THE'
9
2001

This is to certify that the

dissertation entitled

A Generic Framework for Formalizing
Object Oriented Modeling Notations for
Embedded Systems Development

presented by

William Eugene McUmber

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science and Engineering

Betty H. C. Cheng
Major professor

Date 8/21/00

A G
Mod

A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development

By

William Eugene McUmber

A Dissertation

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

Doctor of Philosophy

Department of Computer Science and Engineeering

August 8, 2000

A GEN

Embe

displays

develop

Applicat

graphica

eling La

provides

formaliz

checking

the den

homomo

formal l

in order

Abstract

A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development

By

William Eugene McUmber

Embedded systems are 10 to 100 times more numerous than traditional systems with displays and keyboards But even though the object-oriented paradigm has been used to develop many traditional systems, it has not been widely applied to embedded systems. Application of the object-oriented paradigm in industry is encouraged and reinforced by graphical methods such as the Object Modeling Technique (OMT) and the Universal Modeling Language (UML), but the semantics of the diagrams are not rigorous. This research provides formal semantics for UML models in order to build embedded systems. This formalization enables the automated analysis of diagrams, such as simulation and model checking, which are particularly critical for the development of embedded systems, given the demand for specifying temporal and concurrency properties. This research constructs homomorphisms between metamodels of the source semi-formal language and the target formal language, so that consistency of formalization rules can be mechanically established in order to provide precise semantics to the semi-formal notation.

To my w

througho

To my wife, Cheryl, and my sons, Robbie and Weston, who have supported me throughout while enduring my long hours and lack of availability

I have kn

family, an

excellent

would no

student a

It was

asked the

and I tri

I wish

homomo

formaliti

So ma

I wish to

Stirewal

Final

through

the sacri

ACKNOWLEDGMENTS

I have known Dr. Betty H. C. Cheng for quite some time because working, caring for family, and earning a Ph.D. all at once, takes a long time. Without her extreme patience, excellent advice, frequent encouragement, and expert editing capabilities, this dissertation would not have occurred. I'll bet she'll be careful about taking a "non-traditional" Ph.D. student again.

It was during my friend Enoch Wang's dissertation defense, that Dr. Anthony Wojcik asked the critical question that became central to this dissertation. It was a good question and I tried to build on Enoch's work to answer that question.

I wish also to thank Dr. Kurt Stirewalt for our many discussions about many things from homomorphic mappings, to semantics. Dr. Stirewalt was extremely helpful formulating the formalities and meanings of the homomorphisms.

So many others provided help along the way it is impossible to list them all in this space. I wish to thank my committee members: Dr. Betty H. C. Cheng (chairperson), Dr. Kurt Stirewalt, Dr. Anthony Wojcik, and Dr. Jacob Plotkin.

Finally, my wife, Cheryl, and my sons Rob and Weston, shared, and at times suffered, through this long process. I couldn't have done it without them and I deeply appreciate the sacrifices they made because I didn't have time for this or that. I do now.

LIST OF

Table of Contents

List of Figures

Cha

Intr

Embedde
parts [1].
software
bedded s
situation
embedde
they con
in a criti
and rigo
Currentl
embedde
level, pr
language
ity and r
informal

Chapter 1

Introduction

Embedded systems are typically 10 to 100 times more common than their desktop counterparts [1], residing in systems ranging from engine systems, to toasters, to autopilots. The software for embedded systems is, in general, more difficult to write and debug because embedded systems software usually involves time-dependent sections in difficult to instrument situations. For example, embedded systems rarely have a keyboard or display. Nonetheless, embedded systems usually must achieve a higher level of robustness and reliability because they control real-world physical processes or devices upon which we depend, frequently, in a critical way. Consequently, methods for developing and modeling embedded systems and rigorously verifying behavior before committing to code, are increasingly important. Currently, much of the embedded systems industry use *ad hoc* approaches for developing embedded systems [2]. Frequently, there are few, if any, intermediate steps between high-level, prose descriptions of requirements and code written in the target implementation language, such as C. With automotive and appliance applications, requirements traceability and reuse are two important tasks, but they are difficult to accomplish when using only informal development techniques.

Object-

of embedde

inefficient

between re

As a resul

through th

A num

recently, i

object-ori

and gain

addition,

how obje

At prese

individua

diagrams

UML mo

testing.

In or

languag

research

use wide

as the t

enable a

While U

Object-oriented development methods have not been widely used for the development of embedded systems, perhaps because of the perception that object-orientation produces inefficient code. On the other hand, especially in an embedded system, the correlation between real components in the physical system and software objects is often very good. As a result, our premise is that the benefits in software quality gained in other domains through the use of object-orientation should also apply to embedded systems.

A number of object-oriented techniques and notations have been introduced [3, 4, 5], but recently, it appears that the *Unified Modeling Language* (UML) [6] could be an approach to object-oriented modeling that is broad enough in scope to represent a variety of domains and gain widespread use. Currently, UML comprises several different notations [6, 7, 8]. In addition, there appears to be interest from the embedded systems community in exploring how object-oriented modeling, specifically UML, can be used for embedded systems [1]. At present, however, UML is only a notation, with no formal semantics attached to the individual diagrams, nor is there a formally defined semantics for the integration of the diagrams. Therefore, it is not possible to apply rigorous automated analysis or to execute a UML model in order to test its behavior, short of writing code and performing exhaustive testing.

In order to address this problem, we have developed a framework for deriving formal language specifications from a subset of the UML models. One overarching goal of this research, motivated by technology transfer objectives, is to enable developers to continue to use widely accepted development techniques, both in terms of the modeling language as well as the target specification language. We have developed a set of formalization rules that enable automated techniques to generate specifications from the individual UML notations. While UML offers several different notations, our preliminary investigations indicate that for modeling re

modeling e

formalizati

of embedd

metamodel

written in

modeling l

semi-forma

enable us

1.1 P

A simple,

tem's abst

Intuitive

semi-form

certain co

tails are e

point, th

long befo

Three

itive, gra

precisely

a general

target fo

modeling requirements and high-level design, the class and state diagrams are sufficient for modeling embedded systems. This dissertation presents results from investigations into the formalization of the class and state diagrams for capturing the structure and the behavior of embedded systems, respectively. The formalizations are based on mappings between *metamodels* of the modeling notation. A metamodel is a model of the notation itself, written in the class-structure notation where "classes" represent syntactic components of the modeling language. Homomorphic mappings are established between the metamodels of the semi-formal (source) and the formal (target) languages. These homomorphic relationships enable us to rigorously establish the consistency of the formalization rules.

1.1 Problem Description

A simple, intuitive notation for describing a system that also accurately captures the system's abstract behavior has long been a goal in the Software Engineering Community [9, 10]. Intuitive notations such as those included in UML are gaining widespread use, where their semi-formality is probably one of the reasons for their popularity. Abstraction requires that certain components and concepts not be specified too tightly because the appropriate details are either not known or because the detail is "implementation specific". But, at some point, the specific behavior of the system being designed must be ascertained, hopefully long before the system is actually implemented in code.

Three major objectives guide this research. First, we wanted to enable the use of intuitive, graphical notations for the design of embedded systems while providing a means of precisely specifying the semantics of the written diagrams. Second, we wanted to develop a general framework for creating mappings from diagram components to elements of the target formal specification language. The general framework provides the rigor to ensure consiste

work ena

exercise

specifica

can be u

their res

1.2 I

As a me

industria

Because

language

The

standard

executed

straightf

behavior

and thus

precise ti

in many

In ord

of the co

describe n

execution

consistency between the diagrams and formal specification. In addition, the general framework enables us to alter the semantics of the diagram. The third objective was to enable and exercise complementary analysis techniques for the semi-formal diagram via their formal specifications. Specifically, we wanted to investigate how model checking and simulation can be used independently and in an integrated fashion to analyze the UML models via their respective specifications.

1.2 Research Program

As a means to facilitate technology transfer for our results, we use notations already in industrial use, both in terms of source semi-formal language and target formal languages. Because UML is the *de facto* standard for object-oriented modeling, we use it as the source language.

The initial formal target language we chose is VHDL [11, 12, 13]. VHDL is an IEEE standard [14] and is commonly used in industry. Since VHDL specifications can be directly executed, they provide precise semantics. In addition, simulation of a VHDL model is straightforward, therefore, a system mapped to VHDL allows rapid examination of system behavior without directly writing code. VHDL was initially designed to model hardware, and thus has a rich set of constructs for specifying timing properties. The ability to model precise timing constraints in an embedded systems model is very useful, and is not present in many other formal languages.

In order to explore complementary analysis techniques, we also formalized UML in terms of the commonly used language Promela/SPIN [15, 16]. Promela is the language used to describe models that are analyzed with SPIN. SPIN analyses also include simulation through execution but, in addition, state reachability and *model checking* analyses can be performed.

Model che

a system t

Forma

20, 21, 22

framewor

have disti

examined

"incorrect

quently, t

language i

We int

case) thro

which for

mapping.

constructs

between s

the semi-f

1.3 T

This resea

Object Or

for bindin

Thesis St

Model checking provides the analyst with a set of tools for checking temporal properties of a system through exhaustive state exploration.

Formal semantics have been added to graphical OO languages previously [5, 17, 18, 19, 20, 21, 22, 23], but the mapping from semi-formal to formal was not based on a general framework. There is, of course, no one "correct" mapping of semantics; but some mappings have distinct advantages over others in providing perspectives on system properties to be examined or analysis to be performed. On the other hand, it is possible to describe an "incorrect" mapping that produces inconsistent, or even contradictory, semantics. Consequently, the means used to achieve a mapping from a semi-formal language to a formal language is relevant and important.

We introduce formal semantics to a semi-formal language (having chosen UML in this case) through a *metamodel mapping* that dictates which semi-formal constructs map to which formal target language constructs. We use a homomorphic mapping to achieve this mapping, thus the homomorphic mapping provides a binding of semi-formal source language constructs to the formal language. Construction of the homomorphism (that is, the mapping between semi-formal components and formal components) dictates the semantics given to the semi-formal language.

1.3 Thesis Statement

This research is intended to define a methodology for designing embedded systems using Object Oriented (OO) notation. The approach relies on a flexible formalization framework for binding semantics to semi-formal graphical OO models.

Thesis Statement *Constructing a homomorphic mapping between metamodels of an* *o*

n

g

b

o

b

a

1.4

The ren

ground

tempora

semi-for

morphis

notation

a rigoro

designin

tailed ru

example

formalizi

ing. In (

Hydra th

generate

study of t

object-oriented modeling notation and a formal language provides the basis for precise mapping rules that enables automated generation of specifications in a formal language. Any one of several formal languages can provide the target, where the domain being modeled determines the requirements for the target language. The consistency of the mapping rules is established by the formalism of the homomorphic mapping between metamodels. Automatically generated formal specifications enable automated analysis of the diagrams, specifically model checking and simulation.

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2 we review background material regarding UML, VHDL, Promela/SPIN, model checking and the respective temporal logics they use. This chapter also discusses the relationships between informal, semi-formal, and formal modeling notations. Chapter 3 describes metamodels and homomorphisms between metamodels. Since metamodel notation is based on the class model notation, this chapter also discusses formalization of the class model in order to provide a rigorous basis for the homomorphic mapping. Chapter 4 presents our methodology for designing embedded systems using our formalization technique. Chapter 5 contains detailed rules for formalizing the UML class and dynamic models and is illustrated by a small example of a system designed using these rules. Chapter 6 contains the detailed rules for formalizing a second language, Promela, which can be used for simulation and model checking. In Chapter 7 we review related work. Chapter 8 describes a prototype tool called *Hydra* that has been developed to implement the detailed mapping rules and automatically generate VHDL and Promela specifications. Chapter 9 presents an extended industrial case study of the design of an embedded system. This chapter also contains a detailed account of model che

presents c

model checking and simulation used together on an industrial system. Finally, Chapter 10 presents conclusions, summarizes contributions, and discusses future work.

Cha

Bac

This cha
overview
used thr
the form
checking
the simu
be check
checking
Since th
we provi
models a

2.1 U

The Unif
modeling

Chapter 2

Background

This chapter provides overviews on five topics central to the research. We first provide an overview of UML because it is the source semi-formal object-oriented modeling language used throughout our research. Next, the VHDL language is described because it is one of the formal languages to which UML is mapped. We then discuss Promela/SPIN and model checking for two reasons: first, model checking is an analysis technique complementary to the simulation capabilities provided by a language such as VHDL. Secondly, the model to be checked is first encoded into a formal language, such as Promela, from which the model checking procedure (SPIN) can determine the validity of assertions made about the model. Since the model checking language is formal, we use it as a target formal language. Finally, we provide an overview of previous work on integrating formal, semi-formal and informal models and more precisely define the meanings of these three terms.

2.1 UML Overview

The Unified Modeling Language (UML) [7, 8] is described as a "general-purpose visual modeling language that is designed to specify, visualize, construct, and document the artifacts of a

structure

system. U

Object M

namic m

model to

UML

Dynamic

Compone

we deal v

Class D

cla

ann

cla

ass

sta

ma

agg

dia

tria

rela

end

part

coll

facts of a software system" [24]. UML is based on a series of diagrams that depict the class structure, dynamic properties, and event sequencing for an object-oriented (OO) software system. UML is an extension and melding of several modeling languages, most notably the Object Modeling Technique [3] (OMT) and StateCharts [25]. UML class diagrams and dynamic model diagrams use notation similar to OMT. Unlike OMT, UML has no functional model to depict data flow.

UML contains seven distinct types of diagrams: Class Diagrams, Use Case Diagrams, Dynamic Models (state diagrams), Interaction Diagrams (sequence and activity diagrams), Component Diagrams, Collaboration Diagrams, and Deployment Diagrams. In this thesis we deal with the class, dynamic, use case, and interaction diagrams.

Class Diagram: The class diagram depicts the classes and the interrelations between the classes. Classes are drawn as rectangles with relationships between classes drawn as annotated lines between the rectangles. There are four types of relationships between classes. *Association* is a binary relationship between two classes. Multiplicities on associations are written as a number at each end, with the number applying to instances at that end of the line. An optional instance is denoted by "0..1", "*" indicates many, and "1..*" denotes one or many. Three additional relationships are *subtype, aggregation*, and *composition*, which are drawn as a small hollow triangle, an empty diamond, and a filled diamond, respectively. Subtyping is denoted by a small hollow triangle on the superclass end of the association. The derived class end of the subtype relationship is not marked. Filled and empty diamonds are placed on the aggregate end of composition and aggregation relationships, while the classes that constitute the parts of the aggregation are not marked. Both aggregation and composition indicate collections where the part plays a role in the behavior of the whole but composition is the s

of th

Dynamic

scrib

are

arro

Tran

Figu

a su

ing

con

one

stat

Use Cas

req

An

con

dia

wit

can

the

use

use

the stronger relationship because the existence of the part depends on the existence of the whole. In aggregation, this dependency does not hold.

Dynamic Model: The dynamic model is based on Statechart [25] conventions and describes dynamic behavior of objects through the use of Statechart-like notation. States are drawn as boxes with rounded corners with transitions between states drawn as arrows between the boxes, where the arrow indicates the direction of the transition. Transitions are labeled with the transition event, which has its own syntax as shown in Figure 2.1. Composite states are drawn as a large rounded corner rectangle containing a sub-state diagram. This construction may continue recursively to an arbitrary nesting level. Concurrency of composite state machines is indicated by partitioning the composite states with dotted lines. Each rectangular component bordered on at least one side with a dotted line executes concurrently with the other denoted composite states.

Use Case: The use case diagram documents *Use Cases* and is used to capture system requirements. A use case contains *actors*, the system, and the use cases themselves. An actor is an idealization of an external process, a person, a system, or some external component that interacts with the system. The system is represented in the use case diagram as a large rectangle containing oval use cases. Each use case is connected with lines to labeled actors on the outside of the system rectangle. Each use case can be instantiated as a *scenario*, often presented as a sequence diagram, which gives the specific actions for actors and the system in a particular situation. Use cases are useful for capturing high level requirements and the goals of the system. Increasingly, use cases are providing information useful for validating the system during system analy

Interactio

withi

ally s

of sy

agru

acros

and

diag

deep

prov

The s

els [26].

of the ty

as an ins

is a met

analysis.

Interaction Diagrams: Interaction diagrams depict the interaction between objects within the system, and with actors outside of the system. Interaction diagrams visually show objects and the flow of messages between objects for a particular scenario of system operation. The interaction diagram we use in this work is the *Sequence Diagram.* Figure 2.2 shows a sequence diagram listing objects **car**, **radar**, and **control** across the top along with one external actor. The vertical lines depict time downward and the horizontal lines show message interaction between the objects. Interaction diagrams are useful for showing the operation of the system at a level higher than the deep view provided by dynamic models. As mentioned above, interaction diagrams provide snapshots of system behavior that can be verified with analysis tools.

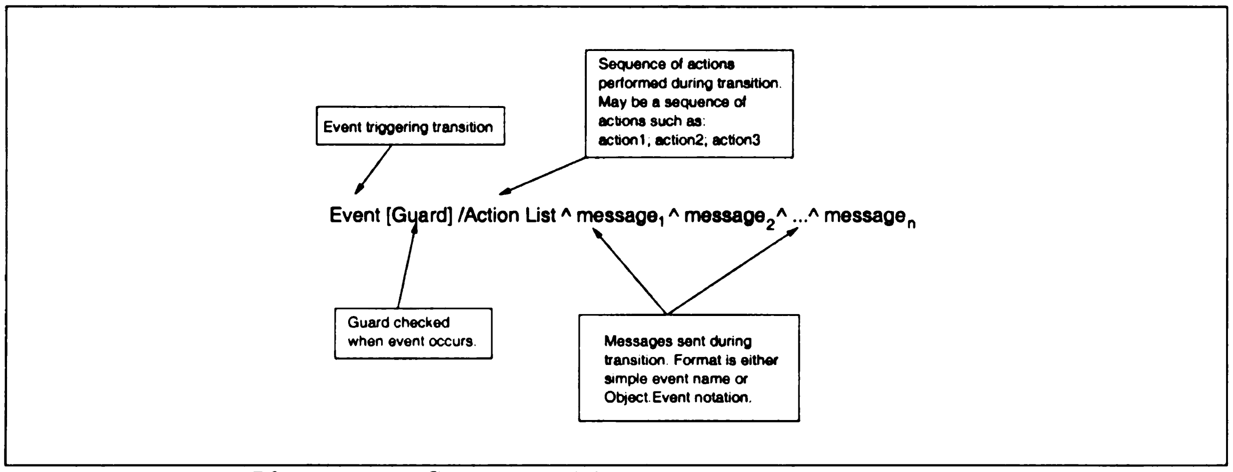
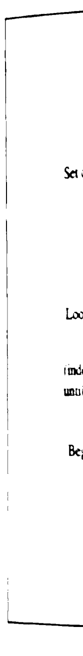

Sequence of actions performed during transition. May be a sequence of actions such as: action1; action2; action3
Event triggering transition
Event [Guard] /Action List ^ message1 ^ message2^ ...^ messagen
Guard checked when event occurs.
Messages sent during transition. Format is either simple event name or Object.Event notation.

Figure 2.1: State transition event syntax conventions.

The syntax of UML diagrams is described in a small subset of UML itself using *metamodels* [26]. A metamodel is a class diagram where the classes describe syntactic components of the type of diagram being described. An instance of a metamodel is a UML diagram just as an instance of a UML class diagram is a set of specific objects in a system. Figure 2.3 is a metamodel of the UML dynamic model. The metamodel describes the interrelation-



ship betw

Classes d

nents in

signifies

or **SubN**

pear in

SubMac

A dia

decorate

Both spe

stronger

of the co

composit

In Fi

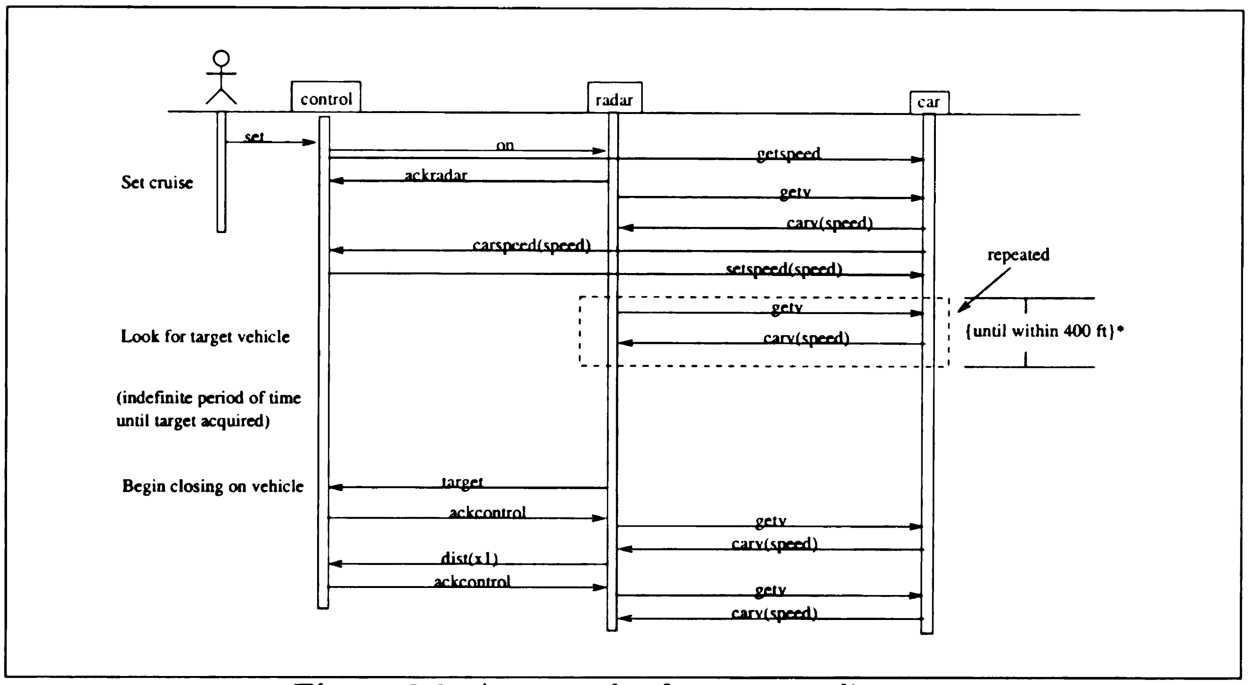


Figure 2.2: An example of a sequence diagram.

ship between states, transitions, events and the other parts of the UML dynamic model. Classes decorated with italics are abstract, and therefore are not realized by actual components in a UML dynamic model diagram. The empty arrow on class **State** in Figure 2.3 signifies a specialization association between **State** and **CompositeState**, **SimpleState**, or **SubMachineState**. Since **State** is abstract, components of type **State** will not appear in a dynamic model but components of types **CompositeState**, **SimpleState**, or **SubMachineState** can appear wherever a type **State** is required.

A diamond on an association end specifies a collection with the aggregate class on the decorated end. UML distinguishes two types of collections, *aggregation* and *composition.* Both specify that the part plays a role in the behavior of the collection but composition is a stronger relationship characterized by dependence of the existence of the part upon existence of the collection. Aggregation is signified in UML diagrams with a hollow diamond while composition is denoted with a filled diamond.

In Figure 2.3 type **State** (and its subtypes) contain compositions of **ActionSequence**

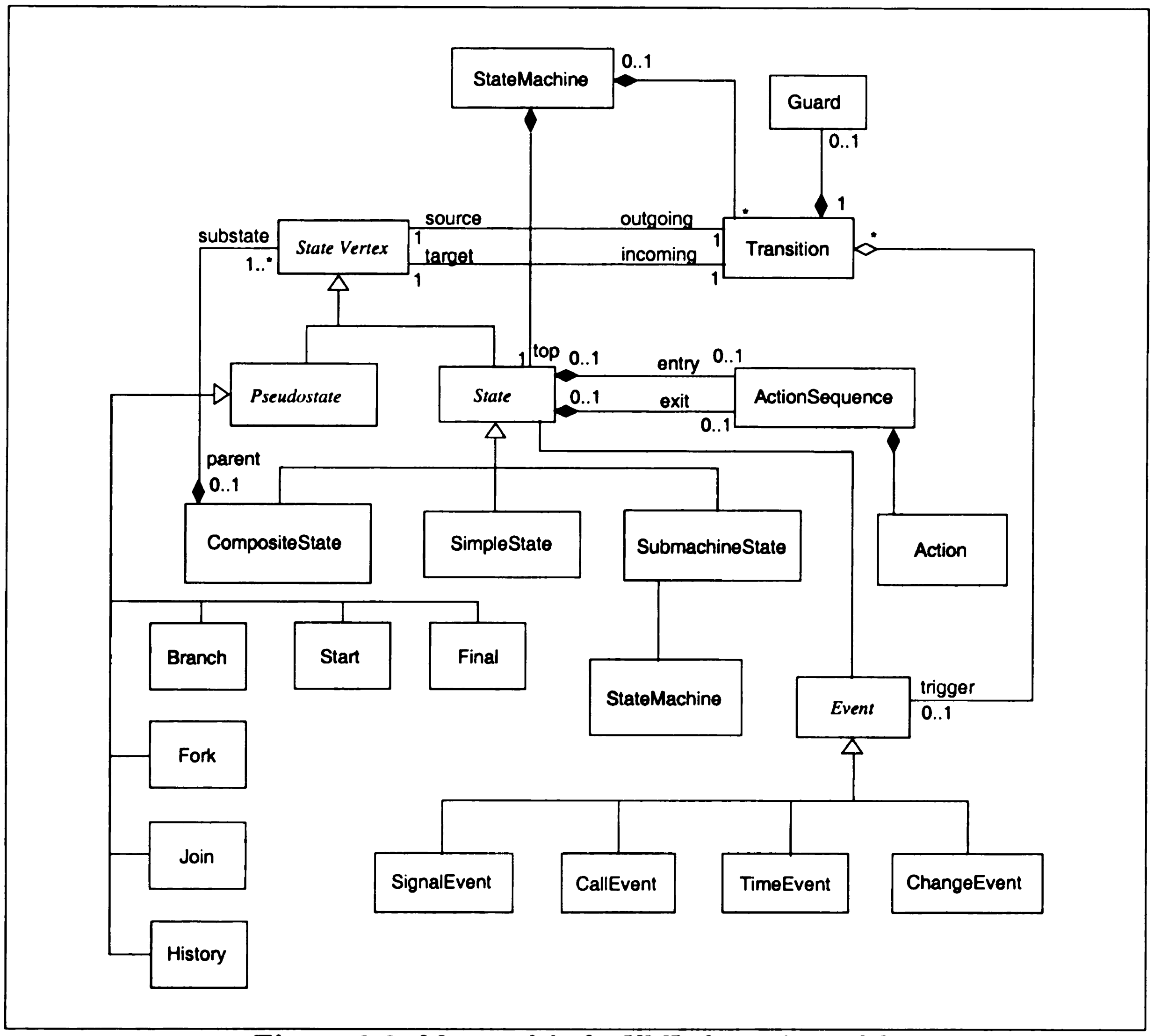


Figure 2.3: Metamodel of a UML dynamic model.

under tw

with sets.

In im

reference

contain t

necessari

Anot

part. Co

be memb

hidden re

example.

and **q** are

side effec

potentiall

Based

tion of **St**

chines.

each of w

StateVer

another in

under two different roles. Collection relationships are similar to membership relationships with sets, with composition closely mimicking set membership.

In implementations, simple aggregation might be the relationship where an object holds references to other (child) objects. In a composition relationship, the parent object would contain the child data type such that if the parent were destroyed, the child object would necessarily also be destroyed.

Another distinction can be made by the number of collections that can hold a given part. Composed parts can only be a member of one collection, where aggregated parts can be members of more than one collection. If this were not true for composition, very complex hidden relations would exist between collection classes by virtue of their aggregations. For example, suppose objects **p** and **q** contained **c** as a composition component (the types of **p** and **q** are not important). Now suppose object **p** is destroyed. This necessarily implies the side effect of destroying **c**, thus destroying the composition relation between **c** and **q** and potentially altering the behavior of **q**.

Based on the metamodel in Figure 2.3 a **StateMachine** is constrained to be an aggregation of **States**, which are realized as **CompositeStates**, **SimpleStates**, or **SubStateMachines**. A **CompositeState** is further specified to be an aggregation of **StateVertices**, each of which is either a **State** or a **PseudoState**, thus forming a recursive relationship. **StateVertex** has two required associations with transitions, one in an outgoing role and another in an incoming role.

2.2 VI

VHDL[1] is

adopted by

dard as ad

This resear

In form

Very High

language p

closer exam

communic

syntax, V

and comm

Althou

perhaps i

discrete t

series of e

events sch

are execu

transactio

[1]The na
Integrated

2.2 VHDL Overview

VHDL[1] is a language for describing digital electronic hardware systems and has been adopted by the IEEE as Standard 1076 [14]. Two versions are commonly available: the standard as adopted in 1987 (VHDL'87) and an updated version adopted in 1993 (VHDL'93). This research assumes the use of VHDL'93.

In form, VHDL resembles Ada, likely because VHDL arose out of the U.S. Government's Very High Speed Integrated Circuits (VHSIC) program. VHDL is a hardware description language primarily intended for description of hardware component behavior. However, on closer examination, it is evident that VHDL is a language for describing multiply concurrent, communicating processes similar to CSP [27] or LOTOS [28]. Instead of a process algebra syntax, VHDL uses an Ada-like syntax with *procedures* and *signals* to describe processes and communication channels.

Although VHDL is useful for describing the behavior and structure of hardware systems, perhaps its most useful aspect is the ability to execute a VHDL specification. VHDL discrete time simulators simulate the behavior of a model through time by scheduling a series of events based on the VHDL model statements. At each simulated time step, all events scheduled for that time are carried out (in zero simulated time). As statements are executed, more transactions are scheduled for future simulated times. When no more transactions are available, the simulation ends.

[1]The name *VHDL* is a double acronym with the **V** derived from VHSIC, which denotes Very High Speed Integrated Circuit, and **HDL** derived from Hardware Description Language.

2.2.1 S

As in Ada

interface.

descriptio

implement

a *port* des

the outsid

represent

```
- De
use
use
entit


end
```

Figure 2
portion o

The b

that refe

declaratio

ing beha

Player st

defines a

signals d

architect

2.2.1 Structure and Statements

As in Ada, most blocks of VHDL model statements have two parts: a declaration of the interface, and a separate description of behavior. The basic building block of a VHDL description is the **entity**. Figure 2.4 shows a VHDL entity declaration for a state machine implementing a simple CD Player. Entities begin with a declaration of the entity name and a *port* description. The port description declares the signals passed into the entity from the outside, analogous to formal parameters on a procedure. In hardware, ports essentially represent "pins" on a component, but in VHDL their semantics are much richer.

```
- Define the CD Player entity. Entity description is declared first
use std.textio.all;
use work.easyio.all;
entity cdplayer is
     port (e_on :  in boolean; - on button event.
          e_off :  boolean; - off button event.
          play :  in boolean; - play button
          stop :  boolean); - stop button
end cdplayer;
```

Figure 2.4: A VHDL **entity** specification for a simple state machine implementation of portion of a CD player

The behavior description of an entity is contained in a separate **architecture** section that references the entity declaration. The **architecture** section itself may also contain declarations for local variables and signals, and contains a set of VHDL statements describing behavior. Figure 2.5 contains an excerpt of the architecture description for the CD Player state machine (the initial states and one state are shown). The **process** statement defines an independent, concurrent process for the entity. Input to the process comes from signals declared in the **entity** statement and from other variables and signals within the **architecture**.

```
- Be
archi
- Stc


    :
    :
- int
    s
    s
    s
begin

- Int
    i
    b



    e
    s
    b





    e
```

Figure 2.

```
- Behavioral portion of the CD Player
architecture one of cdplayer is
- Standard state support header
    type states_r is (none_r, r1, r2);
    type states_c is (none_c, c1, c2, c3);
    signal state_r :  states_r;
    signal state_c :  states_c;
- internal events between substates
    signal i_on, i_off, i_play, i_stop :  boolean;
    signal instate_r :  states_r;
    signal instate_c :  states_c;
begin

- Initial process that starts the other processes.
    initial :  process
    begin
        state_r <= r1, none_r after 1 fs;
        state_c <= c1, none_c after 1 fs;
        wait;
    end process;
    state_r1 :  process
    begin
        wait until state_r = r1;
        say("in state r1");
        wait until e_on;
        i_on <= true, false after 1 fs; - internal event
        state_r <= r2, none_r after 1 fs;
    end process;
```

Figure 2.5: VHDL specification for a simple state machine implementing a CD Player.

Entit

instantia

instantia

with the

VHD

and Con

suggests.

zero sim

construc

in proce

statemen

CASE. a

As ment

the susp

A pr

current

process

ment blo

the **proc**

never te

"wait;"

Entities can incorporate other entities through an instantiation process. In order to instantiate an entity, a *port map* must be specified to describe how the signals in the instantiating entity are connected to signals in the instantiated entity. Communication with the instantiated entity is strictly through the instantiated entity's ports.

VHDL "executable" statements are divided into two groups: Sequential statements and Concurrent statements. Concurrent statements are executed in parallel, as the name suggests, and generally reference a signal in some way. Sequential statements execute in zero simulated time and are written in blocks inside procedure-like concurrent statement constructs. Sequential statements are analogous to common programming statements found in procedural languages and provide the major source of behavior specification. Sequential statements include all the statements commonly found in a procedural language such as IF, CASE, assignment statement, function definitions and calls, and a variety of loop constructs. As mentioned earlier, the **wait** statement is an important sequential statement that causes the suspension of the current section of the model until the specified event occurs.

A primary concurrent statement is the **process** statement. Each **process** defines a concurrent procedure with its own variables and sequential statements. The statements in a **process** block are contained in an implicit loop. At the logical conclusion of a **process** statement block, where a procedural language sub-program normally executes an implicit return, the **process** statement restarts from the beginning. Consequently, once started, **processes** never terminate, although a process can be made to suspend indefinitely by executing a "`wait;`" statement, waiting essentially for no event.

2.2.2

One of t
is the *si*
type. T
is callec
event is
occurs.
using th
value,
variable
state_c.

Sign
lation er
in the **w**
suspensi
Notice t
sstateme

Signa
incremen
the valu
ules a tr
only *cha*
two for s

2.2.2 Signals and Communications

One of the important features of VHDL that makes it useful for embedded systems modeling is the *signal.* Signals are VHDL distinguished variables that can assume any arbitrary data type. The signal mechanism works as follows: Each write to a variable declared as a signal is called a *transaction.* If the variable's value is changed because of the assignment, an *event* is scheduled for the simulated time when the event is to fire. When an event's time occurs, every reference to the signal, usually through a VHDL **wait** statement, is executed using the new signal value. A special kind of assignment statement of the form `signal <= value, [value...]` is used for signals to denote their fundamental difference from regular variables. Statements 17 and 18 in Figure 2.5 show assignments to the signals *state_r* and *state_c*, respectively.

Signals are the implicit or explicit objects of a VHDL **wait** statement. When the simulation encounters a **wait**, execution of the enclosed unit "suspends" until the event specified in the **wait** occurs. For example, the statement "`wait until state = my_state;`" causes suspension of the containing process until the signal `state` *next* assumes the value `my_state`. Notice that execution stops even if `state` already has the value `my_state` because the **wait** sstatement is waiting for an *event* to occur (the changing of signal `state`).

Signal assignments allow several values to be specified in one statement, each with an incremental time specification. For example, `sig_X <= alpha, beta after 1 ns;` assigns the value `alpha` to `sig_X` now (with appropriate transaction–event processing), and schedules a transaction to assign `beta` to `sig_X` one nanosecond in the simulated future. Since only *changes* produce events, this statement will produce at least one event, and perhaps two for signal `sig_X` depending on the current value of `sig_X`.

2.2.3 S

Often a s
presence
the resul
bus impl
computat
drivers fo
results in
dictates h
multiple c
function b
value of a
data type
signal by
the driver
and remo
`null aft`
femtosecon
event to b
complete s

2.2.4 D

VHDL's da
ing strings

2.2.3 Signal Busses – Multiple Signal Drivers

Often a signal is used to represent a hardware bus. The distinguishing feature of a bus is the presence of multiple drivers (multiple assignments to the signal). In hardware situations, the result of driving a bus with more than one source depends on the specifics of the bus implementation, and thus, the appropriate final signal value requires some sort of computation for resolution. In our UML to VHDL mapping, we frequently use multiple drivers for a signal in order to simulate messages and events, however our mapping rarely results in two *active* signal drivers at once. Nonetheless, good VHDL modeling practice dictates handling the multiple driver situation. The appropriate VHDL facility to resolve multiple drivers is called a *resolved signal.* In essence, a resolved signal inserts a user-defined function between signal assignment and transaction processing to calculate the desired final value of a signal. In practice, the user function is passed an array of values for the signal data type with each element of the array set to the current value being contributed to the signal by some driver statement. A special assignment value called **`null`**, "disconnects" the driver from the signal, effectively eliminating this driver's contribution to the signal and removing its value from the array. A statement of the form **`state <= new_state, null after 1 fs;`** assigns a new value to **`state`**, then disconnects from the signal after 1 femtosecond (the shortest possible simulated time increment). Such a construct causes an event to be scheduled for **`state`** and disconnects the driver from further contributions. The complete semantics of signals can be complex and is covered in [11].

2.2.4 Data Typing

VHDL's data typing model closely parallels that used in Ada. Arbitrary data types including strings, various forms of numbers, boolean (logical), and bit strings are all provided.

Arrays a

types ca

contains

defined f

2.3

SPIN [15

ing langu

Promela

tive stat

more det

The s

of *proces*

global ob

variables

2.3.1 S

Figure 2.

intended

and are n

Every

a conditio

blocks. If

(such as v

Arrays and enumerated types are a common tool in a VHDL model and new aggregate subtypes can be easily formed through typing and subtyping mechanisms. In addition, VHDL contains special data types for units, including time, so that new units of measure can be defined for specialized purposes.

2.3 Promela/SPIN Overview

SPIN [15, 16] is a tool for analyzing concurrent systems, specifically protocols. The modeling language for SPIN is called *Promela*. SPIN can either perform random simulations on Promela models or generate a C program to perform a variety of analyses based on exhaustive state explorations. Sections 2.4 and 9.7 describe SPIN's model checking capabilities in more detail.

The syntax of Promela is loosely based on the language C. Promela programs consist of *processes*, *channels*, and *variables*. Processes, which are called called *proctypes*, are global objects that execute asynchronously and can be created dynamically. Channels and variables may either be local or global.

2.3.1 Structure and Statements

Figure 2.6 shows an example of a model written in Promela. The specification is only intended to illustrate Promela statements. The line numbers have been added for clarity and are not part of the language.

Every Promela statement in a model potentially has a dual role. Every statement is a condition that is evaluated prior to execution. If the result is false, then the statement blocks. If the result is true, then the statement executes with its associated side-effect (such as variable assignment), if any. Therefore, any statement becomes a guard for the statemen

either by

The p

Figure 2.

are starte

initial pro

11. forces

within ar

A do-

long as o

include a

on line 1

true, ther

chooses c

An if

execution

Assig

of the "fe

Varial

which is a

Prome

statement

ated in lin

12 and 19

statement following it. To make the syntax easier to read, statements may be separated either by semicolons or **->**.

The primary structuring element in Promela is the *proctype*. Lines 10–14 and 15–30 in Figure 2.6 are complete processes, called **proctypes** in Promela. One or more **proctypes** are started by the **run** statement, as on line 13. The **init proctype** is distinguished as the initial process started during a Promela execution. The **atomic** construction, shown on line 11, forces all statements within its context to execute as one. No interleaving is permitted within an **atomic** block.

A **do-od** construct is shown on lines 18–20. The **do-od** "loop" continues executing as long as one of the conditions, delimited by "::", is true. Constructs after a "::" typically include a guard and an executable statement, although this is not required. The condition on line 18 guards the assignment statements on the same line. If no guard in the loop is true, then the **do-od** block as a whole blocks. If more than one guard is true, then Promela chooses one of the true guards nondeterministically and executes that statement.

An **if-fi** construct is shown on lines 22–26. The semantics of the guards and statement execution are the same as in the **do-od** block, except the **if-fi** block is only executed once.

Assignment statements and logical expressions are the same as in C. Promela has many of the "features" of C, including the increment (**i++**) and decrement (**i--**) statements.

Variables in Promela include integers, boolean, bit (the same as boolean) and *mtype*, which is an enumeration type. Line 9 defines three enumeration values, **on**, **off**, and **none**.

Promela allows for the creation of C-type data structures through the use of **typdef** statements. Lines 1–6 define a **typedef** named **A_type**. The structure is actually instantiated in line 8. References to elements of the structure are the same as in C, as seen in lines 12 and 19.

2.3.2

Promela

and sema

channel

operatio

three hol

contain r

Recei

field. W

a messag

statemen

When a

matched

the varia

switch a

three con

copied to

2.3.2 Channels and Communications

Promela uses channels for communicating between, and even within, processes. The syntax and semantics of channels are similar to CSP [27]. Line 21 in Figure 2.6 shows a send on the channel named **queue**. The value sent is the enumeration constant **on**. Line 23 is a receive operation on channel **queue**. Channel **queue** is declared on line 7 as a queue of depth of three holding messages of type **mtype**. Channels can be any depth, including zero, and can contain messages of arbitrary length and type.

Receive operations on a channel may have either constant values or variables in each field. When a channel receive contains a constant, the value is considered to be a pattern a message must match in order to be received. Line 29 in Figure 2.6 is an example. This statement blocks until the message **off** is at the head of the message queue named **queue**. When a combination of variables and constants is used in a receive operation, a message is matched by the constant values, then the remaining values in the message are copied into the variables in the receive statement. The statement **xyz?on,x,switch,y**, where **on** and **switch** are constants, and **x** and **y** are variables, receives a message where fields one and three contain **on** and **switch**, respectively. Fields two and four of the matched message are copied to variables **x** and **y**, respectively.

```
typedef A
int x;
int y;
bool unus
mtype val
}

chan queu

A_type A;

mtype={on

init
{
atomic {
run abc(
}

proctype
{
int i;
do
:: A.x >
od;
queue!on

if
:: queue?
:: A.y >
:: A.y >
fi;

skip1: p
skip2: p
    queu
}
```

```
typedef A_type {
int x;
int y;
bool unused;
mtype vals;
}

chan queue=[3] of {mtype};

A_type A;


mtype={on, off, none};


init
{
atomic {A.x = 1; A.y = 2}
run abc()
}

proctype abc()
{
int i;
do
:: A.x > 1 -> A.y = A.y + 1; A.x = A.x + 1
od;
queue!on;

if
:: queue?vals;
:: A.y > 4 -> goto skip1
:: A.y > 6 -> goto skip2
fi;

skip1: printf("we are at skip 1") -> i = 12;
skip2: printf(" we are at skip 2");
    queue?off
}
```

Figure 2.6: Promela model for a simple client-server system

2.3.3 D

Promela w

stra [29] a

Dijkstra's

clusively o

(channel d

tion of me

in guards

The se

other guar

are false. I

and hence

2.4 M

The behav

are often

are frequen

complex an

of accurate

on the use

require a g

In the

temporal lo

2.3.3 Differences From Other Guarded Languages

Promela was influenced significantly by the "guarded command languages" of E.W. Dijkstra [29] and C.A.R. Hoare's CSP [27] language. There are, however, important differences. Dijkstra's language has no primitives for process interaction. Hoare's language was based exclusively on synchronous communication, constructed in Promela as an unbuffered channel (channel depth of zero), but Promela also permits buffered channels, allowing the construction of message queues. Also in Hoare's language, the type of statements that can appear in guards is restricted, while Promela has no restrictions.

The semantics of the `do-od` and `if-fi` statements in Promela are also different from other guarded command languages in that these statements are not aborted when all guards are false. Instead, a false guard blocks execution until true. In fact, any statement can block and hence can become a guard.

2.4 Model Checking

The behavior of objects are described in UML by finite state machines and since there are often many occurrences of objects in a real system, concurrent finite state machines are frequently found in a system design. The behavior of such a system can be extremely complex and verification through testing is notoriously troublesome because of the difficulty of accurately reproducing event sequences. Although there has been considerable research on the use of automated theorem provers, these techniques are time consuming and often require a great deal of manual intervention.

In the model checking approach, system specifications are expressed in propositional temporal logic and the system itself is expressed as a state transition system. Efficient search procedu

In a spec

Tempora

A comm

X and Y

temporal

Expre

style of te

atomic pr

true in th

operators

- **X**g :

- **F**g : T

  on th

- **G**g :

  is, g is

- g**U**h :

  assum

- g**R**h :

  the st

procedures are used to determine if the specification is true of the given transition system. In a specification, propositional logic expresses conditions that must be true in a given state. Temporal logic expresses how expressions change throughout the state transition process. A common example might state that when X is true, then eventually Y must become true. X and Y are each propositional phrases while the "eventually" portion is expressed with temporal logic.

Expressions define a condition either relative to a state or to a path, depending on the style of temporal logic used. A *path* is a sequence of states starting at a defined state. An atomic proposition, p, is said to be true in a given state, or if p is true of a path, then it is true in the first state of the path. Given atomic propositions g and h, the temporal path operators common to the logics described are:

- $\mathbf{X}g$: The "next" operator. True if g is true in the next state on the path.
- $\mathbf{F}g$: The "eventually" operator. True if g eventually becomes true, that is, somewhere on the path g is true.
- $\mathbf{G}g$: The "always" or "global" operator. True if g is true now and henceforth, that is, g is true at all states henceforth on the path.
- $g\mathbf{U}h$: The "until" operator. True if g is true until h is true. The semantics usually assumes h will eventually occur (this is called "strong until").
- $g\mathbf{R}h$: The "releases" operator. True if h is true all along the path up to and including the state where g is true.

2.4.1 C

Temporal l

or branchi

and *path f*

a path for

while *a* al

exist are c

addition t

existential

formula. F

hence nest

operators:

- **AXp**

- **AFp**

- **AGp**

- **ApU**

- **ApR**

CTL ex

the tree rep

the stateme

where *e* is t

[2]There ar
The logics for
of temporal l

2.4.1 CTL and LTL Logics

Temporal logics are often classified according to whether time is assumed to have a linear or branching structure. Propositional temporal logics include definitions for *state formulas* and *path formulas.* A state formula is a propositional statement that is true in a state while a path formula expresses conditions along a path. For example, $a\mathbf{U}b$ is a path formula while a alone could be either. Two temporal logics for which computational procedures exist are called Linear Temporal Logic (LTL) and Computational Tree Logic (CTL)[2]. In addition to the temporal operators above, CTL contains universal path quantifier **A** and existential path quantifier **E**, each of which requires a path formula and produces a state formula. Furthermore, CTL limits the operand of a temporal operator to a state formula, hence nested temporal operators are not permitted. Effectively, this produces ten CTL operators:

- **AXp** (and **EXp**) – for all (there exists) paths, in the next state p is true.
- **AFp** (and **EFp**) – for all (there exists) paths, eventually p becomes true.
- **AGp** (and **EGp**) – for all (there exists) paths, p is always true.
- **ApUq** (and **EpUq**) – for all (there exists) paths, p is true until q.
- **ApRq** (and **EpRq**) – for all (there exists) paths, p releases q.

CTL expresses the possible next states in the form of a branching tree with each path in the tree representing one possible computation path. The CTL formula **AG**(**EF**e) expresses the statement, on all paths, from every state, there exists a path on which there is a state where e is true. In other words, it is always possible to reach a state where e is true.

[2]There are *many* temporal logics based on how path quantifiers and temporal operators are combined. The logics form a complex relation of relative expressiveness. See [30] for a discussion of the relative powers of temporal logics.

In

above

paths.

LTL ar

path fo

p becon

formula

is a state

which on

this is no

2.4.2 S

The SPIN

in written

Correct

types of be

mine the va

when possib

techniques a

Since *cl*

inevitable cl

see.

Promela

The assert

In contrast, LTL formulas are written relative to paths using only the path operators above and atomic propositions. In effect, every LTL formula is prefixed with A, for all paths, because there is no sense of branching in LTL. The expressive power of CTL and LTL are not comparable [30] because one applies to state formulas while the other applies to path formulas. The LTL formula $\mathbf{AFG}p$, meaning eventually (on all execution sequences) p becomes, and stays true, is not expressible in CTL. On the other hand, consider the CTL formula $\mathbf{AG}(\mathbf{EF}e)$, meaning on all paths, from every state, there exists a path where there is a state in which e is eventually true. The E quantifier permits two branching paths down which on one e never becomes true and another on which it does becomes true. Clearly, this is not expressible in LTL.

2.4.2 SPIN

The SPIN system is used to verify properties expressible in LTL. The system to be verified in written in SPIN's procedurally based language called Promela, as described above.

Correctness claims about the behavior of a Promela model are formalized in SPIN. Two types of behaviors are generally that a behavior is inevitable, to impossible. To determine the validity of a claim, SPIN exhaustively searches the entire state space of a model, when possible. When the state space is too large, both approximation and state reduction techniques are available for partial searching of the state space.

Since checking for an impossible behavior is easier and faster than checking for an inevitable claim, SPIN claims generally express a negative behavior that we no not wish to see.

Promela has several types of constructs for making correctness claims about a model. The **assert(*p*)** statement can be placed anywhere, and when executed, verifies that conditio

asse

S

haus

the p

state

T

are F

asser

never

The

only

langu

produ

repre

2.5

A forr

ificatio

ematio

ficatio

as info

[3]Bu
is define
final sta
forming

dition p is true in the current global state. Global invariants can be checked by placing an **assert** statement alone in a process such that the condition is continuously checked.

SPIN can also check for state reachability in several forms. The most general, is an exhaustive search to determine which states are unreached. SPIN analyses can also determine the presence or absence of cycles, or loops in the model. Depending on how the Promela statements are marked, the loop is considered acceptable behavior or incorrect behavior.

The most powerful form of the claim specification is called a *never claim*. Never claims are Promela statements contained in a block delimited by **never** { ... } . The never claim asserts that the statements in the never block should not execute to completion. Formally, never claims are based on the ability to translate LTL expressions into Buchi automata [3]. The automata representing the product of the LTL automata and the model itself accepts only states for which the LTL claim is true. Since it is much easier to check for an empty language, normally the LTL formula is first negated, hence the term "never claim". The product of the Buchi automata from the negated LTL claim and the model should then represent an empty language.

2.5 Formal and Informal models

A formal software specification is a system behavior constraint expressed in a formal specification language whose vocabulary, syntax and semantics are rigorously defined in mathematical terms. Predicate logic, LOTOS [31] and CSP [27] are examples of formal specification languages, as are computer languages such as Java. Software specifications start as informal descriptions of the tasks to be performed, often expressed in imprecise natural

[3]Buchi automata are finite state automata that can accept infinite input sequences. Buchi-acceptance is defined as passing through an *acceptance state* an infinite number of times. In the case of a terminating final state, SPIN (and others) *stutter* that state, effectively transitioning to the same state endlessly, thus forming an infinite sequence.

langu

have l

compu

ment p

to forn

Fra

gies tha

formalis

Informa

car

Semi-fo

mat

are

Formal:

derly

notat

ples.

their

Fraser.

diary semi-

are used, t

the appro

whether fo

language. If the project is finally realized in executable code, then the specification will have been translated, by definition, to a formal specification language, namely the target computer language in which the project is implemented. Somewhere during the development process the imprecise specification will necessarily cross the boundary from informal to formal.

Fraser *et al.* [32] suggest a taxonomy for classifying software development methodologies that integrate informal and formal specifications. The authors classify categories of formalism as follows:

Informal: Techniques that do not have complete sets of rules to constrain the models that can be created. Natural language and unstructured pictures are examples.

Semi-formal: Techniques that have a defined syntax. Typical instances include diagrammatic techniques with precise rules that specific conditions under which constructs are allowed. OMT and UML fit in this category.

Formal: Techniques that have a rigorously defined syntax and semantics. There is an underlying theoretical model against which a description expressed in a mathematical notation can be verified. Specification languages based on predicate logic are examples, as are actual programming languages, which have precise semantics by virtue of their execution behavior.

Fraser, *et al.* further classify the integration of informal and formal by the use of intermediary semi-formal specifications as bridging mechanisms. When semi-formal specifications are used, the approach is termed *transitional.* When semi-formal specifications are not used, the approach is called *direct.* The transitional approach is further decomposed based on whether formal specifications are produced directly from semi-formal specifications or semi-

for

seq

men

succ

sum

appro

metho

semi-f

providi

a set o

for all t

research.

diagram

we able *t*

the *consis*

approach c

a formal la

formal and formal are produced successively in parallel. The former approach is termed *sequential* while the latter is termed *parallel-successive.* All approaches bifurcate on the dimension of computer-assisted support, consequently we have both the *transitional parallel-successive assisted* and *transitional parallel-successive unassisted* approaches. Figure 2.7 summarizes the taxonomy. Fraser notes no examples of the transitional parallel assisted approach. Subsequently, Wang and Cheng [17] described an object-oriented development methodology using OMT based on the transitional parallel assisted approach. In that work, semi-formal specifications written in OMT are translated into LOTOS [18, 19, 20, 33], thus providing precise semantics for OMT models. The emphasis of that project was to develop a set of rules and a process to enable the automated generation of formal specifications for all three types of OMT diagrams that could be analyzed using existing tools. In this research, we build on that approach by describing a method to attach semantics to *any* diagrammatic semi-formal model in a precise and rigorous manner. That is, not only are we able to assign formal semantics to the diagrams, we are also able to formally establish the consistency of the formalization rules themselves. In order to provide semantics, this approach develops a homomorphism between *metamodels* of the semi-formal notation and a formal language.

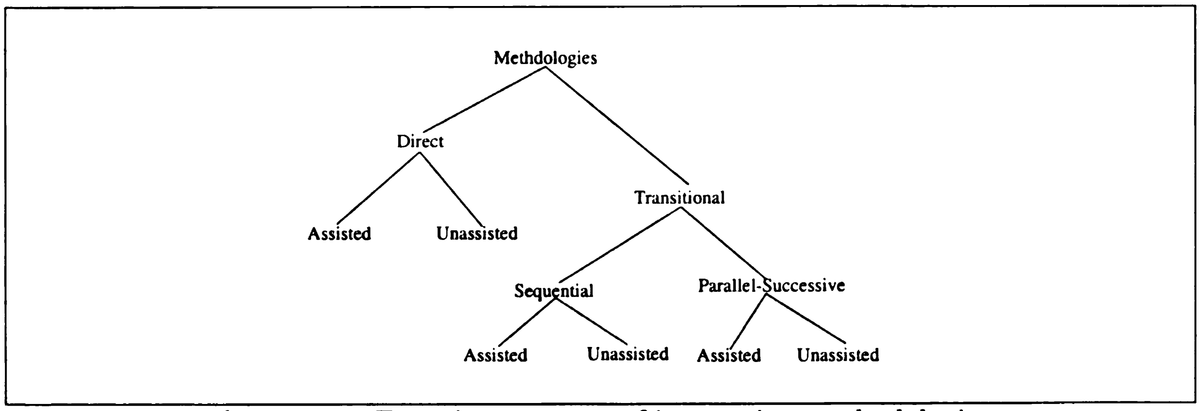


Figure 2.7: Fraser's taxonomy of integration methodologies.

