE statement that senses the `Present_State` variable. The CASE statement will keep track of what the present state is and if a transition to a new state is allowed.

OMT-B4 *Transitions* between states are controlled by 'If' statements within the Case Statement for each state. If the correct event has happened, then the code to handle a new state can be entered.

OMT-B5 A *guard* on a transition is also included in the same If statement. The If statement will include the event that causes the transition and any guard statement that will keep that state from being entered. They will be conjuncted together.

OMT-B6 For *Transition on completion* of a state, or a transition that is not labeled with an external event, a done event will be introduced into the VHDL code. A change to this done event will also be included in the the sensitivity list for the event handling process. This signal will be changed by the state that exits without an event; that is, the signal corresponds to an event completion. This signal will be toggled within the CASE statement for that state.

OMT-B7 *Actions* on transitions in the dynamic model, unless they are atomic actions, will be defined as VHDL procedures with any parameters associated with the action passed to the procedure. *Actions on entry and exiting of a state* will also be handled as a VHDL procedure call. Atomic actions consist of a simple change of a signal or variable value.



OMT-B8 *Activities* within a state in the dynamic model will be added within that state's process from algorithmic information from the functional model.

### 4.3 VHDL Translation Process

Given the above rules for deriving VHDL specifications from OMT diagrams, the complete VHDL specification is created using the following process, where rule usage and user input are explicitly indicated.

1. Create an entity/architecture template for the whole system with the entity name coming from the name of the object model's system aggregate object as shown in Figure 4.4 for the door system. (*Use Rule OMT-S1.*)
  - (a) Developer clarifies all multiplicity in object model. For example, determine whether there are two or four windows in the system.
  - (b) Developer supplies names for any objects with multiplicity in the object model (i.e., supplies instance names). For example, the names `Back_RtWindowSystem` and `Back_LeftWindowSystem` could be supplied.
  - (c) Developer specifies if there is any interaction between multiple objects of the same class. On the object model this would be a circular association from an object back to itself. An example would be a set of bit adders that are hooked together into a larger adder.
2. Add the port names and direction (**in** or **out**) to the system entity. (*Use Rule OMT-S3.*)
  - (a) Each data flow found on the high-level functional model will be a port, as shown in Figure 4.5.

- (b) Create multiple ports for data flows from multiple objects (actors) that have been identified in step (1a).
  - (c) Create unique names by concatenating the sensor or actuator name with the data flow name behind the name from the developer obtained in step (1b), to create unique names.
3. Identify the port types for all ports, refining the object model to add any additional attributes or operations identified. (*Use Rule OMT-S4.*) (See Figure 4.6 )
4. If a controller is an aggregation of other controllers, reflect this aggregation with more entities. (*Use Rule OMT-S2.*)
- (a) Refine the object model to reflect this aggregation with a diamond on the association.
  - (b) Refine functional model with a process bubble corresponding to each aggregate controller and correct placement of signals.
  - (c) Create entity/architecture templates for each aggregate controller.
  - (d) Add new controllers as components to the composite controller. Any multiple instances of components will have had names supplied by the developer in step (1a). Concatenate these names to the controller name.
  - (e) Create any internal signals that are shown in the refined functional model as coming from one controller process to another. For example, the data flow, *window lock state*, leaves the *Window Lock Process* and enters the *Window Process* (Figure 3.9), so it would need an internal signal declared for it.

- (f) Create ports within the aggregate controllers from the refined functional model. Port names will be the concatenation of the aggregate controller and the data flow name. The direction can be determined by the direction of the arrow in the functional model.
- (g) Port mapping for the components within a composite controller can be determined by using the functional model and matching the concatenated signal names. Developer confirmation will be needed in the matching of the semantics of the signal names and the data flows on the functional model.

5. Refine the dynamic model to show new aggregate objects.

The **behavior** of the system is added to the VHDL architectures in the following procedural steps. Please see Figure 4.8 for an example of a behavior architecture.

6. Refine functional model and dynamic model for each controller object.

7. Use the dynamic model for determining the structure of the controller objects' behavioral architecture.

- (a) Create a type declaration called `state_type`. (*Use Rule OMT-B1.*) It should contain the enumerated values of all the states in the dynamic model.
- (b) Create two signals to contain the present state of the diagram as well as one for the next state. (*Use Rule OMT-B1.*)
- (c) Create two concurrent processes within the behavior architecture for this controller object. One will accept all the events that happen in the system and will initiate the state change. The other process will handle the actual states, to what states they transition, as the actions and activities related to these states. (*Use Rule OMT-B1.*)

- (d) Add all the events on the transition on the dynamic model to the sensitivity list for the event handling process. (*Use Rule OMT-B2.*)
  - (e) Create a CASE statement in the state changing process that checks the value of the last state. Based on the last state, a group of if-else-statements will be placed directing the process to the next 'state.' (*Use Rule OMT-B3.*)
  - (f) The 'If' statements within the 'CASE' statement will contain the events and guards for that transition conjuncted together. (*Use Rules OMT-B4, OMT-B4.*)
  - (g) Add additional signals to handle the cases of circular state changes and state changes that happen on completion of an activity, not because of an event. (*Use Rules OMT-B6.*)
8. Before leaving a state section, the name of the present state should be placed in the 'present state' variable.
  9. Write any needed procedures to handle any actions. (*Use Rule OMT-B7*)
  10. Fill in activities within the state processes. (*Use Rule OMT-B8*)
  11. Compile or analyze each behavior entity as it is completed.
  12. Compile or analyze any composite controller when its aggregate parts are analyzed correctly.

---

```
entity Door_System is
  port ( );
end Door_System;

architecture structural of Door_System is
begin
end structural;
```

Figure 4.4: Main entity and architecture templates for Automotive Door System.

---

---

```
entity Door_System is
  --port info. added this step.
  port (
  --in port signals
  Foblock : in ;
  FobUnlock : in ;
  DoorAjar : in ;
  .
  .
  .
  BkLftWindowMotorDown : out ;
  BkLftWindowMotor : out);
  HornHonk :
end Door_System;
```

Figure 4.5: Ports added to Door\_System entity from the Functional Model and the number of instances from the Object model.

---

---

```
entity Door_System is
  --port info. added this step.
  port (
  --in port signals
  Foblock : in BIT ;
  FobUnlock : in BIT;
  DoorAjar : in BOOLEAN;
  .
  .
  .
  BkLftWindowMotorDown : out BIT;
  BkLftWindowMotorStop : out BIT);
  HornHonk : out BIT);
end Door_System;
```

Figure 4.6: Input port type information added.

---

---

```
architecture structural of Door.System is
  component Window_System
  end component;
  component Door_Lock_System
  end component;
  component Window_Lock_System
  end component;
begin
  driver_window : Window_System
  port map ();
  pass_window : Window_System
  port map ();
  .
  .
  .
end structural;
```

Figure 4.7: Component information added to the declarative section of the structural architecture.

---

---

```
ARCHITECTURE behavioral OF WindowSystem IS
TYPE State_type IS (Idlestate, WindowMovingUpState, WindowMovingDownState);
SIGNAL PresentState : State_type <= Idlestate;
SIGNAL State : State_type := Idlestate;
BEGIN
  StateChanging : PROCESS
  BEGIN
    WAIT ON WSWindowDown, WSWindowUp, WSMotorTorque;
    PresentState <= State;
    CASE PresentState IS
      WHEN Idlestate =>
        If ((WSWindowUP = '1' )and (WSWindowLockState = FALSE))
          THEN
            WSWindowMotorStop <= '0';
            WSWindowMotorUp <= '1';
            State <= WindowMovingUpState;
          ELSEIF ((WSWindowDown = '1') and (WSWindowLockState = False))
            THEN
              .
              .
              .
            WHEN WindowMovingDownState =>
              IF ((WSMotorTorque = '1') or (WSWindowDown= '0'))
                THEN
                  WSWindowMotorDown <= '0';
                  WSWindowMotorStop <= '1';
                  State <= Idlestate;
                END IF;
            END CASE;
          END PROCESS;
```

Figure 4.8: Behavior architecture for a controller.