Chapter 3

Metamodels and Mapping Framework

As described in Section 2.1, a UML metamodel is a type of class model that describes the syntax of models written in UML notation. Unless the class model itself is formalized, the mappings developed between metamodels will not rest upon a rigorous basis. This chapter presents a formalization of the class model, which enables the formal description of homomorphic mappings between metamodels. Using a formalization of the class model, the mapping framework central to generating semantics for UML models is then presented.

3.1 Class Model Formalization

The class model, upon which the metamodel is built, has been formalized by Bourdeau and Cheng [18]. Bourdeau and Cheng showed that a class model[1] can be formalized using algebraic specifications and specific algebras related to object instance diagrams. We draw

[1]Bourdeau and Cheng's work pertained to OMT class models but the difference between OMT and UML class models are very slight and most symbols are common between UML and OMT.

up

for

3.1

An

how

that

In

UML

specif

set of

Each in

to the

in Figu

approac

diagram

the set o

consisten

In this

upon Bourdeau and Cheng's formalization with the intent of providing a class formalization for metamodels to support a mapping between metamodels.

3.1.1 Foundation to Class Formalization

An algebraic specification consists of a set of *types* or *sorts*, a set of *signatures* that describes how functions in the specification map types to types, and a set of axioms or constraints that describe the interrelationship between the types and functions.

In Bourdeau and Cheng's class diagram formalization, an object model (the same as a UML class model) is formalized as an algebraic specification. The semantics of the algebraic specification are derived from their corresponding algebras. *Instance diagrams*, one possible set of instantiations of objects from the object diagram, represent algebras in their approach. Each instance diagram consistent with the object model generates an algebra that conforms to the algebraic specification derived from the object diagram. The commuting diagram in Figure 3.1 is adapted from that used by Bourdeau and Cheng [18] and shows their approach to formalization. There are two ways to generate the algebras for an instance diagram: compute the the algebraic specification from the object diagram and examine the set of algebras that satisfy the specification, or determine the set of instance diagrams consistent with the object model and compute the corresponding set of algebras.

$$\begin{tikzcd}
\text{object models} \arrow[r, "\text{diagram semantics}"] \arrow[d, "\text{formalized as}"'] & \text{Instance Diagrams} \arrow[d, "\text{formalized as}"] \\
\text{Algebraic Specifications} \arrow[r, "\text{algebraic semantics}"'] & \text{Algebras}
\end{tikzcd}$$

Figure 3.1: Bourdeau and Cheng's basic approach to formalization.

In this approach, a class is considered a *type*, or *sort* in an algebraic specification. Re-

l

ir

ti

m

rel

in t

of a

Th

and *Ch*

lations.

plicity c

3.1.2

Our class

but we ab

a class is

concept of

lationships between classes are represented as binary relations between types and appear in the signature section of the algebraic specification. The special relationship *aggregation* is represented as a predicate $hasPart(Aggregate, Part)$, true exactly when $Part$ is a member of the $Aggregate$ aggregation. Multiplicity constraints modify the type of binary relationship depending on the constraints at each end of the association.

Bourdeau and Cheng formalize multiplicity constraints on associations between objects in terms of four relational properties: *functional, injective, surjective,* and *total.* In terms of a relation predicate R, each is defined as follows:

Functional: $$\forall x, y, z\ (R(x,y) \wedge R(x,z) \Longrightarrow y = z) \tag{3.1}$$

Injective: $$\forall x, y, z\ (R(x,z) \wedge R(y,z) \Longrightarrow x = y) \tag{3.2}$$

Surjective: $$\forall y \exists x R(x,y) \tag{3.3}$$

Total: $$\forall x \exists y R(x,y) \tag{3.4}$$

This generates fifteen possible constraints on each relation between classes. Bourdeau and Cheng's work enumerated seven relations that in combination represent all fifteen relations. In Section 3.1.3 we show that four relations are sufficient to generate all the multiplicity constraints.

3.1.2 Enhanced Class Formalization

Our class formalization draws on many of the ideas in Bourdeau and Cheng's formalization, but we abandon the algebraic specification and resulting algebras. We retain the idea that a class is a type and that relationships are represented as binary relations, however our concept of a type varies slightly from Bourdeau and Cheng's in that we consider *type* to strict

to be

associ

In

class c

with a

x, in c

Cla

For exa

of type

As a

to relati

types, n

As d

position.

model, an

no "comp

definition

Since p

is *injectiv*

which is e

strictly rely on a typing predicate and not form a signature. We consider all the objects to be the universe of a model for a predicate calculus, over which we can define predicates associated with class relationships.

In our formalization, a class model is formalized as a set of typed components where the class of the component corresponds to the component type. Class membership is expressed with a typing predicate. We write T_{class} for a class predicate, and therefore for a component x, in class y, $T_y(x)$ is true.

Class relationships are formalized as 2-ary predicates between members of each class. For example, in Figure 3.2 a model component x of type **StateVertex** and a component y of type **Transition** are related by the predicates $source(x, y)$ and $outgoing(y, x)$.

As a shorthand, we will occasionally write types, or class names, as the arguments to relationship predicates, for example $source(StateVertex, Transition)$. Elements of the types, not the entire type themselves, are related by the relationship predicates.

As described previously, UML has two aggregation relationships, *aggregation* and *composition*. We describe aggregation with the *hasPart* predicate as in Bourdeau and Cheng's model, and extend the description with the *hasComp* predicate for composition (OMT has no "composition" concept). For classes W and P, and components $w \in W$ and $p \in P$, the definition of $hasPart(w, p)$ requires p to be a part (an aggregate, not a component) of w.

Since parts of compositions can only be members of one collection, the *hasComp* relation is injective. For aggregates x and y and part p, we have:

$$hasComp(x, p) \wedge hasComp(y, p) \Longrightarrow x = y$$

which is exactly the constraint that *hasComp* is injective. Furthermore, both *hasPart* and

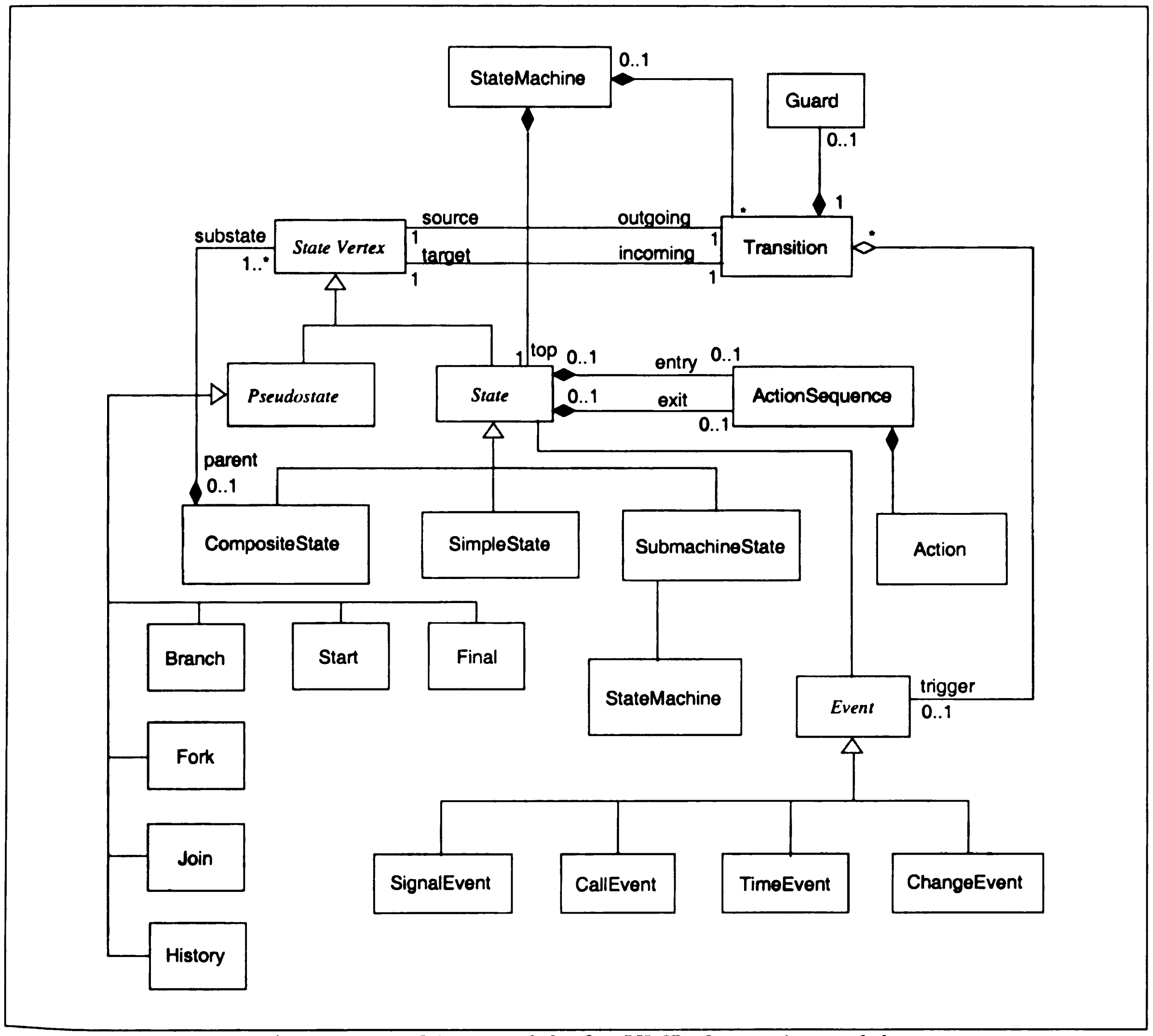


Figure 3.2: Metamodel of a UML dynamic model.

h

o

Th

tru

Thi

simu

(

supe

P,

Expres

themse

in *each*

3.1.3

We adop

and show

hasComp are transitive. If class **B** is a component of class **A**, and class **C** is a component of **B**, then clearly **C** is a component of **A**. Formally:

$$hasComp(A, B) \wedge hasComp(B, C) \Longrightarrow hasComp(A, C).$$

The *hasPart* predicate is similarly transitive.

If class **B** is a component of **A** then it is also a part of **A**, however the converse is not true for two reasons: First, in the *hasPart* relation, **B** can still exist when **A** s destroyed. This is not true for *hasComp*. Secondly, **B** could be a *part* of several classes but it cannot simultaneously be a *component* of several classes.

Class specialization, also called *subtyping*, is constrained by requiring the subtype of a supertype to be a member of the supertype class. Formally, for subtype C and supertype P,

$$\forall x \, (T_C(x) \Longrightarrow T_P(x)). \tag{3.5}$$

Expression (3.5) also expresses the relation between virtual classes, containing no objects themselves, and subclasses. Unlike Bourdeau-Cheng, we do not require an "error object" in each class.

3.1.3 Multiplicity Constraints, Revisited

We adopt Bourdeau and Cheng's multiplicity constraints on associations between objects and show the basis set. We also show how the basis set can be used to form the constraint for a

As

ation. t

be form

function

reverse r

and the r

and *inje*

parenthe

Figure 3.
formed.

for any multiplicity. Again, the four constraints are:

Functional: $$\forall x, y, z\,(R(x,y) \wedge R(x,z) \Longrightarrow y = z) \tag{3.6}$$

Injective: $$\forall x, y, z\,(R(x,z) \wedge R(y,z) \Longrightarrow x = y) \tag{3.7}$$

Surjective: $$\forall y \exists x R(x,y) \tag{3.8}$$

Total: $$\forall x \exists y R(x,y) \tag{3.9}$$

As mentioned above, there are fifteen possible relation characterizations for an association, but Figure 3.3 shows the basis set from which any association characterization can be formed. In the figure, the association is labeled with the letters F, I, S and T to denote *functional, injective, surjective* and *total* properties of the relation. The properties of the reverse relations, *i.e.*, $R'(y,x)$, formed by exchanging the positions of the "1" on the right and the multiplicities listed on the left are produced by interchanging the roles of *functional* and *injective*, and interchanging *surjective* and *total.* In Figure 3.3 the italicized labels in parenthesis denote the reverse relationship.

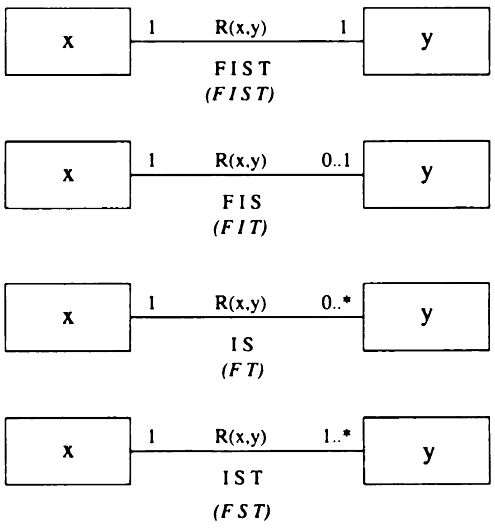


Figure 3.3: The four basis relations from which all other association relationships can be formed.

Any as

"1" on ea

of the fou

intersectio

with prop

injective a

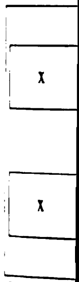


Figure 3
set of con

UML

Multiplici

can descri

1..*n* for pr

$R(x$

Constrain

predicate

R

Setting a

Any association can be characterized by reducing it to two associations with multiplicity "1" on each end. The new characterization is formed by taking the least constraining set of the four properties that characterizes the relation. This effectively means taking the intersection of the properties. For example, Figure 3.4 shows how multiplicities 1 to 1..∗ with properties I S T and 0..1 to 1 with properties F I T are combined to form the constraints *injective* and *total* (the intersection of I S T and F I T).

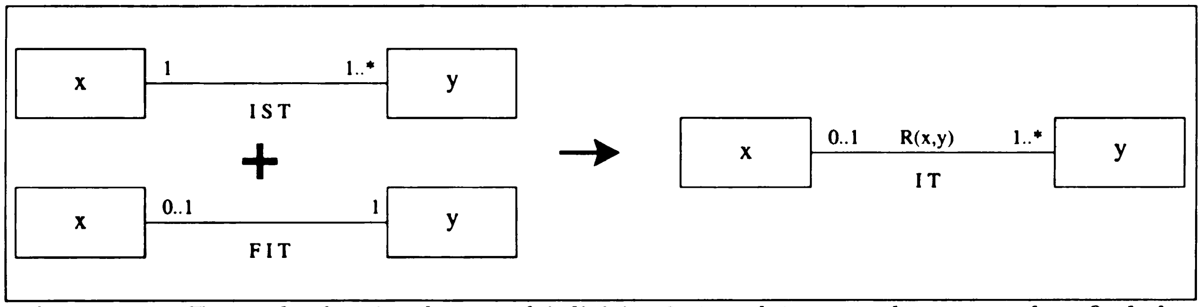


Figure 3.4: Example showing how multiplicities 0..1 and 1..∗ are decomposed to find the set of constraints for an association.

UML also includes cardinal multiplicities with limits of the form $m..n$ where $m < n$. Multiplicities are statically set in both class and metamodels, therefore, first order predicates can describe these constraints. Expressing the upper bound on a multiplicity n, such as in $1..n$ for predicate R is accomplished with a constraint predicate of the form

$$R(x, y_1) \wedge R(x, y_2) \wedge \ldots R(x, y_{n+1}) \implies y_1 = y_2 \vee y_1 = y_3 \vee \ldots y_n = y_{n+1}. \quad (3.10)$$

Constraining the lower bound to be at least m, for example $m..n$ is accomplished with a predicate of the form

$$R(x, y_1) \wedge R(x, y_2) \wedge \ldots R(x, y_m) \wedge y_1 \neq y_2 \wedge y_1 \neq y_3 \wedge \ldots y_{n-1} \neq y_n. \quad (3.11)$$

Setting a specific range of cardinalities is accomplished by conjuncting a predicate of the form of E

pression

nonethel

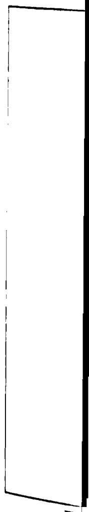
3.1.4 E

As an ex

metamod

dostate.

with the f



Fi

form of Expression (3.10) with Expression (3.11). While predicates of the form of Expression (3.10) and Expression (3.11) can be complex, constraining specific cardinalities is nonetheless possible.

3.1.4 Example of a Class Formalization

As an example, Figure 3.5 shows a small model taken from a part of the UML dynamic metamodel. For conciseness, class **SV** refers to **State Vertex**, **S** to **State**, **PS** to **Pseudostate**, **SS** to **SimpleState**, and **T** to **Transition**. The metamodel is fully characterized with the following predicates:

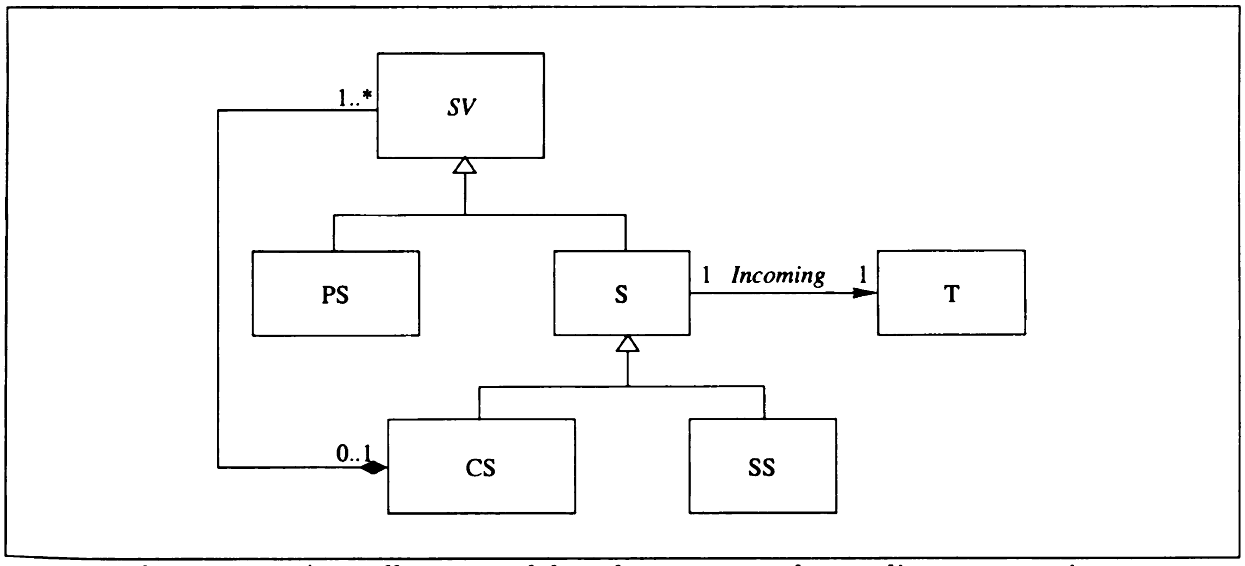


Figure 3.5: A small metamodel to demonstrate the predicate constraints.

$\forall x\,(T_{CS}$

$\forall x\,(T_{SS}$

$\forall x\,(T_{PS}$

$\forall x, y, z$

$\forall x \exists y\,(T$

$\forall x, y, z$

$\forall x, y, z$

$\forall x, y, z$

$\forall y \exists x\,(T$

$\forall x \exists y\,(T$

Predic
(or class)
Predicate
icate (3.1
componen
quiring a
toward th
tive. Sin
Predicate
one-to-on

$$\forall x\ (T_{CS}(x) \Longrightarrow T_S(x) \Longrightarrow T_{SV}(x)) \tag{3.12}$$

$$\forall x\ (T_{SS}(x) \Longrightarrow T_S(x) \Longrightarrow T_{SV}(x)) \tag{3.13}$$

$$\forall x\ (T_{PS} \Longrightarrow T_{SV}(x)) \tag{3.14}$$

$$\forall x, y, z\ (T_{SV}(x) \land T_{CS}(y) \land T_{CS}(z) \land \mathit{hasComp}(x, y) \land \mathit{hasComp}(x, z) \Longrightarrow y = z) \tag{3.15}$$

$$\forall x \exists y\ (T_{SV}(x) \land T_{CS}(y) \land \mathit{hasComp}(x, y)) \tag{3.16}$$

$$\forall x, y, z\ (T_{CS}(x) \land T_{CS}(y) \land T_{SV}(z) \land \mathit{hasComp}(x, z) \land \mathit{hasComp}(y, z) \Longrightarrow x = y) \tag{3.17}$$

$$\forall x, y, z\ (T_S(x) \land T_T(y) \land T_T(z) \land \mathit{incoming}(x, y) \land \mathit{incoming}(x, z) \Longrightarrow y = z) \tag{3.18}$$

$$\forall x, y, z\ (T_S(x) \land T_T(y) \land T_T(z) \land \mathit{incoming}(x, z) \land \mathit{incoming}(y, z) \Longrightarrow x = y) \tag{3.19}$$

$$\forall y \exists x\ (T_S(x) \land T_T(y) \land \mathit{incoming}(x, y)) \tag{3.20}$$

$$\forall x \exists y\ (T_S(x) \land T_T(y) \land \mathit{incoming}(x, y)) \tag{3.21}$$

Predicates 3.12 and 3.13 convey inheritance by expressing that any object that is type (or class) CS or SS is also type (or class) S, and objects of type S are also type SV. Predicate (3.14) expresses the inheritance relationship between classes PS and SV. Predicate (3.15) says if there is a CS component for an SV object, then there is only one SV component. Predicate (3.16) expresses the aggregation relation in the other direction, requiring an object of type SV for each CS object. The *hasComp* predicate is directional toward the collection for the purpose of setting properties such as functional and injective. Since this aggregation is composition, the injective constraint has been added as Predicate (3.17). Predicates (3.18) through (3.21) constrain the relation *Incoming* to be one-to-one and onto, that is, for every S-type object there is a T-type object, and vice ve

De

or

in

by

th

in

R

3

a

a

1

a

p

versa.

UML contains the concept of a *dependency* either between classes or between relations. Dependencies are drawn in UML as a dotted line between the elements and denote that one association (class) is dependent on the other. For example, in Figure 3.6 the diagram indicates that any A associated to C by relation $R1$ has a dependent instance of B associated by $R2$. Similarly, for each D associated with E there is a dual relation, $R3$ and $R4$ to the instance(s) of E. We formalize the dependency relation by including all components in a single predicate. Therefore, the relation between D and E would be formalized as $R34(D, E)$.

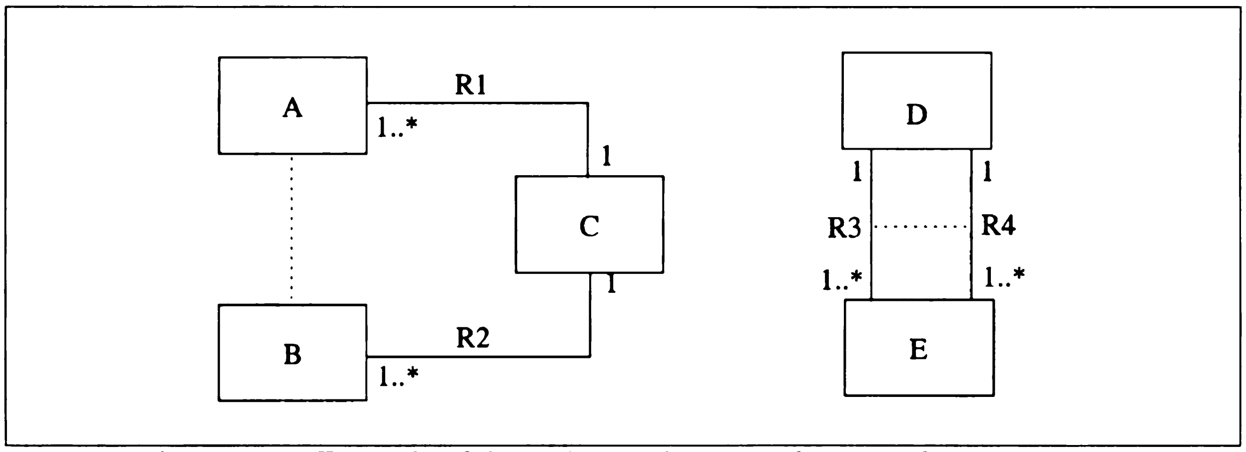


Figure 3.6: Example of dependencies between classes and associations.

3.2 Mapping Framework

As described in [32], transforming a semi-formal model such as a UML dynamic model to a formal model is largely a matter of generating precise semantics. Generating semantics is accomplished by a rigorous mapping from the semi-formal language to a formally defined language possessing precise semantics. We call the semi-formal language the *source language* and the formal language the *target language*. The combination of a set of mapping rules plus a language that has formal semantics generates the semantics for UML. To ensure cor

me

3.2

UN

for

mo

Co

sta

tra

co

sy

is

Th

we

do

in

ci

co

pi

st

st

wi

consistency and rigor, the mapping rules rest upon a homomorphic mapping between the metamodel of the source language (UML) and a metamodel of the target language.

3.2.1 Informal Discussion of the Mapping Framework

UML does not have a precise semantics. In other words, there are diagrammatic constructs for which the interpretation is ambiguous. For example, consider the section of a dynamic model shown in Figure 3.7. The figure shows two concurrent composite states `Concur1` and `Concur2` each containing two simple states. Normally, one would expect these concurrent states to run in parallel as two separate threads, so what exactly is the meaning of the transition with the event `Start` ending on state `On1`? Should only the top concurrent composite state begin, or should both concurrent states begin? Should this transition be syntactically valid at all? The purpose of the mapping framework described in this section is to solve problems like this by assigning specific meaning to each construct in UML. This is accomplished by essentially "defining" a UML construct in terms of some other, well-defined, formal language. Once a mapping of this type is complete, the question "what does this UML construct mean?" can be answered with "it means the same as *this* construct in the formal language".

This approach itself is not new. Denotational semantics is largely concerned with precisely defining the meaning of a syntactic construct through a set of rules that cover the construct. Perhaps a more intuitive description can be drawn from the actions of a compiler. For example, a "C" compiler precisely defines the meaning of a particular C-language statement by producing a set of machine instructions that implement the semantics of the statement. For example, one may write the statements: `i = 0; foo(++i, ++i);`. What will be the values of the two parameters passed to function `foo()`? Depending on how the

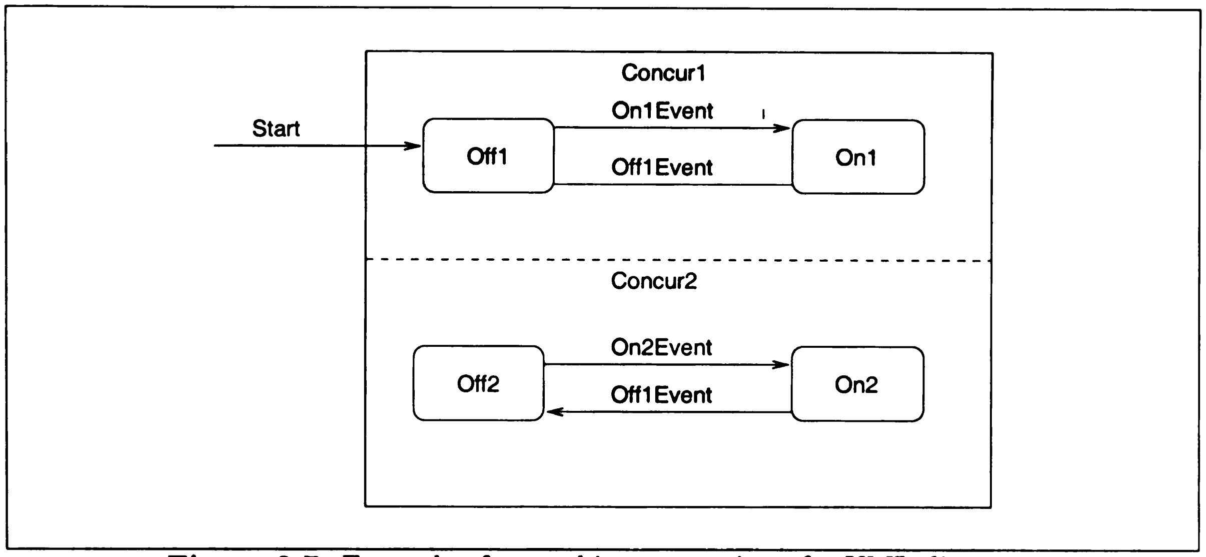


Figure 3.7: Example of an ambiguous section of a UML diagram.

compiler pushes arguments onto the stack for the procedure call, or indeed, whether it uses registers instead of the stack, and depending on how the increment operator is implemented, the possibilities include `foo(1,2)`, `foo(2,1)`, and `foo(1,1)`, although the last one seems least likely. The exact meaning of the two statements above are given by a specific mapping from the syntactical structure `i = 0; foo(++i, ++i);` to a formal language, namely, a machine language. Actual execution of this statement provides a concrete meaning for the syntactical construct.

Similarly, our framework "compiles" UML to a formal language that has a precise definition, either by virtue of being executable or because the formal language is well-defined. In our case, the "compiler semantics" are a precise set of rules for mapping UML to a formal target language. The question of "how do we know the mapping makes sense?" is answered by referring to the homomorphism between the language metamodels.

The UML metamodel describes the syntax of UML in a graphical form. Similarly, we construct a metamodel of the target formal language. To connect the two languages, we construct a homomorphic mapping between the metamodels and constrain the mapping rules with

preserves s

classes of t

between th

formally la

classes are

formally, w

Thus,

by the ho

to specify

structure.

constraint

Const

fect: it as

mapping.

phism ca

the home

complete

With

of a sour

instance

source m

target, h

rules with the homomorphic mapping. The mapping is "homomorphic" in the sense that it preserves structural relationships within the metamodel. The homomorphic mapping sends classes of the source metamodel to classes in the target metamodel and maps relationships between the classes from the source metamodel to the target metamodel. As is explained formally later, the mapping must preserve relationships. In an analogy to an algebra, the classes are the elements of the algebra and the relationships are the operators, although formally, we do not consider the metamodel to be an algebra.

Thus, the "compiler semantics" between UML and the formal language are constrained by the homomorphic mapping between the respective metamodels. A rule is not allowed to specify a transform that violates the homomorphism. In this way, we are assured that structure, and hence meaning, is preserved by the mapping rules. Further, the homomorphic constraint provides consistency within the rules.

Constraining the mapping rules with the homomorphism has an additional beneficial effect: it assures completeness. Every rule must correspond to some part of the homomorphic mapping. Conversely, sections of the metamodel that are not mapped by the homomorphism cannot occur in the mapping rules. Therefore, by comparing the mapping rules and the homomorphic mapping between metamodels we can ensure structural consistency and completeness.

With a complete set of mapping rules and the corresponding homomorphism, an instance of a source model can be transformed into an instance of a target model. Since the target instance model has precisely defined semantics, these semantics are conveyed to the UML source model. Figure 3.8 summarizes diagrammatically the relationship between the source, target, homomorphism, and mapping rules.

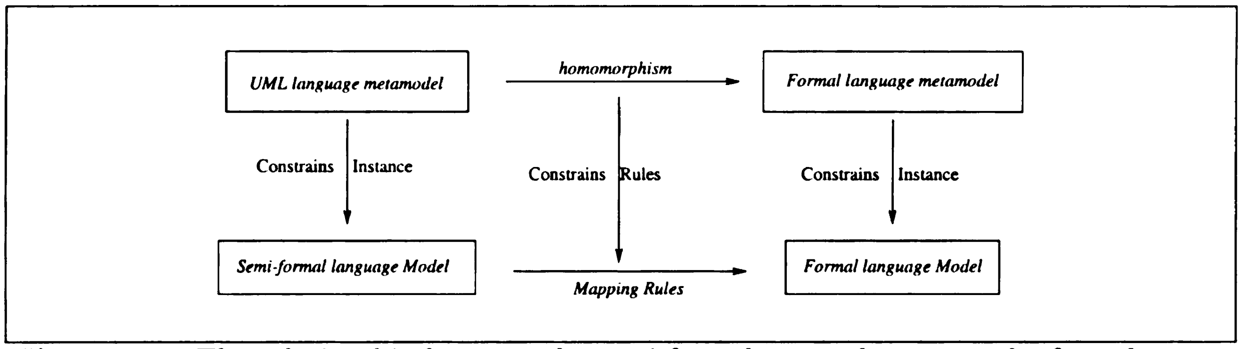


Figure 3.8: The relationship between the semi-formal source language, the formal target language, the homomorphic binding and the instance-specific mapping rules.

3.2.2 Homomorphic Mappings on Metamodels

The formal basis of the homomorphic mapping between metamodels is discussed in this section.

Homomorphic mappings have the important property that they preserve structural relationships between entities in two different systems. This enables compositional, semantic-preserving mappings from one system to another.

In algebras, a homomorphic mapping maps one algebra to another with the property of preserving operations [34]. For algebra A with binary operation $\oplus$, algebra B with operation $\otimes$, and a homomorphic mapping h that maps elements of A to B, we have

$$a, b \in A : \quad h(a \oplus b) = h(a) \otimes h(b).$$

We define a homomorphic function from the source metamodel to the target metamodel that preserves metamodel class relationships. The function maps classes from one metamodel to another and relationships from one metamodel to another, while preserving relationships between metamodel classes. In terms of our class formalization, for each typing predicate in the source metamodel (which represents a class in the source metamodel), the homomorphic function maps to a typing predicate in the target model. Similarly, for each relationship predicate in the source metamodel, the mapping function maps to a relationships predicate in the target model. For example, suppose the source language contains classes (types) A and B that are mapped by function h to classes (types) A' and B', respectively, in the target language. Also assume there exists a relation predicate R between A and B that is mapped by h to R'. Then our definition of a homomorphic function requires h to satisfy:

$$\forall x, y \; (T_A(x) \wedge T_B(y) \wedge R(x,y)) \implies (T_{A'}(h(x)) \wedge T_{B'}(h(y)) \wedge R'(h(x), h(y))). \qquad (3.22)$$

In Expression (3.22), x and y are members of metamodel classes A and B, and are therefore instances of a diagram component. Similarly, $h(x)$ and $h(y)$ are components of the formal language. Expression (3.22) requires classes A and B that are associated by R to map to classes A' and B', respectively. Expression (3.22) further requires the existence of an association R' in the target metamodel connected to R by h between classes A' and B'. We note that h is not necessarily injective, and with actual metamodels, is very rarely so. In the above example, there may be other relations between A' and B' other than R', but minimally, R' must exist. This case is further discussed below.

3.2.3 Handling Structurally Different Metamodels

It is rarely the case that the source metamodel and target metamodel are structurally identical. In general, we are given a metamodel of the source language, but since the goal is to produce a mapping to supply semantics to the source language, we have some flexibility when constructing the target metamodel to choose and arrange the parts of the target language that are semantically relevant. In order to make the mapping straightforward, when possible, a target class is constructed for each source class in the source metamodel.

Sinc

be n

in th

but

ther

situ

h n

A' a

has

How

due

Th

3.2

The

ther

Since the mapping function is homomorphic, it need not be surjective, therefore, there may be more classes in the target than in the source. Similarly, there may be more relationships in the target than in the source.

It may be the case that an association in the target metamodel is not explicitly depicted, but rather inferred. For example, the predicates *hasPart* and *hasComp* are both transitive, therefore, there may be inferred aggregation levels not explicitly drawn in the target. The situation is depicted in Figure 3.9. In the source diagram, B is a part of A. Assume function h maps A to A' and B to B'. Also assume it is necessary to introduce class C between A' and B' because of the constraints of the target language. The mapping rules require $hasPart(A, B)$ to be matched by a *hasPart* relation in the target. That is:

$$\begin{aligned} &\forall x, y\ (T_A(x) \wedge T_B(y) \wedge hasPart(x, y) \Longrightarrow \\ &T_{A'}(h(x)) \wedge T_{B'}(h(y)) \wedge hasPart(h(x), h(y))) \end{aligned} \tag{3.23}$$

However, as seen in Figure 3.9, there is an inferred *hasPart* relation between A' and B' due to the transitivity of *hasPart*. Formally:

$$\begin{aligned} &\forall x, y, z\ (T_{A'}(x) \wedge T_C(y) \wedge T_{B'}(z) \wedge hasPart(x, y) \\ &\wedge hasPart(y, z) \Longrightarrow hasPart(x, z)) \end{aligned} \tag{3.24}$$

Therefore, the mapping as described is valid.

3.2.4 Target Metamodel Templates

The target languages that we have typically selected are at least executable in simulation and therefore often have statements that describe more details of execution than the constructs

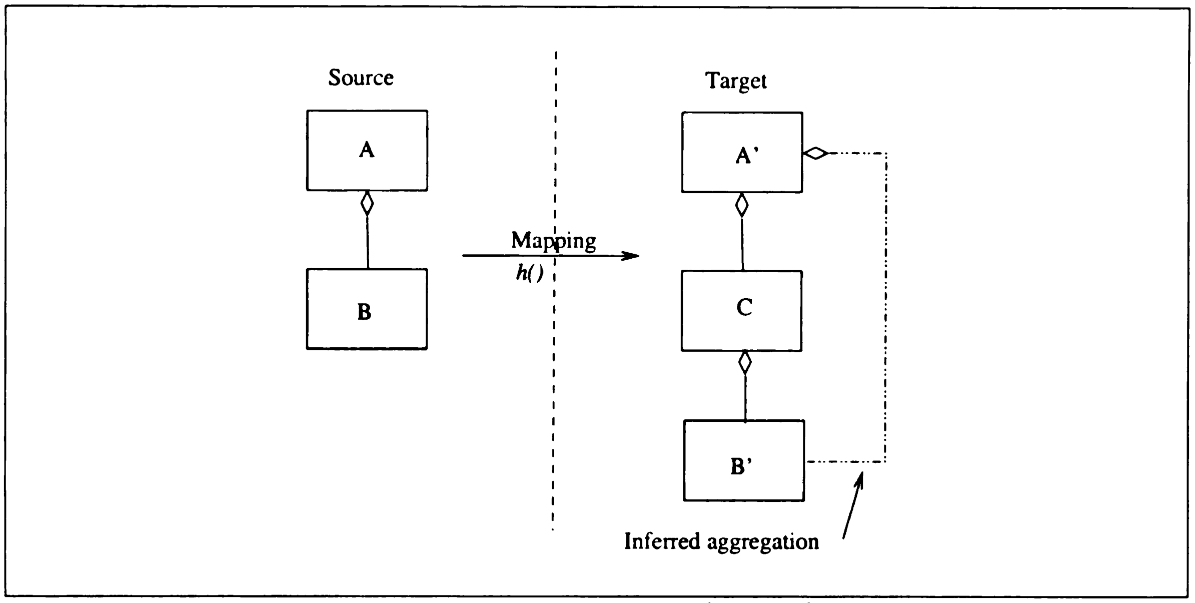


Figure 3.9: Aggregations between A' and B' are inferred

in the UML metamodel. For example when using the target language VHDL, a UML dynamic diagram *State* maps to a series of VHDL statements presented as a *template*.[2] Each template is represented in the target metamodel by one or more classes connected by zero or more connectors, and serves as a target language metamodel building block. Some templates (represented by a class) require aggregations of other templates (also represented as classes) for completion. This relationship is drawn in the target metamodel as an aggregation or composition relation. For example, the template that represents **CompositeState** requires the inclusion (through aggregation) of templates for **SimpleStates** in order to be complete. Using detailed templates permits the automatic generation of target language specifications that are executable. More examples of templates are in Chapters 5 and 6 where mapping rules for VHDL and Promela are described in detail.

[2]By *template*, we do not mean *template class* as found in C++, but rather a pattern of target language statements.

3.3 Unified UML Class and Dynamic Metamodel

A complete system model contains, at a minimum, two *types* of UML diagrams: a class diagram, and at least one dynamic model diagram. Each of these types of diagrams are described by separate metamodels in the UML references [24], however there is only one target metamodel and a single homomorphic mapping. Since our approach describes a mapping from a single UML metamodel to a target metamodel, multiple metamodels describing various UML diagrams must be combined into a unified UML metamodel. The new metamodel is called the *Unified Class/Dynamic Metamodel.* The homomorphic function is defined to provide a mapping from the Unified Class/Dynamic Metamodel to the target metamodel.

Figure 3.10 shows the UML class diagram metamodel modified to include *Model* at its highest level. From this diagram, we see that a system model is composed of one or more classes and zero or more relationships. Behavior of the entire system model is manifested through the behavior of each class as the classes interact. Then behavior is a *part of* each class. Figure 3.10 shows this relationship by depicting *Behavior* in a dashed box.

In a UML diagram, behavior is specified with a collection of state vertices and transitions, but this is exactly described by the dynamic model metamodel. Figure 3.11 shows the unified class/dynamic metamodel with the parts of the metamodel corresponding to the original class model enclosed in dotted lines.

Note in Figure 3.11 that *Behavior* is an abstract class and therefore is not instantiated directly. Rather, behavior resides in the class model as a set of state vertices and transitions, each of which is further defined by the portion of the metamodel corresponding to the dynamic model.

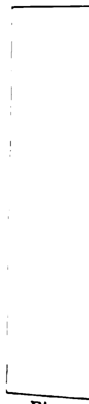
The metamodel for a formal language should follow the structure of the UML unified metamodel. In fact, this makes the homomorphic mapping more straightforward because mapping

dynamic

from its

instance

of the mo



Figure

mapping rules used to construct executable models need to draw on both the class and dynamic diagrams in concert. At a minimum, a dynamic model has no identity apart from its class. Associations, which form the basis of communication between objects, and instance variables, which hold state information, both are woven into the executable part of the model from the class diagram.

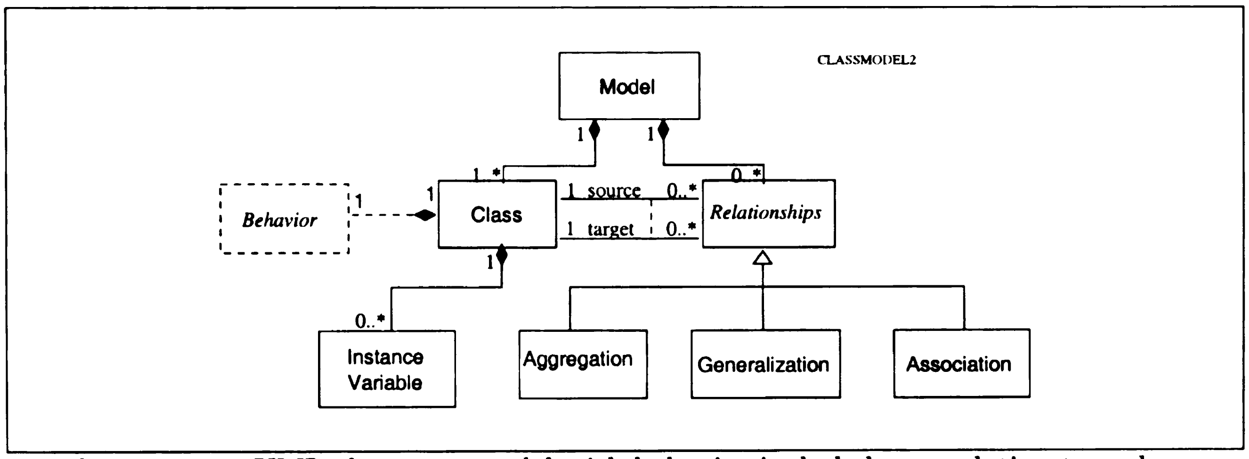
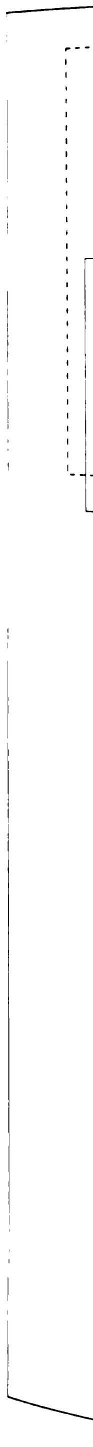


Figure 3.10: UML class metamodel with behavior included as a relation to a class.



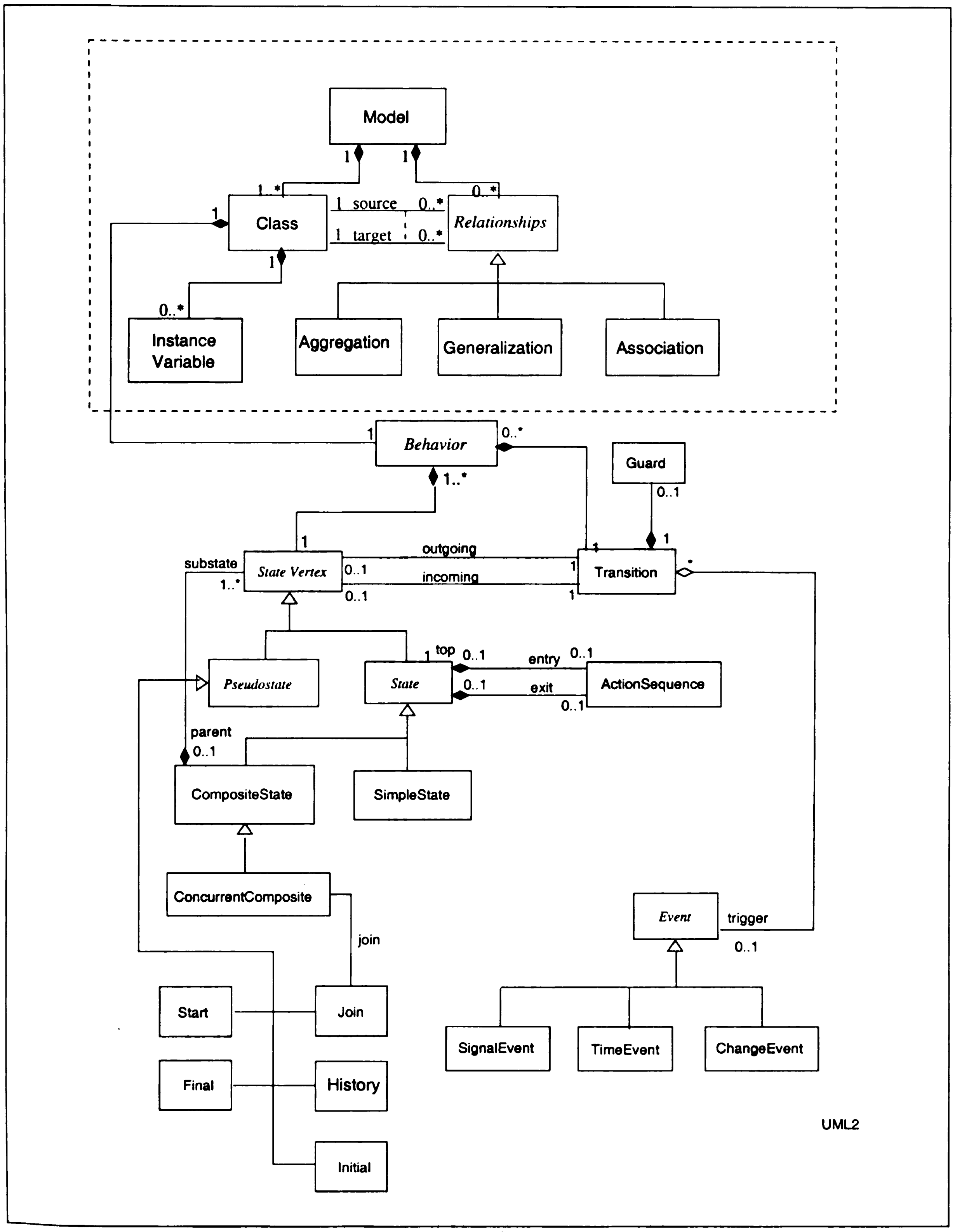


Figure 3.11: The unified class/dynamic UML metamodel.

Chapt

Desi

This chapt

mapping fr

UML diagr

results of t

the system

is actually

4.1 D

A *data fl*

that syst

modified

often lab

stores, w

cessed in

by a lab

Chapter 4

Design Process

This chapter describes a embedded systems design process built around the metamodel mapping framework. The objective of the design process is to move from requirements to UML diagrams to a formal language specification that can be simulated and analyzed. The results of the simulation and analysis form the basis of feedback to refine the design until the system exhibits the desired properties. This process ends in early design, before code is actually written.

4.1 Design Process Data Flow

A *data flow* model shows how information moves through a system and is transformed by that system. Ovals represent transformations or processes where in flowing information is modified to produce an outflow. Information flow itself is represented by directed arrows, often labeled with the information content (unless it is obvious from the context). *Data stores*, written as a label between two horizontal parallel lines, denote a place where processed information is retained for input to other processes. *External information* is denoted by a labeled rectangle. Data flow models do not necessarily imply a particular processing order ot

Our

-system

Since th

is perfor

4.1.1

Figure 4

embedde

cases an

The

A use c

idealizat

interacts

rectangl

on the o

and cool

can be i

specific a

capturin

providin

One u

may aris

instances

order other than that required locally by the sequence of transformations.

Our design process is described as a data flow. The information flowing through the "system", or design process in this case, describes various aspects of an embedded system. Since there are numerous feedback loops where refinement of the embedded system model is performed, this depiction better conveys how the process works.

4.1.1 Use Case and Context Model

Figure 4.1 contains an overall data flow diagram for the process we use for developing embedded systems. The methodology begins with the parallel construction of UML *use cases* and a *context class model* from requirements statements.

The use case diagram documents *Use Cases* and is used to capture system requirements. A use case contains *actors*, the system, and the use cases themselves. An actor is an idealization of an external process, a person, a system, or some external component that interacts with the system. The system is represented in the use case diagram as a large rectangle containing oval use cases. Each use case is connected with lines to labeled actors on the outside of the system rectangle. Figure 4.2 shows an example use case for a heating and cooling system. The actors are the user, the furnace, and the A/C unit. Each use case can be instantiated as a *scenario*, often presented as a sequence diagram, which gives the specific actions for actors and the system in a particular situation. Use cases are useful for capturing high level requirements and the goals of the system. Increasingly, use cases are providing information useful for validating the system during system analysis.

One use case may have many scenarios based on the various conditions and events that may arise while the system is processing the required behaviors. Use case scenarios are instances of use cases themselves.

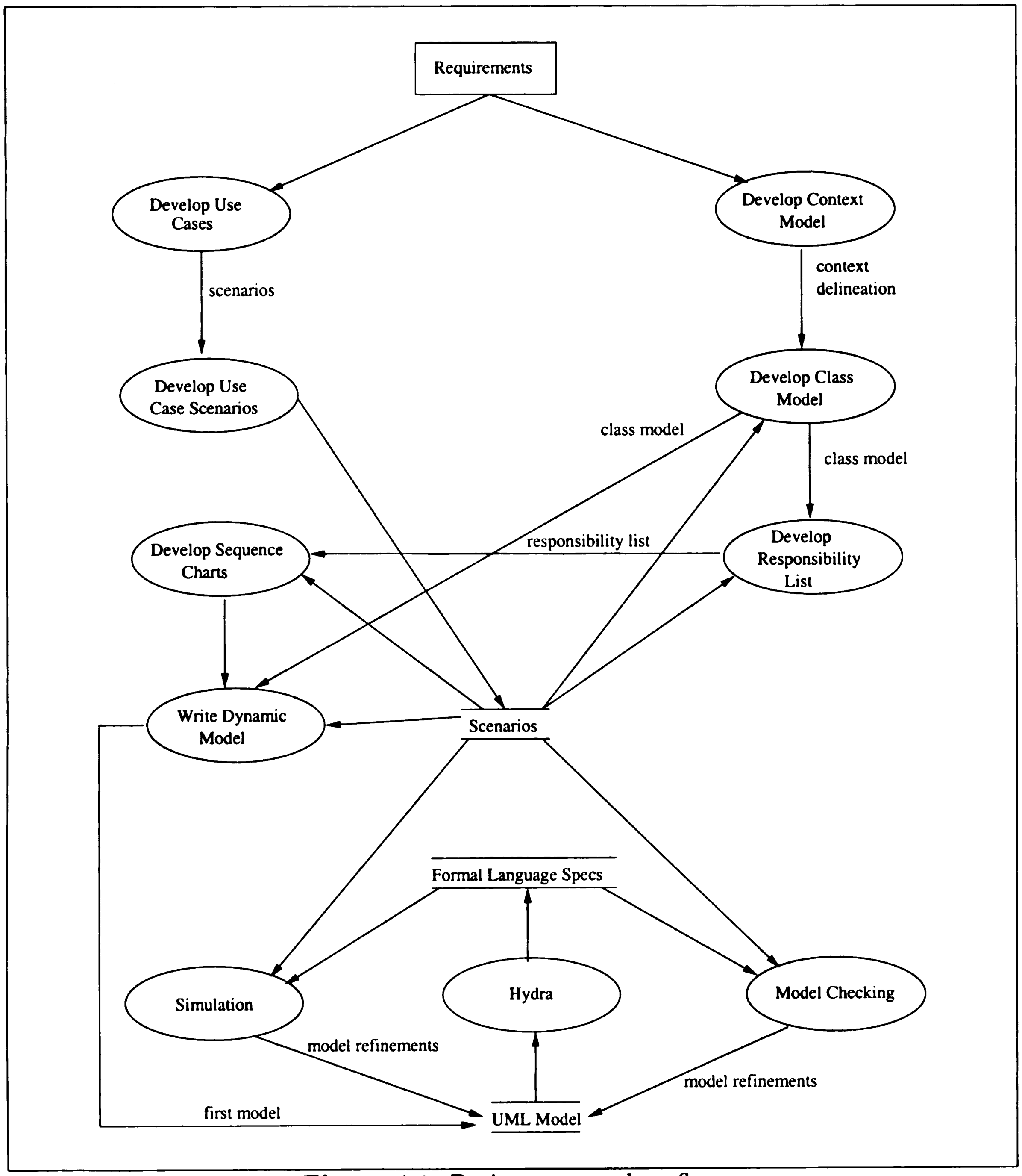


Figure 4.1: Design process data flow

Use cas

Nonetheles

Wang [35].

others [5] i

context mo

single class.

relationship

and cooling

in the syste

teraction be

the importar

of boundarie

perception/ac

boundary les

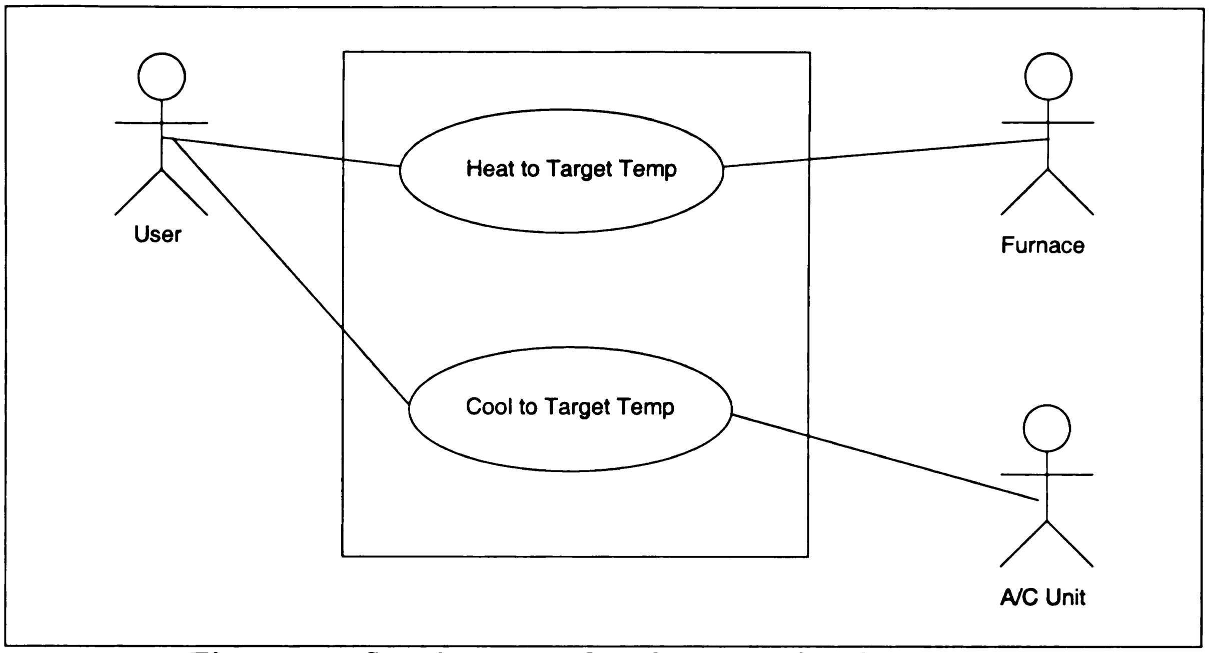
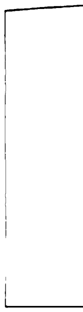


Figure 4.2: Sample use case for a heating and cooling system.

Use cases show to some extent the actors external to the system versus the system itself. Nonetheless we find an explicit context diagram useful. We follow the approach taken by Wang [35], Douglass [36], Ellis [37], Ward and Mellor [38], Hatley and Pirbhai [39], and others [5] in using a context model, which is currently not explicitly used in UML. The context model is essentially a class model in which the system itself is represented by a single class. The objects in the system's environment appear independently showing their relationship to the system itself. Figure 4.3 shows a simple context diagram for a heating and cooling system. The context model's contribution is a clear delineation of components in the system versus components external to the system and a high level view of the interaction between the external components and the system. Ward and Mellor [40] stressed the importance of setting early the system–environment boundary correctly. Multiple levels of boundaries are often suggested [37], such as *System*, *interfaces*, *sensors/actuators*, and *perception/action*, or simply *system*, *interface*, *world.* We believe the classification of the boundary less important than noting which objects are inside of the system's purview and those that



Fig

4.1.2 U

In the ne

circumsta

the class

an instan

the syste

We try to

4.1.3 C

After the

In an em

objects i

potential

evident.

those that are external.

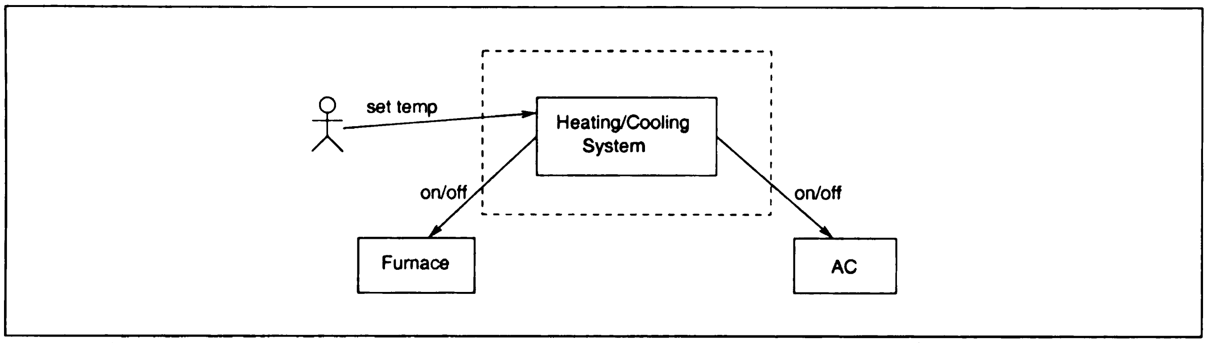


Figure 4.3: System Context Model for a hypothetical heating/cooling system.

4.1.2 Use Case Scenarios

In the next step of the process, we develop detailed use case scenarios for particular system circumstances. As can be seen in Figure 4.1, use case scenarios are important to developing the class model, and later in driving simulations and model checking. A use case scenario is an instance of a use case. A use case scenario develops one sequence of behaviors through the system, serving as a concrete example of system behavior in at least one, narrow case. We try to choose a scenario as close to an actual situation the system will be in as possible.

4.1.3 Class Model

After the use case scenario and context model development, we develop a UML class model. In an embedded system, there is usually an obvious one-to-one correspondence between objects in the real world and system classes. As the class model is developed, there are potential minor refinements to use case scenarios as the interaction of the classes becomes evident.

4.1.4

When

constru

in the l

The

are resp

importa

class an

exist. T

may mor

messages

the respo

latter cas

We b

because i

hardware

on the fir

4.1.5 S

It is ofte

CRL to d

diagram f

in Figure

across the

4.1.4 Class Responsibility List

When the initial class model is complete, a *Class Responsibility List* (CRL) for classes is constructed. The assignment of class responsibilities was not initially included in UML, but in the latest versions [41] setting class responsibility plays a prominent role.

The class responsibility list that we use is simply a listing of classes and what they are responsible for written in natural language. Relative responsibilities are particularly important. For example, in the heating system example, we determine that a controller class and a thermostat class are required, and that several possibilities for class interaction exist. The thermostat may deliver nothing more than a temperature level indication or it may monitor the temperature and control hysteresis parameters itself, thus providing only messages to demand heat and cooling. If the thermostat delivers a level, the *controller* has the responsibility for determining when the target temperatures require action, while in the latter case the *thermostat* carries this responsibility.

We believe setting class responsibilities is particularly important for embedded systems because it is not uncommon that sets of classes are implemented in physically separate hardware components. Carefully refining class responsibilities can have an overall impact on the final cost of the system and on its ability to be extended.

4.1.5 Sequence Diagrams

It is often useful at this point to employ the use case scenarios, the class model, and the CRL to develop one or more *sequence diagrams.* Figure 4.4 shows an example of a sequence diagram for three objects and an external actor. A sequence diagram shows, as is seen in Figure 4.4, interactions between objects on a relative time scale. The classes are listed across the top and time flows downward. Messages flow horizontally to and from the lines under the

The d

Control.

message *g*

between a

The d

top from a

entering i

the dotted

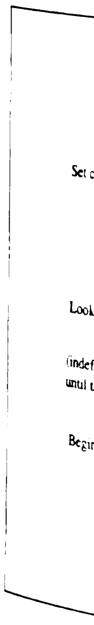
The se

sibilities a

use case s

diagram is

hundreds.



under the objects.

The diagram in Figure 4.4 shows an external actor first sending the message *set* to object *Control*. Object *Control* then sends two messages, *on* to class *Radar* followed quickly by message *getspeed* to class *Car*. *Radar* and *Car* respond, followed by a complex interaction between all three classes.

The dotted line across the sequence diagram delineates the beginning sequences at the top from a cyclic behavior pattern that eventually develops. The diagram shows the objects entering into a continuous cycle of interaction at some point in time. The interaction below the dotted line shows one possible message exchange sequence during the cycle.

The sequence diagram visually depicts complex interactions and helps ensure responsibilities are correctly assigned between objects, and lays out the interaction for typical use case scenarios in order to aid in developing the dynamic model for each class. This diagram is not a substitute for careful analysis, however. By its nature, only one of perhaps hundreds, or even thousands of interactions can be depicted.

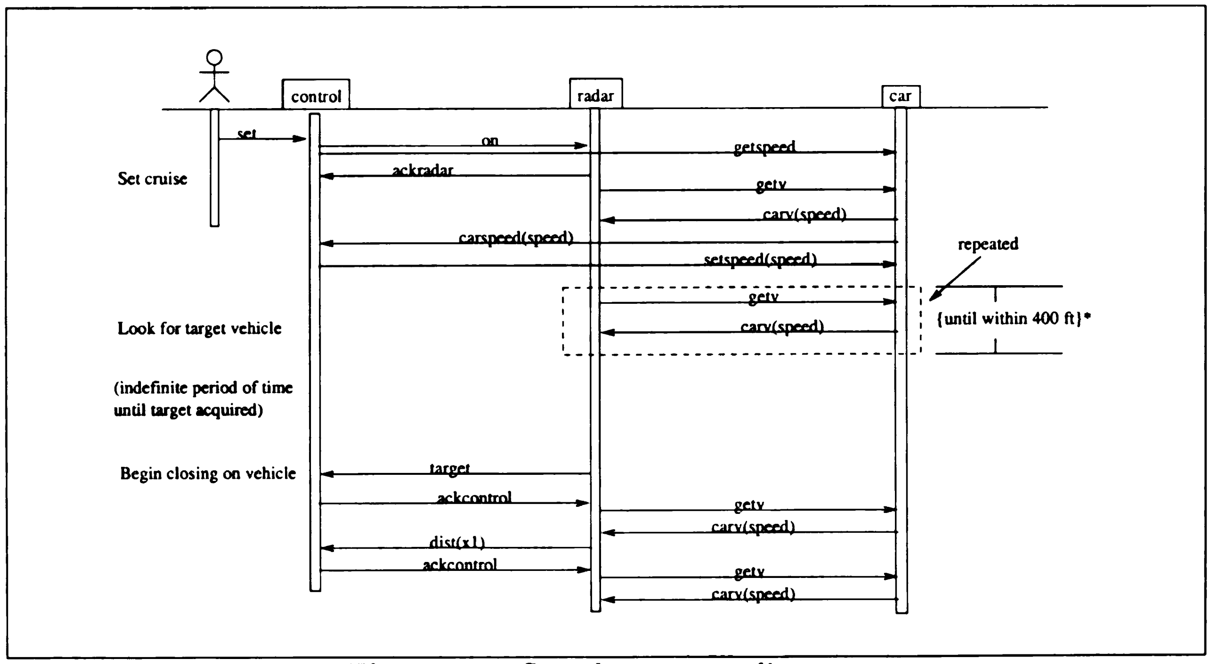


Figure 4.4: Sample sequence diagram.

4.1.6

We ne

the pr

models

classes.

objects

types.

It is

can be

obscure

4.1.7

In the ne

and Pro

Hydra to

as descri

If the

model ca

model eve

may run

Simula

scenarios.

in simula

retranslat

4.1.6 Dynamic Model

We next develop dynamic models for each class, based on the material developed so far in the process. Each dynamic model is constructed within the context of one class. Dynamic models for various classes must take into account the interactions, or signals sent between classes. The dynamic model must know not only that there is an association between two objects, but also the content of the message flow in terms of signal identities and parameter types.

It is reasonable at this stage to not fully develop each dynamic model. More behavior can be added as the overall model is refined. Adding too much detail at this point can obscure problems if, for example, the simulation steps fail to behave properly.

4.1.7 Simulation and Model Checking

In the next step, the class and dynamic models are translated to a formal language (VHDL and Promela are two choices used in this research). The translation is performed by the Hydra tool (Chapter 8), which itself implements metamodel to metamodel mapping rules as described in Chapters 3, 5, and 6.

If the language has simulation capabilities (Promela and VHDL do), the translated model can be executed. The first few executions serve to determine if the behavior of the model even closely approximates the desired behavior. The model may not run at all, or may run but exhibit undesirable characteristics, such as deadlock.

Simulation often uses use case scenarios to step the model through the most expected scenarios. The model should at least properly run the most common use case scenarios in simulation. Changes to the system are fed back into the UML models, the model is retranslated, and another simulation is performed. When the designer is reasonably sure the model

Earlier

refinement

is now, du

If mod

be checked

ploration,

that migh

to the mo

concurrenc

Model

form, "wh

is received

circumstan

Embed

in a stable

run would

once the c

4.2 C

For some a

semantics

The precur

required w

the model performs as expected, more rigorous analysis is possible with model checking.

Earlier we suggested writing an incomplete dynamic model for later completion through refinement and evolution. The appropriate time to refine the model to its final requirements is now, during simulation.

If model checking is available (Promela uses SPIN for model checking), the model can be checked for important properties. Since model checking involves exhaustive state exploration, rare deadlock sequences, for example, can be discovered during model checking that might be missed by simulation. Similarly, in a non-fair scheduling context (refers to the model process scheduling, not the management environment) using high levels of concurrency, process starvation is better detected with a model checker.

Model checking can also verify positive properties. Such properties are usually of the form, "when x happens, y always eventually happens". For example, when a brake signal is received, the vehicle controller must eventually react. This kind of analysis insures rare circumstances do not let important events slip by.

Embedded systems often have a control loop that monitors and maintains the system in a stable state. To test a control loop in simulation, an infinite (or very long) simulation run would be required. A model checker that can detect cycles can, however, verify that once the control loop is entered, there are no unexpected paths out of the loop.

4.2 Choosing Semantics

For some applications, when the designer is sufficiently sophisticated, our process allows the semantics produced by the Hydra translation of UML to a target language to be altered. The precursor of the need for altered semantics usually appears when a universal property required within the infrastructure of the model is uncovered during simulation or model checki

For

messag

sage is

fact, c

object.

action

inter-ob

the desi

the mod

guage ca

mapping

We d

target h

the sour

4.3 F

The diag

loop alter

The m

output of

could also

Since t

mated, we

checking.

For example, suppose the mapping to the formal target language does not provide for messages to be queued between objects. If the receiver object is not listening when a message is sent from one object to another, the message is lost. Non-queued semantics is, in fact, consistent with the informally described UML message passing semantics *within an object.* To continue the example, suppose a designer has built a model with complex interaction patterns such that inter-object messages are consistently lost. Further assume that inter-object queueing would solve the designer's problem. But, to make the model work, the designer will be forced to effectively build inter-object message queuing semantics into the model. Our mapping offers an alternative. The mapping from UML to the target language can be modified to assume queued messages between objects. In effect, the modified mapping assumes the object handling infrastructure includes inter-object queuing.

We described the designer above as "sophisicated" because the mapping from source to target has to be refined carefully so that it does not violate the homomorphism between the source metamodel and the target metamodel.

4.3 Feedback Loops

The diagram in Figure 4.1 contains many feedback loops between processes. A feedback loop altering semantics was described above.

The most obvious feedback loops are for refinement of the dynamic model given the output of simulation and model checking. It is possible, however, that class responsibilities could also be modified, causing a refinement of the class and dynamic models.

Since the step from a UML model to an executable or ready-for-analysis model is automated, we anticipate frequent feedback loops and an evolutionary style of development.

4.4 Relation of the methodology to Formalization Integration

The Use Cases and Object Context Model employed early in the development process are at best semi-formal methods, but most likely informal. The transition to use case scenarios, UML class diagrams, and UML dynamic models produce semi-formal specifications for the system. The transition to a formal specification occurs when the model is translated into the target language. The translation from UML class and dynamic models to the target language is accomplished through an homomorphism that maps metamodels describing UML models to the target language, which are formal specifications of the model. Since our homomorphisms are rigorously defined, it is possible to implement the mapping process in a software tool (The tool is Hydra. See Chapter 8.). Use case scenarios provide the equivalent of putative theorems to test the specification and the results of the target language simulation can then be used to refine the dynamic model. Similarly, use case scenarios can be used to write temporal logic formulas to verify properties through model checking. The sequence of actions clearly follows the semi-formal to formal (in parallel) assisted scenario (discussed in Section 2.5).

Chapter 5

UML to VHDL Mapping Rules

In this chapter we apply the framework described in Chapter 3 to the construction of a homomorphism and mapping rules for mapping UML to VHDL. The homomorphic function is described first, followed by a description of the mapping rules.

5.1 UML to VHDL Diagram Homomorphisms

Figure 5.1 contains the combined metamodel for UML class diagrams and behavior diagrams. The classes in the UML metamodel are enclosed in rectangles in the diagram. The portion of the metamodel attributable to the class model is enclosed in dotted lines. The list of normal relationships (*i.e.* not aggregations or generalizations) generated by the formalization of the UML metamodel is shown in Figure 5.2. The occurrences of the hasComp() and hasPart() predicates generated by the formalization of the UML metamodel are show in Figure 5.3. As described in Chapter 3, each class and relationship in the UML metamodel must have a mapping to a component in the VHDL metamodel. Further, the mapping must be consistent so that relationships between classes are preserved.

The strategy for writing the VHDL metamodel and formulating the UML to VHDL

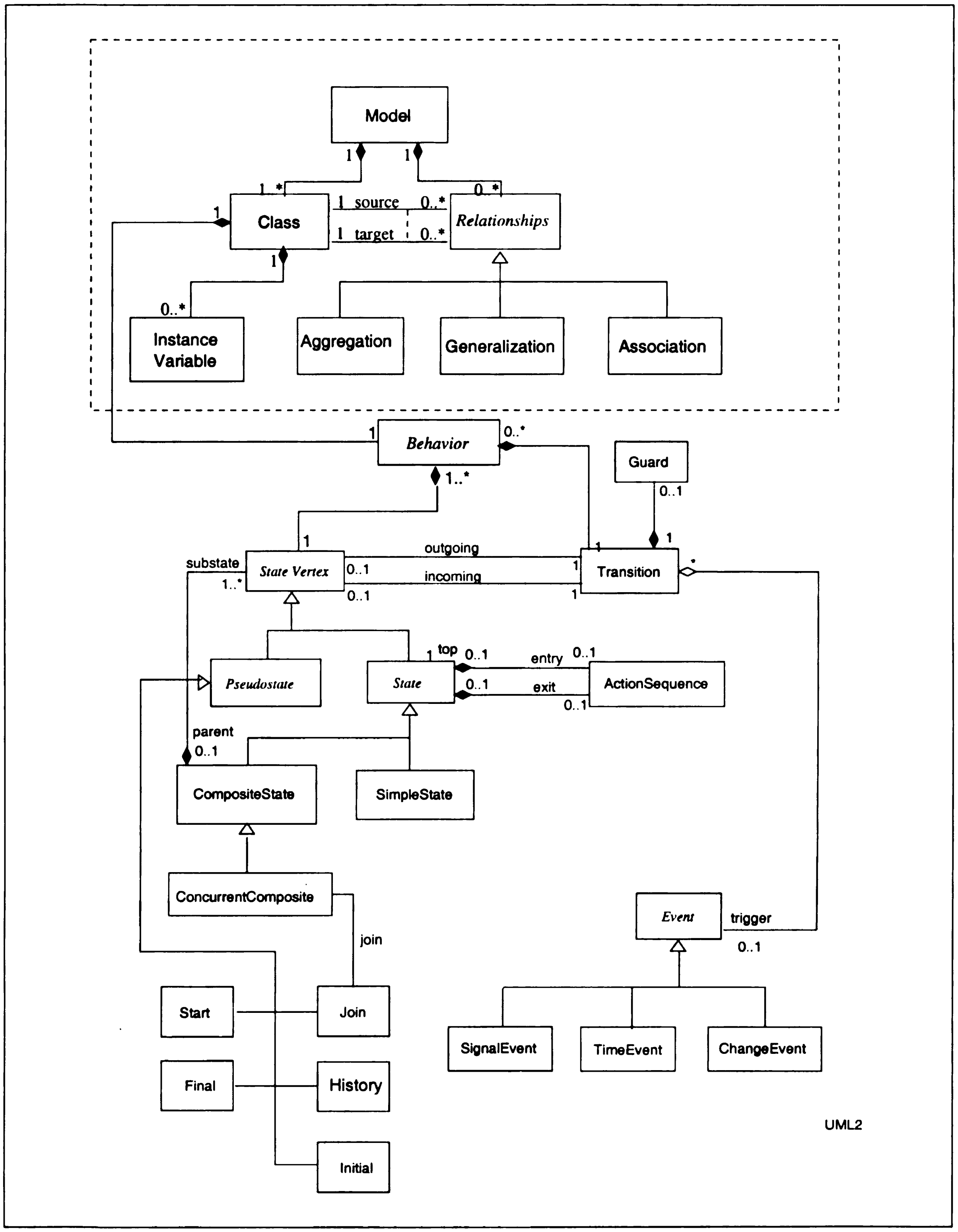


Figure 5.1: Unified UML class/dynamic diagram metamodel.

mapping

Section 3

UML me

in the V

statemen

when the

the VHD

mapping

1. Ma

   mo

2. Wh

3. As

   in t

The l

the forma

hasComp

are show

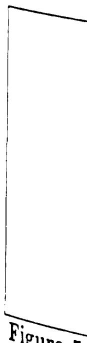


Figure 5.
the formal

mapping rules is as follows: First, the VHDL metamodel is constructed. As described in Section 3.2.3, we begin by constructing a metamodel that has a structure as close to the UML metamodel as possible. Templates are constructed, where necessary, for the classes in the VHDL metamodel. As described earlier, a *template* is a parameterized series of statements in the target language that represents a target language metamodel class. Next, when the VHDL metamodel is complete, the homomorphism from the UML metamodel to the VHDL metamodel is constructed. Finally, when the VHDL metamodel is complete, the mapping rules are written. Three goals drive the construction of the mapping rules:

1. Map every UML class and relationship to a class and relationship in the VHDL metamodel.

2. When a template is used, implement the desired semantics in terms of VHDL.

3. As a rule is written, ensure that relationships in the UML metamodel are preserved in the target metamodel.

The list of normal relationships (*i.e.*, not aggregations or generalizations) generated by the formalization of the UML metamodel is shown in Figure 5.2. The occurrences of the hasComp() and hasPart() predicates generated by the formalization of the UML metamodel are show in Figure 5.3.

```
source(Class,Relationships)
target(Class,Relationships)
outgoing(StateVertex,Transition)
incoming(StateVertex,Transition)
entry(State,ActionSequence)
exit(State,ActionSequence)
join(ConcurrentComposite,Join)
```

Figure 5.2: The list of non-aggregate and non-generalization relationship predicates from the formalized representation of the UML metamodel.

Figure
tion of t

The

As descr

UML me

consisten

classes a

5.1.1

Figure 5.

ure 5.6 sh

To de

non-aggre

VHDL m

the mapp

Figure 5.

preserved

homomorp

hasComp($Model, Class$)
hasComp($Model, Relationships$)
hasComp($Class, InstanceVariable$)
hasComp($Behavior, Transition$)
hasComp($CompositeState, StateVertex$)
hasPart($Transition, Event$)

Figure 5.3: The list aggregation relationships predicates from the formalized representation of the UML metamodel.

The VHDL metamodel resulting from the process described above is shown in Figure 5.4. As described in Chapter 3, the homomorphism must map each class and *relationship* in the UML metamodel to a component in the VHDL metamodel. Further, the mapping must be consistent with the homomorphic properties of the function so that relationships between classes are preserved.

5.1.1 Homomorphic Mapping

Figure 5.5 shows the mapping of UML metamodel classes to VHDL metamodel classes. Figure 5.6 shows the mapping of the relationships from one metamodel to another metamodel.

To demonstrate completeness, Figure 5.5 and Figure 5.6 show that every class and non-aggregate relationship in the UML metamodel maps to a class or relationships in the VHDL metamodel. Figure 5.6 shows that each non-aggregate relationship is preserved by the mapping in the VHDL metamodel, satisfying the requirement for a homomorphism. Figure 5.7 further shows that the hasComp() and hasPart() relationship predicates are preserved in the VHDL metamodel, completing the demonstration that the mapping is homomorphic.

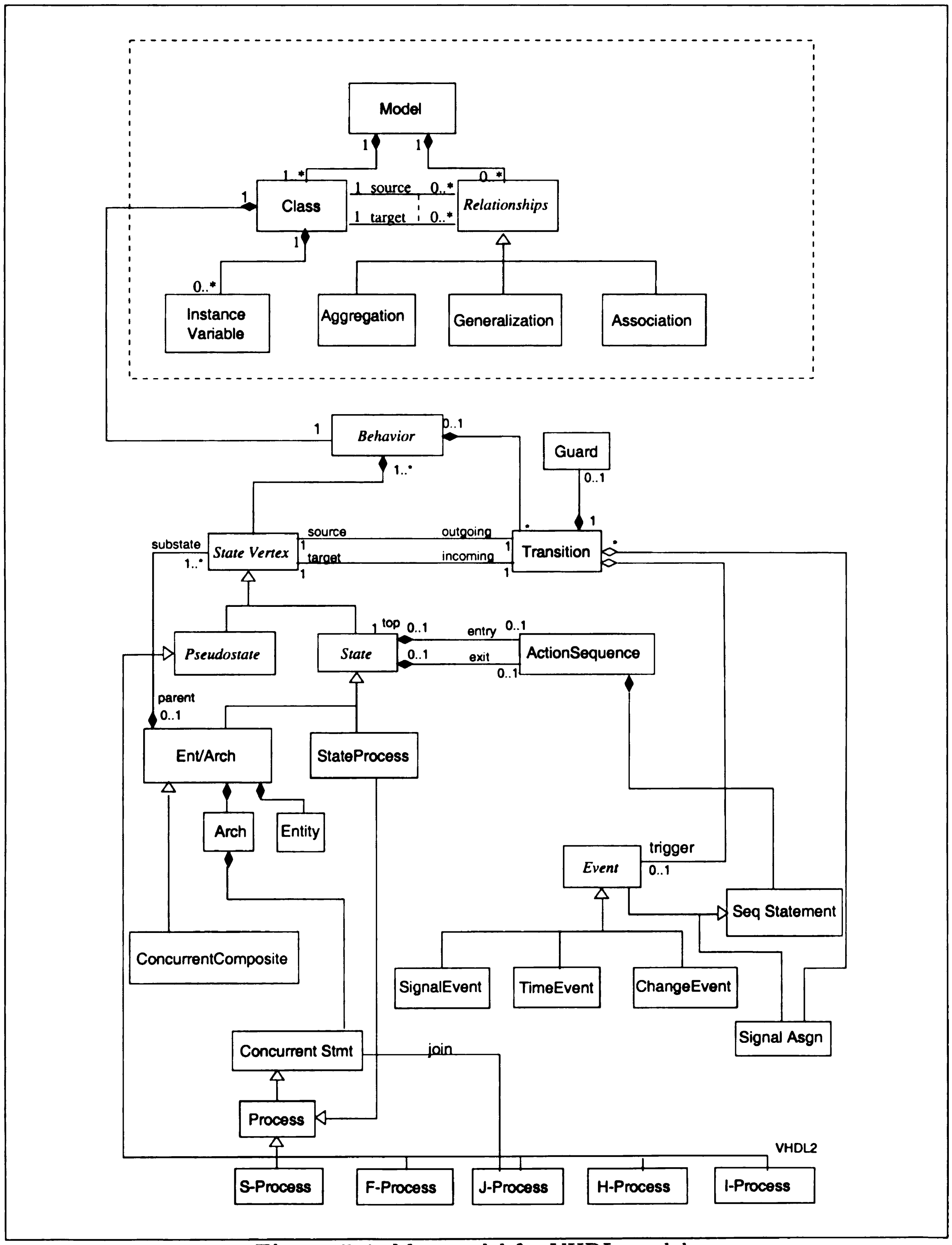


Figure 5.4: Metamodel for VHDL models

Figure 5.
model to t

UM

outgo
incom
e

join
Figure 5.
the UML n

UML Metamodel Class		VHDL Metamodel Class
Model	⟶	Model
Class	⟶	Class
Relationships	⟶	Relationships
InstanceVariable	⟶	InstanceVariable
Aggregation	⟶	Aggregation
Generalization	⟶	Generalization
Association	⟶	Association
Behavior	⟶	Behavior
Guard-	⟶	Guard
StateVertex	⟶	StateVertex
Transition	⟶	Transition
Pseudostate	⟶	Pseudostate
State	⟶	State
ActionSequence	⟶	ActionSequence
CompositeState	⟶	Ent/Arch
ConcurrentComposite	⟶	ConcurrentComposite
SimpleState	⟶	StateProcess
Start	⟶	I-Process
Final	⟶	S-Process
Join	⟶	J-Process
History	⟶	H-Process
Event	⟶	Event
SignalEvent	⟶	SignalEvent
TimeEvent	⟶	TimeEvent
ChangeEvent	⟶	ChangeEvent

Figure 5.5: The definition of the homomorphic mapping of classes from the UML metamodel to the VHDL metamodel.

UML Metamodel Predicate		VHDL Metamodel Predicate
source(Class,Relationships)	⟶	source(Class,Relationships)
target(Class,Relationships)	⟶	target(Class,Relationships)
outgoing(StateVertex,Transition)	⟶	outgoing(StateVertex,Transition)
incoming(StateVertex,Transition)	⟶	incoming(StateVertex,Transition)
entry(State,ActionSequence)	⟶	entry(State,ActionSequence)
exit(State,ActionSequence)	⟶	exit(State,ActionSequence)
join(ConcurrentComposite,Join)	⟶	join(ConcurrentComposite,J-Process)

Figure 5.6: The definition of the homomorphic mapping of relationship predicates from the UML metamodel to the VHDL metamodel.

ha

hasCor

F

5.2 U

This sect

of the me

to preser

An in

classes ar

Many

of the rul

name uni

true, then

5.2.1 C

Figure 5.8

formalizat

parts of C

ciation $R1$

association

UML Aggregate Relationship		VHDL Aggregate Relationship
hasComp($Model, Class$)	$\longrightarrow$	hasComp($Model, Class$)
hasComp($Model, Relationships$)	$\longrightarrow$	hasComp($Model, Relationships$)
hasComp($Class, InstanceVariable$)	$\longrightarrow$	hasComp($Class, InstanceVariable$)
hasComp($Behavior, Transition$)	$\longrightarrow$	hasComp($Behavior, Transition$)
hasComp($CompositeState, StateVertex$)	$\longrightarrow$	hasComp($Ent/Arch, StateVertex$)
hasPart($Transition, Event$)	$\longrightarrow$	hasPart($Transition, Event$)
hasComp($Transition, Guard$)	$\longrightarrow$	hasComp($Transition, Guard$)
hasComp($State, ActionSequence$)	$\longrightarrow$	hasComp($State, ActionSequence$)

Figure 5.7: The preserved hasComp and hasPart relationship predicates.

5.2 UML to VHDL Mapping Rules

This section describes the mapping rules from UML to VHDL. The rules are stated in terms of the metamodel to metamodel mapping. As described earlier, the mapping is constrained to preserve the homomorphic mapping described above.

An important part of the UML class model is relationships. All relationships between classes are provided in VHDL by signals, therefore, *relationships* map to *signals*.

Many VHDL compilers use a common name space for named constructs, therefore, some of the rules include naming conventions for names found within the model in order to ensure name uniqueness. We assume that state names are unique within the model. If this is not true, then it can be easily rectified by composing state names with class names.

5.2.1 Class Diagram Formalizations

Figure 5.8 gives a sample class model that will be used to help illustrate the class diagram formalization rules. The diagram contains four classes: A, B, C, and D. Class A contains parts of C (the expression $\exists x, y\, (T_A(x) \wedge T_C(y) \wedge hasPart(x, y))$ is true), and has an association $R1$, with class B. Class B is a supertype for class D, and therefore, there is also an association $R1$ between class A and class D (the expression $\exists x, y\, (T_A(x) \wedge T_D(y) \wedge R1(x, y))$

is true). C

Each c

obj_*class*.

the only w

is true). Class *A* contains one instance variable, *vara*, declared as an integer.

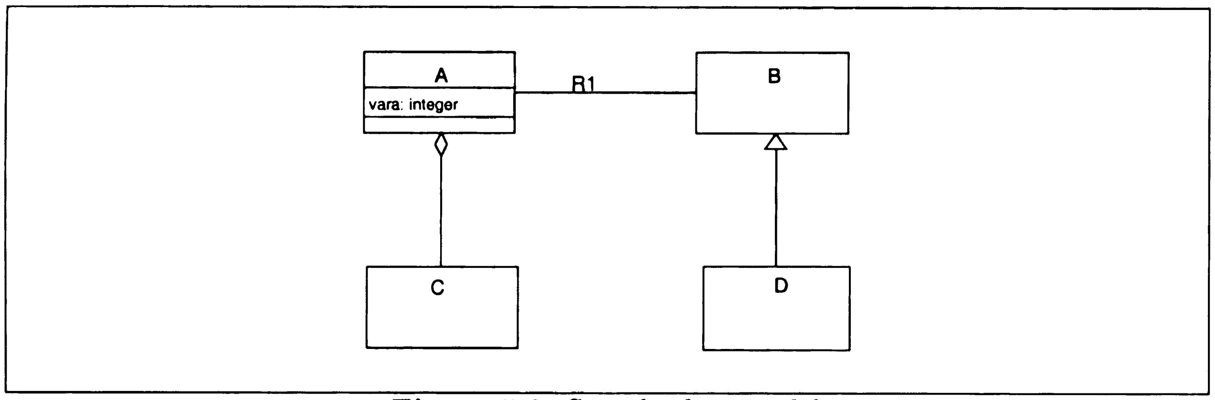


Figure 5.8: Sample class model

Each class is represented as a VHDL entity and architecture pair. The pair is named obj_*class*. Associations between classes are represented in VHDL with signals that provide the only way in VHDL for propagating information between entities.

Rule V
Formal
followi

e

e

ar
be

–
e

sig_1
predica
mappec
ins_st
the IN
Relatio
This ru
are pres
End of

Class ir

rules will e

tity/archit

to be glob

declarative

Rule VHDL 1
Formalize each object of class **CLASS** as an entity/architecture pair according to the following template:

```
entity obj_CLASS is
    port(state:  inout rs_st;
        sig_1:  inout sig_1type;
        .
        .
        .
        sig_n inout sig_ntype;
        instate:  out st;
        ins_state_1:  in st;
        .
        .
        .
        ins_state_n:  in st);
end entity;

architecture abstract of obj_CLASS is
begin
    instate <= state when state/=st'(none);
- Statements for states and composite states go here
end abtract;
```

`sig_1` through `sig_n` are inter-object signals. `instate` is used for the $IN()$ state predicate to convey to other entity architecture pairs (that represent various other mapped templates) the simple states this entity/architecture is in. `ins_state_1` through `ins_state_n` are incoming VHDL signals that simple states of this object can test for the $IN()$ predicate.
Relationship to Homomorphism
This rule satisfies the mapping of **Class** to **Class**. The *source* and *target* relationships are preserved by including `signals` in the VHDL class definition.
End of Rule VHDL 1.

Class instance variables must be accessible from any state within the object. As later rules will explain, a number of entity/architectures can be generated in addition to the entity/architecture representing the class itself. Consequently, instance variables are required to be global to each entity/architecture comprising the object. The place designated for declarative components common to a number of entity/architectures is the VHDL **package**, and th

In a

known

as is ex

details

function

placed i

and the VHDL **package body** contains procedures common to each entity/architecture.

In addition to instance variables, the set of state names used within the object must be known to the entire object. The signal called **state** is used for transitions between states, as is explained in Rule VHDL 5, below. For technical reasons related to VHDL modeling details (explained in Section 2.2.3), this signal must have associated with it a *resolution function* that is common to the entire object. The declaration of the resolution function is placed in a **package body**.

Ru

Cr

for

the

are

```
e
-
p
en
```

Where

each en

Relatio

This rul

aggrega

causing

End of

Figure 5

in Figure 5.

Rule VHDL 2
Create a class **package** and **package body** named pk_*classname* to contain declarations for instance variables, the list of state names, and the resolution function required for the bus signal state (introduced in Rule VHDL 5). The **package** and **package body** are constructed according to the following template:

```
package pk_CLASS is
- State signal names
    type st is (none, state_1, ...  , state_n);
- An array required for the resolution function
    type st_a is array (natural range <>) of st;
- Instance variables
    shared variable var_1 <type>;
    .
    .
    .
    shared variable var_n <type>;
end package pk_CLASS;
- Resolution function for state bus
package body pk_CLASS is
    function resolve_st
        (v :  in st_a) return st is
        variable i :  natural;
    begin
        for i in v'range loop
            if v(i) /= st'(none) then
                return v(i);
            end if;
        end loop;
        return st'(none);
    end function;
end package body;
```

Where **CLASS** is the name of the class, and var_i is an instance variable. Precede each entity/architecture with a **use work.pk_CLASS.all** statement.
Relationship to Homomorphism
This rule complies with the mapping of **InstanceVariable** to **InstanceVariable**. The aggregation is preserved by including the **package** above the definition of the class, thus causing inclusion of the instance variables in the class.
End of Rule VHDL 2.

Figure 5.9 gives the VHDL generated for class A in Figure 5.8 by Rule VHDL 2. Line 3 in Figure 5.9 is the partial list of state names used for state transitions. The rest of Line 3 is filled in as states are added to the specification. Lines 11 through 21 declare the VHDL resolution function that enables the resolved bus.

Figure 5.10 shows the **architecture/entity** pair generated by Rules 1 and 2. The **entity/architecture** is skeletal because a number of details that are obtained from the dynamic model have not been specified. Figure 5.10, Line 5 shows the **port** assignment that corresponds to the association $R1$ in Figure 5.8. The instance variable *vara* is declared in the package in Figure 5.9. Just as with a class, this entity/architecture may be instantiated as many times as required.

```
1   package pk_A is
2   - State signal names
3       type st is (none);
4   - An array required for the resolution function
5       type st_a is array (natural range <>) of st;
6   - Instance variables
7       shared variable vara:  int;
8   end package pk_A;
9   - Resolution function for state bus
10  package body pk_A is
11      function resolve_st
12          (v :  in st_a) return st is
13          variable i :  natural;
14      begin
15          for i in v'range loop
16              if v(i) /= st'(none) then
17                  return v(i);
18              end if;
19          end loop;
20          return st'(none);
21      end function;
22  end package body;
```

Figure 5.9: The VHDL package statements generated for each class.

Class specialization conceptually allows the designer to duplicate the parent class and modify it to become a subclass. Such duplication implies that the behavior, along with relationships and instance variables, are also reproduced. In theory, a subclass object should be able to simulate a parent class object, but the meaning of specialization when the duplicated parent class behavior is radically modified, or even replaced, is not clear.

Fortunately, high levels of specialization are not a characteristic of embedded systems.

```
use std.textio.all;
use work.easyio.all;
use work.pk_A.all;
entity obj_A is
    port(r1:  inout integer);
end entity;
architecture abstract of obj_A is
begin
    l1:  entity obj_C(abstract)
        port map(r1=>r1);
end architecture;
```

Figure 5.10: Entity/Architecture pair from diagram 5.8 class *A*

Therefore, generalization is formalized as duplication of all the mapped components of a class followed by modifications as required to create the subclass definition. The duplication includes all entity/architectures for the object and reuse of the **package** for the class.

Rule VHDL 3
Class specialization is formalized as duplication of the parent class components, with appropriate renaming of the components that are named by the class, followed by modification, as required.
Relationship to Homomorphism
This rule preserves the mapping of **Generalization** to **Generalization**.
End of Rule VHDL 3.

The VHDL for an object of class *D* generated by Rule VHDL 3 from Figure 5.8 is shown in Figure 5.11. Referring to Figure 5.8, class *D* inherits the association *R*1 from class *B*, therefore, the **port** declaration appears in the VHDL for class *D*.

```
use std.textio.all;
use work.easyio.all;
use work.pk_D.all;
entity obj_D is
    port(r1:  inout integer);
end entity;
architecture abstract of obj_D is
begin
end architecture;
```

Figure 5.11: VHDL generated for an instance of class *D* from Figure 5.8

Aggregation in a class model indicates that a class (the "part" class) plays a role in the function of the aggregate class. The class diagram does not contain enough detail to specify exactly how the aggregation is to occur. For example, in some cases, communication may be required between the aggregate and the parts. In other cases, the aggregate may simply keep track of its parts. The details of the aggregation are defined by the specification of the dynamic model. In general, aggregation implies inter-object communication between the aggregate and the part. Aggregation may also require the aggregate to instantiate the part.

Rule VHDL 4
Class aggregation is formalized as follows: In the collection **architecture** body, instantiate as many instances of the part entity as required by the instantiation of a particular model. The **port map** on the instantiation must match the **port** declaration for the subclass **entity**. A unique label must be provided on each entity instantiation statement.
Relationship to Homomorphism
This rule compiles with the mapping of Aggregation to Aggregation.
End of Rule VHDL 4.

Lines 9 and 10 in Figure 5.10 show the VHDL statement generated by Rule VHDL 4 to instantiate one instance of class C.

5.2.2 Dynamic Model Formalizations

This section describes the mapping rules for the portions of the unified metamodel corresponding the the dynamic model.

Figure 5.12 shows a sample dynamic model intended to illustrate the formalization rules pertaining to the dynamic model portion of the UML diagram. It consists of a top state *TOP* that contains one simple state F and a composite concurrent state denoted by *CP1* and *CP2*. The dashed line between *CP1* and *CP2* indicates concurrent execution of these two states. Composite state component *CP1* contains a *history state*, denoted as a circle containing an "H", and three simple states. Composite state component *CP2* contains one simple state, *E*, and a composite state, *D*. The execution threads from *CP1* and *CP2* are rejoined in a *join* construct, denoted as a heavy vertical bar. The transitions for events *T4* and *T5* lead to the join pseudo-state. State *F* is activated only after both transitions leading into the join have occurred.

The initial state of *TOP* is jointly the concurrent composite states *CP1* and *CP2* as denoted by the solid circle indicating a transition to the concurrent composite state boundary. Similarly, the initial state of *CP2* is state *E*. State *E* is entered every time a transition to the boundary of *CP2* occurs.

The history state in *CP1* is a special initial state that remembers the current state when the composite state is exited. For example, if *CP1* is in state *B* when event *T5* occurs, then the next time *CP1* is entered, state *B* will be made the current state. When there is no previous state, the default state as indicated by the transition arc (*A*, in this case) is entered.

Background for Simple State Template

Before stating the rule for generating a simple state in VHDL, the mechanism for implementing a state, a state transition, and signalling an event is discussed.

States.

A UML *simple state* is represented in VHDL as a **process**. The process is named s_*statename*. Activation of the process (*i.e.*, state) is accomplished through a special "state bus" signal named *state*. Signal *state* is declared as an enumerated type named *st* listing the name of every state in the object, including composite states, as well as the distinguished name *none*. Processes representing states all have a similar structure: The initial

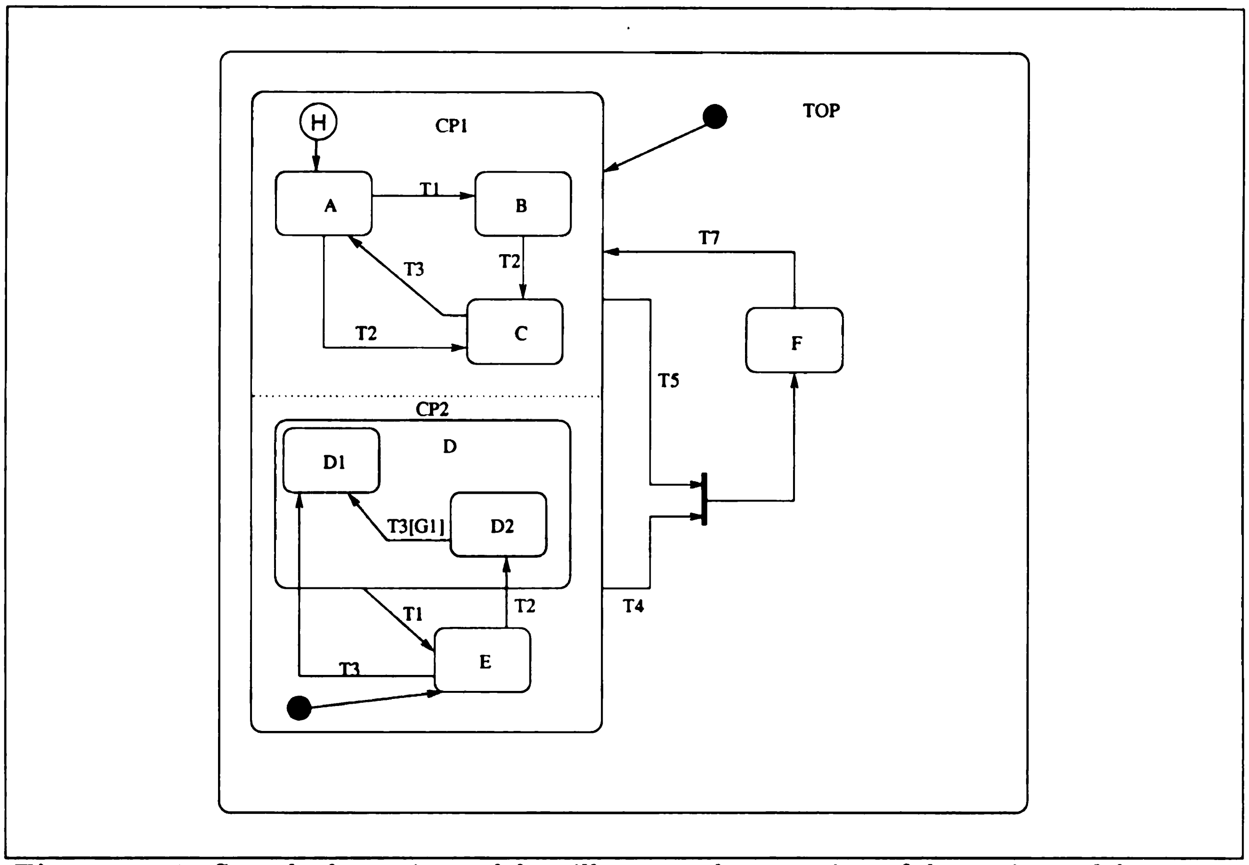


Figure 5.12: Sample dynamic model to illustrate the mapping of dynamic model components.

wait statement waits until the *state* signal contains the name of the state. At that point, the process is activated and can perform the actions required by the state diagram. Events and messages flowing between objects and states are carried by signal declarations on the **entity port** declaration as described in Rule VHDL 1.

Events.

All events within the state diagram are represented as signals that assume a momentary value. All transitions for a state are mapped to **wait** statements inside of a **loop** construct. When the process is activated after an event on the **wait** occurs, a nested set of **if** statements determine which event occurred. Next, the guard, if any, is checked. If the guard is false, control falls through to the bottom of the **loop**, causing the **wait** to be re-initiated.

Transitions.

State transitions are represented as signal assignments. Each transition assigns the name of the destination state followed by 'disconnecting' the state signal driver by using the VHDL 'null' signal assignment. The structure of the "state–process" requires that the signal assignment be the last logical action of the process so that the process resumes waiting for its name on signal *state*.

Rule VHDL 5
For state named *statename*, events E_1, E_2, ... , E_n guarded by guards GD_1, GD_2, ... , GD_n, eventless transitions guarded by guards EGD_1, EGD_2, ... , EGD_n, and destination states $nextstate_1$, $nextstate_2$, ... , $nextstate_n$, formalize each simple state as a **process** according to the following template:

```
s_statename: process
begin
      wait until state=statename;
      { entry: processing statements }
      if $EGD_1$ then
         state <= $nextstate_1$;
      elsif $EGD_2$ then
         state <= $nextstate_2$;
         ⋮
      elsif $EGD_n$ then
         state <= $nextstate_n$;
      endif
      loop
         wait until $E_1$ or $E_2$ or ... $E_n$;
         if $E_1$ and $GD_1$ then
              state <= $nextstate_1$, null after 1 fs;
         { Other transition actions }
         exit;
         elsif $E_2$ and $GD_2$ then
              state <= $nextstate_2$, null after 1 fs;
         { Other transition actions }
         exit;
         end if;
      end loop;
      {exit: processing statements }
end process;
```

If there are no event-less transitions from this state then the **if** checking guards EGD_i above the **loop** can be eliminated.
Relationship to Homomorphism
This rule defines the **StateProcess** template, which is the homomorphic target of the **SimpleState** class. The **StateProcess** is a *State* and *StateVertex*. **StateProcess** has relationships with **ActionSequence** and **Transition** (through the *StateVertex* class). These relationships are preserved through the constructs **if** E_n for events and guards. Action sequences are preserved within the same **if** statement.
End of Rule VHDL 5.

Figure 5.13 shows the **process** generated by Rule VHDL 5 for state *D2* from the example in Figure 5.12. The **loop** statement on Line 5 begins the section that waits for one of the events $T1$, $T3$, or $T4$. The transition for event $T3$ is guarded, which generates an extra conjunct on the **if** statement on Line 9. There are no event-less transitions in this example. Lines 9, 10 and 11 in Figure 5.13 also show the VHDL statement generated by Rule VHDL 5 for the transition from state $D2$ to state $D1$ guarded by $G1$ shown in Figure 5.12.

```
1       s_d2:  process
2       begin
3           wait until state=st'(d2);
4           say("In state d2");
5           loop
6               wait until t3
7                   or t4
8                   or t1;
9               if t3 and g1 then
10                  state <= st'(d1), null after 1 fs;
11                  exit;
12              elsif t4 then
13                  state <= st'(join1), null after 1 fs;
14                  exit;
15              elsif t1 then
16                  state <= st'(e), null after 1 fs;
17                  exit;
18              end if;
19          end loop;
20      end process;
```

Figure 5.13: Example of the generated state *D2*

Composite States.

Composite states are mapped to **entity architecture** pairs named cs_*statename*. The **entity port** description inherits all the signals from its parent state as well as other special signals required for messages between states, and signals required for the $IN()$ 'in state' predicate. Signal *state* is declared in the composite state to be type **inout** to permit bi-directional passing of state information between parent and child.

The $IN()$ predicate permits one concurrent thread to determine the current state of some other, parallel component. The $IN()$ predicate is represented in our mapping with an extra set of signals and ports that communicate the current state of a concurrent component to the other components.

Single threaded and concurrent composite states are mapped differently in order to handle one thread, versus multiple threads of control. The single threaded version of a composite state is discussed first.

A single threaded composite state is mapped to an entity/architecture instantiated by the composite state's parent. The value of the *state* signal is conveyed to the child composite state by connecting the *state* signals in the parent's **port map**. This allows the child composite state to observe every state transition the parent makes, including transitions into the child. When a child composite state observes it's name (through a **wait** statement, as in Rule VHDL 5), that state becomes the current state. If the composite state has an initial state then that state simply listens for the name of the composite state and then transitions to the default initial state.

Rule VHDL 6
Formalize each single threaded composite state named **CSTATE** according to the following template:

```
entity cs_CSTATE is
    port(state:  inout rs_st;
        sig_1:  inout <type>;
        .
        .
        .
        sig_n:  inout <type>;
        lsig_1:  inout <type>;
        .
        .
        .
        lsig_n:  inout <type>;
        ins_CS1:  in st;
        .
        .
        .
        ins_CS2:  in st;
        instate:  out st);
end entity;

architecture abstract of cs_CSTATE is
begin
- the body of the composite state follows. Contents may
- include simple states, composite states, concurrent composite
- states, and pseudo-states.
end abstract;
```

where `state` is the state signal per Rule VHDL 5, `sig_i` are the inter-object signals per Rule VHDL 1, `lsig_i` are intra-object signals, `ins_CS_i` are the signals for the $IN()$ predicate, and `instate` is the signal conveying this entity/architecture's current state to the rest of the object.
Relationship to Homomorphism
This rule complies with the mapping **CompositeState** to **Ent/Arch**. The template defined responds to state transitions and therefore is a *State* and a *StateVertex*. Other **StateVertex** components are included below the **begin** statement in the template in compliance with the substate aggregation. The **signal** declarations provide the links for included **StateProcesses**, which are **StateVertices**.
End of Rule VHDL 6.

Rule VHDL 7
In the parent composite state or object, instantiate the composite state as follows:

```
l1:  entity cs_CSTATE(abstract)
    port map(state=>state, sig_1=>sig_1, ..., sig_n=>sig_n,
            lsig_1=>lsig_1, ..., lsig_n=>lsig_n,
            ins_CS1=>ins_CS1,
            .
            .
            .
            ins_CSTATE=>instate,
            .
            .
            .
            ins_CS_n=>ins_CS_n);
```

where the signals are as in Rule VHDL 6. Note that signal `in_CSTATE` connects to the child `instate` signal.
Relationship to Homomorphism
This rules specifies how to satisfy the outgoing and incoming relationships between *StateVertex* and **Transition** and the *StateVertex* is a composite state.
End of Rule VHDL 7.

Figure 5.14 shows the entity portion of composite state *D* in Figure 5.12 generated by Rule VHDL 6. The body of *D* in Figure 5.14 below line 20 consists of processes for states *D1* and *D2* generated according to rule Rule VHDL 5. Figure 5.15 shows how Rule VHDL 7 generates the parent state's (*CP2*) instantiation of composite state *D*.

Concurrent States.