---

# Chapter 5

## VHDL Analysis

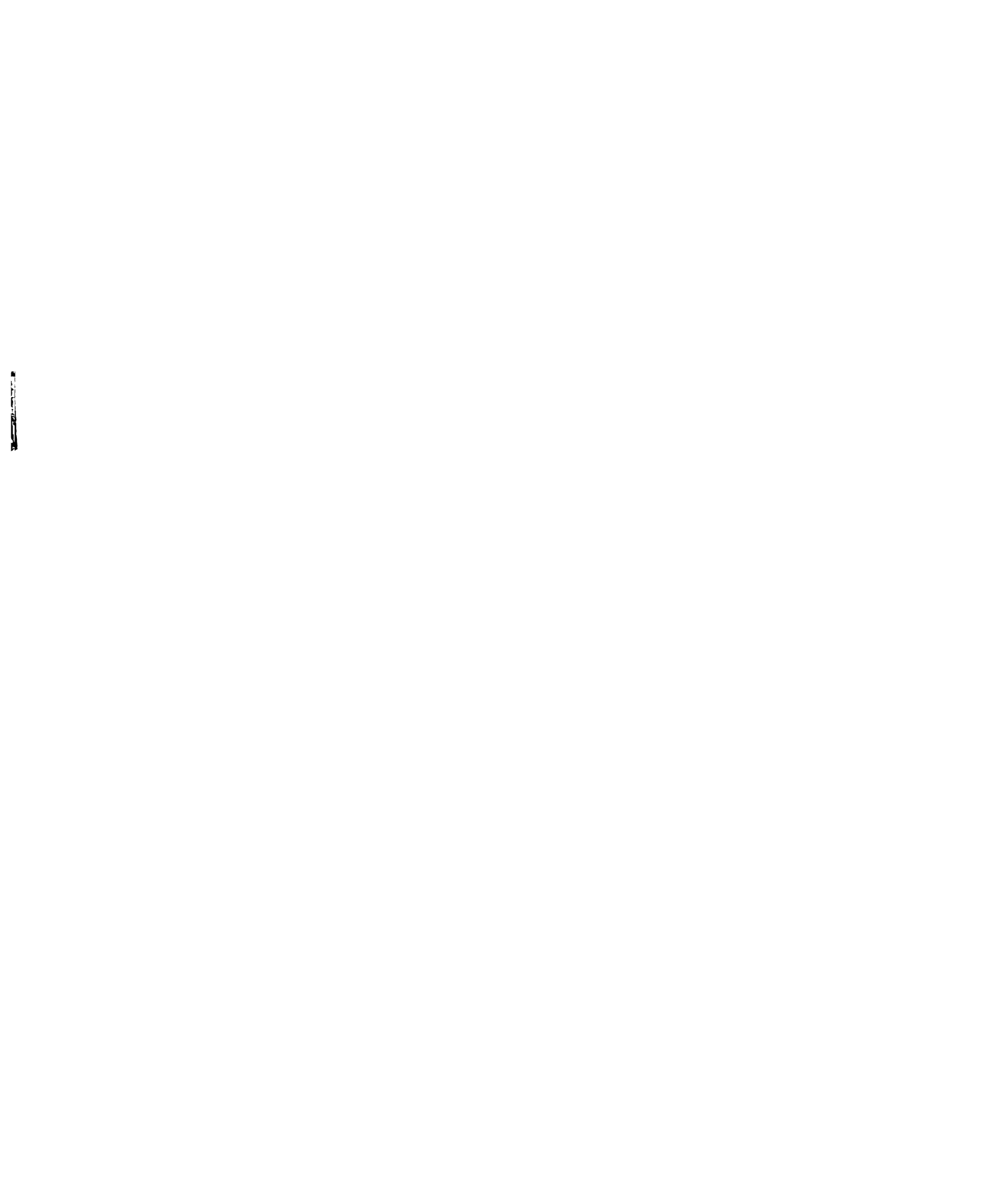
Once the OMT models have been developed, the corresponding VHDL modules can be analyzed using numerous existing tools. The results of compiling of the generated VHDL code will find naming inconsistencies, syntax problems, and also whether all the signals declared in the system were actually used. Also the typing of all ports will be automatically checked.

Once each module has been compiled correctly, the system can be given a specific configuration and its behavior can be simulated. Input testing files can be generated manually to check various system conditions and behaviors. These input files will correspond to the input signals coming from the environmental actors or sensors. The test scenarios will show changing signal values over time.

The result of this simulation using the input files will be a series of outputs for the system's actuators. The input/output pairs are compared to see if the system has the expected behavior. The VHDL simulation cycle is explained in further detail in Section 5.1.

More formal analysis of VHDL systems is also possible. In addition to behavior simulation, different properties of the system can be checked, such as timing and





well-formedness. Several tools that can handle VHDL are described in Section 5.2. The results of applying the VHDL simulation environment to the Automotive Doors System is presented in Section 5.3.

## 5.1 VHDL Simulation

Event-driven simulation is the stimulus-response paradigm used in VHDL simulators [22]. The VHDL event-driven simulation cycle defines the dynamic semantics of VHDL. A VHDL simulator is a software program running a VHDL description that responds to a series of inputs. The simulator computes the output. The input/output behavior of an accurate VHDL model reflects the same input/output behavior of the associated actual hardware [21].

Event-driven simulation is based on the concept that actions are initiated when events or signals change value. More than one signal can change value at one simulation time unit. The model responds to these changes by running processes that in turn will change more signals' values in the future. The simulation time is then advanced to the next event or input, and the model repeats until there are no more events [10].

When simulation begins, a VHDL module is first initialized. See Figure 5.1 for a diagram of the simulation cycle used by VHDL simulators. During initialization, the initial value of each declared signal is computed. There is an assumption that this value has persisted for an infinitely long duration of time in the past. The initial signal value may explicitly be defined or the signal is assigned the default value that is associated with the signal or port type. Based on these signals' initial values, each process in the model is executed until there are no more signal changes to evaluate. After all these activities take place, the model is considered to be initialized.

At this point the event cycle begins. These event cycles continue until there are no more changes to signals, i.e., no more events. If there is a non-terminating process, such as a clock, the simulation typically terminates after a user-specified amount of time has passed [10].

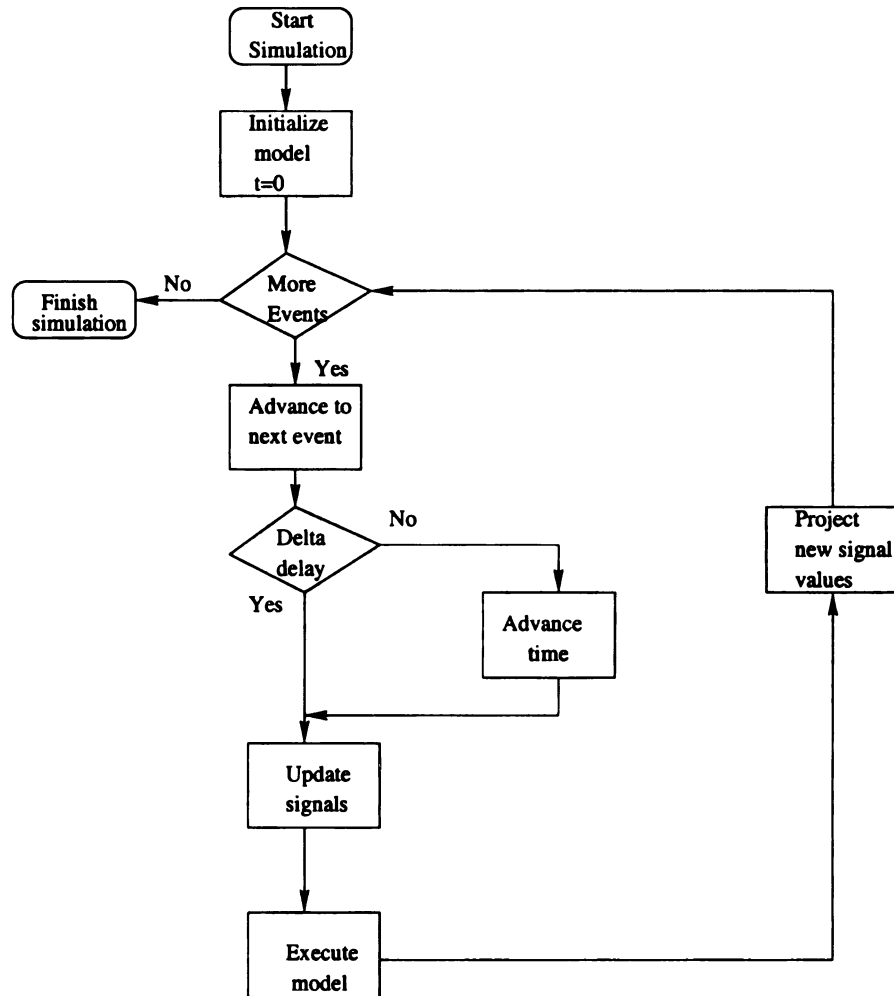


Figure 5.1: Simplified diagram of VHDL simulation cycle [10].

---

Mixed-level simulation is allowed where different VHDL components of the system are defined at different levels of abstraction and are simulated together. Lower levels of abstraction have more timing details and have more functional parallelism. These take longer to simulate compared to higher-level abstract components. Often simu-

lation is done on one lower-level unit of interest, combined with more abstract-level components. This arrangement allows for faster simulation.

## 5.2 Other Analysis Tools

VHDL specifications can be translated into Typed Decision Graphs [18] (a variant of Binary Decision Diagrams [5]) for various verification tests [9]. Thus VHDL code can be transformed into a form where proofs of well-formedness can be generated to recognize problems with bus conflicts, improper asynchronous loops, and incorrect references to signals [19].