Concurrent state components are denoted in UML models by partitioning the composite state rectangle with dotted lines and represent concurrent execution of all the partitioned parts. To create multiple execution threads in VHDL, a separate state signal must be created for each thread and all of the signals normally connected to a composite state must be connected to each composite state component. Creating separate state signals is ac-

```
use std.textio.all;
use work.easyio.all;
use work.pk_top.all;
entity cs_d is
    port(state:  inout rs_st;
        t1:  inout boolean;
        t2:  inout boolean;
        t3:  inout boolean;
        t4:  inout boolean;
        t5:  inout boolean;
        t6:  inout boolean;
        t7:  inout boolean;
        ins_cp1:  in st;
        ins_cp2:  in st;
        instate:  out st);
end entity;

architecture abstract of cs_d is
begin
- body composite state follows...
```

Figure 5.14: Example entity and architecture for composite state *D*

```
1   architecture abstract of cs_cp2 is
2   begin
3       instate <= state when state/=st'(none);
4       l1:  entity cs_d(abstract)
5           port map(state=>state, t1=>t1, t2=>t2, t3=>t3, t4=>t4, t5=>t5,
6           t6=>t6, t7=>t7, ins_cp1=>ins_cp1, ins_cp2=>instate,
7           instate=>ins_d);
```

Figure 5.15: The instantiation of composite state *D* as contained in parent state *CP2*. Line 3 shows the mapping for the 'in state' predicate for state *CP2*.

complished by declaring one state signal per (concurrent) component in the parent and connecting each of the newly declared signals to the child's *state* signal. The parent state containing concurrent components instantiates the components as in the non-concurrent case, but instead of mapping the *state* signal in the child to the *state* signal in the parent, the **port map** statement maps *state* to the locally declared signals, thus effectively creating multiple threads of control. Control is then passed to each concurrent component by assigning the name of its corresponding child state name to each state's signal, thus activating several (concurrent) composite states in parallel. In order to rejoin the threads generated by concurrent states, UML requires a **join** construct, described below. We only permit transitions to the boundary of the concurrent composite state and we require a **join** before the parent state's thread can resume. We believe these restrictions to be consistent with the vast majority of actual embedded systems.

Rule VHDL 8
Given composite concurrent child states $child_1$, $child_2$, ... , $child_n$, separated by dotted lines in the composite state, instantiate each component as a child composite state. Each child *state* signal is connected to a unique state signal (instead of *state* in the parent) in the **port map**. All concurrent states are started in parallel by assigning the initial concurrent state name to the unique signal created. Use the following template for instantiating the child concurrent states:

```
architecture abstract of cs_state is
    signal state : rs_st ;
    signal ss_child1 : rs_st ;
    signal ss_child2 : rs_st ;
        ⋮
    signal ss_childn : rs_st ;
begin
    I1: entity cs_child1(abstract)
        port map(state=> ss_child1, ... )
    I2: entity cs_child2(abstract)
        port map(state=> ss_child2, ... )
            ⋮
    In: entity cs_childn(abstract)
        port map(state=> ss_childn, ... )
    ss_child1 <= child1,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
    ss_child2 <= child2,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
    ss_childn <= childn,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
```

where ss_child_i is the temporary state signals holding cs_child_i's thread.
Relationship to Homomorphism
This rule complies with the **ConcurrentComposite** to **ConcurrentComposite** class mapping. The rule further defines how the outgoing ang incoming relationships on class *StateVertex* are maintained.
End of Rule VHDL 8.

Figure 5.16 shows how Rule VHDL 8 generates the instantiation of composite concurrent states *CP1* and *CP2* in state *TOP* from the diagram in Figure 5.12. Lines 10–12, and 16–18 create the connection between *TOP*, *CP1*, and *CP2*. Lines 19–20 start the composite state when a transition to the composite state boundary occurs.

```
1   architecture abstract of obj_top is
2       signal state :  rs_st;
3       signal ss_cp1:  rs_st;
4       signal ss_cp2:  rs_st;
5       signal ins_cp1:  st;
6       signal ins_cp2:  st;
7       signal ins_d:  st;
8   begin
9       12:  entity cs_cp1(abstract)
10          port map(state=>ss_cp1, t1=>t1, t2=>t2, t3=>t3, t4=>t4, t5=>t5,
11          t6=>t6, t7=>t7, instate=>ins_cp1, ins_cp2=>ins_cp2,
12          ins_d=>ins_d);
13      ss_cp1 <= st'(cp1), null after 1 fs when state=st'(cp1) or
14      state=st'(cp2);
15      13:  entity cs_cp2(abstract)
16          port map(state=>ss_cp2, t1=>t1, t2=>t2, t3=>t3, t4=>t4, t5=>t5,
17          t6=>t6, t7=>t7, ins_cp1=>ins_cp1, instate=>ins_cp2,
18          ins_d=>ins_d);
19      ss_cp2 <= st'(cp2), null after 1 fs when state=st'(cp1) or
20      state=st'(cp2);
```

Figure 5.16: This example illustrates the mapping from Rule VHDL 8 to generate two concurrent composite states.

5.2.3 Joining Concurrent Threads

We require UML **joins**, which combine several threads of control into a signal thread, to be in the parent state (one level higher) that instantiated the concurrent components. Since a **join** is a pseudostate according to the UML dynamic metamodel, it is mapped to the process named `join`$_n$ where n is a sequential number assigned in order to create a unique state name. The *join* must remember the occurrence of each incoming transition as it occurs, and when all have occurred, cause a transition to the outbound side of the *join*. Remembering the occurrence of each inbound transition requires listening to each threads' state signal, and a new set of signals to remember when the transition has occurred.

Rule VHDL 9
In the following template $csname_n$ are the names of the concurrent composite state components, join is the name of the join state, and *newstate* is the name of the state entered after the join operation. A *join* pseudostate is formalized with one concurrent signal assignment per inbound transition arc and a join process. The signal assignments and the process take the form:

```
join_csname1 <= join when ss_csname1=join;
join_csname2 <= join when ss_csname2=join;
⋮
join_csnamen <= join when ss_csnamen=join;
process join; begin
    wait until join_csname1 = join
            and join_csname2 = join
            ⋮
            and join_csnamen = join;
    join_csname1 <= none;
    join_csname2 <= none;
    ⋮
    join_csnamen <= none;
    state <= newstate, null after 1 fs;
end process;
```

Each join_$csname_n$ must be declared in the enclosing **architecture** as an enumerated signal type matching *state* in the current component. The name of the join process must be unique.
Relationship to Homomorphism
This rule complies with the mapping **Join** to **J-Process**. A **J-Process** is a kind of *StateVertex*, thus the rules defines how the outgoing and incoming relationships are maintained.
End of Rule VHDL 9.

Figure 5.17 is an example of the *join* named *join1* generated by Rule VHDL 9 that joins threads from *CP1* and *CP2* in Figure 5.12. As seen in Figure 5.12, after the threads are re-joined, state *F* is activated.

```
joinl_cpl <= ss_cpl when ss_cpl=st'(joinl);
joinl_cp2 <= ss_cp2 when ss_cp2=st'(joinl);
s_joinl:  process
begin
    wait until joinl_cpl=st'(joinl) and joinl_cp2=st'(joinl);
    say("joinl join complete");
    state <= st'(f), null after 1 fs;
    joinl_cpl <= st'(none);
    joinl_cp2 <= st'(none);
end process;
```

Figure 5.17: Example of the join pseudo-state generated by Rule VHDL 9 in state *TOP*

Pseudo States.

Default initial states are mapped to a process named *s_initial.* This process simply waits for the state name, then performs an immediate transition to the state indicated in the state diagram.

Rule VHDL 10
Map default initial states to a state process named *s_initial* according to the following template:

```
s_init:  process
    begin
        wait until state=st'(CSTATE);
        state <= st'(DEFAULT), null after 1 fs;
    end process;
```

where CSTATE is the name of the enclosing composite state or object, and DEFAULT is the name of the deafult initial state.
Relationship to Homomorphism
This rules complies with the **Initial** to **I-Process** mapping.
End of Rule VHDL 10.

UML inherits the concept of a "history state" from Statecharts [25] as an alternative to the default initial state. *History* either transitions to the indicated initial state or to the last active state in the state component. *History* is simulated in VHDL by adding an additional history signal named *history* that continuously tracks the last value of the *state* signal variable. The initial state is modified to assign the current value of *history* instead of the static value of the initial state. The new signal *history* is initialized to the default initial state. Clearly, there cannot be an initial state and a history state in the same composite state.

Rule VHDL 11
Map history states to an initial state named *s_initial* according to the following template:

```
s_init:  process
    begin
        wait until state=st'(CSTATE);
        state <= st'(DEFAULT), null after 1 fs;
    end process;
```

where CSTATE is the name of the enclosing composite state or object, and DEFAULT is the name of the default initial state. In addition, add the following statement between the **architecture** and **begin** statement for the composite state:
signal history: st := st'(DEFAULT);
Relationship to Homomorphism
This rule complies with the **History** to **H-Process** mapping.
End of Rule VHDL 11.

Figure 5.18 shows how Rule VHDL 11 generates VHDL statements for the history state in *CP1* from Figure 5.12.

```
1   architecture abstract of cs_cp1 is
2       signal history:  st := st'(a);
3   begin
4   -
5   -
6       history <= state when state=st'(a) or state=st'(b) or state=st'(c);
7       s_init:  process
8       begin
9           say("In state history");
10          wait until state=st'(cp1);
11          state <= history, null after 1 fs;
12      end process;
13
```

Figure 5.18: Example of the generated history initial state generated by Rule VHDL 11. Non-history initial states, per Rule VHDL 10, are the same except the assignment on line 11 is a constant and lines 2 and 6 are not needed.

Final states simply cause a thread to end. A **wait** with no parameters performs the same function in VHDL.

Rule VHDL 12
Map a final state to a state process named *s_final* whose body contains a single **wait** with no parameters.
Relationship to Homomorphism
This rules complies with the mapping of **Final** to **S-Process**
End of Rule VHDL 12.

5.3 Example

In this section we demonstrate the analysis of a UML model by mapping it into VHDL and executing the model. The embedded system we show is a heating and cooling system that operates separate furnace and air conditioning (AC) units. The user interface consists of a thermostat and five buttons to control heating mode, cooling mode, off, fan continuously on, and fan-auto mode. In fan-auto mode the fan cycles as required by the AC or furnace. The controller starts and stops a separate furnace and AC unit. The initial class diagram for the system is shown in Figure 5.19. The mode buttons and fan buttons are subclasses of class *button* that generates a momentary signal when pushed. In this example, we model the *Controller* class by supplying a system context that provides several user case scenarios. The UML dynamic model for the controller class is shown in Figure 5.20.

From the class diagram, we see that the controller must handle and remember stateless (momentary contact) mode messages from the control panel, which are essentially events, and must respond to demands for heat and cooling from the thermostat. The responsibility of temperature overrun and undershoot, or hysteresis, lies with the thermostat and not with the controller.

In order to respond to and maintain the mode buttons, the controller consists of five

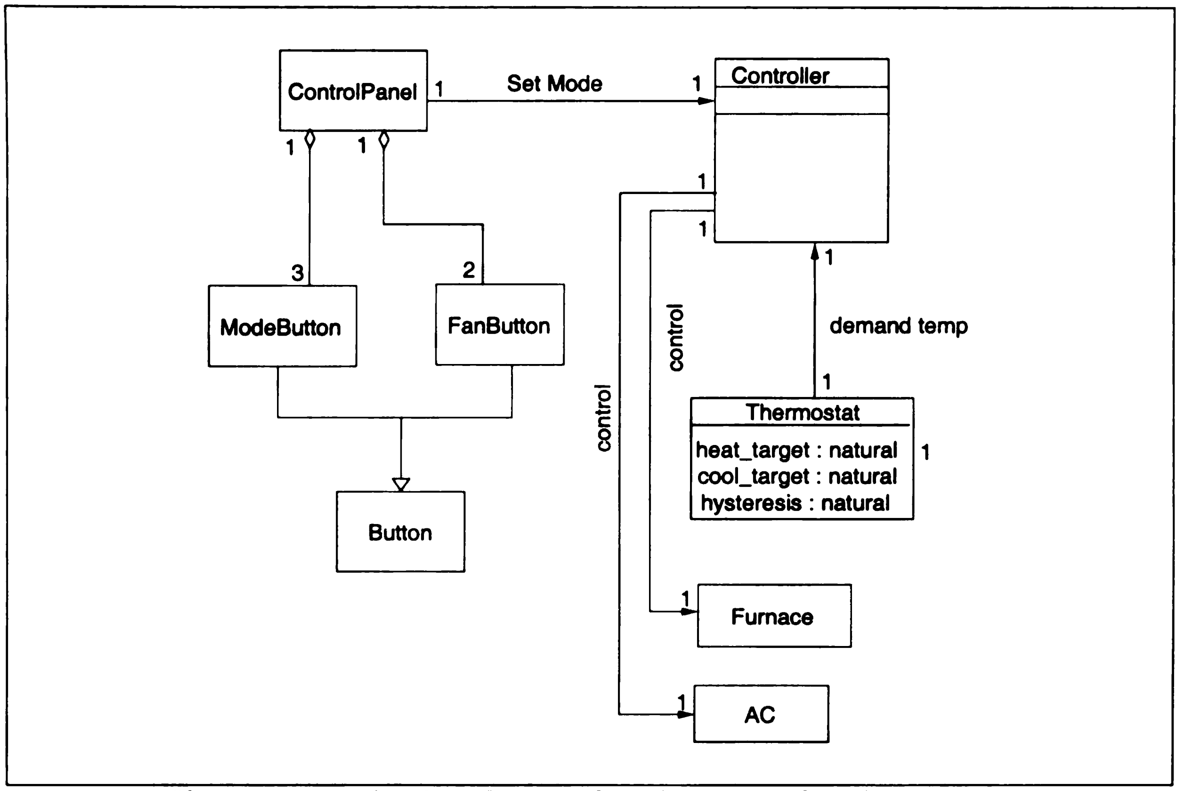


Figure 5.19: The class diagram for a heating and cooling system.

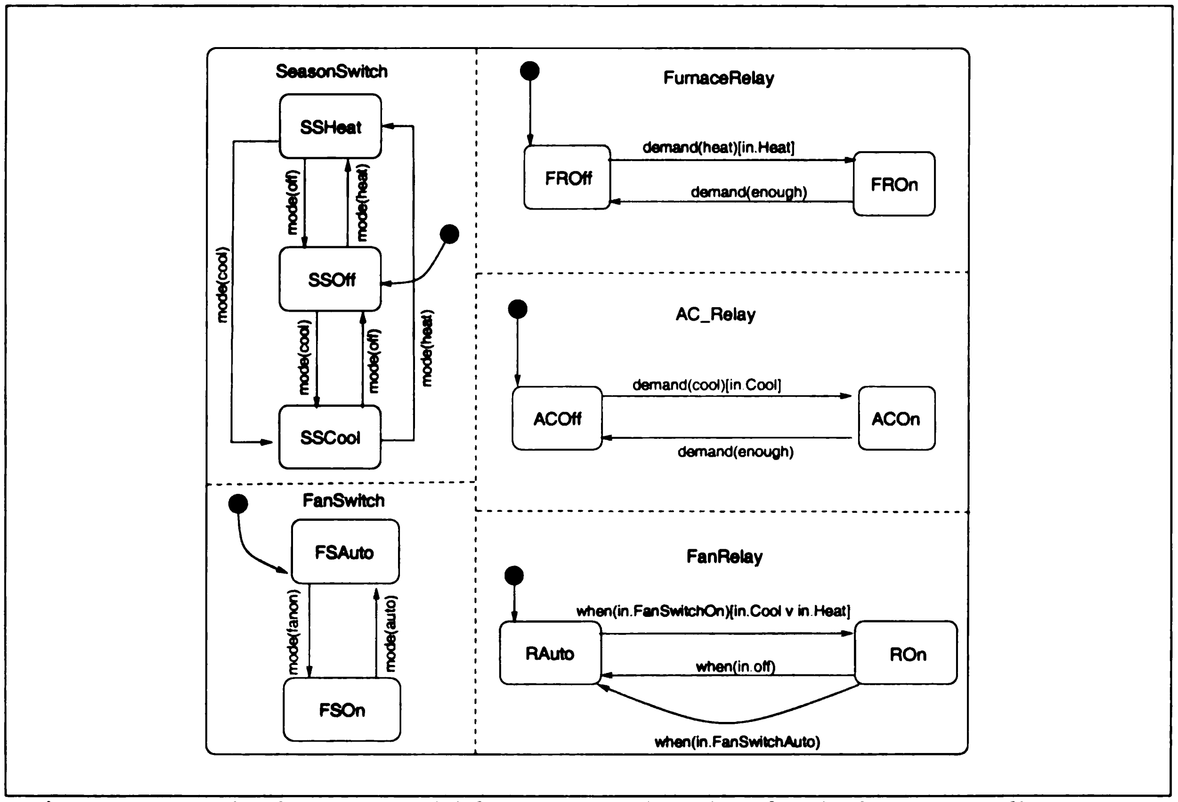


Figure 5.20: The dynamic model for the controller class for the heating–cooling system

concurrent composites states. States *SeasonSwitch* and *FanSwitch* respond to button pushes to preserve the current state of system mode. The state *FurnaceRelay* decides when to start the furnace and stops the furnace when the thermostat commands **enough**. The *AC_Relay* state is analogous to state *FurnaceRelay*, turning the AC on and off.

5.3.1 VHDL Specifications

Figures 5.21 and 5.22 show VHDL for the *FurnaceRelay* component of the model generated by our UML to VHDL translator, Hydra (see Chapter 8). The entire controller specified in VHDL as generated by Hydra is given in Appendix A.

```
– Furnace Relay component
entity cs_furnrelay is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        ins_seasonswitch:  in st;
        ins_fanswitch:  in st;
        instate:  out st;
        ins_acrelay:  in st;
        ins_fanrelay:  in st);
end entity;
```

Figure 5.21: The **entity** section from the FurnaceRelay component of the top state from Figure 5.20. The matching **architecture** is in Figure 5.22

5.3.2 Behavior Validation

Use case scenarios can be applied directly to the model to validate behavior. In scenario one of the use case, we move the system to heat mode, then demand heat, and terminate by sending the signal **enough**. When the model is executed in a VHDL simulator, the output shown in Figure 5.23 is produced. Since time is relative in a simulation, the numbers to the left of the event annotations in the figures are relative times. At time 0, all initial states are entered. Next we see the system cycling to heat mode, starting the furnace and turning

```
architecture abstract of cs_furnrelay is
begin
    instate <= state when state/=st'(none);

    s_FROff:  process
    begin
        wait until state=st'(FROff);
        say("In state FROff");
        loop
            wait until demand=demandtype'(heat);
            if demand=demandtype'(heat) and ins_seasonswitch=st'(SSHeat)
            then
                state <= st'(FROn), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_FROn:  process
    begin
        wait until state=st'(FROn);
        say("In state FROn");
        loop
            wait until demand=demandtype'(enough);
            if demand=demandtype'(enough) then
                state <= st'(FROff), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(furnrelay);
        state <= st'(FROff), null after 1 fs;
    end process;

    s_ACOn:  process
    begin
        wait until state=st'(ACOn);
        say("In state ACOn");
        loop
            wait until demand=demandtype'(enough);
            if demand=demandtype'(enough) then
                state <= st'(ACOff), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;
end abstract;
```

Figure 5.22: The **architecture** section from the FurnaceRelay component of the top state from Figure 5.20 The matching **entity** is in Figure 5.21

it off again in response to thermostat signals.

```
0 : In state SSOff
0 : In state FSAuto
0 : In state FROff
0 : In state ACOff
0 : In state RAuto
3 : setting mode to Heat
3 : In state SSHeat
8 : Demanding heat
8 : In state FROn
13 : signalling Enough
13 : In state FROff
```

Figure 5.23: Output from driver scenario one.

As expected, the furnace turned on in response to a demand for heat and subsequently shut down with a command of *enough.*

In the second scenario we start with a cold building and the system in the off mode. The thermostat will already have demanded heat but the system will not be in heat mode. When we switch to heat mode, we expect the furnace to start since there is an outstanding request for heat. Figure 5.24 shows the result of this simulation. Clearly, the furnace did not start as required. The problem lies in the loss of the event **demand(heat)** and we can see from the dynamic model that the same situation exists with regard to cooling. One possible modification would be to add another state to composite states *FurnaceRelay* and *AC_Relay* so that heating and cooling demands always cause a state transition. We would do this by adding another state to the input to the UML to VHDL specification file and repeating the simulation.

```
0 : In state SSOff
0 : In state FSAuto
0 : In state FROff
0 : In state ACOff
0 : In state RAuto
3 : Demanding heat
8 : setting mode to Heat
8 : In state SSHeat
```

Figure 5.24: Execution output for scenario two. Turning on the heat after the thermostat demands heat should result in the furnace starting. This sequence shows it does not.

Safety is an important and often tested condition in an embedded system. Safety means that nothing bad ever happens in the system. In this system, we must ensure that the furnace and air conditioning unit are not simultaneously running. In scenario three, the system starts in heat mode. After the thermostat has demanded heat, the scenario calls for changing the system mode to cooling and abruptly lowering the demanded temperature. Assuming the system was producing heat, leaving the temperature above the desired cooling threshold, the thermostat should now demand cooling. Figure 5.25 shows the model results from running this scenario. At time 8 the furnace starts, then, while the furnace is still on, the AC comes on at time 18. Several causes for the improper system response in this scenario are possible. From the class diagram we see that *Thermostat* has responsibility for demanding heating and cooling. Perhaps the demand sequence {"heat", "cool"} without an intervening "enough" is invalid, or the mode sequence "heat" to "cool" without an intervening "off" is invalid. If the responsibility for safety lies with *Controller*, then it would have to be modified to handle both input sequences. This modification would take the form of assuming an intermediate "off" state whenever a switch from "heat" to "cool", or vice-versa, occurs. Cycling through "off" would shut down the system in operation before starting the alternate system.

If, on the other hand, the responsibility lies with *Mode*, then perhaps this simulation sequence is itself invalid, indicating an inconsistency between the scenario and the behavior of *Mode*. In this case, a scenario where the switch is moved from "heat" to "cool" without passing through "off" may be physically impossible, therefore, the scenario would have to be modified to include an intermediate "off" signal. Another alternative is that the user really does intend a direct transition from "heat" to "cool", perhaps via separate push buttons. In this case, the scenario is correct and *Mode* should generate an "off" signal to allow *Controller* to reset for the new mode.

```
0 : In state SSOff
0 : In state FSAuto
0 : In state FROff
0 : In state ACOff
0 : In state RAuto
3 : setting mode to Heat
3 : In state SSHeat
8 : Demanding heat
8 : In state FROn
13 : setting mode to Cool
13 : In state SSCool
18 : Demanding cool
18 : In state ACOn
```

Figure 5.25: Scenario three. Driving the controller into a furnace on and AC on condition.

5.3.3 Discussion

Our translation to VHDL and simulation has shown several important properties of even this small system during design. In the first scenario, we showed that the system does work at a basic level. In the second scenario we detected a problem of losing events in the design of the state machine. The state machine could have been quickly modified at that point and the scenario rerun to determine if the event loss problem were remedied. In the final scenario, we detected a more serious problem that could lie with the class diagram and the responsibilities distributed between the classes, or it may lie with *Controller* itself. If the mode switch is driven by a physical three-position switch, then the specification for *ControlPanel* is incorrect since it is probably physically impossible to move the switch from one extreme to another without passing through an intermediate "off" setting. Whatever the failure, this simple scenario calls for a deeper review of the design.

All of these results were obtained without writing code directly from the UML class and dynamic model diagrams. Since no code writing is involved, refinement of the model involves a straightforward modification of the UML and a re-translation to VHDL.

Although this model is small, we could have included other objects in the VHDL simulation to test inter-object interactions. And, if there had been VHDL models of actual hardware, these models could have been integrated into the generated VHDL model.

Chapter 6

UML to Promela/SPIN Mapping Rules

In this chapter we apply the framework described in Chapter 3 to the construction of a homomorphism and rules for mapping UML to Promela, the input language for the model checker SPIN. The Promela metamodel and homomorphic function is described first, followed by a description of the mapping rules.

6.1 UML to Promela Diagram Homomorphisms

The strategy for writing the Promela metamodel and formulating the UML to Promela mapping rules is as follows: First, the Promela metamodel is constructed. As described in Section 3.2.3, we begin by constructing a metamodel that has a structure as close to the UML metamodel as possible. The UML metamodel is shown in Figure 6.1 and the Promela metamodel is shown in Figure 6.2. The portions of the metamodels attributable to the class diagram is enclosed in dotted lines. Templates are constructed in the Promela metamodel, where necessary. As described earlier, a *template* is a parameterized series of statements in the target language that represents a target language metamodel class. Next, the homomorphism from the UML metamodel to the Promela metamodel is constructed. Finally, when the Promela metamodel and homomorphism are complete, the mapping rules are written. Three goals drive the construction of the mapping rules:

1. Map every UML class and relationship to a class and relationship in the Promela metamodel.
2. When a template is used, implement the desired semantics in terms of Promela.
3. As a rule is written, ensure that relationships in the UML metamodel are preserved in the target metamodel.

As described in Chapter 3, each class and *relationship* in the UML metamodel must have a mapping to a component in the Promela metamodel. Further, the mapping must be consistent with the homomorphism so that relationships between classes are preserved. The list of normal relationships (*i.e.*, not aggregations or generalizations) generated by the formalization of the UML metamodel is shown in Figure 6.3. The occurrences of the hasComp() and hasPart() predicates generated by the formalization of the UML metamodel are shown in Figure 6.4.

6.1.1 Homomorphic Mapping

Figure 6.5 shows the mapping of UML metamodel classes to Promela metamodel classes. Figure 6.6 shows the mapping of the relationships from one metamodel to another.

To demonstrate completeness, Figure 6.5 and Figure 6.6 show that every class and non-aggregate relationship in the UML metamodel maps to a class or relationship in the

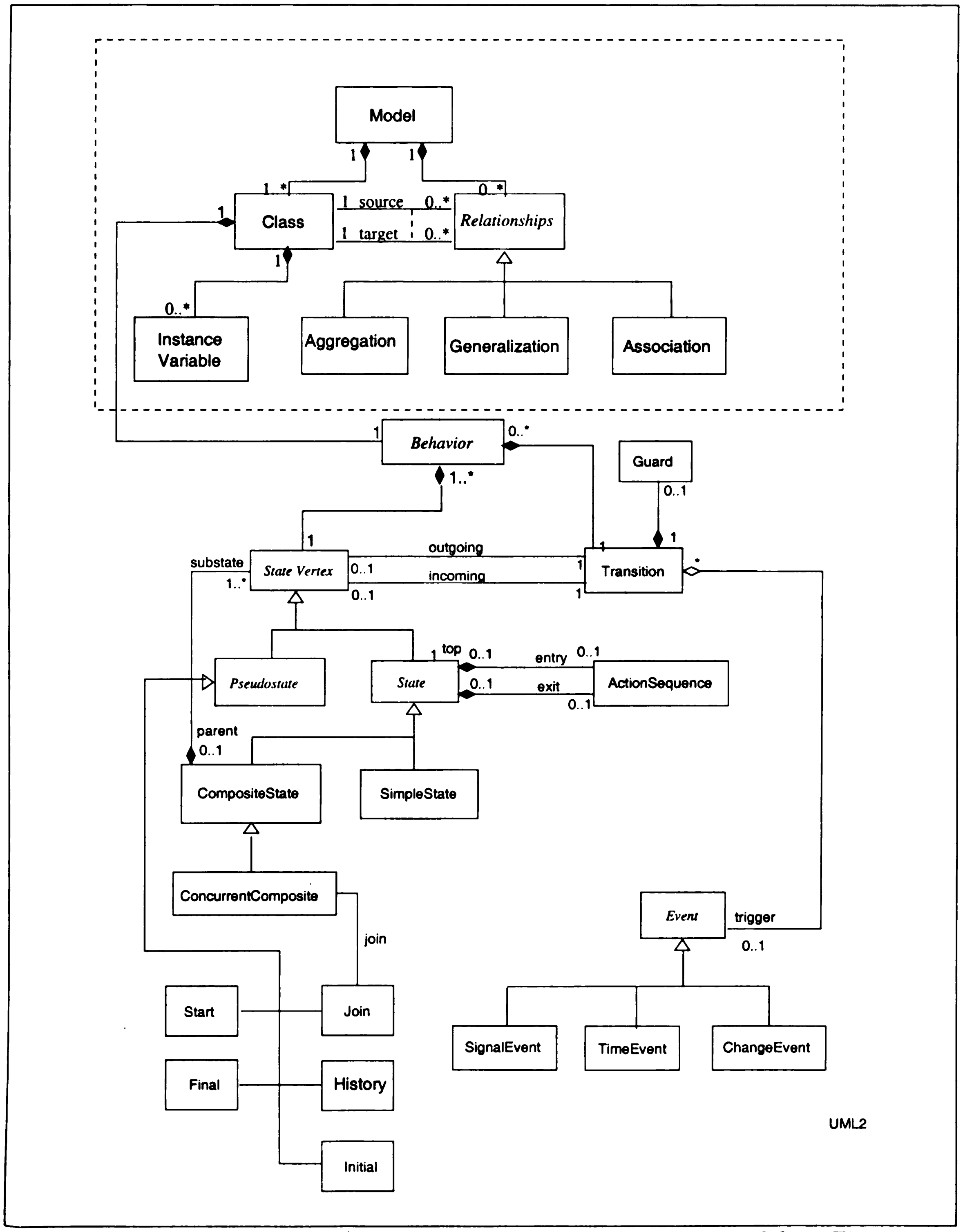


Figure 6.1: The unified class/dynamic metamodel for UML. Repeated from Figure 5.1

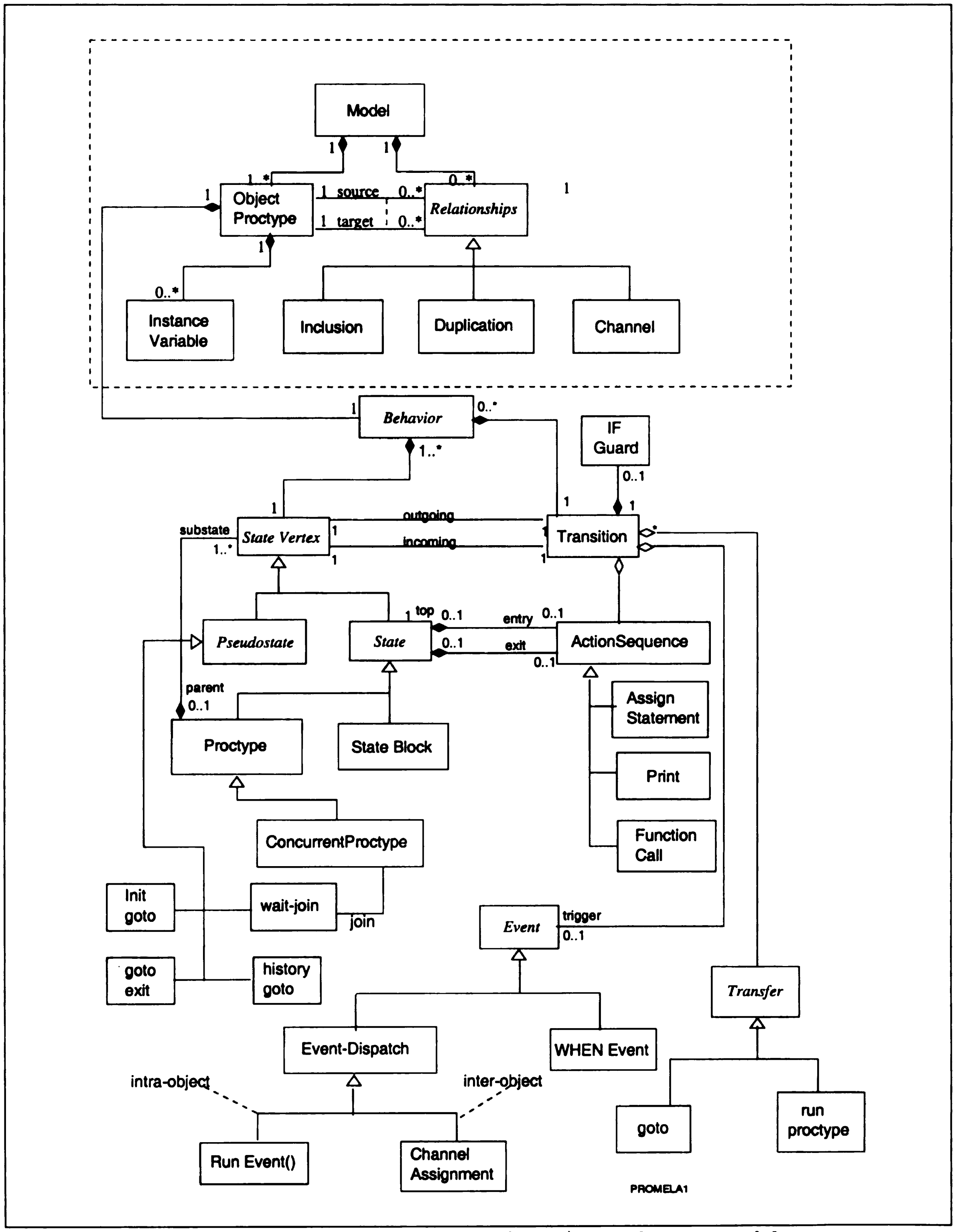


Figure 6.2: The metamodel for SPIN/Promela metamodels.

source(Class,Relationships)
target(Class,Relationships)
outgoing(StateVertex,Transition)
incoming(StateVertex,Transition)
entry(State,ActionSequence)
exit(State,ActionSequence)

Figure 6.3: The list of non-aggregate and non-generalization relationship predicates from the formalized representation of the UML metamodel.

hasComp($Model, Class$)
hasComp($Model, Relationships$)
hasComp($Class, InstanceVariable$)
hasComp($Behavior, Transition$)
hasComp($CompositeState, StateVertex$)
hasPart($Transition, Event$)

Figure 6.4: The list of non-aggregate and non-generalization relationship predicates from the formalized representation of the UML metamodel.

Promela metamodel. Figure 6.6 shows that each non-aggregate relationship is preserved by the mapping in the Promela metamodel, satisfying the requirement for a homomorphism. Figure 6.7 further shows that the hasComp() and hasPart() relationship predicates are preserved in the Promela metamodel, completing the demonstration that the mapping is homomorphic.

6.2 UML to Promela Mapping Rules

This section describes the mapping rules from UML to Promela. The rules are stated in terms of the metamodel to metamodel mapping. As described earlier, the mapping is constrained to preserve the homomorphic mapping described above.

UML Metamodel Class		Promela Metamodel Class
Model	⟶	Model
Class	⟶	ObjectProctype
Relationships	⟶	Relationships
InstanceVariable	⟶	InstanceVariable
Aggregation	⟶	Inclusion
Generalization	⟶	Duplication
Association	⟶	Channel
Behavior	⟶	Behavior
Guard-	⟶	IFGuard
StateVertex	⟶	StateVertex
Transition	⟶	Transition
Pseudostate	⟶	Pseudostate
State	⟶	State
ActionSequence	⟶	ActionSequence
CompositeState	⟶	Proctype
ConcurrentComposite	⟶	ConcurrentProctype
SimpleState	⟶	StateBlock
Start	⟶	Init-goto
Join	⟶	Wait-join
History	⟶	History-goto
Final	⟶	goto-exit
Event	⟶	Event
SignalEvent	⟶	Event-Dispatch
TimeEvent	⟶	Event-Dispatch
ChangeEvent	⟶	WHEN-Event

Figure 6.5: The definition of the homomorphic mapping of classes from the UML metamodel to the Promela metamodel.

UML Metamodel Predicate		Promela Metamodel Predicate
source(Class,Relationships)	⟶	source(Class,Relationships)
target(Class,Relationships)	⟶	target(Class,Relationships)
outgoing(StateVertex,Transition)	⟶	outgoing(StateVertex,Transition)
incoming(StateVertex,Transition)	⟶	incoming(StateVertex,Transition)
entry(State,ActionSequence)	⟶	entry(State,ActionSequence)
exit(State,ActionSequence)	⟶	exit(State,ActionSequence)
join(ConcurentComposite,Join)	⟶	join(ConcurrentProctype,Wait-join)

Figure 6.6: The definition of the homomorphic mapping of relationship predicates from the UML metamodel to the Promela metamodel.

UML Aggregate Relationship		Promela Aggregate Relationship
hasComp($Model, Class$)	$\longrightarrow$	hasComp($Model, ObjectProctype$)
hasComp($Model, Relationships$)	$\longrightarrow$	hasComp($Model, Relationships$)
hasComp($Class, InstanceVariable$)	$\longrightarrow$	hasComp($Class, InstanceVariable$)
hasComp($Behavior, Transition$)	$\longrightarrow$	hasComp($Behavior, Transition$)
hasComp($CompositeState, StateVertex$)	$\longrightarrow$	hasComp($Proctype, StateVertex$)
hasPart($Transition, Event$)	$\longrightarrow$	hasPart($Transition, Event$)

Figure 6.7: The preserved hasComp and hasPart relationship predicates.

6.2.1 Global Requirements

Several fixed constructs are required for every model mapped from UML to Promela. These include the queues used for intra-object event dispatching and a Promela **proctype** to manage these queues. An additional queue, called **wait**, is used to determine child process termination and pass return codes used in transitions back to the parent process. The complete discussion of the **proctype event()**, event dispatching, and the transition process is given in Section 6.2.3.

Rule Promela 1
Every model mapped from UML to Promela contains the following channel definitions and the dispatching `proctype`, called `event()`, to handle intra-object transitions:

```
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};

/*
        The body of the model goes here.  This section will contain
        proctypes for objects and composite states
*/

/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
mtype type;
int pid;

atomic {
do
::  evq??[eval(msg),pid] ->
evq??eval(msg),pid;
evt!msg,pid;
do
::  if
::  evq??[type,eval(pid)] -> evq??type,eval(pid)
::  else break;
fi
od
::  else -> break
od}
exit:   skip
}
```

Relationship to Homomorphism
This rule provides details to complete the mapping from **Model** to **Model**
End of Rule Promela 1.

6.2.2 Class Diagram Formalizations

As in VHDL, static structure is defined primarily in the class model. Figure 6.8 shows a sample class model intended to provide examples of the class diagram formalization rules.

As described in the previous chapter, the diagram contains four classes: A, B, C, and D. Class A contains an aggregate of C, and class A has a relation with class B. Class D is a specialization of class B.

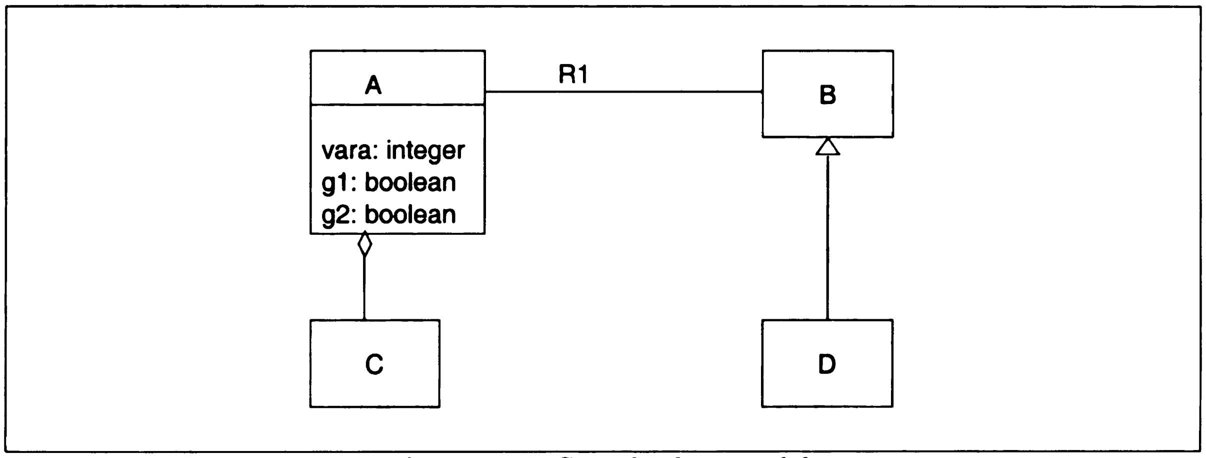


Figure 6.8: Sample class model.

Classes map to **proctype** Promela definitions. Simple states, if any, are represented by sequences of statements within the **proctype** representing the class.

Rule Promela 2
Each class is formalized in Promela as a **proctype** definition. The body of the **proctype** is enclosed in an **atomic** statement. The **proctype** accepts no parameters. The following template shows the class container:

```
proctype CLASS()
{atomic{
        mtype m;
        /* Statements for the top level of the object
           go here */
exit:   skip
}
```

Relationship to Homomorphism
This rule follows from the **Model** to **ObjectProctype** mapping. The aggregation in **Model** is preserved because each class is contained within the Promela model file.
End of Rule Promela 2.

Figure 6.9 shows the Promela specifications produced for class A in Figure 6.8 by Rules

Promela 1 and Promela 2. The queues produced by Rule Promela 1 are shown on lines 1-3 and the intra-object dispatching **proctype** also generated by Rule Promela 1 is shown on lines 24–44 of Figure 6.9. The class container generated by Rule Promela 2 is shown on lines 17–22 of Figure 6.9.

Class specialization conceptually allows the designer to duplicate the parent class and modify it to become a subclass. Such duplication implies that the behavior, along with relationships and instance variables, are also reproduced. In theory, a subclass object should be able to simulate a parent class object, but the meaning of specialization when the duplicated parent class behavior is radically modified, or even replaced, is not clear.

Fortunately, high levels of specialization are not a characteristic of embedded systems. Therefore, generalization is formalized as duplication of all the mapped components of a class followed by modifications as required to create the subclass definition. The duplication includes the **proctype** definition for the class and all associated **proctypes** and variable declarations belonging to the class (other **proctypes** are generated by rules that follow).

Rule Promela 3
Class specialization is formalized as duplication of the parent class components, followed by modification, as required.
Relationship to Homomorphism
This rule follows directly from the depiction of the metamodels. The type relationship with *Relationships* is preserved.
End of Rule Promela 3.

Aggregation means two (or more) classes have a relationship in which the aggregate class tracks the parts, but the communication in an aggregate relationship is no different that normal inter-object messaging.

```
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};
mtype={R1};
typedef A_T {
              bool g1;
              bool g2;
              int vara;
              }
A_T A_V;
chan A_q=[5] of {mtype};
chan A_p1=[5] of int;
mtype={free};

/* User specified driver file */

proctype A()
{atomic{
mtype m;
int dummy;
exit:      skip
}}

/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
mtype type;
int pid;

atomic {
do
::  evq??[eval(msg),pid] ->
evq??eval(msg),pid;
evt!msg,pid;
do
::  if
::  evq??[type,eval(pid)] -> evq??type,eval(pid)
::  else break;
fi
od
::  else -> break
od}
exit:   skip
}
```

Figure 6.9: Promela specifications generated for class *A* in Figure 6.8.

Rule Promela 4
Aggregation is formalized as an instantiation of the "part" class as many times are required or specified. The aggregate object and part objects communicate with messages.
Relationship to Homomorphism
This rule follows directly from the depiction of the metamodels. The type relationship with *Relationships* is preserved.
End of Rule Promela 4.

Instance variables are declared at the top of the Promela model using **typedef** structure definitions. Since instance variables must be accessible to all transitions within an object, the mapping generates global Promela variables.

Rule Promela 5
Instance variables are formalized as members of a **typedef** structure statement named uniquely for the class according the the following template:

```
typedef CLASS_T {
            <type> var_1;
            .
            .
            .
            <type> var_n;
}

CLASS_T CLASS_V;
```

where CLASS is the name of the class. The mapped instance variable declarations are placed at the beginning of the Promela specifications before any proctypes.
Relationship to Homomorphism
The template listed above forms the **InstanceVariable** class in the Promela metamodel. The aggregation is preserved by naming conventions and the generation of Promela statements for the class, which reference the typdef and CLASS_V declaration described above.
End of Rule Promela 5.

Lines 5–10 in Figure 6.9 show the declaration of class A's instance variables generated by Rule Promela 5.

Objects communicate with each other using *messages*. Three major semantic possibilities exist for message arrival: queue the message for later use, drop the message if the object is not ready to use it, or the sender waits until the receiver can accept the message. Queued semantics are easier for the designer because objects can run asynchronously with less regard to the operation of other objects, but queueing requires extra mechanisms that may be expensive to implement in an embedded system. Non-queued, semantics is simpler to implement, but places a high burden on the designer. Each message exchange must be carefully choreographed to ensure that the objects rendezvous at the proper time to avoid message loss, or for waiting semantics, deadlock[1]. When many objects are present and waiting semantics are used, deadlock avoidance is very difficult.

Associations in the class model represent communication between related classes. Inter-object communication is formalized in Promela with channels. Unlike VHDL, we have a wide choice of semantics for the inter-object messaging, as described above. One option is to use the same semantics as suggested for state to state messaging, which does not queue messages at the object. If an object is not ready to receive a message, then the message is lost. The mechanism to implement non-queuing semantics is the same as intra-state messaging, and is discussed in Section 6.2.3.

Another alternative is to queue messages at each object. There are several possibilities for retrieving messages from queues. One is to formalize inter-object messaging as a FIFO queue. If the receiving object is not ready to accept a message, then the message is queued until a transition is ready to consume it from the head of the queue. To obtain FIFO queued semantics, a Promela channel with simple "sends" and "receives" (the "!" and "?" channel operators) is used. It is also possible to allow retrieval of any message in the queue, regardless of its arrival order. To obtain any-order queued semantics, the Promela receive

[1]Deadlock occurs, for example, and A is waiting for B, B is waiting for C and C is waiting for A.

operator is replaced with the *poll* operator (the poll operator is "??"). Any-order queue semantics allows message arrival order to be largely ignored during model composition, but it also allows race conditions in the absence of fair process scheduling. This may occur when several messages in the queue can enable a transition at a given state. If there is a continual reception of a new message for a given transition, then nothing guarantees the other message will ever be chosen. We implement FIFO queued semantics for inter-object message passing in this mapping.

Rule Promela 6
Formalize inter-object relationships as a set of Promela channels. For class CLASS, signal types SIG_1 through SIG_n, where each signal type has a number of variables, use the template below. In the template, assume SIG_1 has m variables, SIG_2 has n variables, and SIG_k has o variables. <type> represents the type of the variable passed in the message.

```
chan CLASS_q=[5] of {mtype}
chan SIG_1_p1=[5] of <type>;
        .
        .
        .
chan SIG_1_pm=[5] of <type>;

chan SIG_2_p1=[5] of <type>;
        .
        .
        .
chan SIG_2_pn=[5] of <type>;

chan SIG_k_p1=[5] of <type>;
        .
        .
        .
chan SIG_k_po=[5] of <type>;
```

The result of this template is placed at the beginning of the Promela specifications.
Relationship to Homomorphism
The above template forms the **Relationships** class in the Promela metamodel.
End of Rule Promela 6.

In Figure 6.8, if we assume relationship *R1* has one integer parameter, then Rule

Promela 6 generates lines 11 and 12 in Figure 6.9.

6.2.3 Dynamic Model Formalizations

This section describes the formalization of the portion of the unified UML metamodel corresponding to the dynamic model. Figure 6.10 shows a sample dynamic model intended to illustrate the rules in this section. The model represents no particular system.

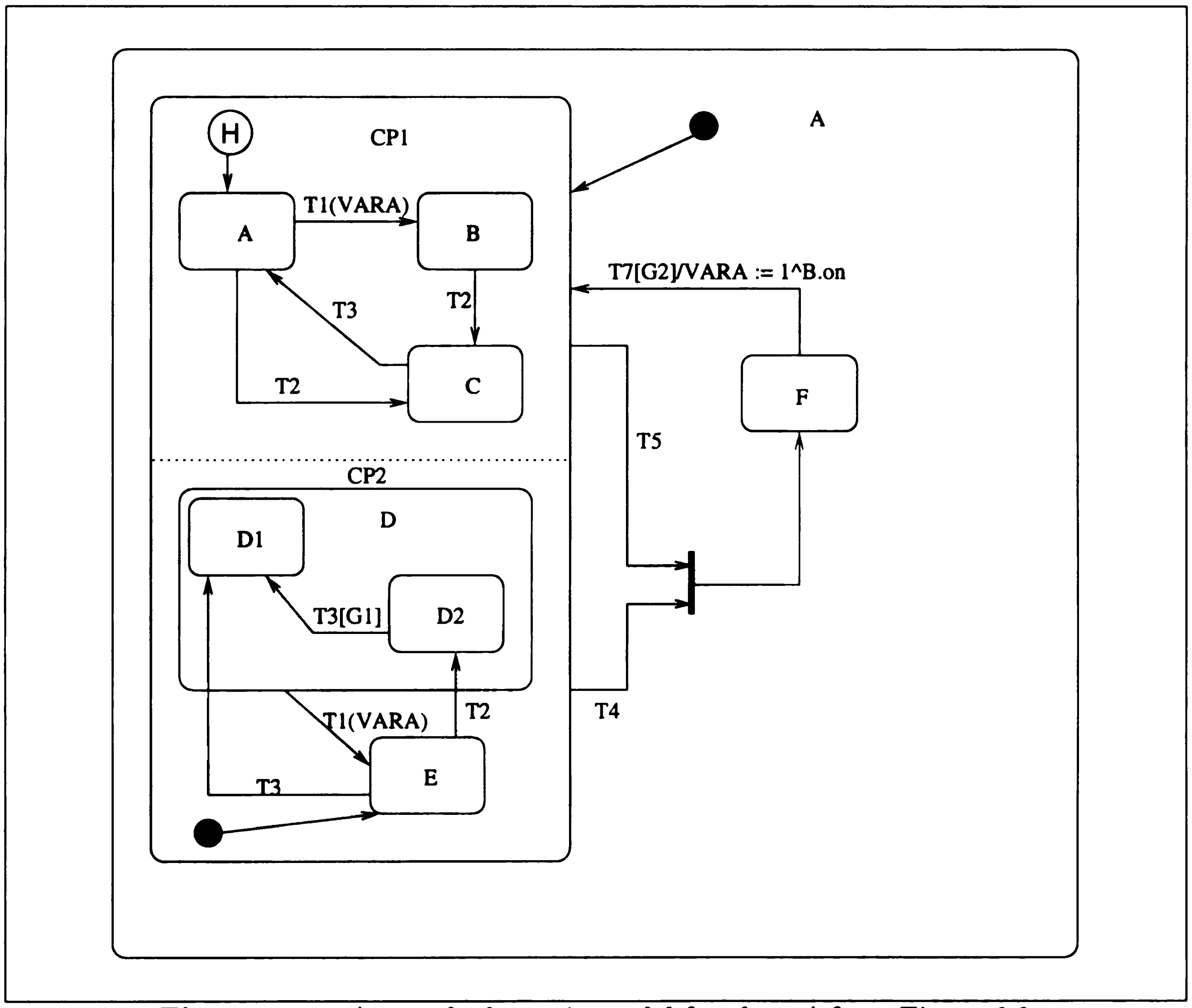


Figure 6.10: A sample dynamic model for class *A* from Figure 6.8

Simple States

Figure 6.11 shows the Promela specifications generated for the top level of object *A* in Figure 6.10. Various parts of the specification in Figure 6.11 are described as the rules are presented.

A UML *simple state* is represented in Promela as a block of Promela statements whose first statement is labeled with the state name. Transfer to the state is through a **goto** statement. The contents of the state block vary according the the types of events used on transitions, the messages sent, and the actions listed for the transitions. Each of these is described in detail in rules below.

```
        chan evq=[10] of {mtype,int};
        chan evt=[10] of {mtype,int};
        chan wait=[10] of {int,mtype};
        mtype={none, t1, t2, st_join1, st_a, t3, st_b, t4, st_c, t5, t6, st_d1,
                st_e, t7, st_d2};
        mtype H_cp1=st_a;
        typedef A_T {
                bool g1;
                bool g2;
                int vara;
                }
        A_T A_V;
        chan A_q=[5] of {mtype};
        chan t1_p1=[5] of {int};
        chan t=[1] of {mtype};
        mtype={free};

        /* User specified driver file */

        proctype A()
        {atomic{
        mtype cp1_code, cp2_code;
        int cp1_pid, cp2_pid;
        mtype m;
        int dummy;
        /* Init state */
                goto to_cp1;
        /* State f */
f:           skip;
                printf("in state A.f");
in_f:
                if
                ::  A_q?t7 -> if
                   ::  A_V.g1 -> A_V.vara = 1; B_q!on; goto to_cp1
                   ::  else -> goto in_f
                   fi
                fi;
to_cp1:
to_cp2:
                atomic {
                cp1_pid = run cp1(none);
                cp2_pid = run cp2(none);
                }
                wait??eval(cp1_pid),cp1_code;
                wait??eval(cp2_pid),cp2_code;
                if
                ::  cp1_code == st_join1 && cp2_code == st_join1 -> goto f;
                fi;
exit:        skip
        }}
```

Figure 6.11: The Promela specifications generated for Object A in Figure 6.10 (the top level of the model).

Rule Promela 7
Simple states are formalized as a *State Block*, which is constructed from the following template:

```
state-name:  printf(''in state object-name.state-name'');
           H_composite-state-name = st_statename;
           evt!<event-name 1>,_pid;
           evt!<event-name 2>,_pid;
               .
               .
               .
           evt!<event-name n>,_pid;
           if
           ::  <transition event expression 1> -> <guard list 1>
              <action list 1>
              <send list 1>
              goto nextstate
           ::  <transition event expression 2> -> <guard list 2>
              <action list 2>
              <send list 2>
              goto nextstate
               .
               .
               .
           ::  <transition event expression n> -> <guard list n>
              <action list n>
              <send list n>
              goto nextstate
           fi;
```

In this template, `state-name` is the name of the state, `object-name` is the name of the enclosing object, and `composite-state-name` is the name of the enclosing composite state (or object if this is the highest level). The meta terms are defined as: `<transition event expression i>` are channel receive expressions for `<event-name i>` as described in Rules Promela 10 and Promela 11, `<guard list i>` is the construction for transition guards as described in Rule Promela 16, `<action list i>` is a list of statements for the action on the transition, and `<send list i>` is a list of channel send operations for messages sent by the transition.
The second line of the template is present only if a history state is present in the enclosing composite state.
Relationship to Homomorphism
This rule provides the **StateBlock** class in the Promela metamodel. The template above describes event dispatching and action sequences that preserve the `outgoing`, `incoming`, `entry`, and `exit` relationships. **StateBlock** is a *State* since it handles events and maintains the state of the object.
End of Rule Promela 7.

Lines 28 through 49 of Figure 6.11 contain state *f* as generated by Rule Promela 7 and the rules that follow.

Composite States

Composite states are represented as **proctype** definitions. The procedures are started (instantiated) when a transition is made to the composite state or one of its members. Each **proctype** representing a composite state takes a parameter of type **mtype** representing the name of state within the composite state to be activated in the case where a cross boundary transition is specified. The transition from state *e* to *d*2 in Figure 6.10 is an example. When the transition is to the boundary of the composite state, then the value of the parameter is set to **none**.

After the parent instantiates the child composite state **proctype**, the parent must wait until the child ends. This is accomplished using the **wait** queue generated by Rule Promela 1. After starting the composite state, the parent composite state (or object) issues a "receive" on channel **wait** specifying the child's process identification number (PID). Just before the child **proctype** ends, it sends it's PID to channel **wait** to notify the parent to resume execution.

Since SPIN uses a fully interleaved statement execution model, it is possible to map UML transitions to semantics that allow concurrency during transitions, but UML semantics generally assume transitions complete before another action is taken in the model [7]. This is called *run to completion* semantics. Run to completion semantics greatly ease model design and make sense for most embedded system designs.

Our mapping for transitions assumes run to completion semantics. This is accomplished by placing each **proctype** within the scope of an **atomic** statement. The **atomic** statement tells the analyzer to execute all the statements within the atomic block as a single statement. When a statement within the scope of **atomic** blocks (such as a channel receive operation), Promela semantics allow ready statements elsewhere to be executed. If there are multiple choices for transfer of control, a choice is made non-deterministically. Effectively, our construction causes each procedure to run until it blocks, then control is transferred to another procedure until that one blocks, *etc.* . Since there are no blocking statements allowed after a transition event, run to completion semantics are produced.

Rule Promela 8
A composite state named `composite-state-name` is formalized as a `proctype` with a formal parameter of type `mtype`. A `proctype` representing a composite state is activated in the parent composite state or object with a `run(first-state-name)` statement that passes the name of the state to begin, or `none` when the transition is to the boundary of the composite state. Use the following template for the composite state `proctype`:

```
proctype composite-state-name(mtype state)
{atomic{
<initial state sequence>
<state sequences>
exit:     skip
}}
```

The meta sequences are defined as: `<initial state sequence>` is either a history state construct or an initial state construct, as required. See Rules Promela 19 and Promela 20. The body of the simple states and transfers to composite states are contained in `<state sequences>`.
Relationship to Homomorphism
Class **Proctype** holds simple states and holds other composite states through references (calling sequences) to the `proctypes` that represent the composite states, hence **Proctype** is a *State* and an aggregation of *StateVertex*.
End of Rule Promela 8.

Figure 6.12 shows composite state *d*, which is the target of transitions from state *e*, generated by Rule Promela 8 from Figure 6.10. Line 6–10 in Figure 6.12 show the selection of the initial state within state *d*. Figure 6.13 shows the statements generated by Rule Promela 8 within composite state *cp*2 for simple state *e*. There are two parts to the transition to state *d*2 (within *d*): line 15 sets a parameter for the initial state for composite state *d* to *d*2, then transfers to label `to_d`. Rule Promela 8 generates line 2 in Figure 6.13 to transfer control to composite state *d*. Line 3 in the same figure contains statements to wait for state *d* to exit.

```
proctype d(mtype state)
{atomic{
int d_pid;
mtype m;
int dummy;
                if
                ::  state == st_d1 -> goto d1
                ::  state == st_d2 -> goto d2
                ::  else -> assert(0) /* no init state - die if bad state */
                fi;
/* State d1 */
d1:             skip;
                printf("in state A.d1");
in_d1:
                if
                ::  A_q?t4 -> wait!_pid,st_join1; goto exit
                ::  A_q?t1 -> t1_p1?A_V.vara -> wait!_pid,st_e; goto exit
                fi;
/* State d2 */
d2:             skip;
                printf("in state A.d2");
in_d2:
                if
                ::  A_q?t4 -> wait!_pid,st_join1; goto exit
                ::  A_q?t1 -> t1_p1?A_V.vara -> wait!_pid,st_e; goto exit
                ::  A_q?t3 -> if
                   ::  A_V.g1 -> goto d1
                   ::  else -> goto in_d2
                   fi
                fi;
exit:       skip
}}
```

Figure 6.12: Promela specifications generated by Rule Promela 8 from Figure 6.10 for composite state *d*

```
/* Link to composite state d */
to_d:      d_pid = run d(m);
             wait??eval(d_pid),m;
             if
             :: m == st_join1 -> wait!_pid,st_join1; goto exit;
             :: m == st_e -> goto e;
             fi;
/* State e */
e:              skip;
             printf("in state A.e");
in_e:
             evq!t2,_pid;
             if
             :: A_q?t4 -> wait!_pid,st_join1; goto exit
             :: evt??t2,eval(_pid) -> m = st_d2; goto to_d
             :: A_q?t3 -> m = st_d1; goto to_d
             fi;
```

Figure 6.13: Promela specifications generated by Rule Promela 8 from Figure 6.10 for the transition to state *d*

Concurrent Composite States

The dotted line between composite states *cp*1 and *cp*2 in Figure 6.10 indicate that these two components are to run concurrently. Concurrent substates are formalized the same as non-concurrent states except that the parent composite state must start several **`proctypes`**, and consequently, wait for all of the **`proctypes`** to exit.

Rule Promela 9
Concurrent composite states are formalized as composite states per Rule Promela 8, but are activated in parallel using the template below. Let cp1, cp2 ... cpn be the names of n concurrent composite state components. Then, in the parent composite state that activates child concurrent composite states, declare variables int cp1_pid, ..., int cpn_pid to hold the child PIDs. Similarly, declare mtype variables, mtype cp1_code, ..., cpn_code for each component. The template for the declares is as follows:

```
int cp1_pid, cp2_pid, ..., cpn_pid;
mtype cp1_code, cp2_code, ..., cpn_code;
```

The declarations are placed at the beginning of the parent proctype. Then, the following template is used:

```
/*
        Compound labels for a transfer to the concurrent
        state component boundary.
*/
to_cp1:
to_cp2:
.
.
.
to_cpn:
cp1_pid = run cp1(none);
cp2_pid = run cp2(none);
.
.
.
cpn_pid = run cpn(none);
wait??eval(cp1_pid),cp1_code;
wait??eval(cp2_pid),cp2_code;
.
.
.
wait??eval(cpn_pid),cpn_code;
/* The join sequences are optional */
<join sequence 1>
.
.
.
<join sequence n>
```

The meta term <join sequence> is either assert(0) if no join construct has been specified, or one or more join sequences as described in Rule Promela 18. The concurrent composite state is activated by transferring control to any of the labels to_cp1 through to_cpn.
Relationship to Homomorphism
A **ConcurrentProctype** is a type of **Proctype** as described above. This rule details the specialization for the **ConcurrentProctype** and also preserves the join relationship by setting up the mechanisms required to map **join** to **Wait-join**.
End of Rule Promela 9.

Figure 6.14 shows the statements generated by Rule Promela 9 by the transition from state f to the concurrent composite state component in Figure 6.10. Lines 4 and 5 start *cp*1 and *cp*2 while lines 7 and 8 wait for both components to exit. The *join* construct to rejoin threads from concurrent components is detailed in Rule Promela 18.

```
1  to_cp1:
2  to_cp2:
3                  atomic {
4                  cp1_pid = run cp1(none);
5                  cp2_pid = run cp2(none);
6                  }
7                  wait??eval(cp1_pid),cp1_code;
8                  wait??eval(cp2_pid),cp2_code;
```

Figure 6.14: Promela specifications generated for class A in Figure 6.8.

Events and Signals

Events and inter-object messages (signals) are mapped to send and receive operations on Promela channels. Informal UML semantics specify events to be lost if the transition for that event is not ready [7]. To implement this policy without filling Promela channel queues, extra statements are required that work as follows: When a transition for event E is ready, the state sends its PID and the event name, E, to channel **evq** with the statement **evq!E,_pid**. Waiting for event E is accomplished with a poll ("??" operator) on channel **evt** using the statement **evq??E,eval(_pid)**. To post the occurrence of event E, process **event(E)** is run. The **proctype event()** is included in the mapping by Rule Promela 1.

When **event(E)** runs, it searches queue **evq** for a matching event, E. A match implies a ready transition in the **proctype** whose PID is in the message, therefore, **event()** sends a message to channel **evt** to cause a transition. The transition message consists of the event, E and the PID.

All of process **event()** executes in an atomic block so statement interleaving is not possible while events are being handled. Line 6 in Figure 6.15 contains state a's notification that it is ready for event $t2$. Line 10 shows the corresponding wait for event $t2$.

Rule Promela 10
An intra-object transition is formalized as a send operation sending a message containing the event name and PID to channel evq to signify the transition's readiness, and a receive statement that waits for the same message to occur on channel `evt`. The following template shows the Promela construction:

```
evq!e1,_pid;
evq!e2,_pid;
.
.
.
evq!e1,_pid;
if
::  evt??e1,eval(_pid) -> <transition actions>
::  evt??e2,eval(_pid) -> <transition actions>
.
.
.
::  evt??e1,eval(_pid) -> <transition actions>
.
.
.
fi;
```

Events are dispatched by running the model-wide procedure `event()`.
Relationship to Homomorphism
This rule complies with the **Transition** to **Transition** mapping and complies with the `trigger` relationships between *Event* and **Transition**. As the metamodel depicts, there are two subtypes of **Event-Dispatch**. This rule also covers the **Run-Event()** subclass.
End of Rule Promela 10.

Figure 6.15, lines 6 and 10 show the statements generated by Rule Promela 10 for the transition from state a to state c in Figure 6.10.

Inter-object messages, also called *signals*, can use either a queued or non-queued semantics, as explained earlier. When non-queued semantics is chosen, the inter-object message passing uses the same event transfer mechanism as is used for intra-object events. When inter-object signals use queued semantics, a Promela channel is declared for each object. Signals are placed directly on the queue by channel send operations, and transitions map

```
/* State a */
a:              skip;
              printf("in state A.a");
              H_cp1 = st_a; /* Save state for history */
in_a:
              evq!t2,_pid;
              if
              :: A_q?t5 -> wait!_pid,st_join1; goto exit
              :: A_q?t1 -> t1_p1?A_V.vara -> goto b
              :: evt??t2,eval(_pid) -> goto c
              fi;
```

Figure 6.15: Promela specifications generated for state *a* in Figure 6.10.

to direct queries of the channel.

Rule Promela 11
An inter-object transition event using queued semantics is formalized as a receive operation on the set of channels corresponding to the event and its parameters generated by Rule Promela 6. If signal parameters are present, then pass each in a channel named for the signal per Rule Promela 6. Each parameter channel is declared as a single entry message queue of the same type as the signal parameter. An inter-object signal send is formalized as a set of channel write statements to send the parameters and signal to the channels associated with the destination object. For class **CLASS**, and inter-object signal E that receives message values into instance variables `var1, var2, ..., varn`, the following template shows the message send operation:
`E_p1!parm1; E_p2!parm2; ... E_pn!parmn; CLASS_q!E;`
and the corresponding receive and transition statements are:

```
if
:: CLASS_q?E -> E_p1?CLASS_V.var1;
                E_p2?CLASS_V.var2;
                .
                .
                .
                E_pn?CLASS_V.varn -> <transition actions>
.
.
.
fi;
```

Relationship to Homomorphism
This rule complies with the **Transition** mapping and also reflects the superclass–subclass structure of **Event-Dispatch** in the Promela metamodel. The `incoming` relationship between **Transition** and *State Vertex* is preserved by the Promela constructs specified above.
End of Rule Promela 11.

Technically, one channel could be used for both signal and parameters, or one for signals and another for all parameters, however, the SPIN analyzer generator apparently has a bug that manifests itself when channels with multiple parameter messages are used heavily. Therefore, this rule specifies a workaround solution that is equivalent, although somewhat more cumbersome.

Line 8 in Figure 6.15 shows the transition statement generated by Rule Promela 11 for signal *t*5 in state *a* from Figure 6.10. Line 9 of Figure 6.15 shows the transition statements in state *a* for event *t*1 that has an integer value associated with it. The value is stored in instance variable **vara**.

UML transition syntax allows a transition expression with no event specified. When this occurs, the missing event is mapped to a wait for value "1", or "true", which never blocks.

Rule Promela 12
Formalize a missing event on a transition as "1", or true.
Relationship to Homomorphism
This rule complies with the **Transition** mapping and the *Event* aggregation. As noted on the metamodel, a transition does not require an event. When absent, this rule provides semantics for the missing event.
End of Rule Promela 12.

Transitions

Transitions are formalized as **goto** statements. For transitions within a composite state, the target is a label on the sequence of statements representing the next state. For transitions to a child composite state, the target label is on a block of one or more `run()` statements, which instantiate the child composite state.

Rule Promela 13
An intra-composite-state transition is formalized as a **goto** transferring control to the label on the *State Block* representing the destination state.
Relationship to Homomorphism
This rule complies with the **Transition** mapping and the relationships with *StateVertex*. This rule also reflects the superclass–subclass relationship for **Event-Dispatch** in the Promela metamodel.
End of Rule Promela 13.

Line 10 in Figure 6.15 shows a **goto** generated by Rule Promela 13 for the transition from state *a* to state *c* in Figure 6.10.

Transitions that cross a composite state boundary go either upward or downward in the state hierarchy. For transitions that require upward traversal in the composite state hierarchy, the name of the destination state is written to channel `wait`, and the procedure exits. Upon, exit, the parent retrieves the name of the destination and executes appropriate statements to transfer to the block of statements representing the destination. For downward transitions, a `run()` statement with a parameter indicating the destination state is used. An entire cross boundary transition may require combinations of upward and downward transitions until the destination is reached.

Rule Promela 14
A Transition that crosses the composite-state boundary downward in the state hierarchy is formalized as a `run()` statement that starts the child composite state `proctype` that either contains the destination state, or is on the transition path to the destination state. If the destination state is not the immediate child composite-state boundary, then the `run()` statement contains a parameter naming the destination state. Assume the next composite state downward on the path to the destination is `CSTATE` and the destination state is named `Dest`, then this portion of the mapping generates the statement:

```
run CSTATE(Dest);
```

Relationship to Homomorphism
This rule further describes the **Transition** mapping and preserves the relationships with *StateVertex*, specifically, down to the **Proctype** Promela metamodel class.
End of Rule Promela 14.

Rule Promela 15
A transition that crosses a composite-state boundary upward in the state hierarchy is formalized as a channel send operation followed by an exit from the composite state. The message sent consists of the current processes PID and the name of the destination state. The statement template for the transition to state `Dest` is:

```
wait!_pid,Dest; goto exit;
```

Relationship to Homomorphism
The comments for Rule Promela 14 are applicable to this rule.
End of Rule Promela 15.

To illustrate Rules Promela 14 and Promela 15 we examine several example sections of Promela specifications. The transition from state e to simple state $d2$ within state d in Figure 6.10 is a downward transition. Rule Promela 14 generates the Promela statements on lines 9 and 22 in Figure 6.16 for this transition. Line 6 in Figure 6.17 shows the statements generated by an upward transition from state $d2$ to the *join* construct via the transition labeled $t4$. Line 7 in the same figure illustrates the transition up one level to state e. The signal for this transition ($t1$) has one integer parameter.

Transitions may be guarded by an arbitrary logical expression. When a transition guard is present, an `if-fi` block is constructed after the event reception statements to test the guard. Further transition actions, if any, are placed after the guard and before the **goto** statements per Rule Promela 7. When multiple transitions with the same event and varying guards occurs, all of the transitions are gathered together to form one event test with the guards enumerated in an `if-fi` block, with each guard representing one guard. If none of the guards are true, then the event should be dropped and waiting resumed. This is accomplished by placing an `:: else goto in_CLASS` statement as the last statement of the `if-fi` guarding block, which causes control to loop back to the start of the state.

```
        proctype cp2(mtype state)
        {atomic{
        int d_pid;
        mtype m;
        int dummy;
        /* Init state */
                 goto e;
        /* Link to composite state d */
to_d:          d_pid = run d(m);
                 wait??eval(d_pid),m;
                 if
                 ::  m == st_join1 -> wait!_pid,st_join1; goto exit;
                 ::  m == st_e -> goto e;
                 fi;
        /* State e */
e:             skip;
                 printf("in state A.e");
in_e:

                 evq!t2,_pid;
                 if
                 ::  A_q?t4 -> wait!_pid,st_join1; goto exit
                 ::  evt??t2,eval(_pid) -> m = st_d2; goto to_d
                 ::  A_q?t3 -> m = st_d1; goto to_d
                 fi;
exit:          skip
        }}
```

Figure 6.16: Composite state *cp2* from diagram Figure 6.10.

```
/* State d2 */
d2:            skip;
               printf("in state A.d2");
in_d2:
               if
               ::  A_q?t4 -> wait!_pid,st_join1; goto exit
               ::  A_q?t1 -> t1_p1?A_V.vara -> wait!_pid,st_e; goto exit
               ::  A_q?t3 -> if
                  ::  A_V.g1 -> goto d1
                  ::  else -> goto in_d2
                  fi
               fi;
exit:       skip
}}
```

Figure 6.17: State *d*2 from Figure 6.10 illustrating the statements generated by Rule Promela 15 for an upward transition.

The semantics of eventless transitions calls for the transition to be taken as soon as the actions on the transition are complete (see Rule Promela 12). If the transition is guarded, then the guards are tested immediately, and if false, the state is blocked (because the *event* cannot reoccur). To produce blocked semantics, the final "`:: else`" is replaced with "`:: else 0`". Since "0" is false, it blocks, and since zero is a constant, it never becomes true. Without this construction, guarded, eventless transitions would behave like "When" clauses because each guard would be constantly re-evaluated.

Rule Promela 16
Formalize a guard on a transition as an `if-fi` block located immediately after the statements representing the event reception. Gather all like events into one wait statement. For one or more transitions on event `E` guarded by `G1` through `Gn`, and wait expression `Q` (`Q` is some form of a wait on a channel per Rule Promela 10, Rule Promela 11, or Rule Promela 12) the following template applies:

```
if
::  Q?E -> if
           ::  G1 -> <transition 1>
           ::  G2 -> <transition 2>
           .
           .
           .
           ::  Gn -> <transition n>
           ::  else goto in_STATE;
           fi
.
.
.
```

For eventless transitions guarded as above, replace the final `else` with `else 0`.
Relationship to Homomorphism
This rule complies with the **Guard** to **IFGuard** mapping and preserves the aggregation between **Guard** and **Transition** in the UML metamodel.
End of Rule Promela 16.

The top half of Figure 6.18 shows the rather extreme case of multiple transitions with the same event and different guards.

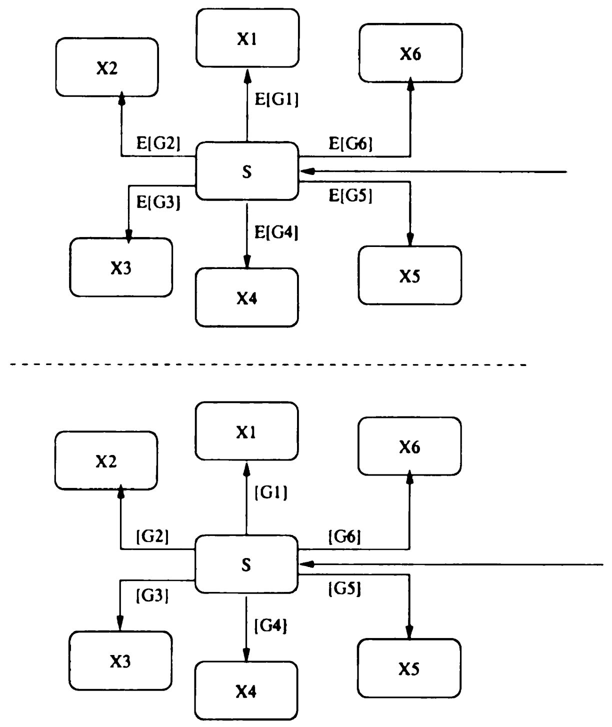


Figure 6.18: An extreme example of multiple transitions on a single event guarded by varying guards. The bottom half of the figure shows eventless guarded transitions.

Figure 6.19 shows the Promela generated by Rule Promela 16 for state S in the bottom half of Figure 6.18. Since the event is the same on each transition, it has been gathered into one wait.

Figure 6.20 shows the result when the event is removed from the transitions as in the bottom half of Figure 6.18. In this case, Rules Promela 11 and Promela 16 work together to produce the `if-fi` block shown. Notice in this case the final `:: else 0`. Once state S has been entered, if no guard is true, then the state is blocked.

Promela contains no constructs or means of simulating the passage of real time, therefore the UML event *TimeEvent* is mapped to a sequentially numbered event called $TE1$, $TE2$, *etc.* . The modeler must make provision to generate these timed events elsewhere in the model.

```
in_S:
        if
        :: CLASS_Q?E -> if
                        :: G1 -> goto X1;
                        :: G2 -> goto X2;
                        :: G3 -> goto X3;
                        :: G4 -> goto X4;
                        :: G5 -> goto X5;
                        :: G6 -> goto X6;
                        :: else goto in_S;
                        fi
        fi;
```

Figure 6.19: The Promela generated by Rule Promela 16 for state S in the top half of Figure 6.18

```
in_S:
        if
        :: 1 -> if
                        :: G1 -> goto X1;
                        :: G2 -> goto X2;
                        :: G3 -> goto X3;
                        :: G4 -> goto X4;
                        :: G5 -> goto X5;
                        :: G6 -> goto X6;
                        :: else 0;
                        fi
        fi;
```

Figure 6.20: The Promela generated by Rule Promela 16 for state S in the bottom half of Figure 6.18

Rule Promela 17
Timed events are mapped to uniquely named events following the naming convention $TE1, TE2, \dots, TEn$. The modeler is responsible for generating these events elsewhere in the model.
Relationship to Homomorphism
Class**TimeEvent** is mapped to **Event-Dispatch** because there are no constructs in Promela for time-constrained events. Therefore, to complete the mapping, timeouts and interval waits are mapped to user-generated events.
End of Rule Promela 17.

Pseudo States

Four pseudo-states are mapped from UML to Promela: The *Init State*, the *History State*, *Exit*, and *Join*.

The *Join* pseudo-state accepts multiple inbound transitions from two or more concurrent state components generated by Rule Promela 9, joining the separate threads into one thread. When all inbound transitions have occurred, the outbound transition from the join is triggered. There may be zero or more *Joins* for each concurrent composite state. When no *Join* exists, ending a thread in a concurrent component is an error. When more than one *Join* is present, each concurrent component must transition to the same *Join* or else a deadlock occurs.

As described in Rule Promela 9, initiation of a concurrent composite state includes statements that issue receives on channel `wait`, causing the parent to wait until each concurrent statement has terminated. When all concurrent Promela procedures have exited, control falls through to the `<join sequence>` in the template in Rule Promela 9. At this point, without a *Join* construct, an `assert(0)` statement is executed, causing the model to terminate in error.

When one or more *Joins* are present, the `<join sequence>` consists of an `if-fi` block containing guards that are strings of conjuncts testing for each *Join* as a destination.

Rule Promela 18
Each *Join* pseudo-state is formalized as a `if-fi` block. For each *Join*, construct a guard consisting of a conjunct testing each concurrent component's return code for a transition to that *Join*. The *Join* template is as follows:

```
if
::  cp1_code == st_join1 && cp2_code == st_join1 ...  && cpn_code == st_join1
      -> goto dest1
::  cp1_code == st_join2 && cp2_code == st_join2 ...  && cpn_code == st_join2
      -> goto dest2
.
.
.
::  cp1_code == st_joinn && cp2_code == st_joinn ...  && cpn_code == st_joinn
      -> goto destn
fi
```

Where **cpn_code** is the return code from the n-th process indicating the join state, `st_joinn` is the name of the n-th join, and `destn` is the destination from the n-th join.
Relationship to Homomorphism
This rule complies with the mapping of **Join** to **Wait-join**. In Promela, the `join` construct occupies the same relative position as a state, and has a destination label.Therefore, it qualifies as a subtype of *Pseudostate*.
End of Rule Promela 18.

Figure 6.21 shows the `<join sequence>` statements generated by Rule Promela 18 for the `join` in Figure 6.10. This `join` rejoins the threads from *cp*1 and *cp*2 through transitions labeled *t*4 and *t*5. The *Join* construct works in Promela as follows: The sequence of `waits` after the `run()` statements blocks until all concurrent components have ended. For example, lines 7 and 8 in Figure 6.21 block until both *cp*1 and *cp*2 have exited. After all threads have ended, the return codes will be set. If the set of return codes matches one of the conjuncts in the `join if-fi` block, the join is complete. Otherwise, a deadlock could occur. On line 10 of Figure 6.21, the `join` is waiting for transitions to `join1`.

An initial state has only to transfer control to the designated start state in the composite component. This is accomplished with a `goto` at the beginning of the `proctype`.

```
to_cp1:
to_cp2:
               atomic {
               cp1_pid = run cp1(none);
               cp2_pid = run cp2(none);
               }
               wait??eval(cp1_pid),cp1_code;
               wait??eval(cp2_pid),cp2_code;
               if
               ::  cp1_code == st_join1 && cp2_code == st_join1 -> goto f;
               fi;
exit:       skip
}}
```

Figure 6.21: The Promela specifications generated by Rule Promela 18 for the `join` in Figure 6.10

Rule Promela 19
Each *Initial* pseudo-state is formalized as a **goto** to the initial state. The **goto** is located at the start of the Promela procedure associated with the composite state or object.
Relationship to Homomorphism
Init-goto is mapped from **Start**. The Promela implementing `init` fits within a composite state or object envelope, and has an outbound transition. Therefore, it qualifies as a subtype of the supertype *Pseudostate*.
End of Rule Promela 19.

Lines 6 and 7 of Figure 6.16 show the initial state mapping generated by Rule Promela 19 for state *cp*2.

A **history** state is a special initial state that remembers the last active state of a composite state after the composite state has exited. When the composite state is re-entered via a transition to its boundary, the history state arranges to make the last active state current again. When there is no prior active state, the history state acts as a normal initial state and transfers to the default start state.

Rule Promela 20
Formalize each *History* pseudo-state as an `if-fi` block at the top of the Promela procedure representing the composite state. The `if-fi` block tests the value of a variable holding the name of the last state. When a match is found, the `if-fi` transitions to the state with a goto as in other transitions. The `if-fi` block follows the following template:

```
if
:: H_composite-state-name == st_state1 -> goto state1;
:: H_composite-state-name == st_state2 -> goto state2;
.
.
.
:: H_composite-state-name == st_staten -> goto staten;
fi;
```

Where `composite-state-name` is the name of the composite state (objects may not contain history pseudo-states at their top level), and `statei` is the name of the i-th state in the composite state. When a *History* state is present, line 2 in the template in Rule Promela 7 must be present and the variable `H_composite-state-name` is declared as type `mtype` in the global declarations. The initial value of `H_composite-state-name` is the default initial state.
Relationship to Homomorphism
The comments for Rules Promela 18 and Promela 19 apply to this rule, which maps **His-troy** to **History-goto**.
End of Rule Promela 20.

Lines 6–10 of Figure 6.22 show the statements generated by Rule Promela 20 for the history state. Figure 6.22 shows the statements for the first part (through state a) of composite state $cp1$ from Figure 6.10. The statement generated by Rule Promela 7 on line 14 shows the history variable being set to state a.

There are two possible semantics for *exit*: one is to return to the next higher level in the state hierarchy, and the other is to execute a "0", causing a block. If this occurs deep within the hierarchy of states, the entire branch will be halted. We have chosen the former semantics for *exit*. The *exit* or *final* state construct is mapped to an exit of the current `proctype`.

```
proctype cp1(mtype state)
{atomic{
mtype m;
int dummy;
/* History pseduostate construct */
          if
          :: H_cp1 == st_a -> goto a;
          :: H_cp1 == st_b -> goto b;
          :: H_cp1 == st_c -> goto c;
          fi;
/* State a */
a:          skip;
          printf("in state A.a");
          H_cp1 = st_a; /* Save state for history */
in_a:
          evq!t2,_pid;
          if
          :: A_q?t5 -> wait!_pid,st_join1; goto exit
          :: A_q?t1 -> t1_p1?A_V.vara -> goto b
          :: evt??t2,eval(_pid) -> goto c
          fi;
```

Figure 6.22: History state statements generated by Rule Promela 20 for composite state *cp*1 in Figure 6.10.

Rule Promela 21
Formalize each *Final* pseudo-state as a transfer (goto) to label **exit**: at the end of the containing Promela procedure.
Relationship to Homomorphism
This rule complies with the mapping of **Final** to **Goto-exit**, and the previous comments for Pseudostates apply.
End of Rule Promela 21.

Rules Promela 2 and Promela 8 both require a final **exit**: label at the end of the **proctype**. An **exit** simply transfers to this label.

Chapter 7

Related Work

In this chapter we overview related work. The objective of the research is to show how to formalize UML so that it can be used for embedded systems design. The formalization then enables behavior validation and other types of analysis on the embedded systems model. The literature related to this objective includes embedded systems development methodologies, both object-oriented and non-object oriented. In addition, we review approaches to formalizing object-oriented graphical modeling techniques, and formalizing UML in particular. Some of the related work covers both formalization and embedded systems; some includes related work containing only one aspect.

There are many graphical notations for embedded systems, some of which are closely associated with specific complete methodologies, while others are associated only with techniques for analyzing or describing a system. We review some of the more widely used notations.

7.1 Embedded Systems Methodologies

Methodologies for building embedded systems cover the spectrum from *ad hoc*, to structured methods, to combined structured and object-oriented to purely object-oriented. Five major approaches are reviewed:

- *Ad Hoc*
- Structured techniques: Real Time Structured Analysis (RTSA) of Ward and Mellor [38] and the similar method from Hatley and Pirbhai [39]
- Hybrid structured and OO: Real Time Object-Oriented Structured Analysis (RTOOSA) [37]
- OO, non-UML based: Realtime Object Oriented Method (ROOM) [5]
- OO, UML based: Real-Time UML [1]

7.1.1 Ad Hoc

Over the past 25 years, the majority of commercial system software houses used an *ad hoc* method to build embedded systems [2]. The starting point of this approach is a set of requirements, sometimes written, other times verbal, which, in the best case, are transformed into an initial design by skilled engineers. The form of the initial system depends heavily on the experience and background of the designing engineer. If the worst case, programmers write code directly from requirements with no intervening process. Unfortunately, this is a very common industrial practice. Since hardware is seen as a recurring cost in the final product, software design is often considered a throw away nuisance and hence a mentality can develop where the primary goal is simply getting the code to run. A common practice is to develop an initial version of the system running in real hardware, then couple it with the intended physical system for initial debugging [42].

7.1.2 Structured Methods

Real Time Structured Analysis (RTSA) techniques are well represented by the methods of Ward and Mellor [38] and the methodologies suggested by Hatley and Pirbhai [39]. The two methods are very closely related and are considered together in this section.

Real-time and embedded systems structured approaches have their roots in classical structured analysis techniques used for large mainframe systems [43]. Structured approaches use modified forms of data flow diagrams to denote the flow of information through the system. The common modification used by Ward and Mellor [38], and later by Hatley and Pirbhai [39], is to augment the data flow with control flows, denoted with dotted lines, that show how a transformation, or "Process" is modified or turned on and off.

The methodology starts with the construction of a *context model*, or *level 0* model, that contains a single transform, or process, and all of the relevant objects in the environment. Special emphasis is placed on the delineation between the system and the environment. The single process representing the system is further broken down into *process schemas* at higher levels of detail. Each transformation at each level can represent a complete set of lower level transformations, thus providing a hierarchical view of the system.

Inherently, data flow models do not show process sequence or control. RTSA introduces sequence numbers on the transformations to provide a sense of how the flow occurs within a system. Similarly, embedded systems must deal with a large amount of control details, hence these methods introduce Control Flow Diagrams, (CFD) to show how control flows through a system. Ward and Mellor explicitly introduce processes drawn as dotted line circles that are control-only in nature. Underlying a control process is a finite state machine, described elsewhere in the RTSA documentation.

Hatley and Pirbhai's notation replace the dotted line "bubble" with a vertical bar that has control flows moving to and from it. The vertical bar also represents a finite state machine documented elsewhere in the model.

Refinement from requirements in the context model to final code proceeds in several steps. First, each process is refined by creating lower level processes. This type of refinement serves to decompose the problem, but it is clearly a top-down methodology, and thus dependent on the initial partitioning of the problem. Hatley and Pirbhai quote Miller's [44] "seven-plus-or-minus-two" principle[1] to determine how much decomposition is required at any particular level. Basically, this principle states that humans deal best with from five to nine objects at a time, and therefore, according to the theory, a requirements model with more or less than these numbers is suboptimal with respect to the reader's understanding.

At the bottom of the decomposition structure, each process or flow is described with a PSPEC for processes, or CSPEC for controls (Hatley-Pirbhai nomenclature). PSPECs and CSPECs contain "structured English" and perhaps equations to describe exactly what is to happen with the control or transformation. This level of the process constitutes the design, and it is the intention that from this material the code is generated.

Structured methods are not highly amenable to computer-based analysis, except for certain consistency checks on the data-flow models. Furthermore, there is a conflict between the flow-based nature of the specification and the fact that most embedded systems are inherently state-based. When perusing the code, it is difficult to trace the flow from the requirements analysis information. In general, there is a problem with the correspondence

[1]Actually, this principle is quoted in a number of works related to graphical methods including Douglass [36] and Ellis [37].

between the analysis view and the detail design view because the code writing process and the data modeling process are fundamentally disjoint [45].

Comparison with our Approach

Although we share the conjecture that graphical models are easier to construct and comprehend than other model forms, we differ on other aspects of the design process. The emphasis of our approach is precise, even executable, semantics while the semantics of the RTSA models are at best semi-formal [32]. Furthermore, we embrace the object-oriented methodology as a superior design paradigm, while RTSA uses a top-down functional decomposition approach. In our approach, the behavior of the system can be continually checked through simulation by executing use case scenarios while in RTSA behavior can only be checked by writing code.

7.1.3 RTOOSA

Ellis proposes a methodology called Real Time Object-Oriented Structured Analysis (RTOOSA) [37] that uses structured analysis of Ward–Mellor's technique and adds objects. The modeling heuristics and notation closely follow structured analysis techniques but the models differ somewhat from RTSA. Essentially RTOOSA replaces RTSA processes with objects. RTOOSA uses two major modeling components called the *essential requirements model* and the *essential objects model.*

The essential requirements model consists of the *external interfaces diagram (EID)*, the *external events/response list (EERL)* and *object relationship diagrams (ORD).* The EID is the counterpart to the context diagram in structured methods and delineates the external environment from the system boundary. The most important component of the

RTOOSA models is the EERL. The EERL is a simple, numbered list of every event that the system expects to encounter and the required system response. The entire list is written in natural language. For example, system responses might be written: (1. Do activity A, 2. Do activity B, *etc.*). The EERL also contains information on event precedence and contains other notations about the type of the event. The EERL is frequently referred to throughout the methodology and provides the source of information for other components of the requirements model.

The object relation diagram (ORD) highlights interdependencies among objects in the problem space. We note that the ORD is not limited to objects in the system, but rather includes environmental objects. The ORD is very similar to an Entity Relationship diagram and details the static data relationship between objects. The ORD also provides candidates for *object flow diagrams* that will model the behavior of the system.

The other major component of the requirements model, called the *essential objects model* contains three components: *object flow diagrams (OFD)*, *object templates*, and the *data dictionary.*

The OFD is equivalent to a process schema in RTSA except processes are replaced with objects. RTOOSA has the explicit convention that objects can be decomposed, and in fact are treated as subsystems. This parallels the RTSA approach for decomposing process schemas. Object templates describe the fields and methods in each object and are auxiliary to the OFD itself.

Creating classes in RTOOSA is considered a design activity. The emphasis is on objects without regard to the class in which they belong. During the design phase, objects are simply "appropriately" grouped into classes for the implementation phase.

Comparison with our Approach

Even though objects are used in RTOOSA, the methodology is still very much a structured approach. Detailed behavior is not attributed to object methods other than through the EERL natural language descriptions. RTOOSA emphasizes capture of requirements in an easy to understand (if inconsistent) manner. By contrast, our method emphasizes formal semantics for notations and places a higher level of emphasis on verifying and validating behavior. On the other hand, the EERL appears to be a useful device for collecting events and responses.

7.1.4 ROOM

Real-Time Object Oriented Modeling (ROOM) [5] is an object-oriented method for real-time and embedded systems based on a formal, executable modeling language. Special emphasis is placed on modeling the architectural levels of the software. The method includes graphically based modeling notations for object relations and for behavioral descriptions.

The object-equivalent in ROOM is called an *actor* and is central to modeling. Actors encapsulate behavior and state, and are only able to communicate via structured messages through *ports*. ROOM is distinguished from many methods by the rigor of the message passing specification and messaging protocol. Syntactically, a ROOM message is a data object that incorporates a message signal attribute, a message priority attribute and an optional data object. Messages are grouped into *protocols* that define a set of incoming messages, a set of outgoing messages, and optionally a valid message exchange sequence (not currently implemented in ROOM tools). A protocol association with a port conveys a type to a port, and the type of an actor is determined by the collection of its port types. Protocols are a central ROOM concept to both modeling and analysis.

Actors can themselves be composed of networks of actors that are connected and communicate within the composite actor. Arbitrary levels of hierarchical decomposition is permitted.

Behavior is considered an optional attribute of each actor, including composite actors. Thus, a composite actor might have its own behavior attribute separate from the behavior of it constituents. Graphically, behavior is not written with the actor notations but rather in ROOMcharts. ROOMcharts are based on event-driven behavior and a variant of Statecharts [25] modified in order to meet the needs of the real-time environment and take advantage of the object paradigm. State transitions are triggered by events received as messages through actor ports.

The ROOM development methodology is suggested as a series of modeling heuristics rather than a formal procedure. The process starts with the construction of a set of requirements written as a set of scenarios covering typical actions of the system. An early step is the determination of the system boundary from the environment, then scenarios from the requirements help derive key protocols that cross the environment-system boundary. Next, an early structural view of the system is captured with an initial set of actors. The initial set of actors is usually implied by the physical components described in the scenarios. Actor structure, protocol structure, and consequently data structure are then further defined. The behavior of an actor is derived by studying the scenarios it must support and the protocols to which it responds, although no mention is given as to how to decide on actor responsibilities. Since ROOMcharts are executable, it is a goal to validate scenarios as early in the process as possible through simulation.

Comparison with our Approach

ROOM shares several commonalities with our approach such as the ability to execute models, early simulation, and the use of objects, but similarities end there. ROOM's semantics are defined by the ROOM language, which describes "actors" instead of objects, while in our approach any semantics is possible depending on the homomorphic mapping and the target language chosen. ROOM appears to have no facilities for model checking or any other form of validation other than simulation. In our approach, the formal language can be the language of a model checking system. Similarly, ROOM has no facilities for modeling precise timing constraints. In our approach, we have mapped UML semantics to SPIN for model checking and to VHDL for precise timing semantics. Although ROOM is growing in popularity, UML is already in widespread industrial use, as are the two languages (VHDL and Promela) we have targeted. Furthermore, as new, better languages appear, either in the industrial realm or in research settings, we can construct a homomorphic mapping to define UML semantics in that language.

The concept of an "actor" in ROOM is slightly askew from the model implied by UML. We are able to match and formalize the UML informally accepted semantics of objects through out metamodel mapping approach. ROOM also emphasizes "protocols" between objects, but the semantics of communications between objects is fixed. In our approach, as seen in Chapter 6, the semantics of message passing between objects can be tailored to suite the application. Dynamic model transitions in ROOM have limited capability, but through our mapping, we can realize the full range of capabilities on a UML transition.

Finally, ROOM is based on no visual notation other than ROOM-charts, for which there is no formal description, while our approach is based on UML metamodels. This has two implications: First, we can check for consistency between visual models and executable models, while ROOM cannot. Secondly, as UML evolves, and its metamodel is modified, our approach can follow the evolution by altering the metamodel to metamodel mappings.

7.1.5 Real-Time UML

Douglass [1] proposes using UML, with slight modifications, for the construction of real-time and embedded systems. Rather than a fixed methodology, a set of OO guidelines, based on UML doctrine, are suggested. The methodology captures requirements with use case scenarios and class models. Douglass suggests starting with a context diagram similar to Ellis [37] and Ward–Mellor [38], except the diagram is expressed in use case notation. The context diagram delineates system packages from the environment and documents major system level inbound events and their corresponding responses.

The methodology next proposes building the class model. Based on the use cases, several strategies are suggested to define the object/class structure. These include:

Underline the Noun. The first strategy works directly on the concept statement prior to the construction of use cases. Fundamentally, it calls for building a table by underlining each noun or noun phrase in the concept statement to discover potential objects in the system. This approach has been widely suggested before [3, 46].

Identify Real-World items and physical devices. This strategy correlates real artifacts of the proposed system with software objects. For embedded systems, this correlation is often very good and produces a reasonable class structure.

Identify key concepts. Key concepts are important abstractions within the system description. For example, a task schedule could be an important object type for a controller.

Identify transactions. Some types of transactions arising out of the interaction of objects are objects themselves. In the previous example, a schedule (an object) might have an originator, a modifier and a user, whose uses are based on the transactions surrounding the use of the schedule.

Next, use case scenarios are applied to the constructed class structure for refinement of the structure, if required, and to determine behavior for each class/object. Use case information also drives the construction of UML dynamic models, one per object, which will define the dynamic behavior of each class.

Douglass advocates the application of *design patterns*, which are abstract generalizations of a type of problem. No specific process is provided to find, then transform a design pattern into a design. However, once a pattern has been chosen, the methodology suggests rearranging and adapting the class structure to fit the pattern deemed appropriate for the application.

Comparison with our Approach

Because our primary focus for designing embedded systems is UML, we have much in common with this approach in terms of methodology. We also suggest the context model be constructed early and generally follow a similar procedure for arriving at the class model. However, we differ in a number of significant points. First, we advocate the delineation of class responsibilities early where Douglass does not. Perhaps the greatest difference between Douglass's approach and ours lies is verification of behavior. We assign rigorous semantics to the UML diagrams and advocate an iterative approach of simulation and analysis where Douglass's methods is largely based on paper analysis of UML notation and a reliance on design patterns. Douglass relies solely on a visual analysis of semi-formal diagrams that are analyzed by hand. A key to our research is the ability to not only know precise semantics for diagrams, but also to able to simulate and analyze system behavior based on the UML diagrams.

7.2 Graphical Languages for Embedded Systems

In this section, we describe graphical modeling languages used for embedded systems and discuss analysis techniques that are possible with each. All languages below, except Petri nets and FSAs, are considered semi-formal since each has a clearly defined set of syntactic rules for constructing a graphical depiction but no consistent underlying formal semantics.

7.2.1 DFD-CFD

Data flow diagrams and Control Flow diagrams are used primarily in structured methods [43, 38, 39, 37], although the *Functional* model of OMT is a data flow model. In DFD notation, circles represent processes or transformations and the directed arcs connecting circles represent flows through the model. The model can be checked for consistency to ensure inbound flows match outgoing flows, but otherwise little behavioral modeling is possible.

Comparison with our Approach

Use of DFD's is incomparable with our approach. UML has no data flow diagrams and data flow diagrams have no description of behavior, and therefore are no executable.

7.2.2 Flowcharts

Flowcharts [47], also known as control-flow graphs, are algorithm-oriented models that depict the flow of control through a set of nodes. Basically, a flowchart is a set of nodes of various shapes connected by directed arcs that depict control flows between the nodes. Rectangular nodes contain a series of one or more statements, often in a programming-language-like notation and specify a specific set of actions to be taken at that point. Diamond-shaped nodes depict a decision point where the flow of control can proceed along one of several (usually two or three) outbound arcs. Some flowchart notations contain further node types, such as circles to depict an I/O operation or an oval to depict a procedure call. Flow between nodes is strictly on a transition on completion (TOC) basis. Each node must complete its action before a transition occurs, and there is no support for event transition. Concurrency is not supported nor is any form of hierarchical decomposition. Flowcharts are used in ad hoc methods and in structured methods.

Comparison with our Approach

Flowcharts are algorithm-based diagrams of models and do not directly represent state machines. Furthermore, flowcharts have no sense of objects, or even modularization. UML sequence and collaboration diagrams are similar to flowcharts, but we do not provide mappings for these two diagrams.

7.2.3 Finite State Machines FSA

Simple finite state machines [48] consist of boxed nodes connected by directed arcs. Each node represents a state and transitions between states are written on the arcs between states. There are a number of analysis techniques for this simple and well-known computer science computing model, and for very simple systems, the state model works well. The major problem from which the FSA suffers is state explosion. Describing even simple processes, like counting, requires a potentially large number of states, called the *state explosion* problem.

State explosion arises because simple FSAs require a separate state to handle any changes in state. A variable that could take ten values requires at least ten states, and where more than one variable is involved the combinations explode exponentially. For this reason, the simple addition of variables to states greatly adds to the state machine model's usefulness.

Finally, it is well-known that a FSA can only represent a class of problems that can be described by regular grammars [48]. Consequently, the *extended* state machine is more widely used for system design.

Comparison with our Approach

Finite state machines are a subset of UML dynamic models. Any FSA can be encapsulated within an object.

7.2.4 Extended Finite State Machine

Extended Finite State Machines are finite state machines in which extended actions, such as setting and testing variables, can appear within a state. Depending on the particular convention for the extended state machine, actions can be carried out while in a state, or during the state transition. Transitions can occur based on simple events, or based on predicates, such as testing the value of a variable in the state machine. The range of actions permitted within a state vary, but can be as extensive as the 'execution' of small subprograms. Extended State Machines are commonly used to describe embedded system behavior because they are reasonably clear while retaining a large amount of expressibility power. Furthermore, extended state machines representing moderately complex systems can be written with far fewer states than a FSA. Extended state machines do not totally solve the state explosion problem, but they do address it to a large extent.

Comparison with our Approach

Extended FSAs are a subset of UML dynamic models and lack most of the facilities of UML.

7.2.5 Statecharts

Statecharts [25] are a particular type of extended finite state machine notation introduced by Harel. The most notable feature of Statecharts is its hierarchical composition of state diagrams. Each state within a Statechart diagram can itself contain a lower-level state machine. Statecharts permit programming-language-like notation within a state to carry out actions on entry to a state, exit from a state and at certain other times specified by event arrivals. Statecharts also assume the existence of *events* that are broadcast across the entire Statechart model and used for transitions. Statechart transitions can be guarded, and action-on-transition is defined for simple actions.

UML has adopted a modified form of Statecharts as its dynamic modeling language. An important modification to Statecharts for UML is that events are not broadcast, but rather must be sent to specific objects.

The principle drawback to Statecharts is determining the exact semantics of any particular Statechart model. Statecharts itself has no official semantics [49] other than is defined by simulators[49] that can execute Statechart models. For example, UML adopts Statecharts transition notation of **`event[guard]/action^message^message...`**. The UML semantics seem to suggest that when **`event`** occurs, the transition is taken if **`guard`** is true [50], which matches Statecharts semantics [49]. It is possible to write a UML transition with no event. The semantics seem to then suggest that a transition occurs when the source state actions are complete (transition on completion concept). What should happen though, on a noevent, guarded transition when the guard is false? The semantics seem to suggest that the model should indefinitely remain in the source state since state actions cannot re-execute, and therefore the triggering "event" (action completion) cannot occur again. However, examples in the UML Notation Manual [50] indicate that the transition guard is evaluated continuously.

Comparison with our Approach

UML uses modified Statecharts for dynamic models. As mentioned above, there is no formally defined semantics for Statecharts, other than that defined by its execution. In our approach semantics is generated by the homomorphic mapping from the dynamic model to a formal language. Therefore, in a sense, we provide semantics for a Statechart-like notation.

7.2.6 Speccharts

Speccharts [51] is a language based on VHDL designed for the specification of embedded systems. Speccharts are based on the *Program-State Machine* (PSM) model, which is a combination of hierarchical/concurrent finite state machine model as found in Statecharts, and a programming model. The PSM model is essentially the same as Statecharts except that *leaf states* may contain an arbitrarily complex sequence of programming language constructs. Each program state in a Speccharts model is one of three types:

Leaf is a program state that is not further decomposed and contains an arbitrarily complex sequence of programming language constructs.

Concurrently-composed is a program state that consists of concurrent sub-states. This is the same concept as found in UML dynamic models. Both Speccharts and UML dynamic models use the same dotted-line notation between concurrent elements of the sub-states.

Sequentially-composed is a program state composed of sub-states that are not concurrent.

The PSM model delineates two kinds of transitions within the state model: *Transition on Completion* (TOC) and *Transition Immediate* (TI). Both types of transitions may be guarded. A TOC arc is traversed when the source program state has finished its computation and the guard is true. A TI arc is traversed when some action specified on the arc occurs.

The textual version of Speccharts is almost identical to VHDL with a few additional constructs. The programming language used within a leaf state is, in fact, VHDL sequential statements. The entire behavior of an embedded system is specified in the Speccharts graphical language except at the leaf states where programming statements are used. The authors claim that behavior difficult to capture graphically in state notation can be easily captured with sequential programming language statements in a much more compact manner.

Behavior of the model is determined by simulation, which is achieved through execution of a VHDL model representing the system. In order to simulate the model, the graphical portion of the Speccharts embedded system description is converted to a textual form that is based on extended VHDL. Next, the extended VHDL statements are translated into VHDL program statements. Finally, the VHDL model can be executed.

Comparison with our Approach

Speccharts is really and extension of VHDL while our approach uses VHDL as a formal target language. While both Speccharts and our approach produce executable models, our executable model is based on a mapping of a visual modeling notation (UML).

7.2.7 Petri Nets

Petri nets are usually represented as bipartite directed multigraphs consisting of *places*, represented by circles, *transitions*, represented by bars, an input function, represented by directed arcs from places to transitions, and an output function represented by directed arcs from transitions to places. Petri nets are *marked* when there are one or more *tokens*, drawn as large dots, in one or more *places*. Formally, a petri net is a 4-tuple (P, T, I, O) representing a set of places, a set of transitions, the input function that maps transitions to places (sources for the transitions), and an output function mapping transitions to places (destinations from a transition).

The execution of a petri net is controlled by the number and distribution of tokens in the places. A petri net executes by *firing* one or more transitions, which moves tokens along each transition's output arc to each place at the end of the arc. A transition is *enabled* if there are tokens in all the input (pointing to the transition) places. A transition is fired if it is enabled and a predicate guarding the transition is true.

Petri nets are versatile as a modeling tool, although large systems require large petri net graphs, usually larger than equivalent state machine models (in terms of the number of nodes and arcs). In addition, a large petri net is not as easy to read as a finite state machine.

Applied Petri Nets [52] are used primarily for modeling highly concurrent processes and have been used in a number of domains. The semantics of petri net execution is formally defined, and an entire theory of petri nets has been established. Petri nets are at least as powerful as finite state machines because finite state machines can be translated to petri nets, although the converse is not universally true.

Comparison with our Approach

Petri nets and our approach are orthogonal, but share some common properties. Petri nets have a formal semantics defined for a petri net diagram, although the semantics is fixed. In our approach, the semantics of the diagram can be varied. Petri nets do not have the concept of an object, while our approach is centered on the object concept. Both approaches produce executable models that can be analyzed with automated tools, but analysis of petri nets is different from analysis of state machines. Petri nets may be more expressive than UML (an FSA can be expressed as a petri net, but the converse is not necessarily true) but their power is most likely incomparable. For example, descriptions of objects and use case scenarios in a petri net seem difficult or impossible. For a give application, petri nets are generally larger and much less intuitive than UML diagrams to the extent that a large system design entirely in petri nets is probably not feasible. UML diagrams are more intuitive, although semi-formal, but our mapping to a target language provides semantics as precise as petri net semantics.

7.3 Formalization of UML

The PUML Project [53, 22, 23, 21] is an effort to formalize the meaning of UML diagrams using the Z language as its formalization vehicle. In [23] the authors suggest three general approaches to formalizing OO modeling concepts: *supplemental*, *OO-extended formal notation* and *methods integration* approaches. The supplemental approach adds more formal statements to supplant the informal parts of a modeling language. In the OO-extended approach, an existing language, Z for example, is extended with OO features. One such example is Z++ [54]. In the methods integration approach, informal modeling techniques are integrated with, or mapped to, formal languages to make them more amenable to formal analysis. Both our approach and the methods of Bourdeau-Cheng [18], and Wang [35] fit into this latter category.

Rather than generate formal specifications from informal OO models and then require designers to manipulate the informal specifications, the PUML approach [21] provides formal semantics directly for the graphical UML notations. The mapping to a formal language is then used as a justification for subsequent manipulations of the graphical form of UML.

In [23] it is pointed out that the diagrams alone are usually not expressive enough to define all properties. They suggest the use of OCL or Z as a non-graphical, precise language to rigorously define properties not expressible in the UML graphical notation.

A major drawback of the current PUML work is its inability to deal with anything but the static structure of UML [23, 21]. In effect this means PUML adds semantics to the class and object diagrams but adds nothing to any of the dynamic diagrams, such as the Statechart diagram or the sequence diagram. This difficulty primarily stems from the nature of Z, in which it is very difficult to describe dynamic behavior.

Comparison with our Approach

According to [23], our approach would be classified as *methods integration* but it is difficult to see how mapping UML class structures to Z differs. In PUML, the mapping from UML to Z is called a *meaning function* but it performs essentially the same function as our homomorphism between metamodels. The major differences, then, are that we map to a variety of formal languages such as VHDL and SMV. Since the formal languages we use include behavior semantics, we can provide a rigorous semantics for behavior as well as structure. Their approach uses only class diagrams while we map class diagrams and dynamic diagrams, and we may be able to map more UML diagrams for precise semantics. On the other hand, the PUML use of Z for UML class structure opens up more avenues for sophisticated analysis such as proving equivalence between class structures (but, again limited to classes). Our intent is to provide a precise semantics so that a practicing designer can precisely determine the properties and behavior of a system under construction, while it appears the PUML effort leans towards attempts to mathematically modify and manipulate class structure. This is not to suggest that we cannot manipulate the class structure. We do, in fact, suggest metamodel modification steps for the target metamodel, which is model manipulation. We feel that direct execution of the model and verification of model properties through analysis is a more practical means for verifying system properties.

7.3.1 OCL

The Object Constraint Language [55] (OCL) is a proposed part of UML intended to constrain class models beyond what is possible through graphical notations alone. The UML class model includes notational constraints but these constraints are written in natural language and are considered to have the status of notes. OCL is an attempt to formalize the class model constraint process.