Several tools can analyze standard VHDL modules during system level design to verify timing aspects of complex systems. An example is the Dynamic Stimulus Generation and Response Validation (DSGRV) [3]. Hierarchical formalized timing diagrams [11] compose time specifications for a system. A waveform editor called Shadow, developed by Bell Research, is used to generate these timing diagrams in a graphical environment [11]. The timing specifications are used to generate inputs to the VHDL simulator and the tool is able to automatically compare the simulated outputs with the expected outputs.

## 5.3 Analysis of Automobile Door System

The automotive door system was developed using OMT analysis as presented in Chapter 3 to create object, dynamic, and functional models. These models are given in their entirety in Appendix A. The OMT to VHDL translation process as presented in Chapter 4 was used to generate VHDL entities and architectures, found in Appendix B. These VHDL components were compiled and simulated using the tool WorkView

Office [29], which includes a VHDL analyzer or compiler, and a VHDL simulator.

### **5.3.1 Static Analysis**

When the VHDL entities and architectures were compiled, several types of errors and warnings were generated. The most common were signal name inconsistencies, which would not be a problem if this VHDL generation process were automated. Type inconsistencies were found. Several of these were Boolean data types that were defined as bit types at a corresponding port. Improper structure of 'if then else if' statements were flagged. Also the absence of needed punctuation and key words were found. The most important problem revealed by the static analysis was the changing of the same signal by several concurrent processes. This resulted in a restructuring of a rule in the OMT to VHDL translation process.

### **5.3.2 Dynamic Analysis**

During the simulation of the automotive doors system, several design flaws were revealed, forcing sharper thinking about the true requirements of the system. The first case of finding a flaw related to the action of locking all the vehicle's doors. All types of locks were placed into one state in the dynamic model. Locks from the keyless fob unit entered this state, as well as lock requests for the driver and passenger doors. During model refinement, the actions of blinking the lights and honking the horn were added to this state from the original requirements for a lock request from the fob. However, the lock requests that come from the driver and the passenger doors were not supposed to cause these actions. The developer did not see this flaw with a visual inspection of the models. However, the simulation within the VHDL simulator environment did reveal this requirements error. This design flaw needed to

be corrected, not only in the VHDL door lock behavioral architecture, but also in the OMT dynamic models and accompanying documentation.

The second design flaw was associated with the transitions from various states in the window system. As originally modeled, more events caused transitions than could physically be the case. For example, the *Window Moving Up State* in the system-level dynamic model had three transitions leaving that state. One was for *torque limit reached*. One transition occurred when the *up toggle released*. And the last transition was for *window toggle down pushed*. During simulation, the result showed an inconsistency. On closer inspection, the developer realized that the *up toggle released* event would always precede the *window toggle down pushed* event because of the physical nature of a window toggle. Therefore, the *window toggle down pushed* transition was removed from the dynamic model for the window system and the translation process was re-executed, resulting in VHDL that generated the expected results.

Also pin-pointed during the dynamic analysis was the incompleteness of the modeling of some needed events within the dynamic model. During simulation, the system became trapped in a state and never transitioned from that state. For example, all of the states within the *Door Locking Mechanism Control* dynamic model needed to have an *unlock accomplished* event to cause a transition from these states. These events were added to the dynamic model. Also, several states in the dynamic models needed explicit events upon entry to these states. The events were implied by the state names, such as *UnLock Driver*, but the states needed explicit events specified, such as *entry/unlock Driver Door*. The addition of these events resulted in more complete and accurate models.

The creation of OMT models for the automotive door system and the use of the translation process presented in Chapter 5, followed by the use of a VHDL simulator

resulted in more complete models for the system. Also the requirements for the system were able to be simulated and checked.

# Chapter 6

## Related Work

This chapter overviews other projects that address the formalization of OMT models, the object-oriented design of embedded systems, or the generation of VHDL modules from graphical models.

### 6.1 The Formalization of OMT models

The notation of the object model for OMT was analyzed by Bourdeau and Cheng [4]. A-schemata, or *analysis object schemata*, was used to represent the object models. *Larch Shared Language* was used to change this A-schemata into a algebraic specification. The result of their work is that the semantics of the OMT object model are now well-defined.

The formalization of the remaining OMT models was continued by Wang and Cheng [27]. The dynamic[26] and functional models[25] were integrated formally with the object model. Algebraic specification, as well as process algebras, were used. LOTOS and ACT ONE [2] were the target specification languages. A rigorous design process was also defined. As a result, rigorous analyses can be applied to a specification and the OMT models can be checked.



## 6.2 Object-oriented design of VHDL modules

Chung and Kim[8] proposed an object-oriented design of VHDL components. Standard VHDL is extended with additional fields to represent the constructs of object-oriented inheritance in their research. Libraries of previously designed modules can be reused while managing version control. No specific process for developing a high-level design is proposed nor is a graphical modeling of components used.

## 6.3 Generation of VHDL from Graphical Models

BetterState [1] supports the automatic generation of executable VHDL code from either StateCharts [14] or PetriNets, but it does not support the depiction or the specification of the structural or data flow aspects of the system. This approach [1] only handles the control portions of a system and cannot handle any data transformations. Only architecture units are created in VHDL, not the corresponding entities. The VHDL code must also be supplied for all events and transitional actions. An object-oriented design of embedded system is also not supported by BetterState.

SpecCharts [12, 13] is a high-level specification language designed specifically for modeling embedded systems' requirements. SpecCharts has two forms: state transition diagrams and an equivalent textual form that is an extension of VHDL. The state diagrams capture control information but do not show any data transformations or structural information about the system.

Translation of CASCADE [20] control graphs, similar to Petri Nets, into VHDL is proposed in [15]. CASCADE control graphs can be represented graphically or textually but cannot capture data transformations.

What distinguishes our research from the approaches discussed above is:

1. A process for object-oriented modeling of embedded systems in terms of a commonly used graphical modeling technique.
2. The specific object-oriented approach includes explicit depiction of data transformations and system structure.
3. A specific process and rules for translating object-oriented diagrams into standard VHDL entity and architecture specifications.

# Chapter 7

## Conclusions and Future Work

As the complexity of embedded system increases, new design and development techniques are necessitated. Object-oriented design and graphical modeling help to provide high-level abstractions based on real world objects. Using a modeling technique such as OMT, allows the designer to explore requirements before committing to any implementation details.

The modeling of an embedded system in OMT has been described and illustrated with the development of graphical models for an automotive door system. Rules for formalizing OMT into VHDL are presented, as well as a stepwise development process.

This translation process was applied to an embedded system, the automotive door system. The automotive door system has been used throughout this document to help illustrate the modeling and translation process.

One of the initial results of the translation process is finding missing items in the OMT models. The identification and correction of this missing information results in a more thorough documentation of the system. For example, some states in the dynamic model had actions within them that were not explicitly specified, just

implicitly assumed. Also, consistency between names on the various diagrams were checked by the translation process. This forced clarification of possible design and requirements inconsistencies.

The translation process itself was reasonably straightforward. One of the most difficult aspects was the existence of multiple objects of the same class in the system and in the resulting VHDL code. The need for well thought out naming conventions was illustrated. The possible addition of an instance diagram that explicitly shows all the objects in the system might be helpful. Explicit instantiation of actors within the system-level functional model would also be a possible improvement.

One area of future research revolves around an interesting dilemma of the proposed translation technique, as well as any translation technique. How can there be verification that a specification language has captured the meaning of a model. Also, how can the completeness, correctness, and consistency of the technique be proven. OMT is presented by Rumbaugh with a syntax, but without an exact semantics for the models and their symbols. The meaning given to the elements of the OMT model has been previously researched by Wang and Cheng [4, 27]. See Section 6.1 for more information about their approach. Their understanding of the meaning of the OMT notation was the foundation of this work. Their work also had to wrestle with the problem of how to prove that the semantics assigned to the syntax was accurate. The issue does not stop once there is an agreed upon semantics for the OMT syntax. How can anyone be assured that the actual specification of this meaning is accurately defined in the target language, in this case VHDL? How can it be proven formally that the semantics of the language is completely and correctly captured?

The completeness, correctness and consistency of the OMT to VHDL translation presented in this thesis has not been studied in depth. If the general understanding of what is meant by a model is not reflected in the resulting VHDL code, the translation

rules themselves could be flawed, rather than the original design itself. However, this research still has much validity as a process bringing more formality to the design of embedded systems.

In the future, more case studies of embedded systems should be developed with this design technique, including extending the model refinement process through actual fabrication. The automation of this translation process is also a logical step for this work. Work is underway to extend VisualSpecs [7], a graphical environment that supports the construction of the OMT models and generates the corresponding formal specifications, to support this VHDL translation process and to interface with existing VHDL analysis and simulation tools.

## APPENDICES

# Appendix A

## Automotive Door System: OMT Models

This appendix consists of the OMT models for the automotive door system. First presented is the completed object model in Figure A.1. This is followed by the system-level dynamic model in Figure A.2 and refined dynamic models for various superstates on the system-level model. Figure A.6 is the system-level functional model for the doors system. Finally, a refinement of the functional model, that shows the processes divided in relationship to the processes defined in the object model, is presented in Figure A.7.

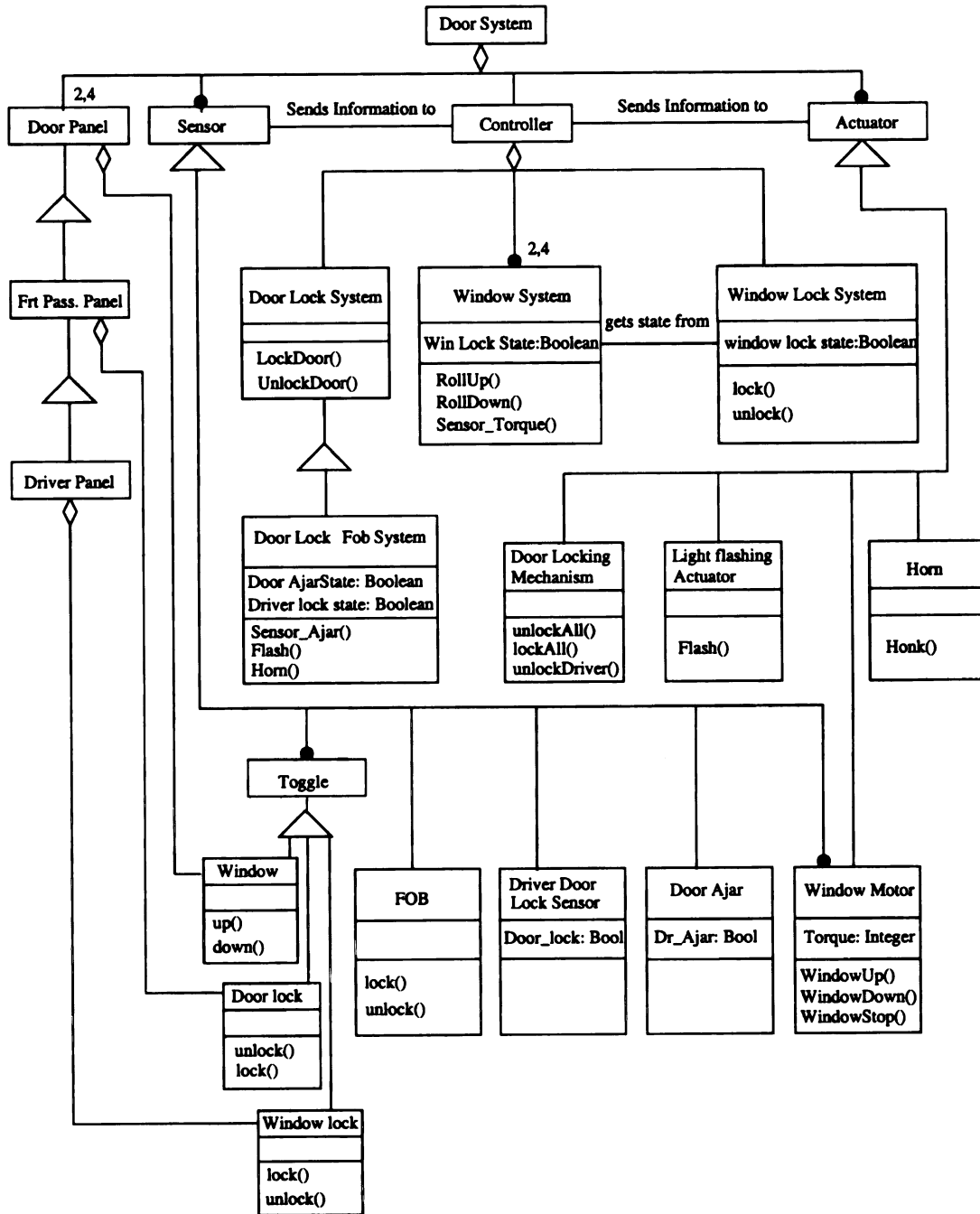


Figure A.1: Object model for door system



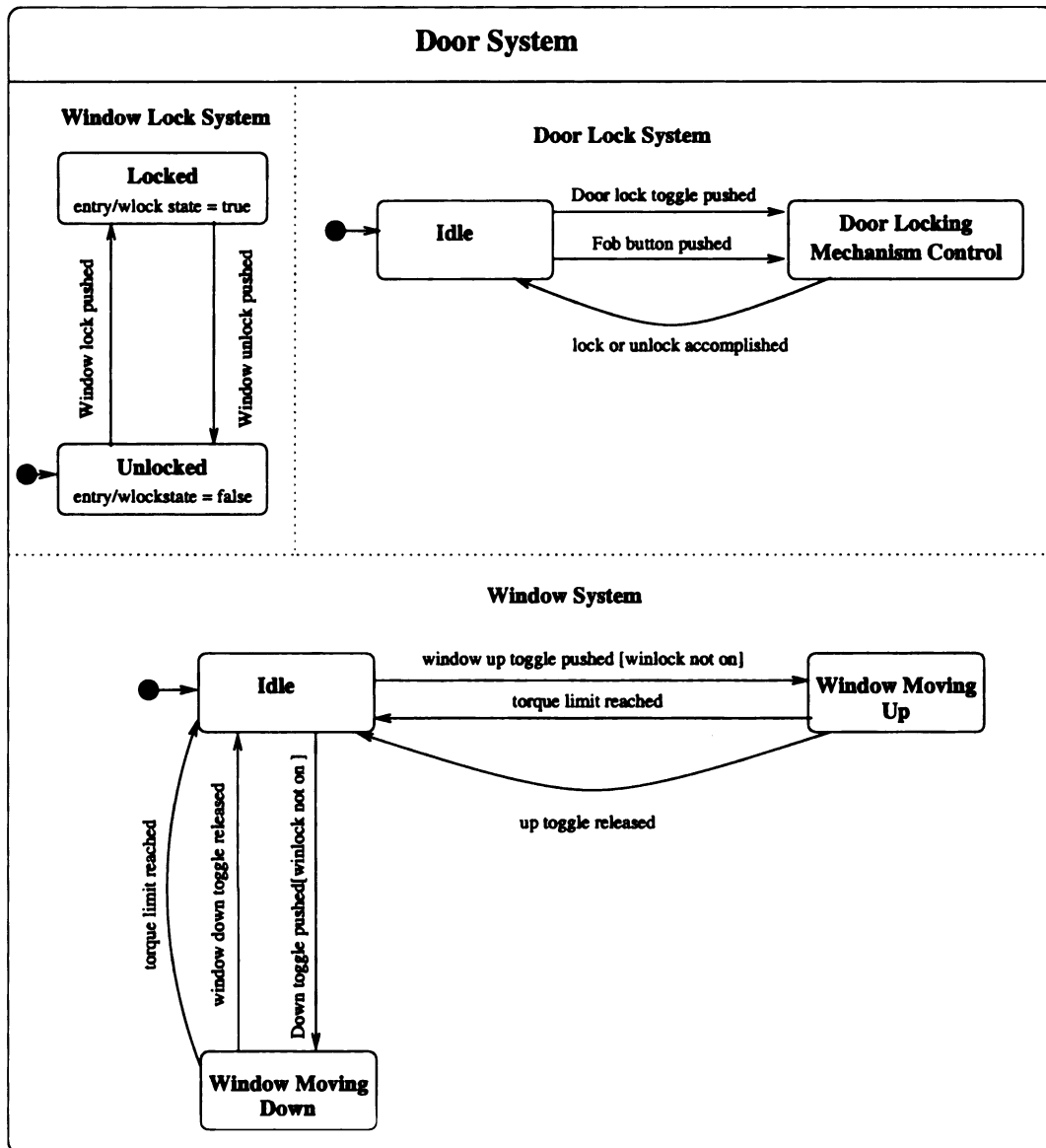


Figure A.2: System-level Dynamic Model

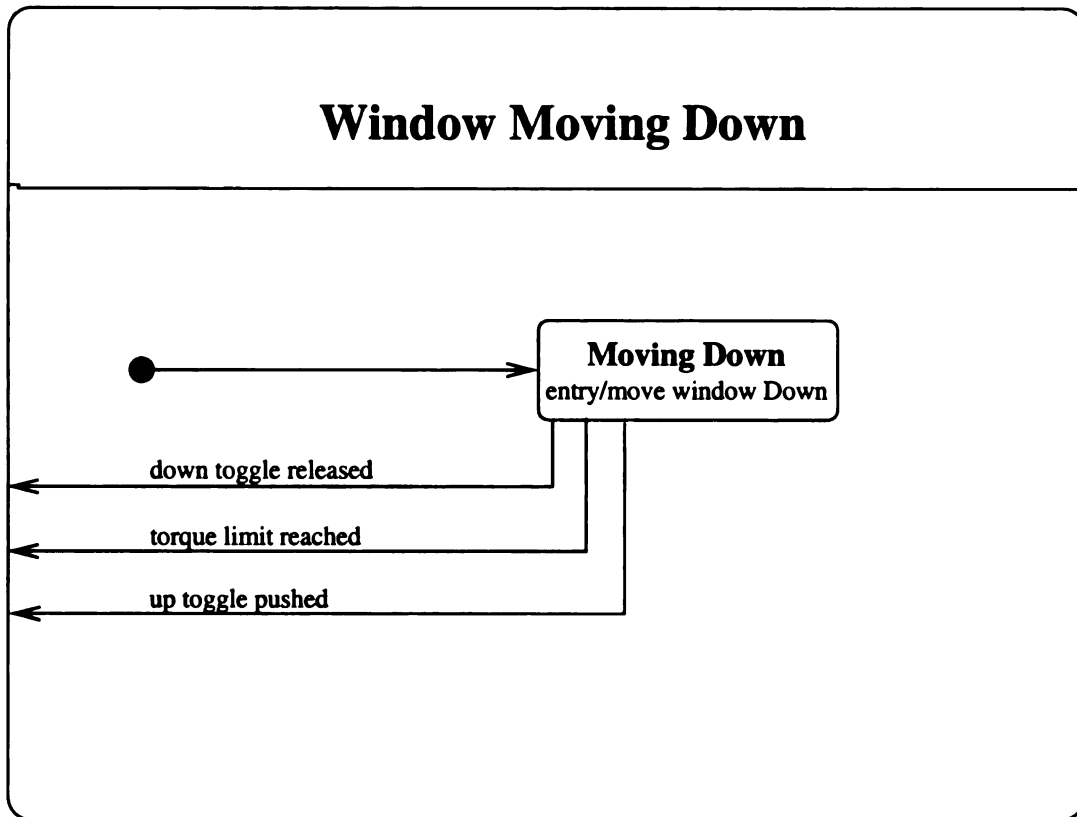


Figure A.3: Dynamic Model for Window Rolling Down State

---

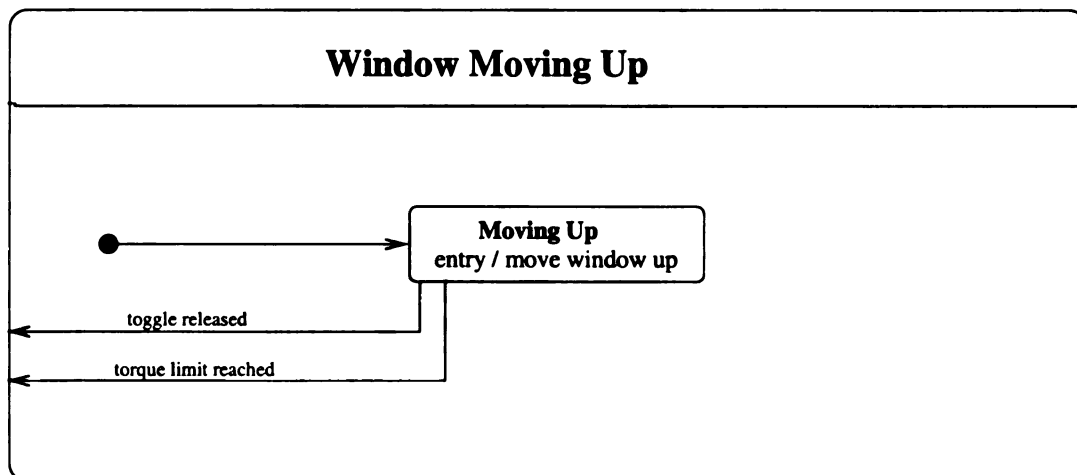


Figure A.4: Dynamic Model for Window Rolling Up State

---

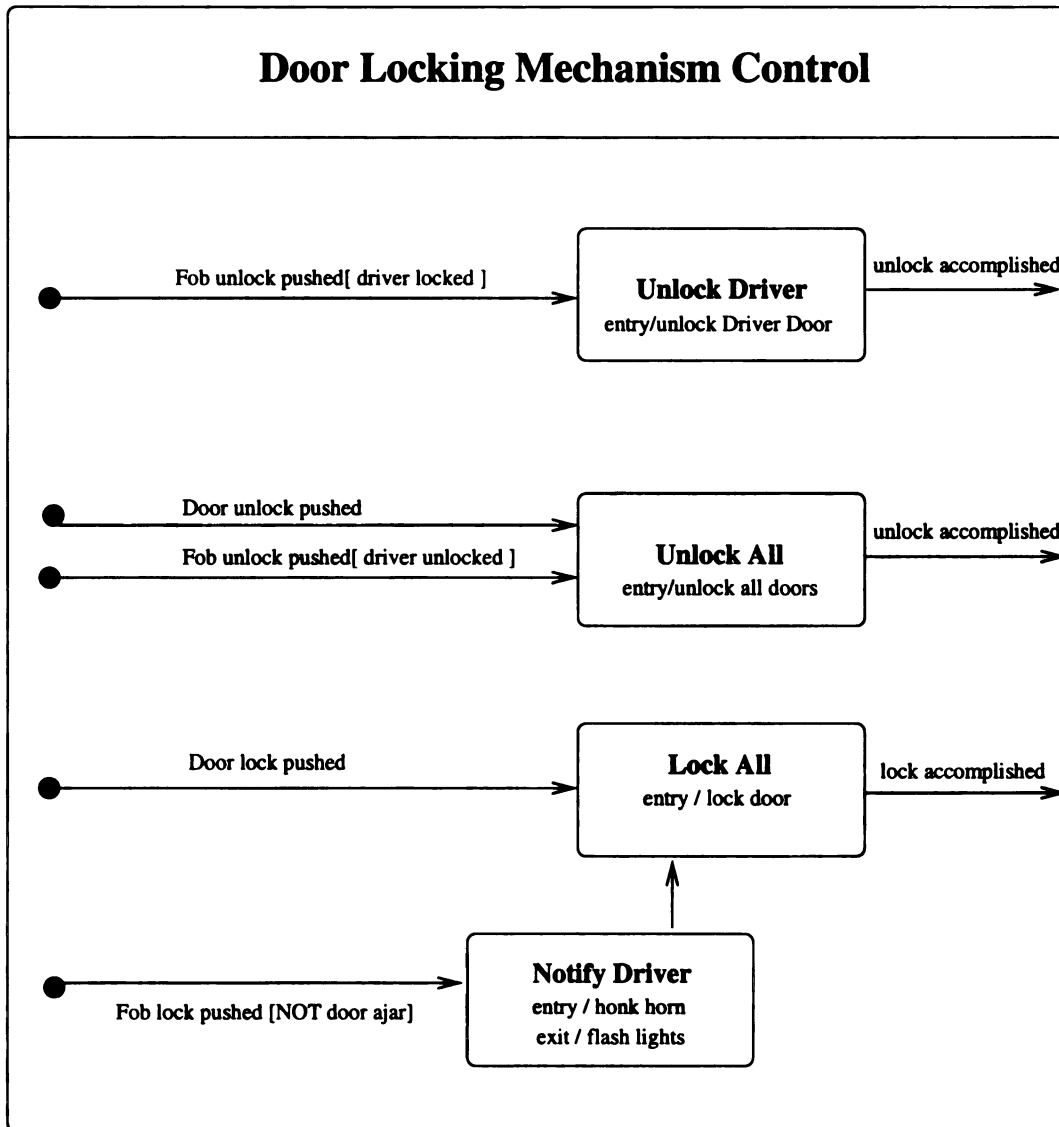


Figure A.5: Dynamic Model for Door Lock State

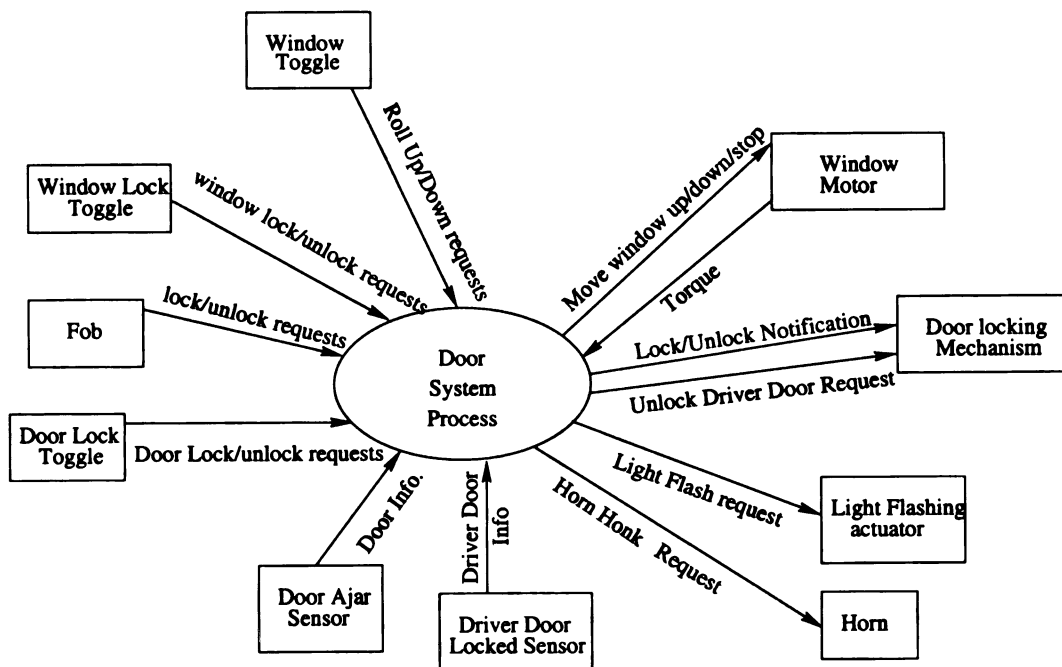


Figure A.6: System-Level Functional Model

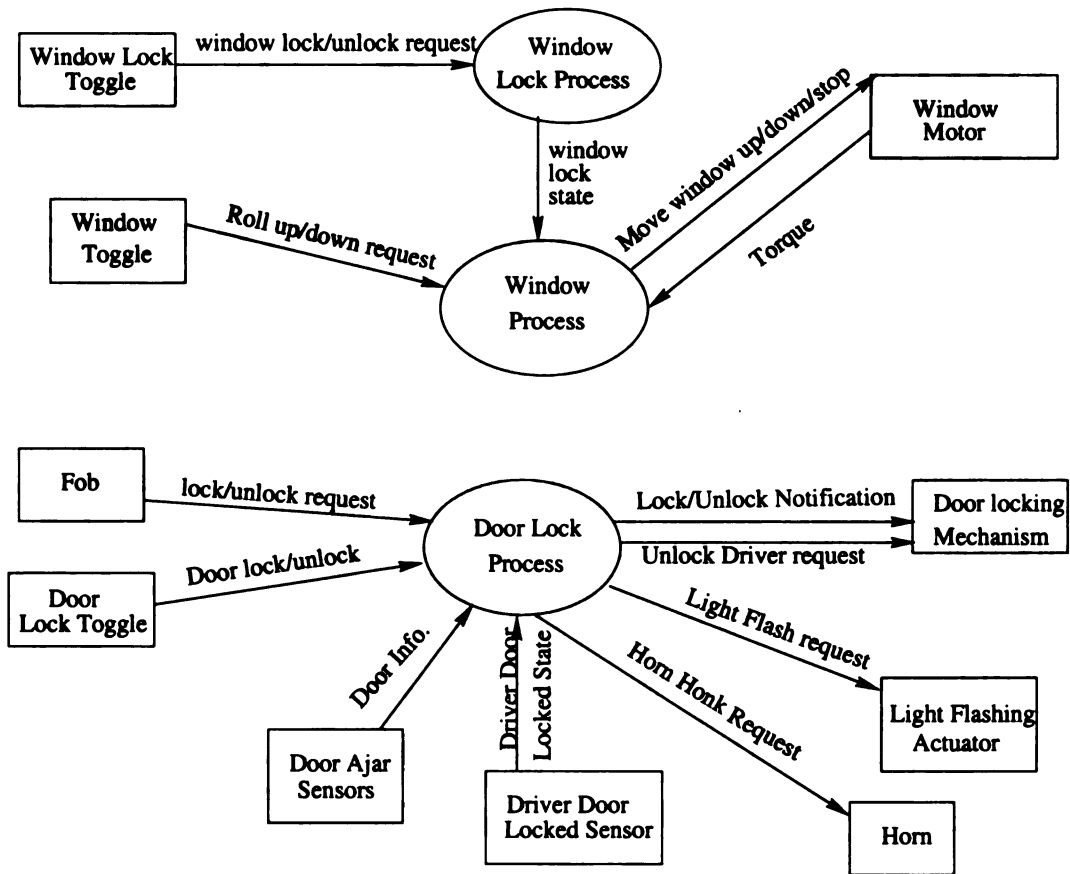


Figure A.7: Functional Model refined by controller objects

---

# Appendix B

## Automotive Door System: VHDL Entity/Architectures

Contained in this appendix are all the VHDL entity and architecture modules needed for a four-door automotive door system. These VHDL modules were generated from the OMT models in Appendix A using the translation process presented in this thesis. Figure B.1 shows the various entities and the