OCL expressions are either of a predefined OCL type or of a type related to the class model in which the OCL occurs. Dot notation is used to refer to attributes of a class, or more correctly, instance variables of potential objects of a class. For example, for a class *Circle* describing circles in a graphical environment, `Circle.radius > 0` constrains the radius attribute to be greater than zero. OCL expressions can refer to OCL types, classes, and association endpoints. In the latter case, the reference is to the class at the end of the association endpoint. OCL includes aggregate types set, bag and sequence, although an aggregation of an aggregate, such as a set of sets, is always flattened by OCL rules. This turns out to be a fatal flaw in OCL [56] because sets of tuples cannot be represented.

While not actually executable, OCL uses procedural, execution paradigm to specify constraints. Consequently, a constraint frequently includes algorithmic or programming-like portions that specify how some portion of the constraint is to be computed. Included with this capability, OCL has the notion of *navigation* through the class diagram in order to gather values or reach a specific object. It is demonstrated in [56] that OCL is not Turing complete due to its lack of ability to generate `while`-like loops.

A frequent use of OCL is to specify a condition as a query of the object diagram. For example:

```
Polygon.allInstances->select(p :  Polygon | p.vertices->size=3)
```

queries all instances of objects of class *Polygon* looking for those with the *vertices* attribute set to 3. The `select` portion is the actual query while the leftmost part of the statement forms an iterator over the set of objects. This particular query constrains all polygons to be triangles.

Codd [57] proposed that the minimum capability of any reasonable query language using the relational model could be described with *Relational Algebra.* Moreover, any relational algebra must possess five operations: projection, join, union, difference, and selection, in order to be *complete* [58]. It is shown in [56] that OCL is not complete with respect to this model. We must therefore conclude that OCL is not an acceptable language for the formalization of constraints within UML.

Comparison with our Approach

OCL is primarily intended for constraining class diagrams and has no role in the UML dynamic model, while we are primarily concerned with model behavior from a dynamic standpoint. Therefore, OCL and our approach address fundamentally different problems.

7.4 Formalization of OO and Other Notations

7.4.1 Fusion

Fusion [59, 60] is a well-defined (although not formal) development process that leads the development team from initial requirements through implementation of an object-oriented system. The methodology uses a variety of models throughout three primary stages:

- **Analysis:** produces a declarative specification of what the system does.
- **Design:** produces an abstract model of how the system will implement the required behavior.
- **Implementation:** produces the code from the design.

Throughout all phases of the development process important concepts and assumptions are recorded in a *data dictionary.* A data dictionary is a compilation of the structure and properties of every data item used in the system. A data dictionary removes the ambiguity on diagrams when a data item is mentioned only by name.

The process begins with an informal requirements document from which is produced an *object model* and an *interface model.* The object model is a static description of the relationship between potential objects in the system and the environment. A modified form of Entity-Relationship diagrams are used for this model. The model deliberately contains no notion of methods or other dynamic behavior, all of which is deferred until later. Furthermore, classes are not described at this time, also deferred until later in the design phase. An important modeling step is the determination of the *system object model* that is delineated on the object model by enclosing within a system boundary only those objects considered within the system. This step is important because it defines the requirements of the *interface model*, addressed next.

The interface model, which describes the behavior of the entire system, actually consists of three submodels: *operational schema*, *life cycles,* and *scenarios.* The operational model decomposes system behavior into discrete operations based on the exchange of events. The operational model is written in natural language and describes preconditions and postconditions for system operations. Life cycles and scenarios are based on *use cases* and define allowable sequences of operations and events.

During the design phase, responsibilities for behavior are distributed between objects. The Fusion design approach focuses first on deciding how the system operations are to be realized, with actual class structure deferred until later.

The first set of models developed during the design phase are the *object interaction graphs.* One object interaction graph is developed for each system operation to show which objects are involved and how they communicate to realize the system operation. The information for the object interaction graph is derived from the operational model. Objects are denoted on the graph with rectangles and the message flow is denoted by directed arcs. The arcs are annotated to indicate the type of communication and the sequence of interaction. Refinement takes place during the construction of an object interaction graph when a *controller* is chosen from the set of objects involved in the operation. The controller requires *collaborators*, some of which may have not yet been defined. Next, extra objects and communication paths are added as required to complete the object interaction graph.

The design is completed with *visibility graphs*, *class descriptions*, and *inheritance graphs*. When the object interaction graphs are constructed, the assumption is made that all objects have visibility to all other objects. In an implementation, this may not be true, hence the visibility graph is constructed to resolve permanent and transient reference requirements for objects.

Analysis through the Fusion process is implemented as a series of checklists and guidelines. Models are not executable. Coding proceeds from the final set of design models directly.

Comparison with our Approach

Fusion is intended as a methodology for building larger scale systems and is not specifically oriented to real-time and embedded systems. Furthermore, even though the Fusion process is carefully described, the models are at best semi-formal, and mostly informal, because they are written in natural language. Our approach is based on formal semantics and verification of the constructed models. Fusion provides no way to verify a model until the code is constructed.

It has been proposed [61], however, that Fusion be modified to add a Z-like notation to object behavior. The notation is limited to pre- and postconditions on methods and is primarily intended as a refinement tool.

7.4.2 TROLL

TROLL [62] is a formal language designed for the conceptual modeling or requirements phase in the development of a system. TROLL is composed of several sublanguages supporting a declarative, logic-based style of system specification with structuring mechanisms from object-oriented data models. All of TROLL's semantics are defined in terms of translations to temporal logic, including both future-tense and past-tense dialects, so that assertions can be made about the history of objects as well as the evolution of events and attribute values.

TROLL brings together real-time concepts (in terms of events and processes) with database concepts. In this framework, an object is defined as a possible sequence of *events* and *attributes* that are observable properties changed by events. A class is considered a type and is defined by a class template that describes possible instances of the class. In addition to attributes, each object specification contains a list of allowable events along with any parameters the events carry and *valuation rules* that define how attributes values change with the occurrence of defined events. Valuation rules are based on a form of first order logic.

Event preconditions are specified in a *permissions* section of the template. Events are not allowed to occur if the precondition is not satisfied. Similarly, the *obligations* section specifies postconditions that must be met before the object can end its life-cycle. Other sections of the template specify allowable attribute evolution and invariant conditions using temporal logic notations.

A major portion of a class specification is consumed by the behavior specification. A sufficient number of declarative temporal logic formulae to specify behavior may well be quite large. As a convenient alternative, TROLL supports the specification of behavior through *patterns*. A pattern is essentially a process description written in a CSP-like language. The process description references events and can contain event sequence constraints and notations, and references to attribute values within the object.

In order to specify an entire system, there must be a means to specify the relationship between objects. TROLL includes a *relationship* construct that constrains the data values exchanged between objects and specifies the allowable sequence of events for object interactions. The event sequence specifications may be arbitrarily complex and use the full range of temporal operators.

No specific process methodology is identified with TROLL. The language is intended to be used after all relevant objects and events have been identified in the environment. Instead, it is intended that a precise model with rigorous semantics be possible while the specific process for model construction is left to the user.

Comparison with our approach

While TROLL is a formal language with semantics rooted in temporal logic, there is no mechanism to verify behavioral properties through model execution, model checking, or provers. TROLL is also non-graphical, in the tradition of Larch [63], while our approach is largely based on graphical modeling languages. While we start with a semi-formal graphical model in UML and attach precise semantics through mapping to a formal language, the TROLL modeler must work in a formal language from the start. Coupled with the extensive use of temporal logic, the TROLL practitioner will require more knowledge and modeling sophistication than is required in our approach.

7.4.3 Formalized OMT

The Object Modeling Technique (OMT) is a graphically based methodology developed by Rumbaugh, *et al* [3] to facilitate object-oriented design and analysis. OMT models have three primary components:

Object Model: The object model captures the static relationship between classes and is equivalent to the UML class model.

Dynamic Model: The OMT dynamic model describes the behavior of objects in a variant of the Statechart notation. The OMT dynamic model serves the same purpose as the UML dynamic model.

Functional Model: The functional model is a data flow diagram depicting how data flows through and is transformed by, processes within the system.

Bourdeau and Cheng [18] formalized the object model by defining a mapping from the object model to an algebraic specification. In their approach, sets of object instances drawn from an object model represent an algebra that must conform to an algebraic specification also derived from the object model.

Subsequently, Wang [35, 19, 20, 33] formalized OMT by formalizing and combining the three OMT models and suggested a methodology for designing object oriented systems. Wang's approach [19] uses the Bourdeau-Cheng formalization of the object model and formalizes the dynamic model by mapping it to a process model written in LOTOS. To formalize the functional model, Wang [20] redefined its role and introduced two new models: the *object functional model (OFM)* and the *service refinement functional model (SRFM).* The OFM models the services provided by an object and describes data flows in and out of services. Each OFM shows only the inputs and outputs external to the object that implement the functions. Each service is shown graphically within the object as a oval labeled with the name of the service. No further details of the service are depicted in the OFM.

The SRFM refines the services of an object into finer grained services and objects required to implement the higher level service. In this approach, an aggregate object of more detailed objects is produced by refinement in order to implement the behavior of the higher level (aggregate) object.

Wang's development methodology follows a well-defined eight step process [33]. In the first three steps, the system object model, functional model and dynamic model are produced, respectively. Because of the refinement strategy used, the original system is considered one large object containing services required by the overall system. This view is refined into aggregates of objects as the process continues.

In Step 4, the object model is refined in terms of objects and services required within the system. In Step 5, the OFM is constructed based on the refined object model. In Step 6, the dynamic model is constructed based on the services required and the service data flows. Also as part of Step 6, the SRFM is constructed to further refine the objects providing services. This leads to a refined dynamic model in Step 7, and finally in Step 8 all of the aggregate parts are combined in a communicating model. The methodology is iterative with Steps 4 through 8 constituting the iteration loop. Furthermore, the methodology is parallel, for example, the SRFM and dynamic model are developed in concert.

Comparison with our Approach

Wang's formalization and methodology share much in common with our approach. Except for the superficial difference that we use UML instead of OMT, the major difference is in the mapping from semi-formal graphical model to formal model. Our approach uses a meta-level mapping to rigorously specify a homomorphism in terms of mappings between meta-models, where Wang's approach simply states the rules for the mapping. Consequently, we should be able to formalize in terms of more than one formal language and we can also show mechanically that our formalization rules are consistent and complete.

In terms of methodology, we refine objects by iteratively assigning responsibilities to classes, constructing appropriate dynamic models, and verifying behavior. Our domain is limited to embedded systems, hence our emphasis is on the placement of responsibility (in terms of the class) and on the dynamic behavior of the system.

7.4.4 Extended Hierarchical Finite State Machines

Latella, *et al.* [64] have formalized UML state diagrams through a mapping to extended automata, but not in the context of the class diagram. Their approach begins with a translation of the UML dynamic model to *Extended Hierarchical Finite State Machines* (EHFSA). An EHFSA is a translation of the hierarchy implied by nested composite states to an explicit state tree. Composite states are placed on a branch of the tree with all its states and composite states forming the subtree below it. When a state within a composite state is activated, EHFSA semantics define multiple active nodes down the state hierarchy tree. The advantage of this approach is that transitions are simple. Each transition follows the EHFSA tree to the next state. The approach apparently sees cross boundary transitions are a major semantics problem.

In this approach, semantics of the dynamic model is achieved by assigning semantics to the EHFSA. Classes seem to play no part in the formalization.

Comparison with our Approach

Essentially, the EHFSA approach maps the dynamic model to a formal language with a defined semantics. To be precise, the target language is EHFSA. In our approach, we map to any suitable formal language, but the mapping is direct. In their approach, the model must first be transformed into the intermediate EHFSA form, from which semantics are derived.

In a UML dynamic model, the nesting structure of composite states form a hierarchy. However, UML does not require activation of a state by following the hierarchy structure. Transitions can occur from anywhere within the hierarchy to anywhere else within the hierarchy without regard to the hierarchy structure. The EHFSA approach simplifies this problem, but in VHDL, this is also a straightforward issue. The UML to VHDL mapping uses a "state bus" for state activation. All states are listening at all times for their name to appear on the bus. When the name appears, the state is activated. As long as the VHDL state bus hierarchy is constructed to match the state hierarchy, cross hierarchical transitions require no further action at specification generation time. In other languages, such as Promela, the traditional procedure call mechanism is used as part of the transition semantics. In this case, following the hierarchy is more complex. Nonetheless, considering the two languages, we see that cross boundary transitions are a local problem dependent on the target language chosen. We see no reason to orient an entire approach around this single problem as in the EHFSA approach.

THESIS
4
2001

Chapter 8

Tool Development

This chapter describes the prototype tool and its operation that we have constructed to translate UML diagrams into formal language statements. The tool is called *Hydra*[1]

8.1 Overview of Hydra

The purpose of Hydra is to automate the translation of UML class and dynamic diagrams into a chosen formal language. The formal language can subsequently be executed, or used in analysis to simulate or analyze the model. The data flow model for the Hydra translation system is shown in Figure 8.1. The input to Hydra is a textual representation of the class and dynamic models, and to a limited extent, use case scenarios. Hydra parses the textual form of the model, produces an abstract syntax tree (AST), and then processes the AST with a class library written specifically for the target language. The mapping rules derived from the homomorphism are embodied within the class library, and by changing how target language statements are generated, the semantics of the diagram can be altered.

[1]The *Hydra* is a nine-headed serpent from Greek mythology. When any one of its heads was cut off, it was replaced by two others. Our use of the name stems from our framework that permits many formalizations of a language with many semantic ambiguities (UML) from a single base.

The Hydra code for the VHDL translation is contained in Appendix G, and the Promela/SPIN code is in Appendix H.

8.2 Hydra Language

As described in the Overview, the Hydra language is the textual input to Hydra and represents the UML diagram for the model. The complete syntax for the Hydra language is presented in Appendix D.

The Hydra language, which encodes the model, consists of statements in a style that is a hybrid of the C-Language [65] and LaTeX [66]. Curly braces, "{" and "}" delimit statements that define a block, and semi-colons end statements that do not define a block. For example, the entire model is found within the bounds of a **`Model`** { } statement. The statements that can appear immediately within the scope of the **`Model`** statement include **`Object`** { } statements. Similarly, **`State`** { } statements appear within **`Object`** blocks. The **`Transition`** statement describes a transition and, since it does not define a block, ends in a semi-colon.

The Hydra language is described below. As each statement is listed, its purpose is explained, and the scope in which it can appear is listed.

`Model modelname` {...}

The **`Model`** statement defines a block in which all other model statements appear. **`modelname`** is the name of the model. The **`Model`** statement must appear at the top level of the model file. All other Hydra statements appear within the scope of **`Model`**.

`Object classname` {...}

The **`Object`** statement corresponds to a class in the UML class model. **`classname`**

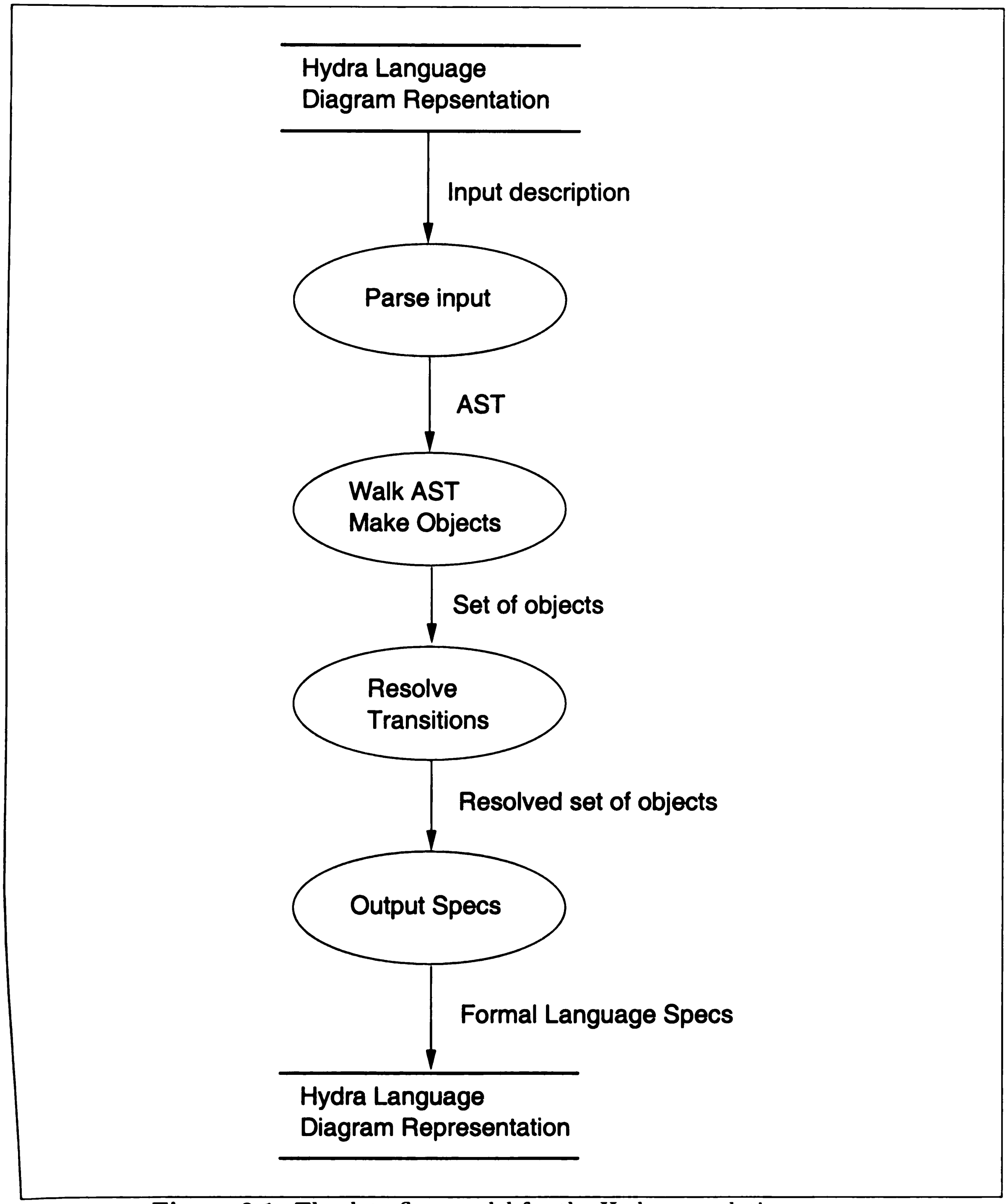


Figure 8.1: The data flow model for the Hydra translation system.

is the name of the class. `Object` may only appear within the scope of `Model` and contains instance variable declarations, signal declarations, composite and concurrent state declarations, and pseudo-state declarations such as `Join` and `Init`. Each `Object` must have a unique name.

`InstanceVar type-name var-name;`

`InstanceVar` declares an instance variable in a class. **`type-name`** is the variable type and can currently be **`int`** or **`bool`**.[2] The scope of an instance variable is the entire object, so the variable can be referred to in any transition within the object. Instance variables are hidden from other objects. An optional initializer of the form **`var-name := value;`** may be included in the declaration. Instance variable declarations may only appear within the immediate scope of `Object`.

`DriverFile ID;`

It is usually necessary to initialize and animate a model in order to determine its properties. For verification and simulation purposes, we define a *System Class* whose sole responsibility is to instantiate the objects of the model, and provide a sequenced set of signals to take the model through a particular scenario. Since the behavior model for the System Class requires the ability to instantiate objects, and may require other formal language specific operations, we encode the System Class dynamic model by hand and include the target language statements in a *Driver File.* The **`DriverFile`** is a means to include target language specific statements and routines into the final model. For example, Promela has a **`timeout`** event that can be used in a narrowly defined set of circumstances to determine when the entire model has deadlocked.

[2]Both types are common to most specification languages used for simulation and model checking, while types like 'float', for example are not. Although VHDL contains float variables, model checking languages such as Promela and SMV do not. So far, text or character types have not been useful in models.

For Promela models, a `timeout` after instantiating the objects of the model ensure that initialization has occurred. In VHDL, one specific **`Entity/Architecture`**, whose name is specified on the command line when the simulation is started, is responsible for loading the balance of the model and providing the initial system events.

`Signal`

The **`Signal`** statement can take two forms: **`Signal signame;`** or **`Signal signame(type-name);`**. The first form declares a simple signal expected from another object. The second form declares a signal with a single parameter of type **`type-name`**. Currently supported types are `int` and `bool`. Signals with more than one parameter are a straightforward extension to Hydra but are not currently supported. **`Signal`** can only appear within the immediate scope of `Object`.

`CompositeState cstate-name` {...}

The **`CompositeState`** block defines a composite state named **`cstate-name`**. The scope of a **`CompositeState`** statement contains declarations for states, pseudo-states, initial states, other composite states and concurrent states. **`CompositeState`** can appear within the scope of `Object`, **`CompositeState`**, or **`ConcurrentState`**.

`ConcurrentState wrapper-name` {...}

The **`ConcurrentState`** block defines a set of concurrent, composite states. In UML dynamic diagrams, a rounded rectangle that is partitioned by dotted lines is a concurrent composite state. The **`ConcurrentState`** statement defines the outer boundary of this UML construct. The **`wrapper-name`** provides a handle used by the `Join` statement and for error messages. **`wrapper-name`** does not name an actual state to which a transition can occur, rather, it names the container for the composite concurrent states.

The only statement that may appear within **ConcurrentState** is **CompositeState**. Even if the concurrent sub-state partition only contains one state, that one state must appear within a **CompositeState** block. **ConcurrentState** may appear anywhere **CompositeState** can occur.

Generating multiple threads and starting a set of concurrent states is accomplished by a transition to any one of the set of composite states contained immediately within the **ConcurrentState** block. In fact, this is the *only* way to start a set of concurrent states. A transition line written to the boundary of a set of concurrent states is written in Hydra as a transition to any of the composite states. Although technically it can be diagrammed, the semantics of a transition to a specific state within a set of concurrent states is difficult to imagine and is therefore not currently supported.

State state-name {...}

The **State** statement defines a block containing statements that describe the state. **state-name** names the state and must be unique across the model. State declarations may occur within composite states or objects, but nowhere else. The only statement that may appear within the scope of a state declaration is a **Transition** statement.

Join name From ccstate to join-state;

The **Join** statement defines combining multiple threads from concurrent state sub-states to a **join-state**. In the UML dynamic model, the **Join** corresponds to a heavy vertical bar. The name of the **ConcurrentState** containing the threads to be joined is **name**. **join-state** is any valid state or composite state. The words "From" and "to" must be written as shown. As mentioned in the mapping sections, joins must occur exactly one level above the concurrent state being consolidated. In other words, transition lines to the join can only cross one composite state boundary. Therefore, **`Join`** must occur within the same scope as the **`ConcurrentState`** statement to which it refers. Currently, a join is the only way of terminating a set of concurrent threads, consequently, a concurrent state declaration will usually have one or more corresponding joins, however, a join is not required. If there is no join for a concurrent state, then Hydra will issue a warning and assume the concurrent thread never ends. Exiting a concurrent thread without a join causes error procedures within the generated formal language specifications to be executed (in Promela, **`assert(false)`** is executed).

`Initial init-state;`

The **`Initial`** statement defines the initial state in an object or composite state. In the absence of an initial state declaration, Hydra assumes at least one direct transition from outside the composite state directly to a state contained in the composite state. This implies that **`Initial`** is not optional for an object. **`Initial`** may appear within an object or composite state.

`History init-state;`

A history pseudo-state is a special initial UML state that, upon exit from a composite state, remembers the last active state within the composite state. In a subsequent transition to the boundary of the composite state, the history state arranges for the last active state in its composite state to be re-activated. **`init-state`** is the default initial state activated when there is no history (that is, on the first entry to the composite state). On a UML diagram, history states are represented as a circle containing "H". History states can appear anywhere an initial state can occur. Specifying both **`Initial`** and **`History`** within the same scope is an error.

`Transition ''trans-expression'' to dest-state;`

`Transition` defines a state transition from the current state to **`dest-state`**. **`dest-state`** may be anywhere within the enclosing object (transitions across objects is not defined). The transition expression **`trans-expression`** follows standard UML syntax and consists of four parts, each of which is optional. The standard form of a UML transition is **`event[guard]/action^msg-1... ^msg-n`**.

<u>**`event`**</u> Event is an internal event name, a signal, or a "when" expression. A signal may take two forms: **`signal`** or **`signal(var)`**, where **`var`** and **`signal`** are a previously declared instance variable and signal. A "when" clause takes the form of **`when(logical-expression)`**, where **`logical-expression`** is an expression that is continuously, but asynchronously, evaluated.

<u>**`guard`**</u> The guard is a logical expression providing a condition on the transition. Logical "and" is written "&", logical "or" is written "|", and negation is written "∼". Comparison predicates (`>, <, >=, <=, ==, !=`) may appear within the expression and some predicates (target language dependent) are permitted. One particular predicate that is always implemented is the **`IN(statename)`** predicate, which evaluates true when **`statename`** is active.

<u>**`action`**</u> **`Action`** consists of a series of one or more statements separated by semi-colons that define actions to be taken during the transition. Currently, three actions are defined: a replacement statement of the form **`var := expression`**, a **`print`** statement, and function call of the form **`function(arglist)`**. The **`print`** statement follows C conventions for a "printf" statement. The first parameter is a string containing text and optionally, "%" qualifiers. Optional additional parameters are instance variable names. The function call provision exists primary to

insert `assert` statements into the action.

msg `msg` is an event to be sent to transitions in the enclosing object or a signal to be sent to another object. Internal messages have only one form: `event`. Signals sent to another object have the form `objectname.signal` or `objectname.signal(value)`. Multiple messages may be specified on a transition, each separated by "^".

Transition expressions must be enclosed in double quotes. Long transition expressions may be written on multiple lines as a list of quoted strings, each separated by a comma.

8.3 Hydra Tool Structure

Hydra consists of four major parts: The main Hydra program, the parser for the Hydra language, the parser for transition expressions, and a class library specific to the target language.

A Hydra translation run starts when Hydra reads the model description and passes it to the parser. When the parsing is complete, the parser returns an AST. Each node in the tree has a type identifier corresponding to the part of the language parsed, and a series of key/value pairs. For example, a node representing a transition contains the node ID "Trans", a key/value pair named "tran" containing the transition expression, and a key/value pair named "dest" containing the name of the destination state.

In the next phase of operation, the Hydra main program walks the tree in depth first order calling internal Hydra routines that correspond to major AST node types such as "Object" and "State". The internal routines have two functions to perform. First, AST nodes corresponding to parts of the grammar that define environments (Object, State, CompositeState, *etc.*) contain lists of nodes that describe the components in the environment, such as States. The internal routines must cycle through these lists calling appropriate methods to handle each item. Second, the class library returns references to objects created as the AST is processed. For example, when a State is processed, a new State object is created within the Hydra code. The Hydra internal routines keep track of these references because subsequent method calls use object references. For example, the "Tran" method in class State expects a reference to a destination, not the textual name of the destination. The Hydra routines must keep tables of name versus reference values both for calling methods and for passing correct parameters to the class library methods. At the completion of the AST walk, Hydra will have created an internal set of objects that correspond to the model. In the final step, the objects created from the target language class library interact to generate the target language specifications.

8.4 Language-Specific Class Library

The actual mapping from UML to the target language is embodied within a *Language Specific Class Library* (LSCL). The LSCL contains a number of standard classes and methods Hydra expects during the AST tree walk. Figure 8.2 shows the generic class diagram for the LSCL. The dotted lines represent class method invocations (calls). The *structure* of the LSCL does not vary much between languages, but the internal functions vary widely. For example, VHDL Entity/Architecture units must occur in the written source file with only forward references. An Entity/Architecture cannot reference another Entity/Architecture that is defined further down in the file. Consequently, the VHDL LSCL must order the Entity/Architecture units to satisfy this requirement. Since Promela has no such requirement, Promela "proctype" definitions can be written in any order. On the other hand, the

UML semantics mapped to Promela require the addition of specific "proctype" routines to handle event dispatching that are not needed in VHDL.

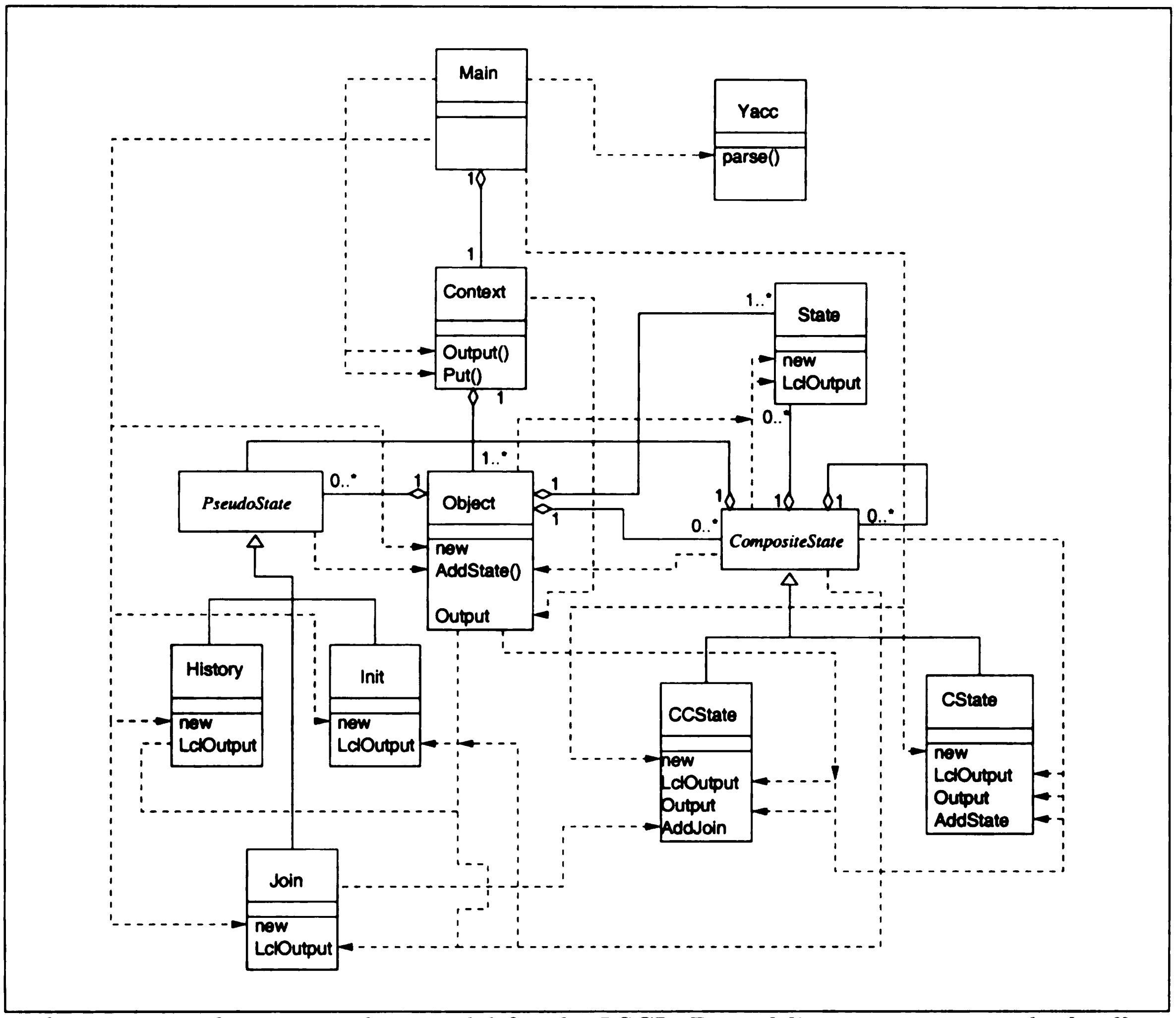


Figure 8.2: The generic class model for the LSCL. Dotted lines represent method calls.

The classes required in the LSCL are as follows:

Class Context: This class is instantiated only once. It is responsible for model-level, or "context" items and serves as the anchor for the LSCL object ontology. For example, the list of all objects in the model are held by Context. Additionally, Context implements standard methods such as "Put", which formats and writes generated specifications to the output file.

Class Yacc: The Yacc class contains the parser for the language. Although the Hydra language is the same across target languages, the parser semantic rules vary between languages. Class Yacc implements a lexical analyzer and the methods "Node" and "Add" used by the semantic rules to build the AST. Technically, the semantic rules could be moved down a level to create a standard parser for all formal languages. If this were done, each Yacc semantic rule would be a call to a method in the LSCL that would build the AST. The capability to construct the parser this way was not realized until the second formal language was completed.

Class Object: New Objects are created when object nodes are found on the AST. The Object class keeps track of all the states, composite, concurrent, and simple, within its scope. Methods in Object handle the creation of instance variables and signals.

Class CState: New ConcurrentStates are created from class CState. CState has many of the same methods as Object, since CState can also contain simple states, concurrent states and composite states.

Class CCState: Class CCState implements the "wrapper" around a set of concurrent composite states. The main purpose of the class is to maintain a list of the composite states that will be running concurrently, and to build the necessary target language constructs to implement concurrency.

Class State: Class State implements simple states. Transitions are implemented as method calls to method "Tran" in State. Class State is responsible for maintaining a list of all transitions, parsing the transitions, and producing the target specifications corresponding to a simple state. Transition parsing is handled by calling a second Yacc parser (class TranYacc) to specifically handle the transition expression. The transition parser is currently tied to the target language, although it would be possible to move the semantic rules into the LSCL as described above for class Yacc.

Transition expressions are enclosed within double quotes in the Hydra language so the Hydra parser (class Yacc) can pass the unparsed, quoted string to method "Tran" without parsing the transition itself. Adding rules to parse the transition directly from within the Hydra language grammar introduces a large number of shift/reduce conflicts and greatly complicates the processing of the language and transition.

Since event dispatching is handled at the state level, class "State" contains methods and routines to generate specifications for the transition mechanism within the target language.

Classes History and Init: These classes create new History and Initial pseudo-state objects within Hydra to support initial, and history states.

Class Join: The Class Join is responsible for implementing the thread joining mechanism when concurrent states are used. Class Join interacts with Class CCState (and hence the reference to the CCState found in the Join language construct) to recombine separately spawned threads within the model.

Class Log: Class Log is used for debugging purposes, but in the two implemented LSCLs it is essentially identical. Class Log instantiates a single class object and is responsible for producing debugging and process information annotated with the object taking the action, and the location within the class library where the action being logged occurs.

8.4.1 Operation of the LSCL

The LSCL begins operation when a set of objects are instantiated, as described above, during the Hydra AST tree walk. Some processing is possible as objects are being created, but in general, the entire model must be instantiated within Hydra as LSCL objects before target language specification generation can begin.

Each class implements method "PreProcess" for performing operations required prior to target specification generation, if any. When the AST tree walk is complete, the Hydra main routine calls method "Output" in class Context to initiate specification generation. Context first cycles through its set of child objects calling the "PreProcess" method. Each child, in turn, calls its child object "PreProcess" methods until all "PreProcess" methods have been called for all LSCL objects.

8.4.2 Resolving Transitions Crossing Composite State Boundaries

In a UML dynamic model, the nesting structure of composite states form a hierarchy. However, UML does not require activation of a state by following the hierarchy structure. Transitions can occur from anywhere within the hierarchy to anywhere else within the hierarchy without regard to the hierarchy structure. In VHDL, this is a straightforward issue. The UML to VHDL mapping uses a "state bus" for state activation. All states are listening at all times for their name to appear on the bus. When the name appears, the state is activated. As long as the VHDL state bus hierarchy is constructed to match the state hierarchy, cross hierarchical transitions require no further action at specification generation time.

Other languages, such as Promela, have a traditional "procedure call" structure. Objects and composite states are mapped to Promela "proctypes", while simple states are implemented using "goto" to transfer to a label within a Promela proctype. Activation of a composite state corresponds to activation of a Promela **proctype**.

The activation necessarily follows the hierarchical structure imposed by the UML model. Therefore, a transition path may lie through a series of proctype procedures that could possibly traverse up and down the hierarchy. As depicted in Figure 8.3 a transition from within composite state "F" to composite state "D" requires an exit to "B", then an exit to "A", and finally, an activation by "A" of the simple state within "D". In passing, we note that other languages within traditional sub-process structures, when used as targets, would posses the same transition difficultly.

For this discussion, denote the CState, CCState, or Object LSCL object that contains a State object as an *environment*. To solve the problem of destination state activation in the Promela case, each LSCL State object asks its parent environment object whether the destination on the transition is local, that is, within the environment that contains the State, or within another environment. If the destination is not local, there are two possibilities:

1. the destination is contained in a child environment object of the current environment (this would be *down* the state hierarchy tree). The transition requires calling a **proctype** representing a lower level environment object.

2. the destination is contained in an environment object that contains the current environment (this would be *up* the state hierarchy tree). The transition requires returning to a **proctype** that has called the current environment.

Within each environment object along the transition path, this decision process is repeated recursively until the destination is local to an environment.

Referring again to Figure 8.3, suppose state "x" in composite state "F" needs to transition to state "y" in composite state "D". First, "x" asks "F" if "y" is local (contained within the procedure represented by "F"). State "y" is not local, so "F" seeks destination "y" downward. This too fails since "y" is not downward. Finally, "F" asks its parent to find state "y". Similarly, "B" also does not find "y" locally or within its children. Each object similarly responds that the destination is local, or it searches its children, and finally, asks its parent to find the destination. Eventually, a series of calls passing from "F" to "B" to "A" to "D" is established revealing the path to "y". Although "D" reports to "A"that state "y" is local, "F" tells State "x" that "y" is up. Since every object in the path also sees the returned resolution of the destination, each object on the path can prepare locally for the transition to "y" when it is required.

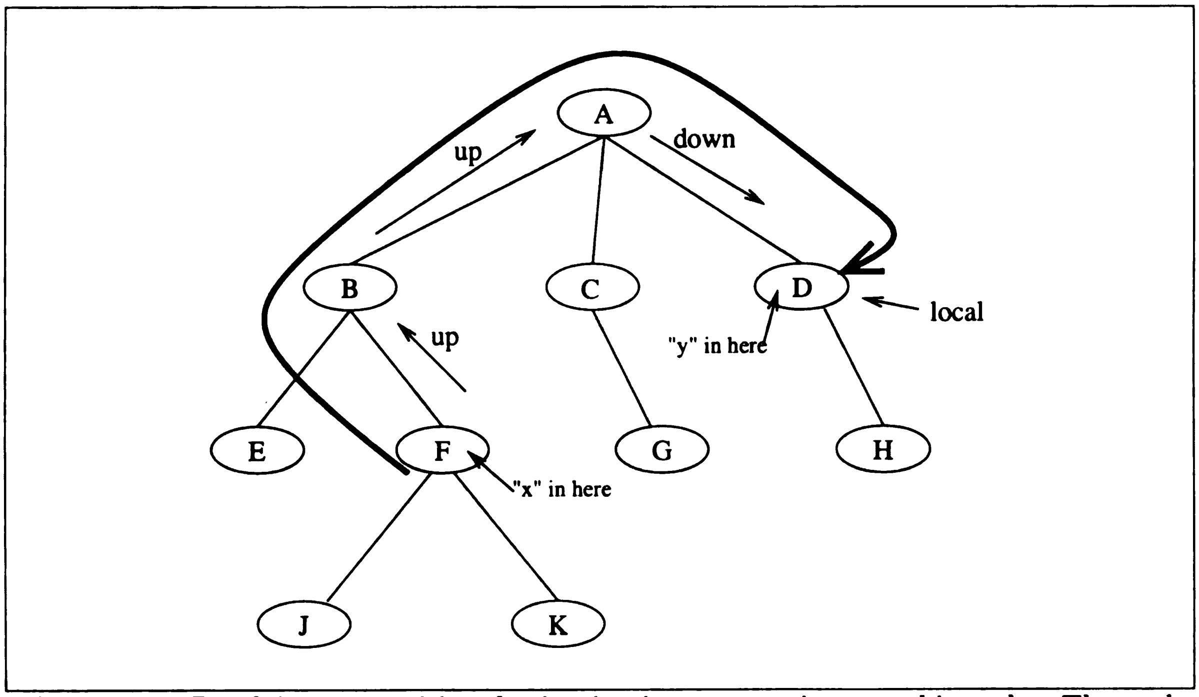


Figure 8.3: Resolving a transition destination in a composite state hierarchy. The ovals represent composite states.

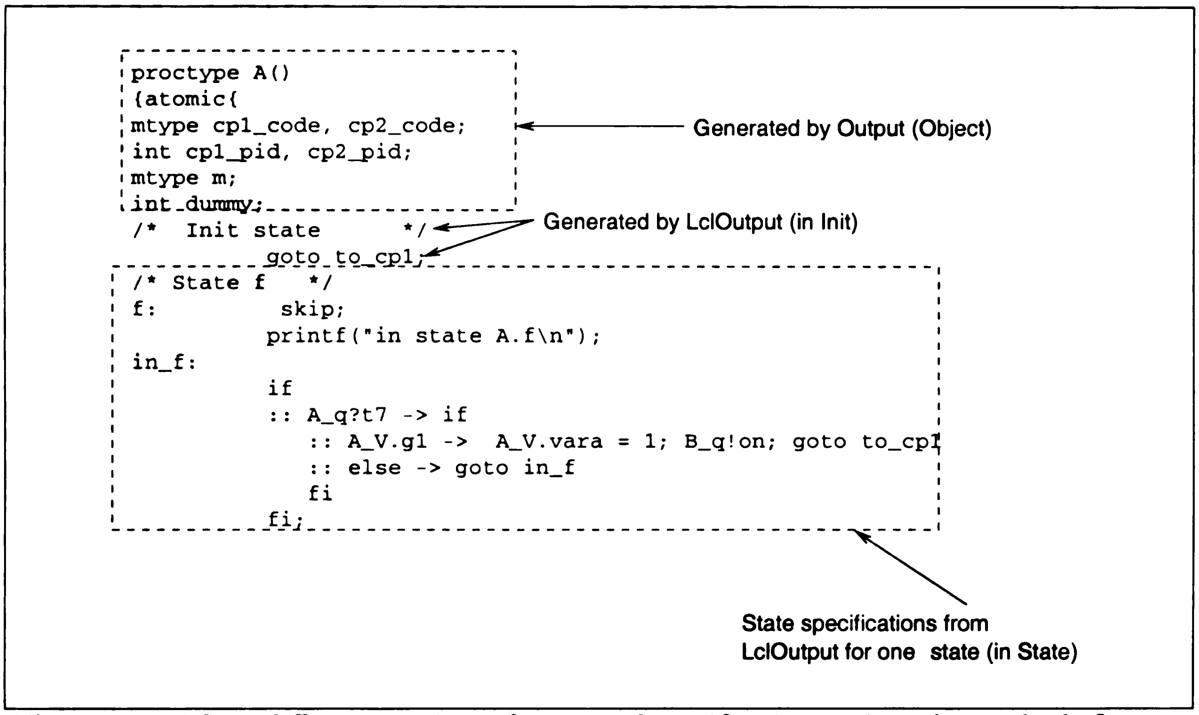
Some mappings from UML to formal languages, Latella's [64] work for example, avoid this type of processing by transforming the entire model to a extended hierarchical state machine early in the translation process. But, as shown above, the degree to which searching the state hierarchy is required depends directly on the target language. For VHDL, a translation to an extended hierarchical state machine would be of little use. Therefore, we implement a search of the hierarchy in the Promela LSCL and omit it in the VHDL LSCL.

8.4.3 Target Specification Generation

At the conclusion of preprocessing, class Context calls method "GlobalOutput" for each object it owns to write any global declarations required by the model. Next, Context calls method "Output" for each object it owns to generate the actual target specifications.

Within the LSCL there are two output methods: "Output" for generating complete structures, usually target language containers such as proctypes and entity/architectures, and "LclOutput" for generating specifications that resides within one of these structures. Figure 8.4 illustrates the different structures written by each method. Class State's processing is primarily contained within method "LclOutput" because it is generating mapped specifications for simple states that resides within the containing structure. For example, a call to class State's "LclOutput" generates a label and a set of "if" structures that will be contained within a proctype definition.

Many of the formal language metamodel's templates are used within class State's LclOutput method, and therefore, many semantic decisions are turned into target language specifications within this method.



```
proctype A()
{atomic{
mtype cp1_code, cp2_code;
int cp1_pid, cp2_pid;
mtype m;
int dummy;
/*  Init state      */
          goto to_cp1;
/* State f   */
f:         skip;
          printf("in state A.f\n");
in_f:
          if
          :: A_q?t7 -> if
             :: A_V.g1 ->  A_V.vara = 1; B_q!on; goto to_cp1;
             :: else -> goto in_f
             fi
          fi;
```

Generated by Output (Object)
Generated by LclOutput (in Init)
State specifications from LclOutput for one state (in State)

Figure 8.4: Three different portions of generated specifications written by methods Output and LclOutput from states Object, Init, and State, respectively.

Chapter 9

Industrial Case Study

This chapter illustrates the use of the design methodology described in Chapter 4 to design a typical industrial embedded system called "Smart Cruise". The method outlined in Chapter 4 relies on many UML diagrams that are shown as the design processes progresses through the chapter.

An objective of the case study is to show how our homomorphic framework, along with our design process using UML diagrams, can automatically translate diagrams to formal specifications using Hydra, which can be executed and analyzed. In particular, the case study illustrates how the UML class and dynamic model diagrams can be transformed to specifications so that model checking can be performed. Checking the dynamic model of an embedded system is notoriously difficult, as described earlier, therefore model checking to validate safety and liveness properties of the system are particularly important. Because the case study uses both simulation and model checking, the formal target specification language used in this case study is Promela and the analysis tool is SPIN, as described in Section 2.3.

Another objective of the case study in this chapter is to demonstrate the practical application of the homomorphic mapping framework to another language. We previously presented a smaller example of a system through a mapping to VHDL in section 5.3. This chapter demonstrates that the framework is useful to obtain formal, executable models in a different language that is amiable to model checking.

Finally, this chapter shows that system refinement can occur quickly and in domains not normally formally accessible in design methodologies. Refinement occurs quickly because it is within the diagram to Promela model loop. As the dynamic and class models are modified, the changes are automatically translated with Hydra to Promela quickly. In addition, the study describes modification of the semantics of message passing within the system. This is accomplished by modifying the homomorphic mapping. In other development methodologies, the semantics of the formal design language, if any, is fixed, and cannot be modified to suite the application.

9.1 Description of "Smart Cruise"

Most current production cars have a facility called "cruise control" that enables the driver to set a given speed to be maintained by the car. The operation is simple: when the driver presses a button on the steering column, the car maintains the current speed until the cruise is turned off or the brakes are applied. Current production cars implement cruise control within the engine control modules, and therefore, the only speed control mechanism available is through throttle position.

A major inconvenience during cruise control operation occurs when the driver is following a slower vehicle. To avoid hitting the leading vehicle, either the cruise must be disengaged by applying the brakes, or by switching it off, or the cruise must be reset to a slower speed to match the leading vehicle.

9.1.1 Requirements

Smart Cruise is intended to increase the convenience typically associated with following a vehicle when using cruise control. The high level use case is as follows:

When Smart Cruise is enabled and detects a vehicle (called the *target*), the Smart Cruise shall re-command the engine control system to a speed to match the target vehicle. The trail distance behind the target is specified as the time the target travels in a given amount of time (usually two seconds). If the equipped car closes to within 90% of the specified trail distance (called the *safe distance*) behind the target, an audio warning must be sounded and the cruise disengaged. Otherwise, the system must match the speed of the target vehicle while maintaining a trail distance close to the specified optimum trail distance, but never less than the safe distance. Speed matching should continue until either the target moves out of range (by speeding up or turning) or until the driver applies the brakes. A brake signal must disengage the entire cruise system.

If the system can determine that it will strike the target vehicle unless driver action is taken, then it must produce a visual warning, but not disengage the system. Only when the safe distance is violated shall the system disengage, thus system disengagement can occur automatically if the car enters the unsafe zone, or by the driver by applying the brakes.

9.1.2 System Environment and Use Case

Figure 9.1 shows the Use Case diagram for Smart Cruise that contains one use case named "Maintain Cruise". The prose description of this use case is as follows:

Use Case Maintain Cruise:

Main flow of events: The primary flow of events occurs when the driver presses the "set"

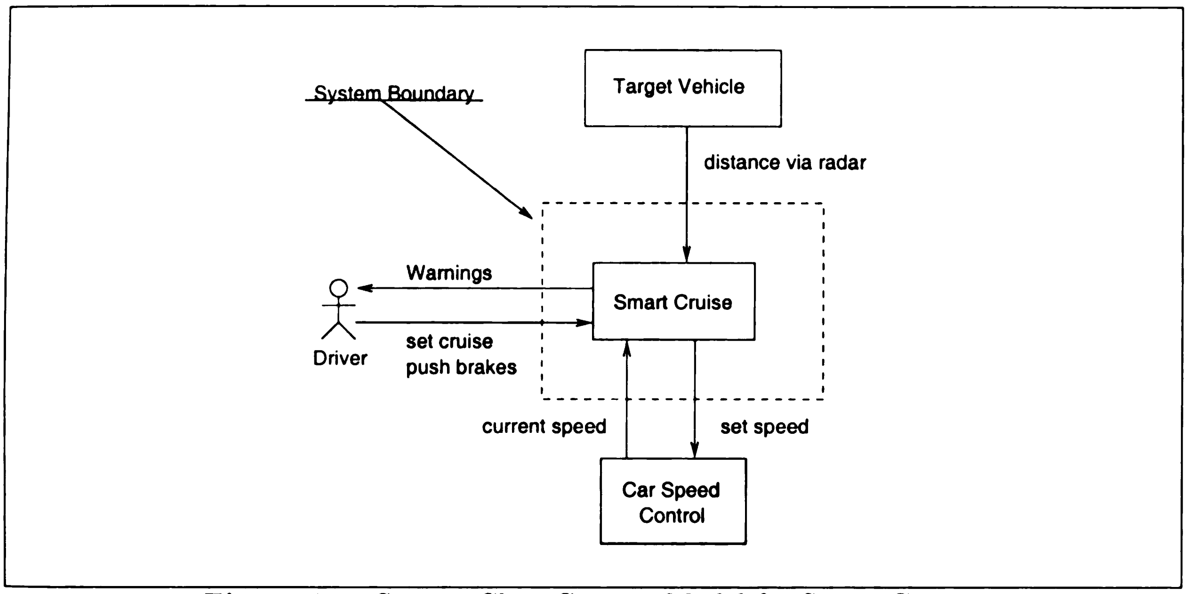


Figure 9.1: System Class Context Model for Smart Cruise.

button on the cruise control. At that time, the car starts maintaining set current speed. If a vehicle appears in front of the car (Called the *target*, the cruise control modifies the speed of the car until a safe trail distance is achieved. Once achieved, the car maintains that position behind the car until the cruise control is turned off.

Exceptional Flow - Car going too fast: If the car is going too fast to decelerate without brakes before arriving at a safe trail distance, the cruise control must notify the driver. If the safe zone is entered, the driver must be notified and the cruise turned off

Exceptional Flow - Target Turns: If the target turns, or otherwise is no longer in front of the car, then the cruise must resume the previous set speed.

Exceptional Flow - Brakes Applied: If the driver depresses the brakes at any time, the cruise must immediately cease speed control, and the main flow takes control.

As discussed in Chapter 4 on Design Process, the use case diagram provides some perspective on actors external to the system versus the system itself. As that chapter also describes, we find a *system context model* also to be useful to make a clear delineation between external and internal system components, and to provide a high level view of the interaction of the system with the environment. Figure 9.1 shows the system context model for Smart Cruise.

As both Figures 9.1 and 9.2 show, the environment of the system is a car whose speed is controlled, a driver, and the target vehicle. The system interacts with the driver by accepting the "set" button on the cruise, and by accepting a "brakes applied" signal. The system produces speed control for the car and alarms for the driver. The target supplies its distance indirectly through radar returns. Other information, such as target speed, must be calculated. The system can command a set speed to the car engine control and it can read the current speed of the car. The system has no means to apply brakes.

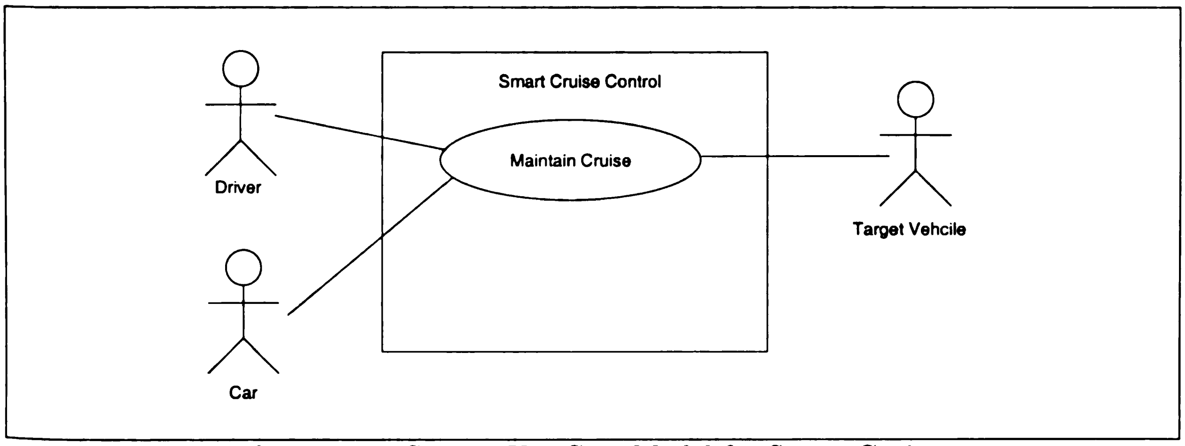


Figure 9.2: System Use Case Model for Smart Cruise.

The primary flow of events occurs when the car comes up behind the target. At this point it must detect the target, determine its speed, calculate when to slow down, then slow to match the target. The target must be trailed until the cruise is shut off or the target disappears.

9.1.3 System Components

The system components include a radar module that can reliably detect distance to the target from about 400 feet, and a control system that uses the radar information and car speed information to command the car's throttle as required to maintain the appropriate trail distance. The car's engine control directly manipulates the throttle to maintain the speed commanded by class **Control**. When the radar first reliably detects a target at about 400 feet, it sends a "target acquired" message, and if the target is lost, a "target lost" message is sent. Targets can be lost when they move ahead out of range or turn, therefore, target loss implies that the car should resume the previously set speed.