related architectures for this system. The Door System has a structural architecture made of components consisting of the three sub-systems; The Window System, the Window Lock System, and the Door Lock System.

The entities for the Door System, the Window System, the Window Lock System and the Door Lock System are presented first. The entities are followed by the structural architecture for the whole door system. Then comes the behavioral architectures for the three subsystem.

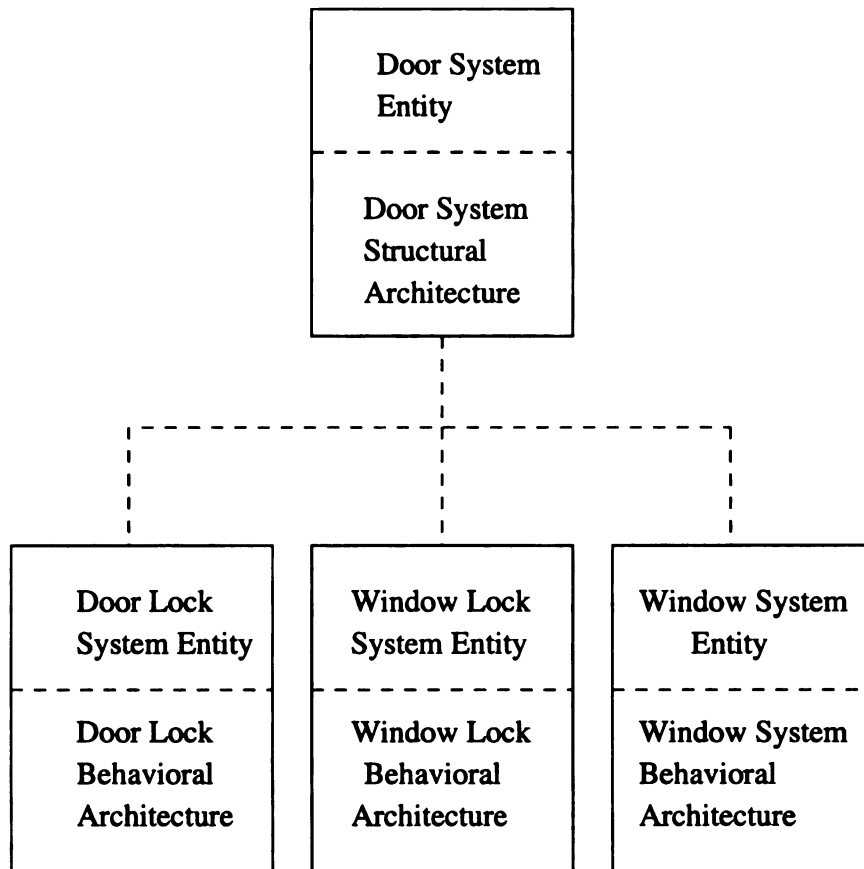


Figure B.1: A view of the entities and architectures in the Automotive Door System.

---

```

-----
-- A Four Door System is being built.
-- Names supplied are Driver, Pass, BkRt, BkLft
-- Multiple objects are WindowSystem (4), WindowMotor (4),
    -- WindowUp (4), WindowDown (4), WindowStop (4),
    -- DoorLock (2 for Driver and Pass),
    -- DoorUnLock (2 for Driver and Pass).
-----

```

```

-----
-- The DoorSystem is the main entity for the whole door system.
-- It contains ports for all the data flowing into and out of
-- the system. This includes all the signals and data from any
-- sensors and control flow and data to any actuators.
-----

```

```

ENTITY DoorSystem IS -- The main entity for the whole system

```

```

PORT

```

```

( -----
  -- In Ports -- -- Ports for the dataflow coming from sensors
  -----
  -- WindowToggle signals (X4)
  DriverWindowToggleUp : IN BIT;
  PassWindowToggleUp : IN BIT;
  BkRtWindowToggleUp : IN BIT;
  BkLftWindowToggleUp : IN BIT;
  DriverWindowToggleDown : IN BIT;
  PassWindowToggleDown : IN BIT;
  BkRtWindowToggleDown : IN BIT;
  BkLftWindowToggleDown : IN BIT;
  -- Window lock signals from driver door panel
  WindowLock : IN BIT;
  WindowUnlock : IN BIT;
  -- Fob button signals from fob unit
  Foblock : IN BIT;
  Fobunlock : IN BIT;
  -- Door locks (X2)
  DriverDoorLock : IN BIT;
  PassDoorLock : IN BIT;
  DriverDoorUnLock : IN BIT;
  PassDoorUnLock : IN BIT;
  -- Door Ajar sensor input
  Doorajar : IN BOOLEAN;
  -- Driver Door Locked sensor input