9.2 Initial System Design

The focus of class **Control** is the speed matching algorithm. Any command to set the car to a new speed will be achieved at an acceleration or deceleration characteristic of the car. For design purposes, we assume the car both accelerates and decelerates at $a = 1.5\ ft/sec$.

The control algorithm must match the behavior, to the degree feasible, of a human driver. The first temptation is to design the algorithm to start slowly decelerating when the target is acquired, but human drivers do not behave this way. Instead, a human continues at cruise speed until she decides that coasting will cause the car to close to a reasonable trail distance behind the target. One practical reason for this kind of behavior is dictated by other driver's behavior in heavier traffic. If a substantial distance is left between the target and the car while the car very slowly closes the distance to the target from 400 feet or more, it is likely that another vehicle will cut into the space between the target and the car. Thus, from a practical viewpoint, we should design the system to continue at set cruise speed for as long as possible.

The distance closing algorithm we employ calculates the distance behind the target at which to start decelerating. Figure 9.3 shows diagrammatically the distances involved. First, we need the closing speed of the car with the target, which requires at least two distance samples from the radar. The car speed can be requested from the engine control, enabling class **Control** to compute the speed of the target and the closing speed. Using the closing speed, the distance to match the target speed at deceleration a is given by the formula $x_m = v^2/2a$. For a trail distance of t_{min}, expressed in seconds at the target speed, and a computed target speed of v_t, we want to begin decelerating when we are x_{coast} feet behind the target, where $x_{coast} = v^2/2a + t_{min}v_c$.

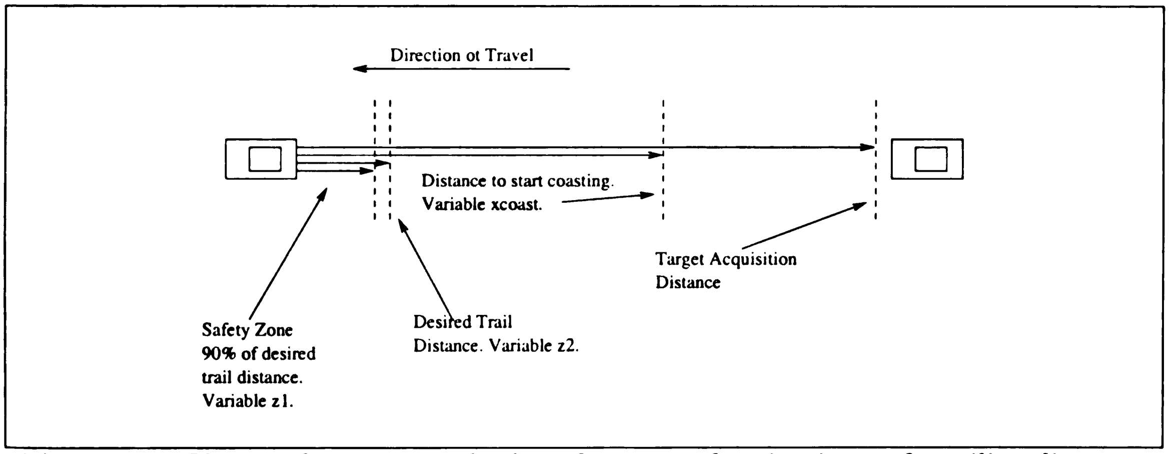


Figure 9.3: Various distances involved to close on and maintain a safe trailing distance.

The most obvious implication of this control policy occurs when the target slows down after the deceleration has begun. In this case, we would have wanted to start decelerating sooner, but it is already too late. Therefore, the best class **Control** can do, since it does not have access to brakes, is to continue decelerating and monitoring the closing speed and distance. When class **Control** computes that the car will collide with the target, it must provide an alarm. If the car enters the unsafe zone behind the target, then it must produce a warning and disengage the cruise control. On the other hand, if the target speeds up, class

Control should command an increased speed, where the maximum is the original set-point for the cruise control.

9.2.1 Use Case Scenarios

The main flow of events in the use case occurs when the driver depresses the "set" button on the cruise, followed by the car approaching the target from behind. At this point it must detect the target, determine its speed, calculate when to slow down, then slow to match the target. The target must be trailed until the cruise is shut off or the target disappears. Figure 9.4 shows the high level sequence diagram for the main flow of events in the use case.

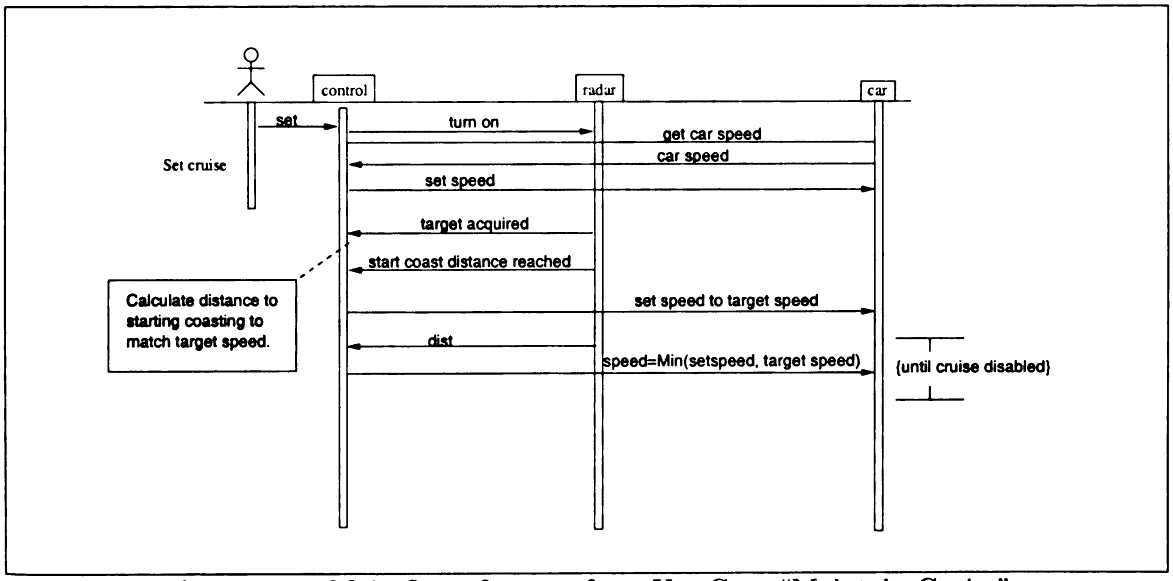

control
radar
car
set
Set cruise
turn on
get car speed
car speed
set speed
target acquired
start coast distance reached
Calculate distance to starting coasting to match target speed.
set speed to target speed
dist
speed=Min(setspeed, target speed)
{until cruise disabled}

Figure 9.4: Main flow of events from Use Case "Maintain Cruise"

In the previous scenario the closing speed was low enough to allow the car to coast to a speed matching the target's speed. If the closing speed with the target is too great, the scenario called "Car going too fast" in Figure 9.5 shows the high level expected behavior for the system. As soon as possible, the system must detect a possible collision and warn the driver. If the safety zone is entered, then the system must provide a final warning to the driver and turn the system off.

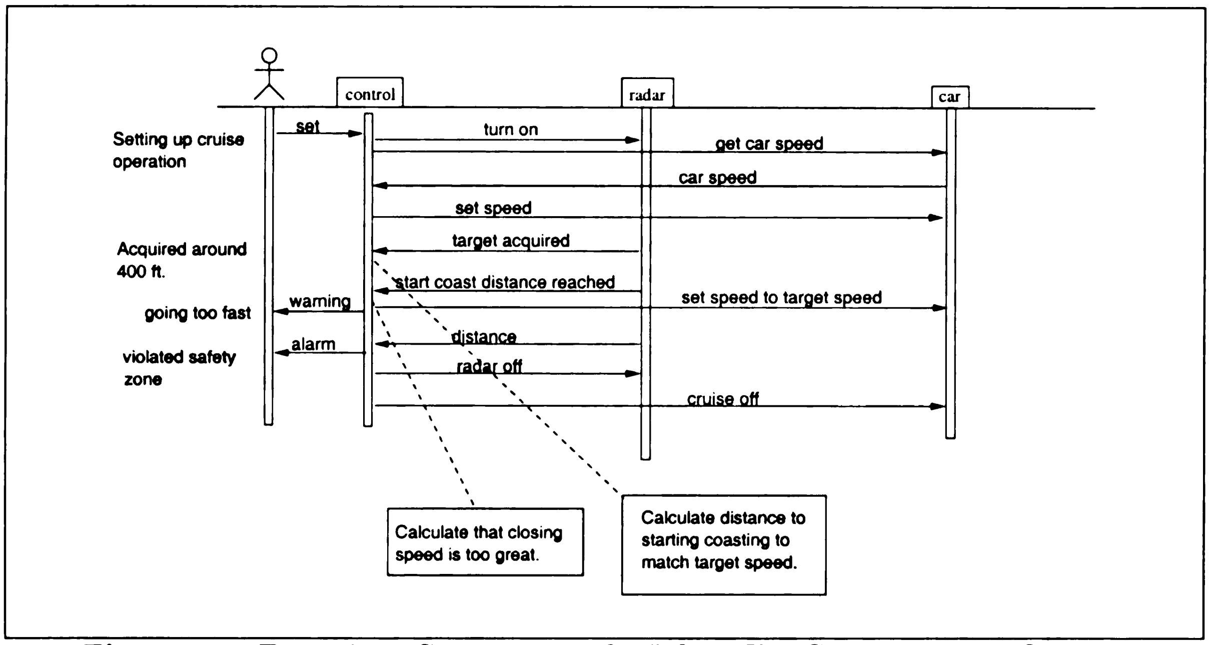

control
radar
car
Setting up cruise operation
Acquired around 400 ft.
going too fast
violated safety zone
set
turn on
get car speed
car speed
set speed
target acquired
start coast distance reached
set speed to target speed
warning
distance
alarm
radar off
cruise off
Calculate that closing speed is too great.
Calculate distance to starting coasting to match target speed.

Figure 9.5: Exception "Car going too fast" from Use Case "Maintain Cruise".

If the radar loses tracking on the target (most likely because it turned), then the car must resume original set speed per Use Case exception "Target Turns". The high level sequence diagram for this scenario is shown in Figure 9.6.

The final use case scenario in Figure 9.7 shows correct behavior when the brakes are applied while the system is functioning. Proper behavior must include shutting down the radar and returning the control system to an idle mode.

9.3 Class Diagram

Our first design included four classes in the system: The **Control** class, the **Car** class (including the throttle control and read-out of car speed information), the **Radar** class, a **target** class, and a **SYSTEM** class. This configuration simplifies simulation because the target is explicitly listed as an object with its own set of responsibilities. This suggests that the target can respond to the radar class with its speed to update the car's x distance along

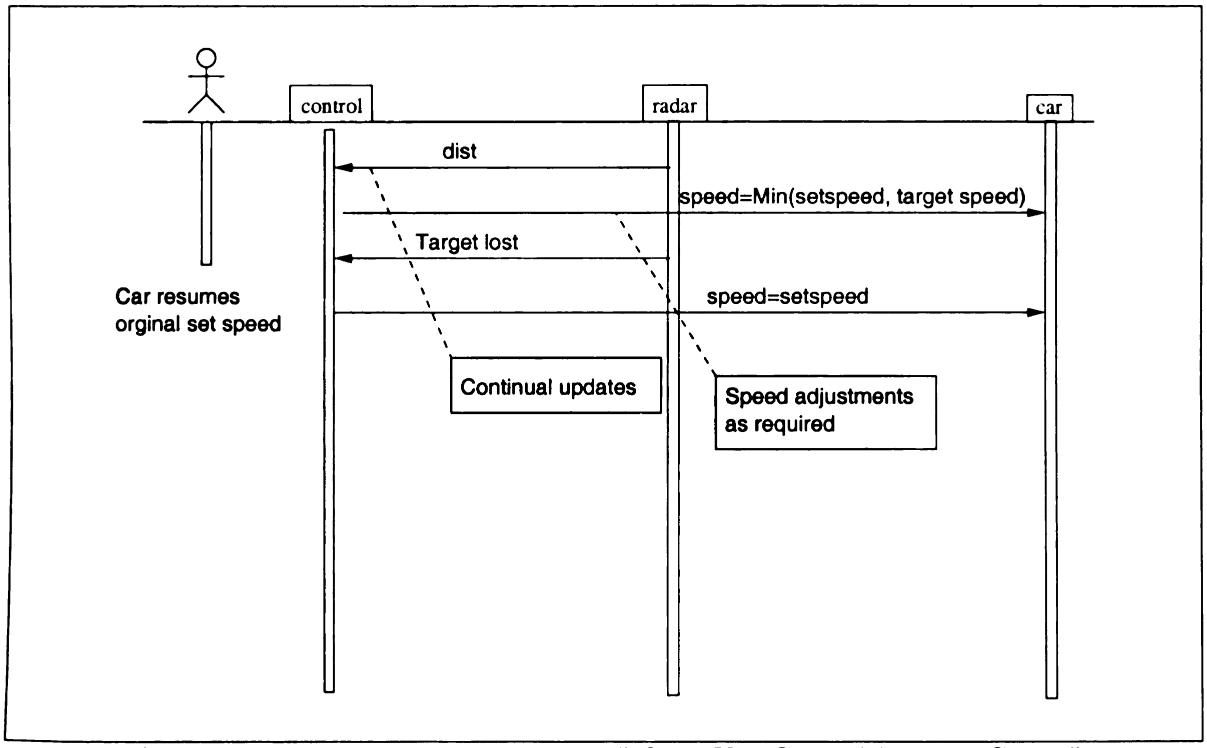

control
radar
car
dist
speed=Min(setspeed, target speed)
Target lost
speed=setspeed
Car resumes orginal set speed
Continual updates
Speed adjustments as required

Figure 9.6: Exception "Target Turns" from Use Case "Maintain Cruise".

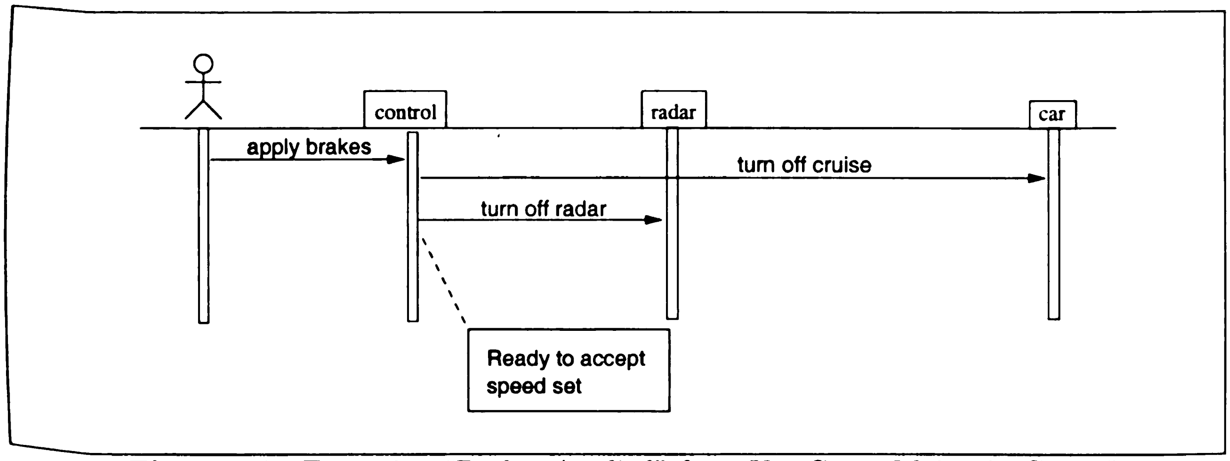


Figure 9.7: Exception "Brakes Applied" from Use Case "Maintain Cruise".

the road. However, this configuration is not accurate for the final system. In particular, the target is not an object within the system boundaries (see Figure 9.1). Instead, the system's knowledge of the target's behavior comes entirely through the radar module. Therefore, a more accurate design includes only the **Control**, **Car**, **Radar**, and **SYSTEM** classes. The class diagram is shown in Figure 9.8. The definition of the instance variables by class, is listed in the data dictionary in Figure 9.9.

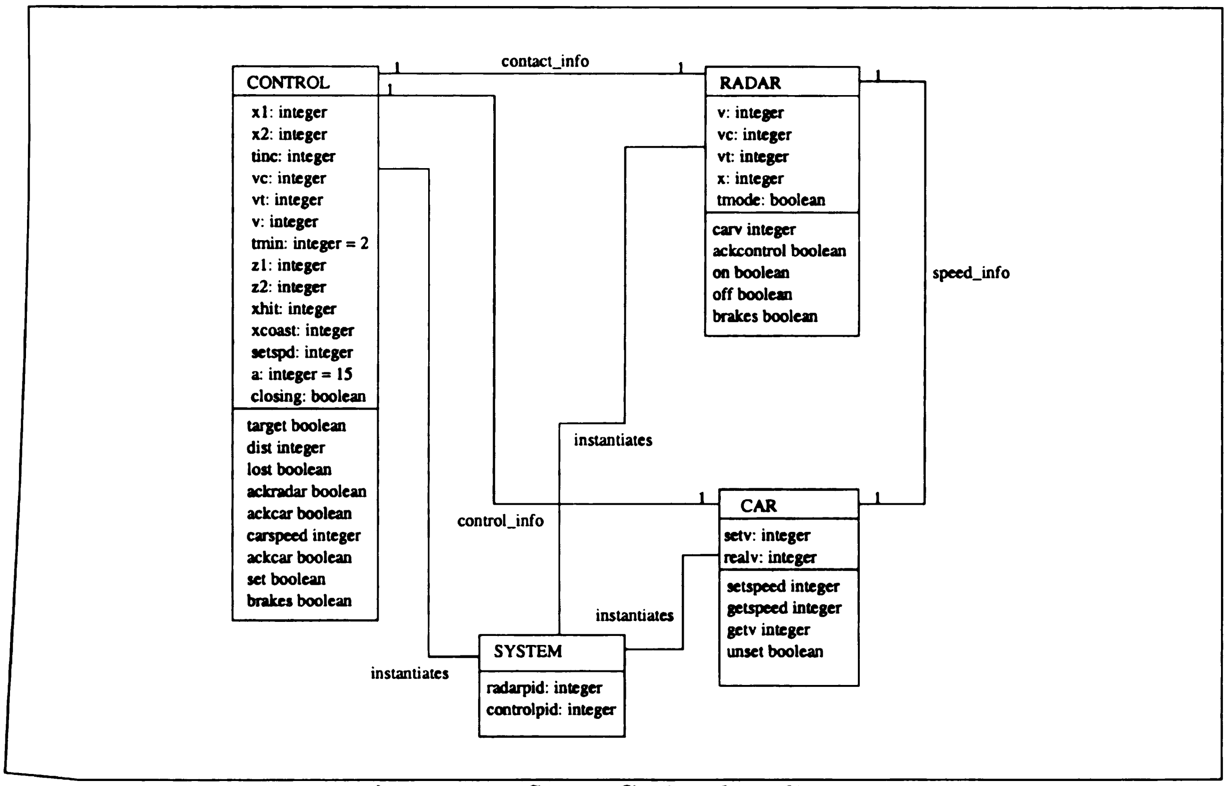


Figure 9.8: Smart Cruise class diagram.

9.4 Class Responsibilities

The next step in the design process is to assign responsibilities to each class. For design purposes, we will abstract the radar module so that it calculates distances that would be obtained via actual radar returns in the final system. Because class **Radar** calculates range information, it requires knowledge of the car speed, which can best be obtained from the

Data Dictionary for Smart Cruise			
Variable	Attributes	Source	Description
		Class **Control**	
$x1$	Integer	**Radar**	The last distance to target.
$x2$	Integer	**Radar**	The current distance to target.
$tinc$	Integer	Constant	The increment of time between radar distance samples.
vc	Integer	**Car**	The speed of the car.
vt	Integer	$vt = cv - v$	The speed of the target.
v	Integer	$v = (x1 - x2)/tinc$	The closing speed of the car towards the target.
$tmin$	Integer	Constant	Desired trail distance in seconds x target speed.
$z1$	Integer	$z1 = z2 - z2/10$	Closest safe distance. Boundary of "safe zone".
$z2$	Integer	$z2 = vt * tmin$	Desired trail distance.
$xhit$	Integer	$xhit = (v^2)/2a$	Distance to start coast to exactly meet target speed at zero trail distance.
$xcoast$	Integer	$xcoast = xhit + z2 + tinc * v$	Distance to start coasting to achieve desired trail distance.
$setspd$	Integer	Class **Car**	The set speed when "set" is pressed.
a	Integer	Constant	acceleration/deceleration speed of car without throttle.
$closing$	Boolean	Set state *waiting.*	Switch used in algorithm when approaching target.
		Class **Radar**	
v	Integer	$v = vc - vt$	Closing speed. Used to calculate next distance.
vc	Integer	Class **Car**	Last car speed.
vt	Integer	Constant	Target vehicle speed
x	Integer	$x = x - v$	Distance to target.
$tmode$	Boolean	Set in state *r3*	Switch indicating target being tracked.
		Class **Car**	
$setv$	Integer	From *realv*	Car "set speed" desired cruising speed.
$realv$	Integer	*setv* and calculated	Current car speed.

Figure 9.9: The data dictionary for Smart Cruise. Definition of the instance variables is in Figure 9.8.

Car class. In the actual system, the **Car** class would only supply speed to the **Control** class and not to the radar class.

The **SYSTEM** class exists solely to initialize and start the model. As described in Section 8.2, under the `DriverFile` description, the behavior model for the system class is encoded (in this case) in Promela and included in a driver file.

The class responsibilities are initially set as follows:

Class Control: The **Control** class must accept input from the driver to set the desired cruise speed, which is the current car speed. This class is then responsible for activating the radar system and waiting for a "target acquired" signal. When obtained, the **Radar** class will continually send distance information until turned off. Each message requires an acknowledgement. Class **Control** is further required to provide the sole input to the throttle control. The **Control** class must also accept a "brakes applied" signal to disengage the system.

Class Radar: The **Radar** class must externally appear to behave the same as the implemented radar module would behave. The **Radar** class does nothing until activated by a message from the **Control** class. Once activated, **Radar** will continually scan for a target. Once the target is 400 feet or less from the car, **Radar** will send a "target acquired" signal to class **Control**. Thereafter, **Radar** will continually send to **Control** an integer distance to the target, but each distance message must be acknowledged by **Control** before the next distance is sent. When the target is lost, class **Radar** must send a "lost" message to the **Control** class and continue searching for a target. The **Radar** class must stop tracking and move to its idle state if it receives an "off" signal. The "off" signal requires no acknowledge.

Class Car: This class has three responsibilities:

1. Provide current car speed upon request.
2. Accept "set" and "unset" messages to set a new desired speed.
3. Continually adjust the car speed to match the set speed, but only at the specified deceleration or acceleration rate. In other words, new speed commands modify the current speed at only $1.5 ft/sec^2$.

Class SYSTEM: The **SYSTEM** class instantiates the **Car**, **Radar**, and **Control** objects. After instantiation, **SYSTEM** waits until each object is waiting on its initial transition, then it simulates the driver of the car by providing a *set* signal to start the simulation.

The **Control** class interacts with both the **Radar** and **Car** class objects. The **Radar** and the **Car** classes interact only for modeling purposes. In the implemented system, distance to the target is determined by a radar return, but for design modeling, we build the **Radar** class to externally respond to other objects as it would in the implemented radar module, while internally calculating the distance to the target based on car speed and target speed.

9.4.1 Sequence Diagram

Figure 9.10 shows a sequence diagram for the interaction between objects from the time the driver presses "set" until a correct trail distance is obtained.

To avoid deadlock and to insure each important message is properly received, the system objects are designed to acknowledge messages it does not initiate. This is called *synchronous* interaction. The response may be either a simple acknowledgement or a reply with requested data .

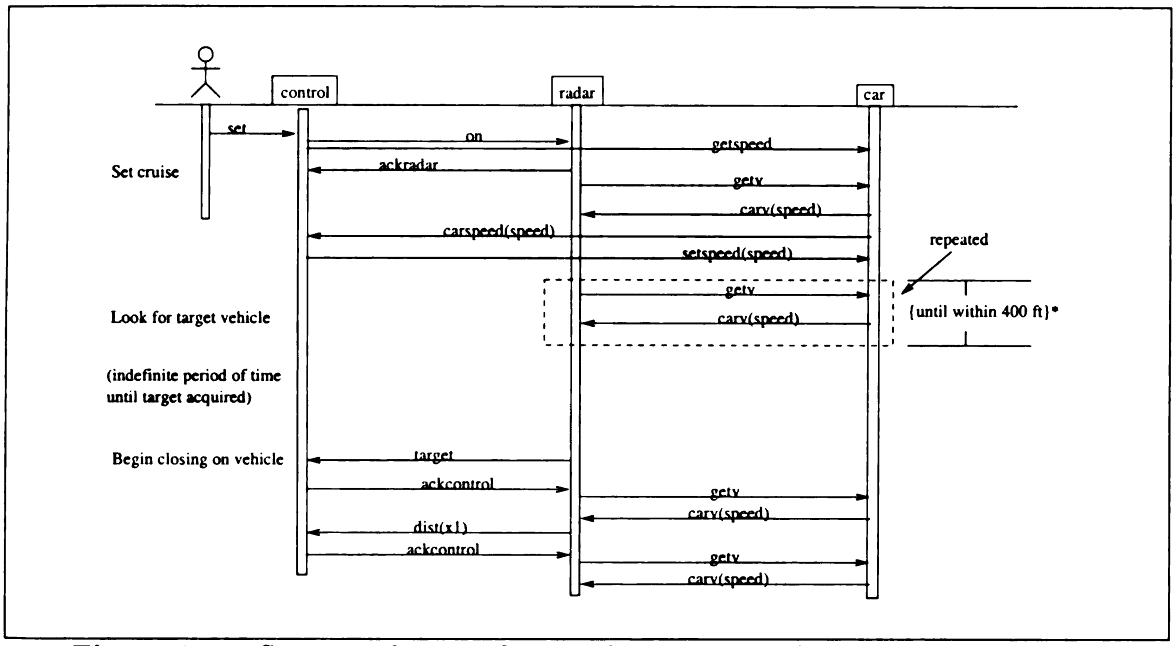


Figure 9.10: Sequence diagram for initial sequence until target is within range.

9.5 Dynamic Behavior Design

The class diagram calls for three objects in the design (artificially, four objects including the **SYSTEM** object). A major semantic decision concerns how the object-to-object message transfer should occur. One possibility is to copy semantics used for event dispatching within an object. This semantics is a no-queueing policy and loses messages if the recipient is not ready to transition when the message is sent. Another possibility is a queueing policy, where each message sent to a object is queued for either immediate or delayed use in a transition.

Non-queueing semantics avoids the problem of multiple transitions waiting for the same event. When the message is sent, every waiting transition can be taken because the set of waiting transitions is known exactly. On the other hand, non-queueing semantics could cause significant coordination problems and potentially lead to deadlocks. From an application viewpoint, a non-queueing semantics requires simpler hardware and software while a queueing semantics requires more expensive hardware and more extensive software.

Queueing semantics for messages passed between objects solves many synchronization problems because the recipient object need not be ready when the message is sent. On the other hand, there can be only one transition within an object that uses the incoming message. Otherwise, we have a dilemma: should the message be removed from the inbound queue when a transition is dispatched? If not removed, which enable multiple transitions for one message, when should it be removed from the queue?

The semantics we apply to UML allows us the flexibility to choose, and perhaps change, semantics that are most appropriate to the application.

We have decided initially to use non-queueing semantics for message communications between objects. As a preview, this decision turns out to be incorrect.

9.5.1 Calculation and Timing Conventions

SPIN contains no 'float' type of variable, and a few quick calculations show that when the radar sample rate is once per second, integer distances are too coarse and do not allow the control algorithm enough distance samples to be effective. To alleviate this problem, we modify the distance scale to *tenths* (tft) of a foot, and velocities to tft/sec. Therefore, a distance of 4000 in the model represents 400 feet.

SPIN has no facilities for simulating time, therefore, we make the assumption that class **Radar** returns a sample once per second. The deceleration is set at 15 tft/sec^2, or 1.5 ft/sec^2. The tradeoffs for using finer, perhaps more realistic scales for distance and time are discussed in Section 9.7.

9.5.2 Notation

Section 2.1 describes UML syntax in detail. For review, Figure 9.11 shows the syntax of transitions. Since all of the calculations are performed during transitions, it is important to understand this notation.

As Figure 9.11 shows, a transition consists of four parts: the triggering event, a guard on the event, actions to be performed during transition, and messages to be sent during transition. Each component of the transition is optional. Actions will not occur, nor will messages be sent, unless the guard is true when the event occurs. Both actions and messages may contain multiple parts. Each action is separated by a semi-colon as shown on the outbound transition for state "calc", class**Control**, in Figure 9.14. Similarly, multiple messages may be sent on a transition, each prefixed by "^".

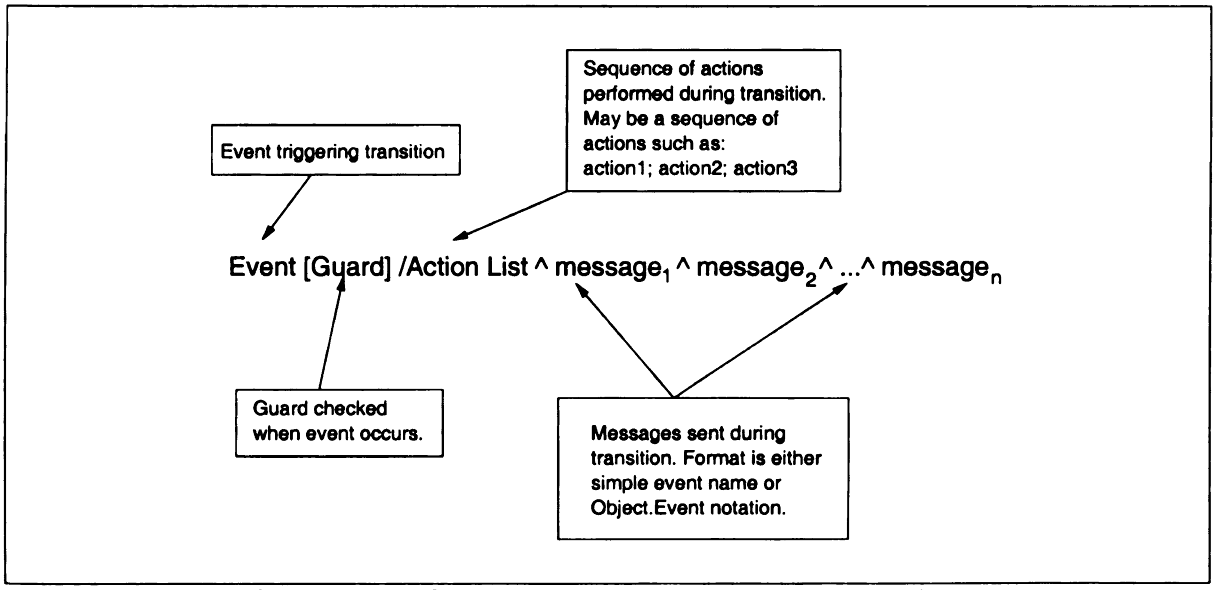

Sequence of actions performed during transition. May be a sequence of actions such as: action1; action2; action3
Event triggering transition
Event [Guard] /Action List ^ message1 ^ message2 ^ ...^ messagen
Guard checked when event occurs.
Messages sent during transition. Format is either simple event name or Object.Event notation.

Figure 9.11: State transition event syntax conventions.

9.5.3 Class Radar Dynamic Model

The **Radar** class dynamic model is shown in Figure 9.12. The model assumes a value for the target vehicle so that it can calculate closing speed, and from that, the distance from the target. The model also assumes an initial trail distance greater than 400 feet so that a target acquired signal can be sent.

The model starts in state "r_off" and remains there until it receives an "on" message. At that point, it transitions to state "r0" and starts a continual loop consisting of states "r0", "r1", and "r3", requesting the current car speed from **Car**, calculating the closing velocity, v, and updating the trail distance, x. At state "r3", when the closing distance is less than 400 feet (numerically 4000 in the model), **Radar** sends message "target" and remembers that it is now tracking by setting variable **tmode** to true. It now begins a another loop consisting of states "r3", "r4", "r_ack", "r0", and "r1", sending updated trail distances until the target is lost (detected in state "r3") or an "off" message is received (detected in state "r4"). Communications with other objects is synchronous. After sending each message, the model expects an acknowledge before generating the next x distance. This is accomplished in state "r_ack", where **Radar** waits the "ackcontrol" message from **Control**.

9.5.4 Class Car Dynamic Model

Class **car's** dynamic model is shown in Figure 9.13. The model consists of four concurrent states, none of which exits. Concurrent states were chosen because class **Car** must wait for four different messages, each of which requires independent computations. Concurrency decreases complexity because each composite state has to only wait for and handle one message. The messages **Car** handles are "getv" from **Radar**, and "getspeed", "setspeed", and "unset" from **Control**. The topmost concurrent component, "updatex", handles message "getv" from **Radar**, returning the current speed. This state also modifies the variable **realv** to match the set speed by increasing or decreasing *realv* commensurate with the acceleration/deceleration characteristics of the car. State "dogetspd" also waits for a re-

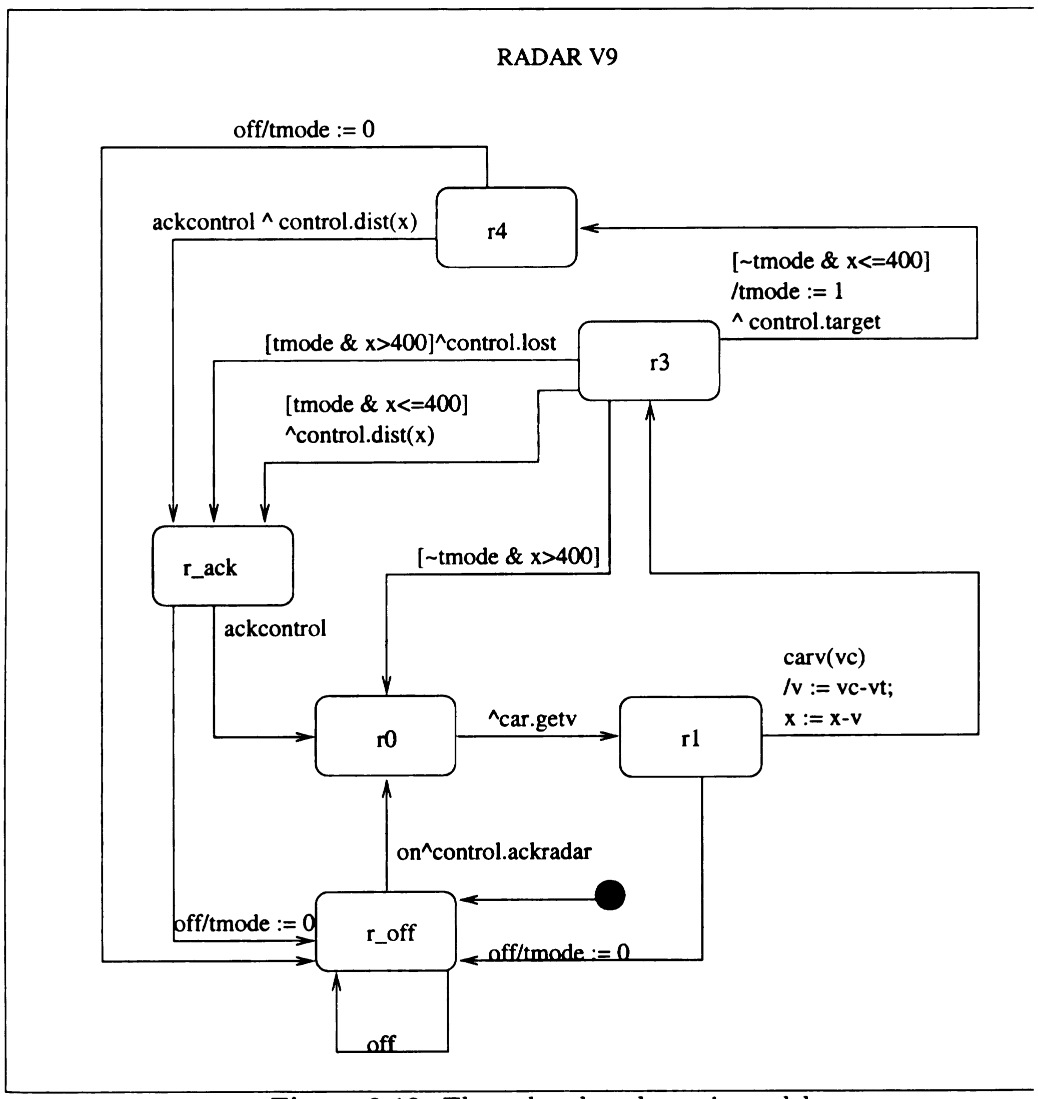


Figure 9.12: The radar class dynamic model

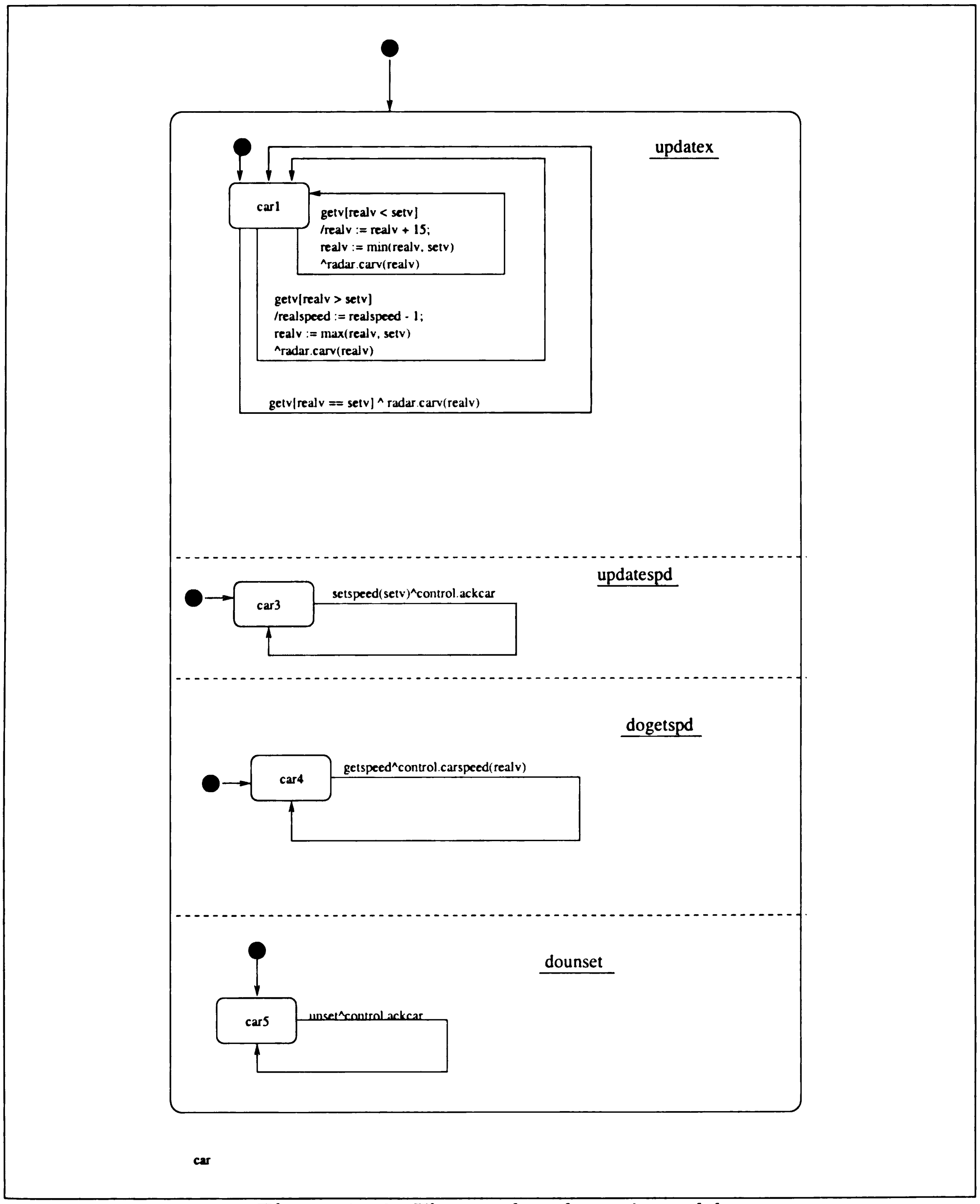
quest for speed, but from **Control**, returning the current speed of the car as computed by "updatex". States "updatespd" and "dounset" set the cruise speed and release the cruise, respectively.

9.5.5 Version 1 Class Control Dynamic Model

The first version of the dynamic model for class **Control** is shown in Figure 9.14. This version of the model does not include brake input. Instead, the plan was to further refine this model to include brakes once the control algorithm was working. As will be explained, this model has a number of problems.

The intent of the class model design is to wait for the "set" command from the driver, set the car speed to the current speed, and then wait until the **Radar** object issues a "target" message. At that time, the model starts two concurrent composite states to close on the target and maintain distance.

The sequence begins in state "idle" waiting for a "set" message with a single integer parameter that is the desired speed. This part of the model is incorrect, per the requirements. As will be seen in the refinement, a "set" signal is received from the driver, then it is the responsibility of class **Control** to obtain the current speed and make it the set speed. After setting the speed, the model remains in state "maintain" until the "target" message is received from class **Radar**. When the first distance from the target ($x1$) is received from **Radar** in state "getx", the transition starts the concurrent composite states. At this point, the top concurrent state "compute" is supposed to obtain the car speed from **Car** (in state "getspd"), accept the next x distance from the target (in state "calc"), and calculate whether the car is close enough to start coasting to the target (variable $xcoast$). During this loop, state "calc2" is supposed to monitor internal event "unsafe" in case the


updatex
car1
getv[realv < setv]
/realv := realv + 15;
realv := min(realv, setv)
^radar.carv(realv)
getv[realv > setv]
/realspeed := realspeed - 1;
realv := max(realv, setv)
^radar.carv(realv)
getv[realv == setv] ^ radar.carv(realv)
updatespd
car3
setspeed(setv)^control.ackcar
dogetspd
car4
getspeed^control.carspeed(realv)
dounset
car5
unset^control.ackcar
car
Figure 9.13: The car class dynamic model

car enters the safe zone. Concurrently, the bottom state, called "monitor", is supposed to monitor the approach to the target watching for either the "*xcoast*" distance to be reached, or the unsafe zone to be entered. In the latter case, the "unsafe" event is generated. State "monitor" obtains its updated x distances from state "compute", checking a new value whenever event "newx_ok" is generated by "compute".

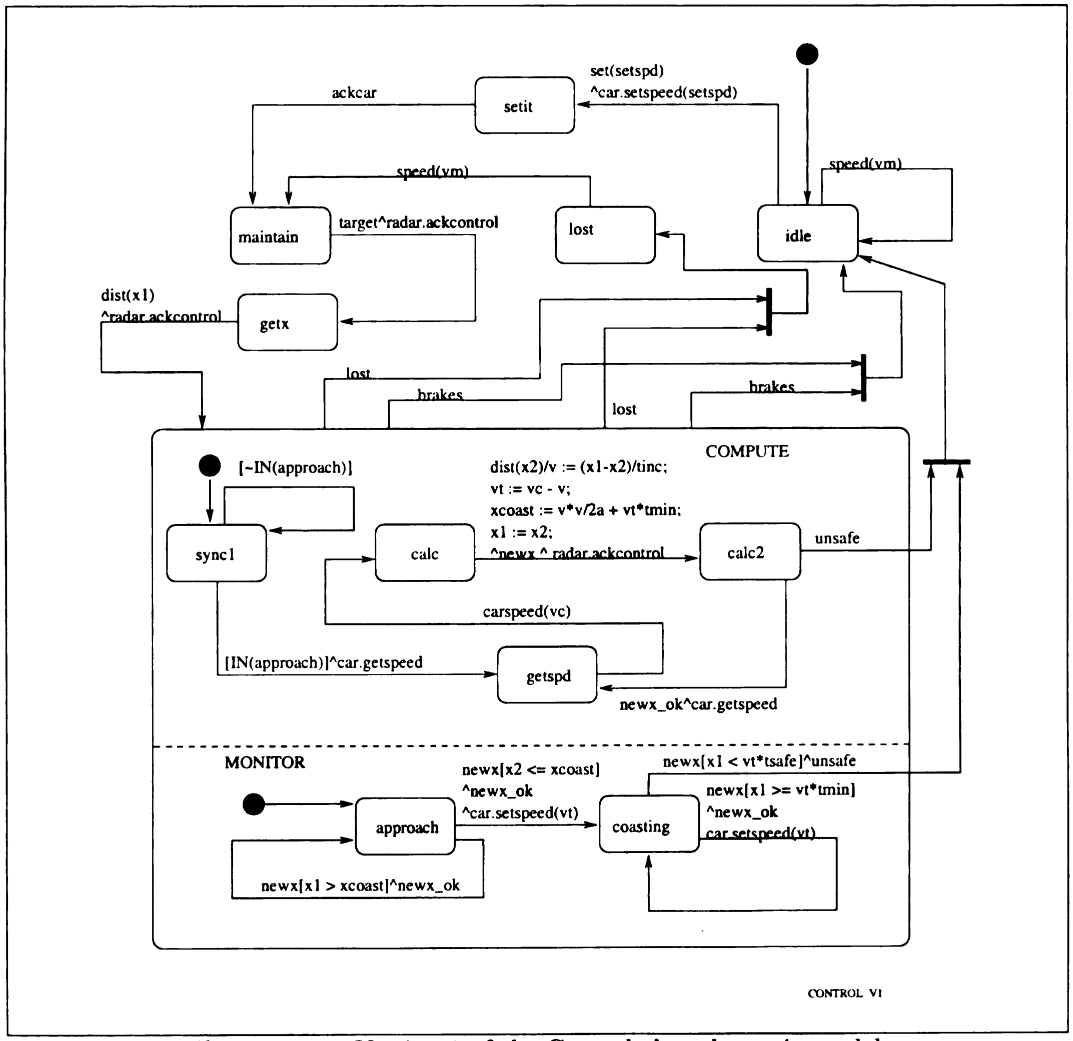


Figure 9.14: Version 1 of the Control class dynamic model.

9.5.6 SYSTEM Class

The **SYSTEM** class dynamic model is shown in Figure 9.15. The equivalent Promela statements are shown in Figure 9.16. As described above, the **SYSTEM** class starts each object and waits via the `timeout` event until the entire model has "deadlocked[1]". Deadlock at this point implies all three objects have completed initialization and are ready for events to start the model. Next, the **SYSTEM** simulates the driver of the car setting the initial speed by sending the *set* signal to **Control**.

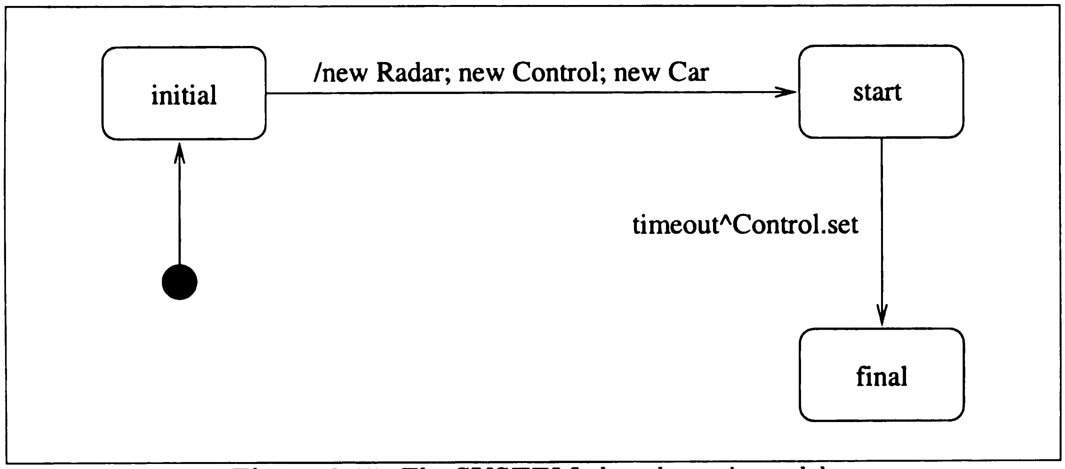


Figure 9.15: The **SYSTEM** class dynamic model.

```
int controlpid;
int radarpid;
init
{

atomic {radarpid = run radar();
controlpid = run control(); run car();}
timeout -> control_q!set;
printf("model started\n");
}
```

Figure 9.16: The Promela statements equivalent to the model in Figure 9.15.

[1]In SPIN terms, *deadlock* as signalled by the `timeout` event means there are no more statements to dispatch.

9.6 Initial Simulation Testing

SPIN contains both random simulation and model checking capabilities. During a random simulation, the model is executed and interleaving choices for various concurrent components are chosen at random. Similarly, if there are non-deterministic steps in the model, the step taken is chosen randomly. Simulation is useful early in the design to determine if the model approximates desired behavior.

The entire Smart Cruise model with Version 1 of the **Control** object was translated into the Hydra language directly from the class and dynamic model diagrams. Hydra was then used to produce Promela specifications for the model. The entire Hydra language specification for the completed model is found in Appendix B. The Promela translated from the Hydra language specification in Appendix B for the completed model is found in Appendix C.

The initial simulations failed to properly control the car. SPIN simulations reported *timeout* conditions, indicating a deadlock. Because the simulation is random, the model would occasionally execute up through target acquisition, while at other times it deadlocked earlier. No version of the model began a controlled approach to the target.

Upon examining the SPIN trace output, it became clear that the common failure mode was for an object, **Radar** for example, to issue a message and not receive a reply. In some cases, the model made little progress because the **Radar** object issued an initial "getv" message to obtain the car speed, received no reply, and deadlocked. The root problem to all unsuccessful executions was lack of coordination between the objects. Much of the difficulties are due to SPIN's lack of *fairness*. The fairness property insures each concurrent object eventually runs. For example, one possible execution sequence includes the **Radar** object transitioning through state "r0" to state "r1" before the **Car** object initializes and begins executing. Consequently, the "getv" message is sent but lost because the **Car** object is not yet listening.

As various alternatives to synchronize the objects were tried with no success, it became clear that the semantics chosen for object message passing were inappropriate for a model of this complexity. Either a queueing mechanism would have to be built into the objects in the model or the semantics of message passing would have to be altered.

Douglass [1] suggests applying queueing semantics between objects as does the UML Reference [7] and UML User's Guide [8], although the latter two are rather informal concerning the expected semantics of a message transfer. Since some type of message queuing is required, we modified the object-to-object message passing semantics to use FIFO queueing by modifying the Language Specific Class Library.

With a queueing policy in place, the problem of object coordination was solved, however object Control still failed to properly close and trail the target. Based on the recent experience with object-to-object coordination, one of the problems was traced to deadlocks between the concurrent components in object Control. The problem was getting both the "Compute" and "Monitor" states to begin coordinated execution. The model in Figure 9.14 is the last refinement for this concurrent approach. The model still fails to run reliably without deadlock.

Another problem centering around the FIFO queue for each object appeared while examining simulation trace output. Lack of careful design can allow messages to be placed in the queue in an order not expected on transitions. For example, if an object sends a message during a transition to both objects A and B, then subsequent transitions must expect the replies to be in either order, otherwise deadlocks occur. A design principle that emerges from this problem is: *deadlock is likely when transitions that send multiple* *synchronous messages to multiple objects is used.*

Even though concurrency in the Control object is unreliable, concurrency in the **Car** object performs well and does not deadlock. The difference lies in the amount of interprocess coordination required. The Control dynamic model requires close coordination between concurrent states without the benefit of event queueing, while the concurrent states in the **Car** dynamic model run independently. An important design principle can be drawn from this experience: *Only use concurrency in an object's dynamic model if the concurrent subparts are loosely coupled.* We found that we should only use concurrency for a dynamic model if:

1. An object is aggregated with a single computation for each aggregate.

2. The concurrent components are loosely coupled (in terms of coordination).

9.6.1 Refined Control Class Dynamic Model

The **Control** class's dynamic model was refined to eliminate concurrency and carefully sequence expected message arrivals as shown in Figure 9.17. The concurrent states were replaced with a sequential series of states that request the car speed, accept the radar's trail distance, and compute the coasting distance. When the car reaches the coasting distance, a loop is entered to trail the target.

Version 1 of the model made no provision for brake input. After the cruise is set, application of brakes should turn the cruise system off. Leaving this out of the initial design was deliberate, with the idea that it could be added as a refinement once the other parts of the model worked. The version in Figure 9.17 contains a provision for brake input.