```



```

DriverDoorLocked : IN BOOLEAN;
-- Window Motors' torque signal (X4)
DriverMotorTorque : IN BIT;
PassMotorTorque : IN BIT;
BkRtMotorTorque : IN BIT;
BkLftMotorTorque : IN BIT;
-----
-- Out Ports -- -- Ports for the dataflows going to actuators
-----
-- Window Motors's signals (X4)
DriverWindowMotorUp : OUT BIT;
PassWindowMotorUp : OUT BIT;
BkRtWindowMotorUp : OUT BIT;
BkLftWindowMotorUp : OUT BIT;
DriverWindowMotorDown : OUT BIT;
PassWindowMotorDown : OUT BIT;
BkRtWindowMotorDown : OUT BIT;
BkLftWindowMotorDown : OUT BIT;
DriverWindowMotorStop : OUT BIT;
PassWindowMotorStop : OUT BIT;
BkRtWindowMotorStop : OUT BIT;
BkLftWindowMotorStop : OUT BIT;
--Door Locking Mechanisms signals
DoorLocking : OUT BIT;
DoorUnLocking : OUT BIT;
DoorUnLockingDriver : OUT BIT;
-- Light Flashing mechanism's signal
LightFlash : OUT BIT;
-- Horn Honking signal
HornHonk : OUT BIT
);
END DoorSystem;          -- end of the main door system
-----

```

```
-----  
-- WindowSystem is a controller  
-- for one particular window, that senses  
-- roll up and roll down request as well as  
-- if the motor is torquing. The WindowSystem  
-- decides if the window motor should be moved  
-- up or down or stopped.  
-----  
ENTITY WindowSystem IS    -- Entity for window system controller  
  
    port  
    ( -----  
      --in ports -- --dataflows into window system from some sensors  
      -----                               for one window.  
      WWindowLockState: IN Boolean; --T locked and F for unlocked.  
      WWindowUp: IN BIT;  
      WWindowDown: IN BIT;  
      WSMotorTorque: IN BIT;  
      -- out ports --    --ports to this window's actuators  
      WWindowMotorUp: OUT BIT;  
      WWindowMotorDown: OUT BIT;  
      WWindowMotorStop: OUT BIT  
    );  
END WindowSystem;  
-----
```

```
-----  
-- WindowLock System is a controller  
-- that senses lock and unlock requests for  
-- the driver or the passenger and sets an  
-- internal global Window lock state variable to True or False.  
-----
```

```
ENTITY WindowLockSystem IS
```

```
  port
```

```
    (WLSlockReq : IN BIT;
```

```
     WLSUnlockReq : IN BIT;
```

```
     WLSWindowLockState: OUT BOOLEAN --T locked and F for unlocked.
```

```
    );
```

```
END WindowLockSystem;
```

```
-----
```

```
-----  
-- DoorLockSystem is an entity for a controller that receives  
-- input from the driver door panel, the passenger door panel,  
-- and the keyless fob unit.  
-- Output from this controller goes to three actuators that  
-- 1. lock all doors, 2. unlock all doors, 3. unlock driver door.  
-----
```

ENTITY DoorLockSystem IS

port

```
( --in ports --  
  DLFobLock : IN BIT;  
  DLFobUNLock : IN BIT;  
  -- Door Lock (X2)  
  DLDriverDoorLock: IN BIT;  
  DLPassDoorLock: IN BIT;  
  DLDriverDoorUnLock: IN BIT; --added during testing  
  DLPassDoorUnLock: IN BIT;   --added during testing  
  DLDoor_ajar : IN BOOLEAN;  
  DLDriverDoorLocked : IN BOOLEAN;  
  --out ports --  
  DLDoorLockingLock : OUT BIT;  
  DLDoorLockingUnLock : OUT BIT;  
  DLDoorLockingDriverUnLock : OUT BIT;  
  DLHornHonking : OUT BIT;  
  DLLightFlashing : OUT BIT  
);  
END DoorLockSystem;
```

```
-----
```

```

-----
-- This is a structural architecture module for the Door_System
-- It is made up of components consisting of four WindowSystems,
-- a DoorLockSystem, and a WindowLockSystem.
--   Components are declared, then specific components
-- are instantiated.  The port map defines what signals from the
-- main door_system are mapped to or become the signals inside
-- of the components.
-- WindowLockStateGlobal is one internal global signal that
-- was declared to hold the state of the window lock system.
-----

```

ARCHITECTURE structural of Door\_System IS

COMPONENT WindowSystem

PORT

( --in ports --

WSWindowLockState : IN Boolean; --T locked and F for unlocked.

WSWindowUp : IN BIT;

WSWindowDown : IN BIT;

WSMotorTorque : IN BIT;

-- out ports --

WSWindowMotorUp : OUT BIT;

WSWindowMotorDown : OUT BIT;

WSWindowMotorStop : OUT BIT;

);

END COMPONENT:

COMPONENT WindowLockSystem

PORT

(WLSlockReq : IN BIT;

WLSunlockReq : IN BIT;

WLSWindowLockState: OUT BOOLEAN --T locked and F for unlocked.

);

END COMPONENT:

COMPONENT DoorLockSystem

PORT

( --in ports --

DLFobLock : IN BIT;

DLFobUNLock : IN BIT;

-- Door Lock (X2)

DLDriverDoorLock: IN BIT:

```

DLPassDoorLock: IN BIT;
DLDoor_ajar : IN BOOLEAN;
DLDriverDoorLocked : IN BOOLEAN;
  --out ports --
DLDoorLocking : OUT BIT;
DLDoorUnLocking : OUT BIT;
DLDoorUnLockingDriver : OUT BIT;
DLHonkHonking : OUT BIT;
DLLightFlashing : OUT BIT
);
END COMPONENT;

-- internal signal needed as seen on refined functional model
SIGNAL WindowLockStateGlobal : Boolean;
BEGIN
-- Window system instances -(X4)--names supplied by developer
DriverWindowSystem : WindowSystem
PORT MAP (  WWindowLockState => windowlockstateglobal,
  WWindowUp  => DriverWindowToggleUp,
  WWindowDown => DriverWindowToggleDown,
  WSMotorTorque => DriverMotorTorque,
  -- out ports --
  WWindowMotorUp  => DriverWindowMotorUp,
  WWindowMotorDown => DriverWindowMotorDown,
  WWindowMotorStop => DriverWindowMotorStop
);

PassWindowSystem : WindowSystem
PORT MAP (  WWindowLockState => windowlockstateglobal,
  WWindowUp  => PassWindowToggleUp,
  WWindowDown => PassWindowToggleDown,
  WSMotorTorque => PassMotorTorque,
  -- out ports --
  WWindowMotorUp  => PassWindowMotorUp,
  WWindowMotorDown => PassWindowMotorDown,
  WWindowMotorStop => PassWindowMotorStop
);

BkRtWindowSystem : WindowSystem
PORT MAP (  WWindowLockState => windowlockstateglobal,
  WWindowUp  => BkRtWindowToggleUp,
  WWindowDown => BkRtWindowToggleDown,
  WSMotorTorque => BkRtMotorTorque,
  -- out ports --

```

```

WSWindowMotorUp => BkRtWindowMotorUp,
WSWindowMotorDown => BkRtWindowMotorDown,
WSWindowMotorStop => BkRtWindowMotorStop
);

```

```

BkLftWindowSystem : WindowSystem
PORT MAP ( WSWindowLockState => windowlockstateglobal,
  WSWindowUp => BkLftWindowToggleUp,
  WSWindowDown => BkLftWindowToggleDown,
  WSMotorTorque => BkLftMotorTorque,
  -- out ports --
  WSWindowMotorUp => BkLftWindowMotorUp,
  WSWindowMotorDown => BkLftWindowMotorDown,
  WSWindowMotorStop => BkLftWindowMotorStop
);

```

```

LockSys: DoorLockSystem
PORT MAP ( DLFobLock => Foblock,
  DLFobUNLock => Fobunlock,
  -- Door Lock (X2)
  DLDriverDoorLock=> DriverDoorLock,
  DLPassDoorLock=> PassDoorLock,
  DLDoor_ajar => Doorajar,
  DLDriverDoorLocked => DriverDoorLocked,
  --out ports --
  DLDoorLocking => DoorLocking,
  DLDoorUnLocking => DoorUnLocking,
  DLDoorUnLockingDriver => DoorUnLockingDriver,
  DLHonkHonking => HornHonk,
  DLLightFlashing => LightFlash
);

```

```

WindowSys : WindowLockSystem
PORT MAP (WLSlockReq => WindowLock,
  WLSUnlockReq => WindowUnLock,
  WLSWindowLockState => windowlockstateglobal
);

```

```

END structural; -- end structural architecture of Window System
-----

```

```

-----
-- The behavior architecture for the WindowSystem entity is
-- presented in this section.
--
-- Input Events: WSWindowDown, WSWindowUp,WSMotorTorque
-- Outputs: WSWindowMotorUp, WSWindowMotorStop, WSWindowMotorDown
-----

```

```

ARCHITECTURE behavioral OF WindowSystem IS
TYPE State_type IS (Idlestate, WindowMovingUpState,
                    WindowMovingDownState);
SIGNAL PresentState : State_type := Idlestate;
SIGNAL State : State_type := Idlestate;
BEGIN
  StateChanging : PROCESS
  BEGIN
    WAIT ON WSWindowDown, WSWindowUp,WSMotorTorque;
    PresentState <= State;
    CASE PresentState IS
      WHEN Idlestate =>
        If ((WSWindowUP = '1' )and (WSWindowLockState = FALSE))
          THEN
            WSWindowMotorStop <= '0';
            WSWindowMotorUp <= '1';
            State <= WindowMovingUpState;
          ELSE IF ((WSWindowDown='1')AND(WSWindowLockState=False))
            THEN
              WSWindowMotorStop <= '0';
              WSWindowMotorDown <= '1';
              State <= WindowMovingDownState;
            END IF;
          END IF;
      WHEN WindowMovingUpState =>
        IF ((WSMotorTorque = '1') or (WSWindowUp = '0'))
          THEN
            WSWindowMotorUp <= '0';
            WSWindowMotorStop <= '1';
            State <= Idlestate;
          END IF;
      WHEN WindowMovingDownState =>
        IF ((WSMotorTorque = '1') or (WSWindowDown= '0'))
          THEN
            WSWindowMotorDown <= '0';
            WSWindowMotorStop <= '1';
            State <= Idlestate;
          END IF;
    END CASE;
  END PROCESS;
END;

```



```
    END IF;  
  END CASE;  
END PROCESS;  
END behavioral;
```

---

```

-----
-- The behavior architecture for the WindowLockSystem entity is
-- presented in this section.
--
-- Inputs:  WLSLockReq:BIT - mapped to LockReg. A '1' causes window
--          lock, if WLSWindowLockState is False (not locked)
--          WLSUnlockReq:Bit - mapped to UnlockReq.
-- A '1' causes windows to unlock,
--          if WLSWindowLockState is False (not locked)
-- Output:  WLSWindowLockState:boolean - True if windows are locked
--          - False if windows are not locked.
-----

```

ARCHITECTURE behavioral of WindowLockSystem IS

```

TYPE state_type IS (LockedState, UnlockedState);
SIGNAL State : state_type := UnlockedState;
SIGNAL PresentState : state_type := UnlockedState;

```

BEGIN

```

StateChanging : PROCESS
BEGIN
WAIT until (WLSLockReq = '1' OR WLSUnlockReq = '1');
PresentState <= State;
CASE PresentState IS
  WHEN LockedState =>
    IF (WLSUnlockReq = '1') THEN      -- enter unlock state
      WLSWindowLockState <= False; -- entry action
      State <= UnlockedState;
    END IF;
  WHEN UnlockedState =>
    IF (WLSLockReq = '1') THEN        --enter lock state
      WLSWindowLockState <= True;    -- entry action
      State <= LockedState;
    END IF;
END CASE;
END PROCESS;
END behavioral;
-----

```

```
-----
-- The behavior architecture for the DoorLockSystem entity is
-- presented in this section.
```

```
--
-- Input Events: DLFobUnlock, DLFobLock, DLDriverDoorLock,
--                                     DLPassDoorLock
-- Outputs: DLDoorLockingLock, DLHornHonking, DLLightFlashing
--          DLDoorLockingDriverUnLock, DLDoorLockingUnLock
-----
```

ARCHITECTURE behavioral OF DoorLockSystem IS

```

TYPE state_type IS (IdleState, UnlockDriverState,
                    UnlockAllState, LockAllState, NotifyDriverState);
SIGNAL State : state_type := IdleState;
SIGNAL PresentState : state_type := IdleState;
SIGNAL Done_Event : BIT;
BEGIN
  StateChanging : PROCESS
  BEGIN
    WAIT ON Done_Event;
    WAIT until (DLFobUnlock = '1' or DLFobLock = '1' or
                DLDriverDoorLock = '1' OR DLPassDoorLock = '1' );
    presentstate <= State;
    CASE presentState IS
      WHEN IdleState =>
        IF ((DLFobUnlock = '1') and (DLDriverDoorLocked = True))
        THEN
          DLDoorLockingLock <= '0';
          DLDoorLockingDriverUnLock <= '1';
          State <= UnlockDriverState;
          IF (Done_Event = '0') THEN --toggle Done_event
            Done_Event <= '1'; --needed for completion event
          Else
            IF (Done_Event = '1') THEN
              Done_Event <= '0';
            END IF;
          END IF;
        ELSE
          IF (((DLFobUnlock = '1') and (DLDriverDoorLocked = False))
              or (DLDriverDoorUnLock = '1') or (DLPassDoorUnLock = '1'))
          THEN
            DLDoorLockingLock <= '0';
            DLDoorLockingUnLock <= '1';
            State <= UnlockAllState;
          END IF;
        END IF;
    END CASE;
  END PROCESS;

```



```
Else
  IF (Done_Event = '1') THEN
    Done_Event <= '0';
  END IF;
END IF;
State <= LockAllState;
WHEN UnlockAllState =>
  State <= IdleState;
WHEN UnlockDriverState =>
  State <= IdleState;
END CASE;
END PROCESS;
END behavioral;
```

# Bibliography

- [1] BetterState. <http://www.isi.com/Products/BetterState/oview.html>, 1997.
- [2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J.V. Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, 1989.
- [3] G. Boriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, University of California, Berkeley, 1988.
- [4] R.H. Bourdeau and B.H. Cheng. A formal semantics of object models. *IEEE Transactions on Software Engineering*, 21:799–821, October 1995.
- [5] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computing*, c-35 N.8, August 1986.
- [6] B.H. Cheng. Software Engineering - CPS470 classnotes. Technical report, Michigan State University, 1997.
- [7] B.H. Cheng, E. Wang, and R. Bourdeau. A graphical environment for formally developing object-oriented software. In *Proc. of IEEE 6th International Conference on Tools with Artificial Intelligence*, pages 26–32, November 1994.
- [8] M. J. Chung and S. Kim. Object-oriented VHDL design environment. In *1990 IEEE Design Automation Conference*, volume Paper 24.3, pages 431–435, 1990.

- [9] Debreil, Alain, Berthet, Christian, and Ahed Jerraya. Symbolic computation of hierarchical and interconnected FSMS. In *The First European Conference on VHDL*, 1990.
- [10] A. Dewey. *Analysis and Design of Digital Systems with VHDL*. International Thomson Publishing, Boston, 1997.
- [11] M. Durfresne, K. Khordoc, and E. Cerny. Using formalized timing diagrams in VHDL simulation. In J. Mermet, editor, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 33–42. Kluwer Academic Publishers, 1992.
- [12] D. Gajski, F. Vahid, and S. Narayan. SpecCharts: A VHDL front-end for embedded systems. Technical Report 93–31, UC Irvine, Department of ICS, 1993.
- [13] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1996.
- [14] D. Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, July 1987.
- [15] C. Le Faou and J. Mermet. Introducing CASCADE control graphs in VHDL. In J. Mermet, editor, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 245–256. Kluwer Academic Publishers, 1992.
- [16] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *Proc. of IEEE International Symposium on Requirements Engineering*, 1993.
- [17] Robyn R. Lutz. Targeting safety-related errors during software requirements analysis. In *SIGSOFT'93 Symposium on the Foundations of Software Engineering*, 1993.

- [18] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviors. In *25th Design Automation Conference*, 1988.
- [19] J. C. Madre, O. Coudert, M. Currat, A. Debriel, and C. Berthet. The formal verification chain at BULL. In *EuroASIC*, 1990.
- [20] J Mermet. Several steps towards a circuit integrated CAD system: CASCADE. In *Proc. of CHDL'85*, Tokyo, August 1985.
- [21] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, Inc., New York, 1993.
- [22] D. Perry. *VHDL*. McGraw-Hill, New York, 1991.
- [23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [24] VHDL. Language reference manual. Technical report, IEEE Standard 1076-1987, 1987.
- [25] Enoch Y. Wang and Betty H. C. Cheng. Formalizing and integrating the functional model into object-oriented design. In *Tenth International Conference on Software Engineering and Knowledge Engineering*, San Francisco Bay, USA, June 1998.
- [26] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *IEEE Proceedings of the 19th International Conference on Software Engineering*, pages 45–55, Boston, MA, May 1997. IEEE.



- [27] Y. E. Wang. *Integrating informal and formal approaches to object-oriented analysis and design*. PhD thesis, Michigan State University, 1998.
- [28] J. Warren, S. Wilkie, W. Johnson, and J. Ipson. System design document automotive door controls. *Software Engineering CPS470 Class Project*, 1997.
- [29] WorkView Office (TM). Version 7.31 by Viewlogic System, Inc., February 1997.

MICHIGAN STATE UNIV. LIBRARIES



31293016885240