In earlier versions of the model shown in Figure 9.17, the outbound transition from state "getspd" obtained the car speed from **Car** and the outbound transition on state

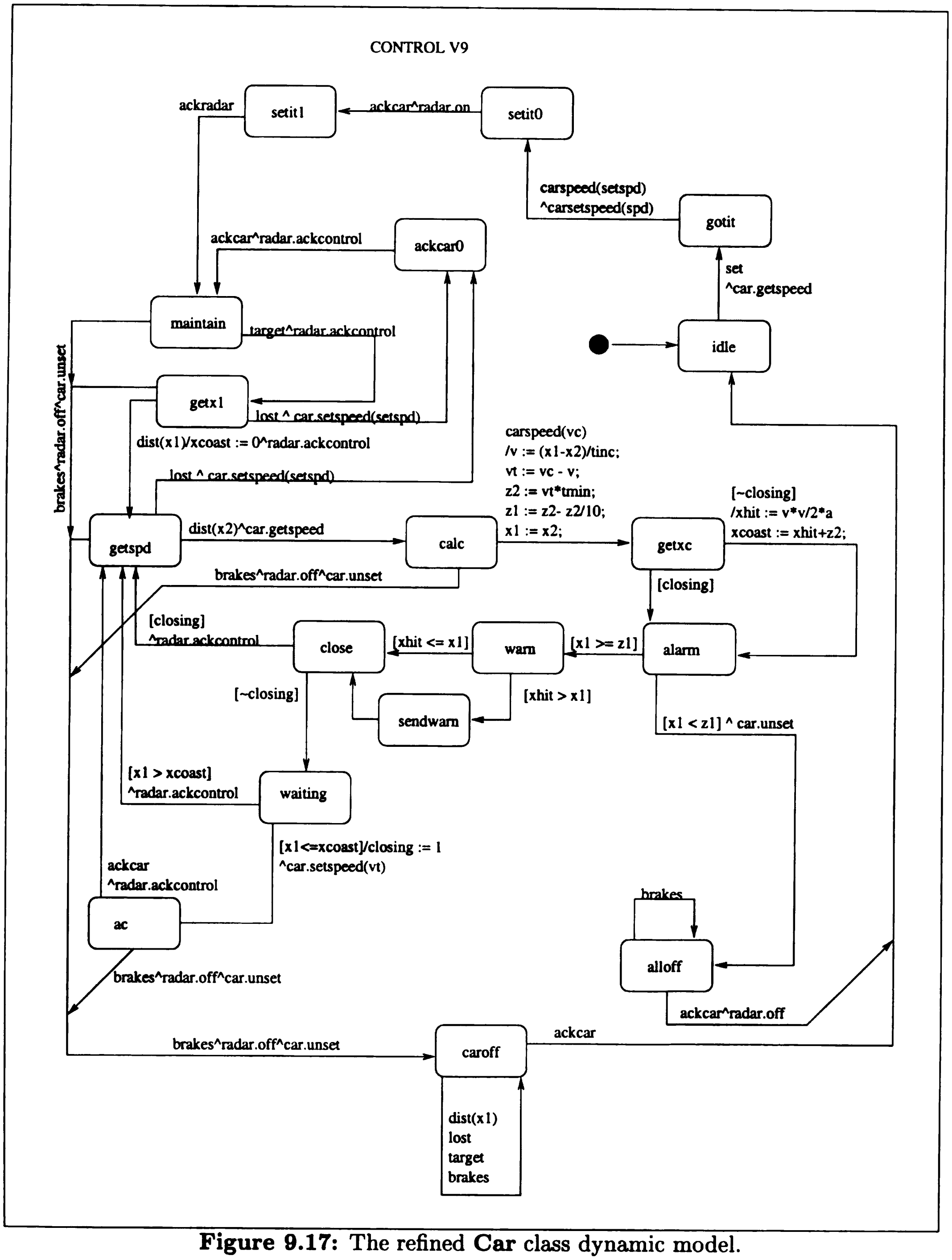


Figure 9.17: The refined **Car** class dynamic model.

"calc" obtained distance from **Radar**. Figure 9.18 shows the two affected states before the refinement and the effect of the reversed message arrivals. Although the model ran, simulation showed erratic behavior. Sometimes the car approached the target much too closely, while at other times the trail distance was too large. Examination of the SPIN simulation traces revealed that the car speed obtained on the transition for "getspd" could be different than the car speed obtained by object **Radar** during approximately the same time period. This occurred because **Radar** could make a speed request to object **Car** after **car's** response to **Control** (the speed request from **Radar** causes **Car** to update its speed). If the car is decelerating, then the values differ. The (incorrect) sequence was as follows:

1. **Control**, state "getspd" gets car speed, s_1, from **Car**
2. **Radar** gets car speed, s_2, from **Car** such that $s_1 \neq s_2$.
3. **Control**, state "calc" gets distance from **Radar** based on s_2, not s_1.
4. The car speed inferred by $x1 - x2$ is different that s_1 resulting in an incorrect assessment of the target's speed.

In other words, because **Radar** uses car speed to compute distance, the distance sent to object **Control** was inconsistent with the car speed that **Control** had just previously obtained. The result was two different values in **Radar** and **Control** for vt, the target vehicle speed. With a inconsistent value for target speed, **Control** computed incorrect throttle control information.

The solution, as shown in Figure 9.17, was to modify class **control's** dynamic model by moving the request for car speed to state "getspd" and delaying acknowledgement of object **radar's** distance message as long as possible. This prevents **Radar** from obtaining a new car speed until after **Control** has the car speed, closing distance, and has computed

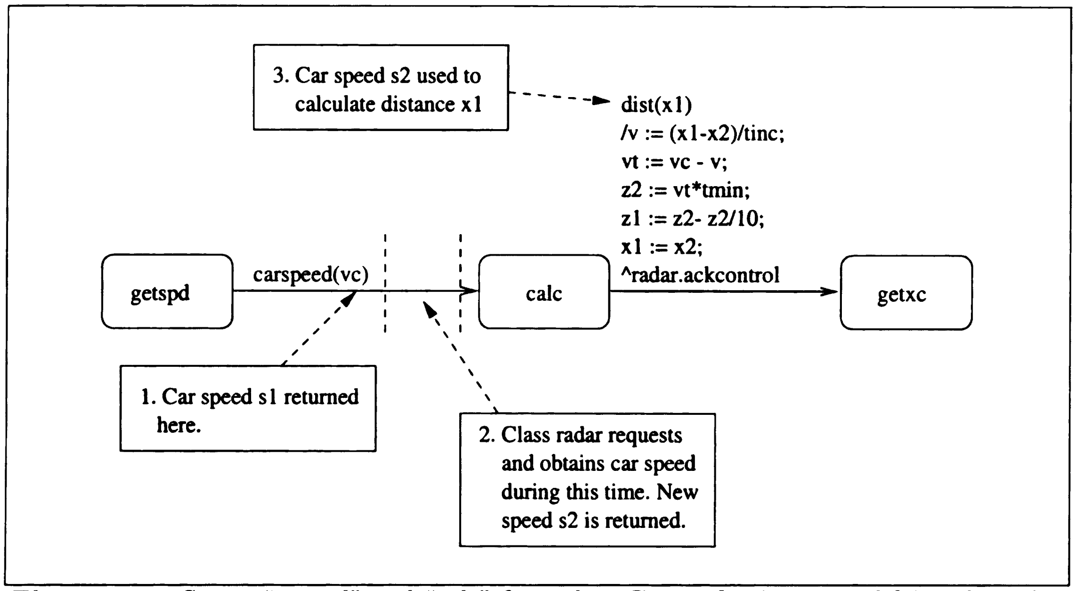

3. Car speed s2 used to calculate distance x1
dist(x1)
/v := (x1-x2)/tinc;
vt := vc - v;
z2 := vt*tmin;
z1 := z2- z2/10;
x1 := x2;
^radar.ackcontrol
getspd
carspeed(vc)
calc
getxc
1. Car speed s1 returned here.
2. Class radar requests and obtains car speed during this time. New speed s2 is returned.

Figure 9.18: States "getspd" and "calc" from class **Control** prior to modifying the order of obtaining car speed and next distance.

all relevant values. With the refinements, the dynamic models shown in Figures 9.17,9.12, and 9.13 work as expected in simulation.

9.6.2 Summary of Simulation-Based Refinements

Two primary improper characteristics of the model were observed in simulation. The first characteristic was the failure to achieve minimal system goals (*i.e.*, accepting a speed "set", and establishing a trail position behind the target). This usually occurred because the model stopped due to deadlock. The second characteristic was erratic behavior from one simulation run to the next. As previously described, sometimes the model entered the safe zone, then, given exactly the same initial values, the model would establish a trail position further behind the target than the specifications allowed.

At the conclusion of simulation testing, the model behaved as expected for all initial conditions that were tried. Unfortunately, without an extremely large number of simulation runs, the analyst cannot be sure the next simulation will not produce an incorrect behavior.

The solution to this problem is *model checking*, explained in the next section.

9.7 SPIN-Based Model Checking

The SPIN model checker verifies the behavior of a model written in Promela using a variety of analysis techniques. Each simulation run only follows one particular execution path. In order to obtain all the possible interleavings of concurrent states and execution paths, a large number of simulation runs would be required. On the other hand, the number of possible behaviors of a given model is finite, although perhaps very large. The key concept behind SPIN's model checking analysis is an exhaustive search of every possible system state to find specified behavior. Rather than simulation, the state search is accomplished, in effect, by expanding every possible execution path into one, large graph where the nodes represent states and arcs represent state transitions.

9.7.1 State Explosion Problem

A major difficulty of exhaustive state exploration is the potentially large number of states. When the number of states becomes too large, SPIN can use approximation techniques by visiting *most* states, but not all. When possible, analysis is much easier and more accurate if the number of states can be kept small enough to fit in available memory. For example, to obtain a more realistic simulation, we could have modified the model's scale to hundredths of a foot (hft) and the time scale to tenths of a second (*tsec*). Using scale factors hft and *tsec*, the radar is assumed to return hundredths of a foot every tenth of a second (or $hft/tsec$). This seems to be a more realistic sampling rate. In initial simulations, we tested both sets of scale factors, and the control algorithm performs about the same for both. The major difference is that the $hft/tsec$ scale factor causes about ten times more state transitions *per possible execution path* than the model with tft/sec. This state explosion is significant and causes analysis to consume all available memory.

SPIN's solution when the number of states is too large to fit in memory is to use an approximation techniques. During an exact, full state search, each visited state is stored in a table by using a hashing function. Every state is uniquely stored by resolving hash table conflicts. When an approximation technique is used, either the state identity is compressed in a non-identity preserving fashion, or collisions in the hash table are not resolved. Since two different states may appear to have the same identity when using these techniques, a newly visited state may appear to have been previously visited. This effectively causes pruning of the state tree because, by the analyzer's records, it has already visited this state, and therefore has already expanded execution paths starting at this state. By ignoring paths forward from the equivalent state, the state tree is effectively pruned. Holzmann [16] claims coverage for approximate analyses are good, but clearly not exhaustive.

The scale $hft/tsec$ also produces a large amount of analysis output (about ten times more) because time increments occur in .1 second intervals.

For this analysis, we felt there were no advantages, and considerable disadvantages, to increasing sampling rate realism at the cost of much more difficult analysis and simulation. We also felt that an exact exhaustive state search was preferable to an approximation. Therefore, we used the tft/sec scale throughout the model.

The actual analysis is performed by a generated C program named "pan.c" The pan.c program encodes the state transitions and calculations specified in the Promela model. Passing parameters on the compiler command line permits various kinds of analyzers to be generated. Each generated analyzer also accepts run-time options for tuning the analysis. The data flow for the process is shown in Figure 9.19. In the first transform, the Hydra language representation of the model is translated to Promela specifications. Next, the "-a" command line option instructs SPIN to produce a C program encoding an analyzer. The analyzer must be compiled in the next step to produce an executable program, called **pan**. Compiler variables defined with the "-D" switch on the command line cause various kinds of analyzers to be built. Finally, the analyst can run the **pan** program with various command line options to cause a various kinds of analyses to be performed.

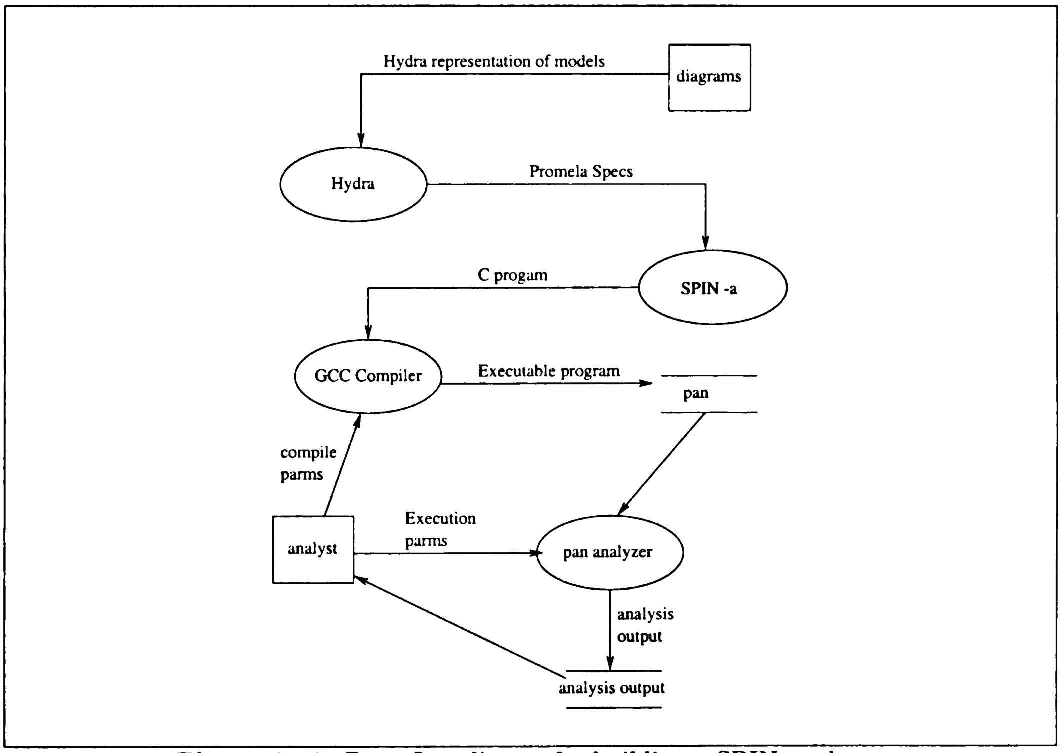


Figure 9.19: Data flow digram for building a SPIN analyzer

9.7.2 Commonly Checked Properties

Several properties of concurrent systems arise so often that common terms for them have been assigned:

Safety: The safety property states that nothing bad will happen in the system. For example, in our system, failing to disengage the throttle after the safety zone had been entered would be a safety violation.

Liveness: The liveness property states that the system makes progress during its execution. No concurrent component is "starved" by failing to execute.

Deadlock: Deadlock means that two or more components are each blocking the other from execution. In a concurrent system, this usually occurs when a message that generates a reply is missed. The sender and the receiver deadlock awaiting each others' responses.

Livelock: Deadlock usually implies a component has blocked, or stopped at some point. Livelock is similar to deadlock but occurs when the components are locked into a cycle of states waiting for a resource or an event that the other component holds. Livelock is harder to detect because it appears as progress unless the cycle is detected. Even then, the cycle may be valid.

Reachability: Reachability is a property of a state, and not a the model, but reachability analysis is common and important. Reachability means a state can be reached by some execution path. An unreachable state usually implies something incorrect in the model.

9.7.3 State Assertions

A variety of model checking options are available. The simplest is an assertion about a state of the system. SPIN uses a predicate expression in a Promela **assert** statement to make a claim about a property of a single system state. The **assert** statement is useful for checking system invariants and for checking a single global state predicate. For example, in the Smart Cruise model, given the distance to the target is $x1$ and the minimum safe distance from the target is $z1$, when the model is given a closing velocity small enough that it can achieve a proper trail distance, the assertion **`assert(x1 >= z1)`** should always be true after state "calc" has completed its calculations. We can add the assertion as an action statement on the transition for state "calc" to test this case. Figure 9.20 shows the modification to the Hydra description of state "calc" and Figure 9.21 shows the output of the analysis. Since there is no mention of an assertion violation, SPIN detected no violation.

```
        State calc { /* perform all the calculations */
           Transition "carspeed(vc)",
           "/ v:= (x1-x2)/tinc;",
           "vt := vc - v;",
           "z2 := vt*tmin;",
           "z1 := z2 - z2/10;",
           "x1 := x2;",
           "print('v=%d, x1=%d, z1=%d z2=%d',",
           " v,x1,z1,z2);",
           "assert(x1 >= z1)" to getxc;
           Transition "brakes^radar.off^car.unset" to caroff;
        }
```

Figure 9.20: Modification of state "calc" to include an assertion statement.

To demonstrate the model checking is working as expected, the closing velocity can be increased to a value that makes it impossible for the car to avoid either hitting the target or violating the safe zone. The closing speed is increased by decreasing instance variable vt, the target speed, in **Radar**, to 80 ft/sec. This provides a closing speed of $30 ft/sec$. Using the distance-required formula, $v^2/2a$, mentioned previously, a closing speed of $30 ft/sec$ requires 600 feet of coasting distance, which far exceeds the 400 ft radar range. The modification was made by directly editing the Hydra language input to change the initial value of instance variable vt. This should cause the **`assert x1 >= z1`** assertion to fail. Figure 9.22 shows the output of the state space exploration when the closing speed is increased. As predicted, the assertion fails.

```
(Spin Version 3.3.1 -- 11 July 1999)

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       - (disabled by -E flag)

State-vector 496 byte, depth reached 1809, errors: 0
    8856 states, stored
   12323 states, matched
   21179 transitions (= stored+matched)
    5881 atomic steps
hash conflicts: 253 (resolved)
(max size 2^18 states)

5.794   memory usage (Mbyte)
```

Figure 9.21: pan output from the model with no assertion violation.

```
pan: assertion violated (control_V.x1>=control_V.z1) (at depth 1125)
pan: wrote sc.v9.pr.trail
(Spin Version 3.3.1 -- 11 July 1999)
Warning: Search not completed

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       - (disabled by -E flag)

State-vector 496 byte, depth reached 1124, errors: 1
     924 states, stored
       0 states, matched
     924 transitions (= stored+matched)
     201 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.903   memory usage (Mbyte)
```

Figure 9.22: pan output from the model with a target speed of 80 ft/sec and a car speed of 110 ft/sec producing an assertion violation.

9.7.4 State Reachability Analysis

Many behaviors require assertions about a sequence of states or the states that can be reached. The simplest verification is to check for state reachability.

SPIN State Reachability Issues

Each block of Promela statements representing a simple state generated by Hydra begins with a labeled "printf" statement announcing which state the model has entered. Apparently, SPIN appears to occasionally ignore "printf" statements in state reachability analysis. Consequently, examining the "unreached states" listing at the end of a pan analysis does not definitely tell the analyst whether or not a state has been entered. In other words, a state may be unreached but the analysis fails to contain the unreached "printf" in the analysis output. To work around this problem, we modified Hydra to place a "skip" statement immediately after the label designating the state, then followed the "skip" with the "printf". This approach works but requires correlation of the unreached state listing, which is given by a Promela line number, with the actual Promela code. A small tool could be constructed to automatically read the analysis output and match it with the Promela code to determine unreached states.

State Reachability Results

In simulation, with a closing speed of less than 10 ft/sec, the model consistently controlled the car to a reasonable trail distance and remained there with a relative velocity of zero. Analysis of the output as described above shows that states "ackcar0", "sendwarn", "alloff", and "caroff" were not reached. The reachability analysis implies that neither a "target lost" or "brakes" signal was received, nor did the model enter the unsafe zone close to the target vehicle.

With the car speed set at 110 ft/sec and the target at a speed of 80 ft/sec, the closing speed is 30 ft/sec. The formula $v^2/2a + 2t_{min}$ shows that 460 feet are required to obtain the proper trail distance. When the radar obtains the target at 400 feet, object Control does not have enough room to position the car in trail.

Figure 9.23 shows the last part of the simulation output. In simulation, Control first starts issuing warnings at 279 feet that the target is going to be struck. Later, it warns that the safety zone has been entered, releases the cruise control, and returns to the "idle" state. In the model, variable v is the closing velocity, $x1$ is the distance to the target, $z2$ is the two second trail distance, and $z1$ is the minimum safe distance.

We would expect more states to be reached when the model is analyzed for state reachability with excessive closing speed, and indeed, all states in **Control** except "ackcar0" and "caroff" were reached. This implies that the entire control loop was executed (states "getspd", "calc", "getxc", "alarm", "warn","close", "waiting", and "ac") followed, by the states composing the exit sequence, starting at state "alarm", then state "alloff", when the safety zone is entered. The only way "ackcar0" is entered is if a message denoting target loss is received, and only a "brakes" message can cause state "caroff" to be entered.

9.7.5 Progress State Analysis

Dynamic models frequently contain cycles, some intended and some unintended. SPIN is capable of identifying cycles, and, depending on how the cycle is marked, can determine if the cycle is "good" or "bad". *Progress state analysis* verifies that there are no infinite behaviors (cycles) of only unmarked states. A state is *marked* by attaching a *progress label* to a Promela statement. Progress state analysis insures lack of deadlock and livelock caused

```
v=165, x1=1575, z1=1440 z2=1600
in state control.getxc
in state control.alarm
in state control.warn
in state control.sendwarn
WARNING: GOING TO HIT TARGET
in state control.close
in state control.getspd
in state radar.r0
in state radar.r1
realv=950 setv=800
in state car.car1
x is 1425
in state radar.r3
Sending x=1425 to control
in state radar.r_ack
in state control.calc
in state car.car4
v=150, x1=1425, z1=1440 z2=1600
in state control.getxc
in state control.alarm
WARNING: TOO CLOSE
x1=1425, z1=1440
in state control.alloff
in state car.car5
in state control.idle
in state radar.r_off
timeout
```

Figure 9.23: Last part of SPIN output when closing velocity is too high.

```
pan: non-progress cycle (at depth 3114)
pan: wrote sc.v9.pr.trail
(Spin Version 3.3.1 -- 11 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)

State-vector 500 byte, depth reached 3264, errors: 1
    1471 states, stored (1473 visited)
       2 states, matched
    1475 transitions (= visited+matched)
     321 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

2.210   memory usage (Mbyte)
```

Figure 9.24: pan analysis run showing a non-progress cycle in the target trail loop.

by an unintended cycle, or race condition, but it also can be used to verify positive behavior by ensuring that the model is visiting intended states.

Progress state analysis in SPIN is accomplished by compiling `pan.c` with a `-DNP` switch and passing the `-l` switch to the resulting analyzer.

Resetting the closing speed to 20 ft/sec by setting vt to 90 ft/sec (to enable a trail loop), and checking for progress loops with no states marked produces the output in Figure 9.24. The first line of the output indicates there is a non-progress loop. Running SPIN in trace mode, and analyzing the output shows that the loop consists of states "getxc", "alarm", "warn", "close", "getspd", and "calc". This is exactly the expected loop for target trailing.

When we add a progress label to state "calc" and re-run the same analysis with a closing speed of $20 ft/sec$. Figure 9.25 now contains no mention of a loop. This implies there is

```
(Spin Version 3.3.1 -- 11 July 1999)
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)

State-vector 500 byte, depth reached 3248, errors: 0
    4677 states, stored (7006 visited)
    9084 states, matched
   16090 transitions (= visited+matched)
    5074 atomic steps
hash conflicts: 107 (resolved)
(max size 2^18 states)

3.746   memory usage (Mbyte)
```

Figure 9.25: The same analysis as shown in Figure 9.24 but with a progress label attached to state "calc".

no other cyclic behavior in the model *except* those global states involved in the trail loop described above. Note that this includes all three objects and not simply object **Control**. The loops in objects **Radar** and **Car** are part of the system-wide loop that includes the trail loop in object **Control**.

The progress label can be added either by directly editing the Promela specifications produced by Hydra, or by inserting a null transition state in the Hydra language specification whose name starts with the word "progress". Since the inserted state has no side effects, it does not affect the behavior of the model.

9.7.6 Never Claims

A SPIN *never claim* is a means of specifying temporal properties across a series of states. The never claim specifies a series of conditions that should never occur, and therefore, checks safety properties of a system. A never claim is written as a modified **proctype** using Promela statements. If a never claim procedure exits by dropping through its final statement, then the claim is violated.

Any statement with no side effects can be contained in a never procedure. The SPIN analyzer interleaves the execution of never claim statements with statements from the model. When a never claim is present in a model, a new state machine is formed as the product of the original model and the state machine formed by the never claim statements. Effectively, the never claim statements are conditions on the current global system state, hence the requirement for non-side effect only statements. Figure 9.26 shows a never claim that illustrates another way of insuring the car achieves the proper trail distance and never stops trailing.

Lines 3 through 6 in Figure 9.26 wait until state "idle" has been entered. Never claims verify a state sequence by always starting at the initial state, thus the claim must cover all prefix states until entry into state "idle". The `skip` statement on line 4 provides a non-deterministic match for any statement until state "idle" is reached. At that point, the loop on lines 3–6 exits and the loop on lines 7–10 begins. Again, the loop skips prefix states while "idle" is still the active state. The `skip` statement is required here because an undetermined number of states could be transitioned in objects **Radar** and **Car** until "idle" exits.

When state "idle" exits, the last loop on lines 11–13 insures that state "idle" is never entered again. If it is, then the loop exits via the `break` statement and the never claim ends, signifying a violated claim. Figure 9.27 shows the results of the analysis with the never claim in Figure 9.26. The lack of a message saying the claim is violated means the claim succeeded.

```
/* Verifies that at low enough closure speeds, the car comes up behind the
   target and stays there forever. If the trail loop is exited, we return
   to state idle    */
never
{
/* wait until state idle is entered */
 do
:: skip
:: control[controlpid]@idle -> break
od;
/* now wait until state idle is exited  */
do
:: skip
:: !(control[controlpid]@idle) -> break
od;
/* and if we come back to state idle, it's an error */
do
:: control[controlpid]@idle -> break
od
}
```

Figure 9.26: A never claim specifying that state "idle" is only entered once, at the start of execution.

```
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
(Spin Version 3.3.1 -- 11 July 1999)
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)

State-vector 504 byte, depth reached 3114, errors: 0
    6314 states, stored
    2919 states, matched
    9233 transitions (= stored+matched)
    3445 atomic steps
hash conflicts: 72 (resolved)
(max size 2^18 states)

4.565   memory usage (Mbyte)
```

Figure 9.27: An analysis run using the never claim from Figure 9.26.

Establishing Liveness Properties

We have now used a number of methods to insure that the model correctly enters the trail position and stays there, but there are other properties that we would like to verify. One such property is to insure that whenever a "target" message is sent by class **Radar**, an acknowledgement is received. Because this is a positive property, we can express it as what should happen, negate that, and use the negated form for the never claim. First, we need a new temporal operator

Leads to, written as $\leadsto$, is a commonly defined temporal operator [67] that makes use of the properties of henceforth and eventually. *Leads to* is defined as:

$$A \leadsto B \equiv \Box(A \Longrightarrow \Diamond B). \tag{9.1}$$

The right side of Expression (9.1) states that from now on, whenever A becomes true then either B is true or will become true. *Leads to* is commonly used to express liveness properties as shown in the examples below.

Our intent is to express "target occurrence leads to acknowledgement", but for a SPIN never claim this has to be negated to say "it is never the case that {target Occurrence leads to acknowledgement} is false".

The SPIN LTL translator only accepts simple, lower case variables, so let p represent "target sent" and q represent "acknowledgement received". Then the formula needed is $\neg(p \leadsto q)$. Translated in terms of SPIN operators, we get:

$$\neg(\Box(p \Longrightarrow \Diamond q)). \tag{9.2}$$

Formula (9.2) says "henceforth, {p implies q is eventually true}, is false". We want to show

Formula (9.2) itself is never true.

The semantics of the Promela statement `A??[B]` is to test queue `A` for the occurence of message `B` *anywhere* within queue `A`. The statement does not modify the queue in any way, and the statement blocks until message `B` is in queue `A`. Promela also contains a **`#define x y`** statement as in C [65] that defines simple variable `x` to textually equivalent `y`, where `y` can be some complex expression. The interpretation of defined variables occurs at compile time.

The name of the input queue for messages to an object is **`CLASS_q`**, therefore using the **`#define`** statement, the variables p and q are defined by placing **`#define p control_q??[target]`** and **`#define q radar_q??[ackcontrol]`** at the top of the driver file for the Hydra translation. The predicate p is true when message "target" is waiting in class **control's** queue and predicate q is true when the acknowledgement "ackcontrol" has reached **radar's** object queue. The sequence p, then q, implies a target has been sensed and has been acknowledged by **Control**.

The LTL formula is passed to SPIN as `!([](p -> <>q))`, so SPIN can generate an analyzer with the corresponding never claim included. When the resulting analyzer is run, the usual SPIN output is produced with no mention of a claim violation, thus it is always true that a "target" signal results in an "ackcontrol" signal.

9.7.7 Acceptance Cycles in Never Claims

The opposite of a progress loop is called an *acceptance cycle.* Checking for acceptance cycles insures there are no infinite cycles that *are* marked. Marking of a state is accomplished by placing a label on a Promela statement for the state starting with "accept". In our model, using acceptance markings would require marking every *bad* cycle. This could be hard to do and may miss an important cycle. But in never claims, the acceptance cycle is very useful.

Figure 9.28 shows the SPIN generated never claim from Formula (9.2). Lines 4–8 contain two guards that can be selected non-deterministically. If p and q are true, then there is a path back to Line 4. When both p and q are false, control transfers to **accept_S4**, which starts the acceptance loop. If q remains false, an acceptance cycle exists and thus the claim is "matched". A matched claim signifies violation of the claim.

```
1          never { /* !([](p -> <>q))
2          */
3          /* >>0,0<< */
4  T0_init:
5          if
6          :: (! ((q)) && (p)) -> goto accept_S4
7          :: (1) -> goto T0_init
8          fi;
9  accept_S4:
10         if
11         :: (! ((q))) -> goto T0_S4
12         fi;
13 T0_S4:
14         if
15         :: (! ((q))) -> goto accept_S4
16         fi;
17 accept_all:
18         skip
19         }
```

Figure 9.28: Never claim to check for the issuance of an acknowledgment when a target message is sent. This claim was generated by the input to SPIN seen on line 1. Line 1 was derived from Formula (9.2).

The requirement to check for negative properties, and the manner in which the acceptance loop functions, can make analysis with SPIN's never claims somewhat confusing. In fact, if the analyzer is to check for an acceptance cycle, as is often found in a never claim, then a special switch (-a) must be passed to pan at execution time. Otherwise (without acceptance labels), no switch should be specified. This means that even if the analyst allows SPIN to generate a never claim from a LTL formula, he must still know if **accept** labels are used to construct the claim so he knows whether or not to use the -a switch.

For comparison with the correct version of the model, we can modify the model at state "maintain" in **Control** by removing the acknowledgement. This should cause the never claim generated from Formula (9.2) to fail. Figure 9.29 shows the modified Hydra statements for state "maintain" to cause a claim violation. Figure 9.30 shows the output from an analysis using the modified model and the never claim shown in Figure 9.28, which was generated by Formula (9.2). As Figure 9.30 shows, pan now reports an acceptance cycle in the never claim, implying that "ackcontrol" is never sent.

```
State maintain { /* wait for target acquisition */
   Transition "target" to getx1; /* ack removed from this stmt */
   Transition "brakes^radar.off^car.unset" to caroff;
}
```

Figure 9.29: Modification of the transition to remove the acknowledge message.

```
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
pan: acceptance cycle (at depth 298)
pan: wrote sc.v9.pr.trail
(Spin Version 3.3.1 -- 11 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)

State-vector 500 byte, depth reached 299, errors: 1
     134 states, stored (135 visited)
       1 states, matched
     136 transitions (= visited+matched)
      32 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

Figure 9.30: Analyzer output using the never claim in Figure 9.30 and a modified model to disable message acknowledgement.

9.7.8 Analysis With Non-Deterministic Transitions

So far, we have not examined the behavior of the brake input to the Control object. Brake input can occur at any time after the cruise is set. To verify the behavior of the model, we would like to specify that brake input can arise at any time. To accomplish this, we modify object **Radar** at state "r3" by adding the new transition {`Transition "[tmode & x <= 4000]^control.brakes" to r_ack;`} to state "r3". Figure 9.31 shows the transition addition graphically, where dashed lines show the existing transitions and the solid line shows the proposed new transtion. When x is less than 400 feet, state "r3" can either send the message `dist(x)` or `brakes`, non-deterministically. Now, we want to verify that whenever brake input *is* sent, object **Control** turns off the radar, disables the throttle control, and returns to state "idle".

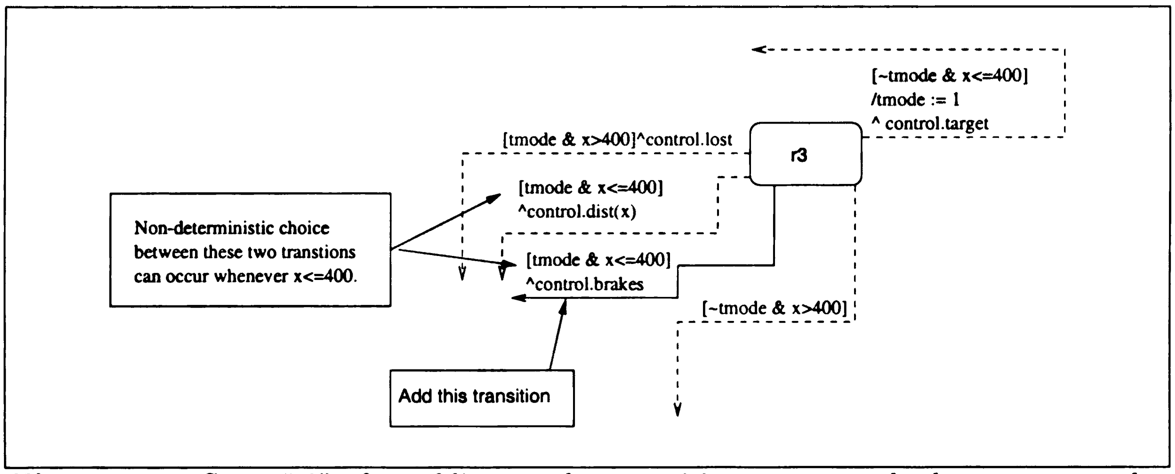


Figure 9.31: State "r3" after adding another transition to cause a brakes message to be generated non-deterministically. The dashed lines show the existing transitions and the solid line shows the new transition.

The positive condition we are testing for is "brakes leads to state idle". To form a never claim, this formula is negated, as before.

This claim follows the form of Formula (9.2) exactly. All that is required is to redefine p and q. For the definition of p, we want a statement expressing the presence of a "brakes" message in the input queue for **Control**. As seen eariler, the form of the define is:

`#define p control_q??[brakes].`

The defintion of q is different from before. We want "brakes" to lead to a return to state "idle" in **Control**. This would indicate that all systems had been reset and the system was ready for a new "set" message. The Promela predicate to test presence at a particular label is

`control[controlpid]@idle`,

where `control` is the name of the **`proctype`** where label `idle` is located and variable `controlpid` is the PID of the **`proctype`** for object **Control**. Therefore, this predicate is true when **`proctype`** `control` is at label `idle`. The entire define is:

`#define q control[controlpid]idle.`

The first time this claim was run, it failed as shown in Figure 9.32. The analyzer reported an acceptance cycle at label `T0_S4` (see Figure 9.28) implying state "idle" was not re-entered. The trace output in Figure 9.32 also shows the previous state in object **Control** was previously "calc", the current state is "caroff", and most importantly, `control_q` contains the message "carspeed". Figure 9.33 shows the three relevant states from Figure 9.17.

The problem is not accounting for a transition on message "carspeed" at state "caroff". In other words, the **Control** object is in state "caroff" waiting for one of the listed messages, but message "carspeed" is at the front of the message queue. This creates a deadlock.

When "carspeed" is added to the self-transition on "caroff" as shown in Figure 9.34, the analyzer reports no acceptance cycles, implying proper behavior for the brake message.

```
<<<<<START OF CYCLE>>>>>
616:    proc 0 (NEVER) line   471 "never"
616:    proc  - (:never:) line 471 "sc.v9.pr" (state 11)
        [(!((control[controlpid]._p==idle)))]
                control_V.a = 15
                control_V.xcoast = 3333
                control_V.setspd = 1100
                control_V.v = 200
                control_V.tmin = 2
                control_V.x1 = 3700
                control_V.vc = 1100
                control_V.x2 = 3500
                control_V.vt = 900
                control_V.z1 = 1620
                control_V.z2 = 1800
                control_V.xhit = 1333
                control_V.tinc = 1
                control_V.closing = 0
                queue 2 (control_q): [carspeed][ackcar]
                radar_V.x = 3500
                radar_V.tmode = 0
                radar_V.vc = 1100
                radar_V.brks = 0
                radar_V.vt = 900
                radar_V.v = 200
                queue 4 (radar_q):
                car_V.realv = 1100
                car_V.setv = 1100
                queue 3 (car_q):
                queue 1 (t): [free]
```

Figure 9.32: Trace output showing acceptance cycle in never claim used to verify behavior of "brakes" message.

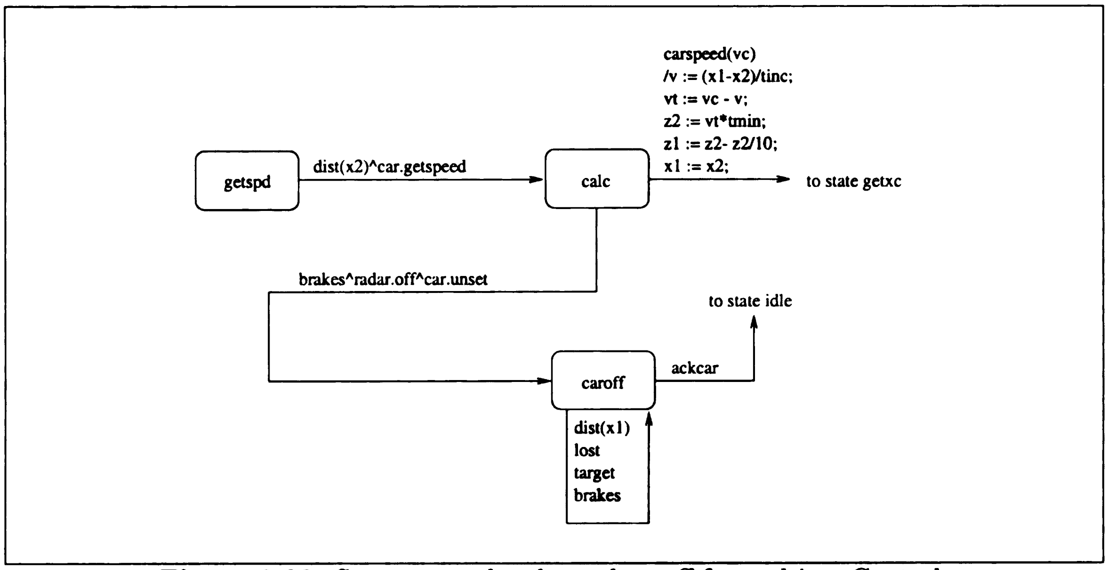


Figure 9.33: States getspd, calc, and caroff from object Control.

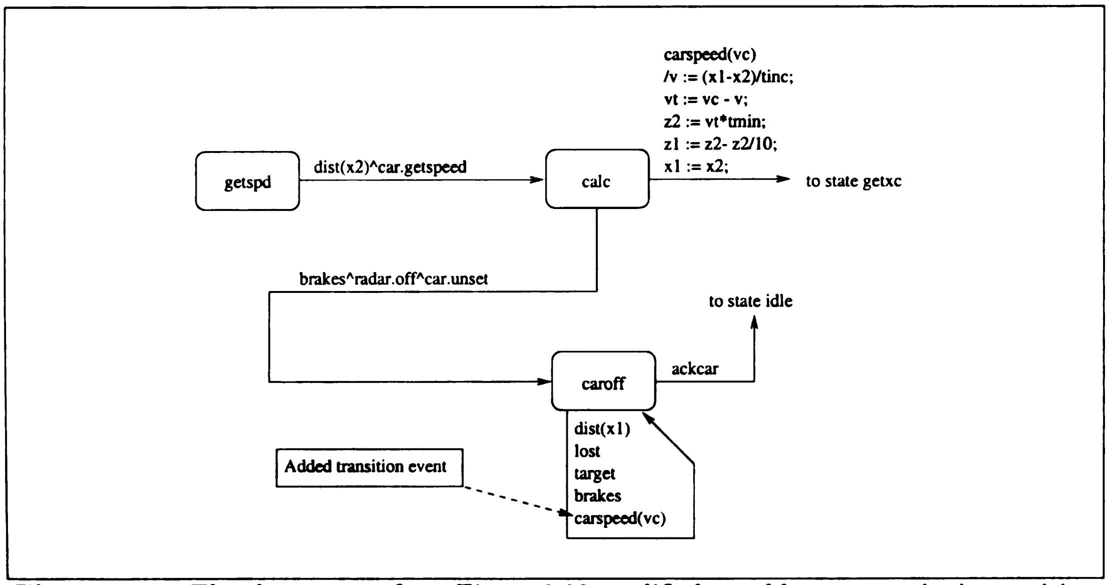


Figure 9.34: The three states from Figure 9.33 modified to add **`carspeed(vc)`** transition to state "caroff"

9.8 Discussion of Using SPIN for Analysis

A number of difficulties arose using SPIN to analyze the semantics of this mapped UML model. This section overviews the difficulties and the causes.

9.8.1 Bugs

The first mapping of semantics to implement queued messages for objects used a two-valued channel declared as:

```
chan control_q=[5] of {mtype, int};
```

where `control_q` is the input queue for class Control. Models constructed with these semantics executed properly in simulation, but during analysis, several assertion statements reported failure, but the trace of the variables in the assertion state indicated that the assertion could not have failed. In some cases, the analyzer reported it had run out of memory after a very long execution. Trace output also contained transitions firing when the trace of variable values showed the transition was impossible.

Diagnostic print statements added to the generated analyzer C code lead us to the conclusion that variables were being inexplicably changed around channel receive statements.

The semantics of the object queue was changed to use two queues with single value message slots. One carried the signal name as before, while the other carried the signal parameter. After making these changes, the same model began exhibiting expected behavior and the analysis runs shortened significantly.

The explanation for the long analysis runs lies in random changes to model variables. This effectively generates a large number of new global system states that increases the storage requirements to the limit of memory and increases run times dramatically.

As mentioned above, another problem concerns the simple analysis of state reachability.

Occasionally, the analyzer omits a state in its unreached state list. As described earlier, we verified the behavior of the model in two ways. First, pan reported not reaching Promela statements immediately after the first statement in the state. Since the first statement of the state (a `printf`) cannot block, it must also have been reached.

Secondly, we placed an `assert(0)` statement in place of the unreached state. Since the state was not listed as unreached, it must have been reached, but then the `assert(0)` would be executed, causing an assertion violation. No assertion violations were encountered. The conclusion is the omitted state is unreached but SPIN failed to report it.

9.8.2 Challenges

As mentioned in Section 9.7.4, SPIN occasionally ignores `printf` as being unreached in a state analysis. While it is true that `printf` has no side-effect, and its output is not required, or even desired during state analysis, the analyzer (pan) should at least report whether the statement was visited or not. This caused us difficulties because `printf` is the first statement in a Hydra generated Promela state. Failure to report it as unreached made the initial analysis of the Smart Cruise model difficult.

9.8.3 Lack of Fairness

Another problem concerns *fairness*. In concurrent systems, fairness is defined as the property of allowing each process some time to run. No process is completely stopped, or *starved*. An early attempt to implement the "brakes" message used an extra concurrent state in class **Radar** as shown in Figure 9.35. The concurrent component contains a single state whose sole responsibility is to listen for the "brakes" message and set a boolean variable true. In the main part of the **Radar** class (RMAIN), the value of the variable is checked, and when set, causes a "brakes" message to be sent to class Control.

Verification of the behavior of the "brakes" message was performed nondeterministically with never claims as described earlier, but the never claim analysis continually failed. Trace output showed that the concurrent state "r_brakes" in RBCHK was receiving the "brakes" message but stopping before setting the variable true. In other parts of the model, the control loop continued to execute, and therefore, the "brakes" message was never communicated. Even though the intended semantics is to complete each transition before enabling another, the transition in RBCHK would not complete. When several processes were ready to execute, the analyzer always chose the worst case. Clearly, this is a fairness issue.

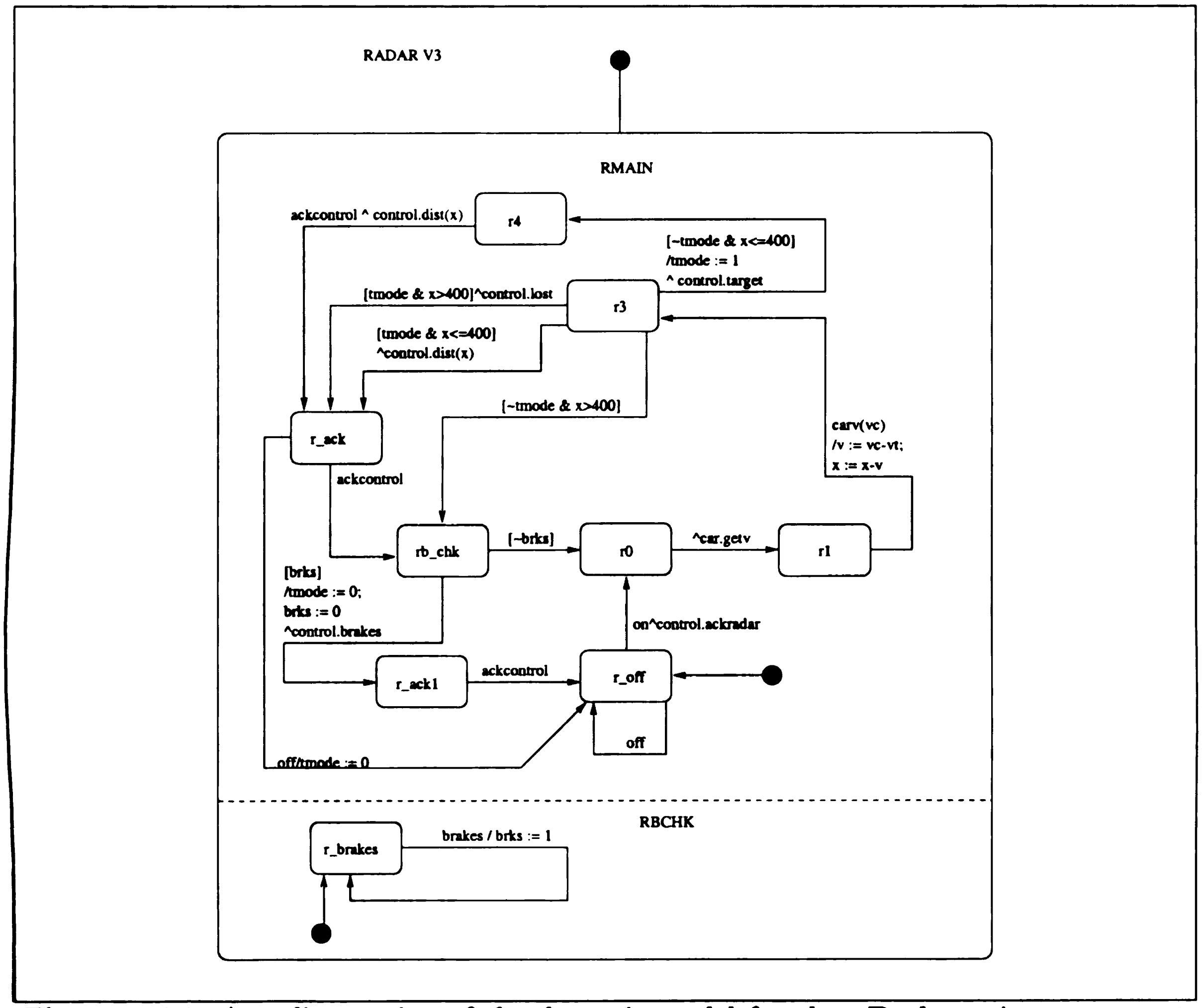


Figure 9.35: A earlier version of the dynamic model for class **Radar** using concurrent substates. The model fails due to lack of fairness.

SPIN has an option to enable "weak fairness" in its analysis, but when enabled, the concurrent process was still starved. To alleviate the problem, queueing semantics were again altered and the model changed to conform to the new semantics.

The original queued semantics allowed any queued message to cause a transition. Queue order was not enforced. Because of lack of fairness, messages can remain in the queue while other messages are inserted and retrieved. The altered semantics requires messages to be dispatched in the order they were sent (a FIFO queue). The altered queue semantics required adding more states to the model and modifying several transitions to ensure messages were always sent in the order they were received.

Although fairness cannot always be assumed, there should be a means of forcing it when it is required. SPIN apparently contains heuristics that always assume *a lack* of fairness.

9.8.4 Never Claims

As mentioned earlier, never claim construction is another area of difficulty. Theoretically, the language accepted by intersection of the automata presented by the model and the language represented by the never claim automata is formed. If positive claims were allowed, the analyzer would have to verify that every execution sequence contained both languages. It is much easier to verify that the intersection is null.

Writing a never claim for safety properties is not too difficult because what *must not* happen is specified, but writing a never claim for liveness, or positive behaviors, contains many pitfalls. The procedure recommended by Holzmann [16] is to express the liveness property, then negate it. This is the procedure we used, but even so, an explanation of what the analysis seeks to match is still difficult.

SPIN sometimes failed to include the never claim when a LTL formula was specified. A missing never claim is not discovered until trace output reports being unable to find a never claim. To be sure the never claim was in place, we generated the never claim separately from the model using SPIN, and manually appended it to the Promela generated by Hydra.

Chapter 10

Conclusions and Future Investigations

Diagrammatic specifications as found in UML are gaining popularity, perhaps because it is easier to comprehend greater complexity in a shorter amount of time. For the same reason, the trend away from command line computer interfaces towards Graphical User Interfaces (GUI) has accelerated. Pictures seem more intuitive. Specifications written as diagrams have a problem, however. They are, at best, semi-formal.

At the same time, interest in formal systems that can be rigorously verified with automated tools has increased. This is especially true for embedded systems, which are 10 to 1000 times more common than their desktop counterparts. One must know with a high degree of certainty that an auto-pilot or steer-by-wire system does not have dark, hidden corners in its behavior.

The two trends seem to be orthogonal. How does one attain rigor from semi-formality? What we have shown in this research is that semi-formal UML models of embedded systems can be mapped to a formal language to gain concrete semantics. Furthermore, by choosing the target language appropriately, analysis tools, such as model checkers can be applied to the graphical design models for verification of important properties and subsequent refinement of the system.

Other research has mapped semi-formal languages to formal languages to provide semantics, but a question that arises is: how can one be sure the mapping is consistent, and in some measure, "correct"? This research has developed a mapping technique that formally maps a metamodel of UML to a metamodel of a formal, target language. This approach insures consistency and provides a means of checking exactly what semantics are generated. The mapping also provides sufficient rigor that automated tools can carry out translations from UML diagrams to target languages. This research has demonstrated mappings to two target languages: VHDL and Promela/SPIN.

Having a formal mapping between metamodels provides extra flexibility to the expert designer. Now the semantics attached to a UML diagram can be modified by altering the mapping (and, consequently, the underlying translation tool). In a sense, the semantics can be tailored to the application.

We have demonstrated the design methodology and mapping techniques are viable for industrial applications through a case study of a design of a smart automotive cruise control. The case study demonstrates how simulation and model checking can be used in concert, and demonstrates how UML semantics can be modified to fit the application.

10.1 Summary of Contributions

There were three major objectives for this research:

1. To enable the use of intuitive, graphical notations for the design of embedded systems while providing a means of precisely specifying the semantics of the written diagrams.

2. To develop a general framework for creating mappings from diagram components to elements of the target formal specification language. The general framework must provide the rigor to ensure consistency between the diagrams and formal specification. In addition, the general framework must enable us to modify the semantics of the diagram.

3. To enable and exercise complementary analysis techniques for the semi-formal diagram via their formal specifications. Specifically, we wanted to investigate how model checking and simulation can be used independently and in an integrated fashion to analyze the specifications.

An overarching objective of this research was to enable practitioners in industry to use formal techniques while not surrendering semi-formal, more intuitive, methods. For this reason, we chose UML as the graphical design language since it has become the *de facto* standard, and chose VHDL and Promela as two formalization languages. Each provides different semantics to UML, and each can itself provide varying semantics.

In order for a practitioner to be able to translate diagrams accurately according to the mapping rules, an automated tool is required. We call the tool Hydra. The structure of Hydra is such that only selected components corresponding to the mapping rules need to be replaced to map to a new target language.

In summary, this dissertation makes several contributions:

- **Overall Framework for Providing Semantics**

  An overarching framework based on mappings between metamodels has been developed, applied, and demonstrated. We have shown that the semantics can be checked for consistency and can be modified as required.

- **Unified Class/Dynamic Model**

  In order to make the mapping more straightforward, the UML class and dynamic metamodels have been merged into a single metamodel. Other approaches often ignore the class-based nature of object-oriented systems and focus on the state model. Class information is central to our approach.

- **Mapping to VHDL**

  We provided a detailed formalization of UML in terms of VHDL. VHDL was chosen because it is already used by a large number of embedded systems practitioners and it is closely related to the hardware used in embedded systems. In addition, a large number of commercial tools, tool environments, libraries, and simulators are available to support VHDL. Finally, VHDL is one of the few formal languages that contains timing constraints.

- **Mapping to Promela**

  We provided a mapping to Promela so that UML model can be either simulated or analyzed with the SPIN model checker. Output from SPIN analysis is useful for either verifying the behavior of the system, or to make refinements to the system, all without writing programming code.

- **Use of Industrially Available Languages**

  As mentioned earlier, an overarching goal of this research is technology transfer to the practitioner. By picking source informal and target formal languages well-known in the industrial community, and illustrating the modeling analysis of an industrial application, we have made significant progress towards this goal.

- **A Development Methodology Based on Our Framework**

Chapter 4 describes a design methodology integrating UML and our framework for formal semantics. Given the proper formal target formal language, the methodology leads to behavioral verification earlier in the design process. Armed early with information about the system behavior, the designer can refine to design cheaper and faster than if refinements are left to a later stage. In addition, the option is available through our homomorphic framework to alter the semantics to dynamic models during design. We demonstrated this with an alteration of the semantics of object message passing in the industrial case study.

10.2 Future Research

Several areas of research can be explored based on the results that we have obtained this far. Several complementary investigations can be directly pursued. First, UML formalization can be expanded to other languages. In particular, the language used for the SMV model checker seems to be a good candidate. In many respects, SMV offers better (although different) model checking capabilities than SPIN. This would be an interesting test in another way: SMV provides no simulation. The specification language for the model is strictly for model checking. Both VHDL and Promela are "executable" in the sense of a program. SMV is not. SPIN uses LTL temporal logic, while SMV uses CTL temporal logic (each was explained in Section 2.4.1). Work on the mapping to Promela revealed that there are several choices of semantics for dynamic models. In the absence of fairness in SPIN's analyzer (discussed in Section 9.8.3), examining alternate semantic interpretations for transitions and message passing would be illuminating. It would also be interesting to compare the semantics available in Promela with those available by using another model checking languages, such as SMV. One reason for this is that SMV has fairness already built in. Another reason is that SPIN uses the logic LTL, while SMV uses the logic CTL. Since the logics are incomparable in expressive power [30], various behaviors not expressible in LTL could be explored in CTL.

The GUI capabilities of Hydra could be improved so that a UML class and dynamic diagram can directly generate the target language, and thus formal specifications. This should largely be a matter of putting the proper GUI drawing interface in front of the Hydra translator. This has already been explored to a certain extent in a related project involving Honeywell's DOME editors [68].

Once this is done, the obvious next step is to use trace output from simulations or model checking to graphically highlight visited states, state cycles, deadlock, and other properties, so that the model's execution can be visualized. Perhaps more useful would be the ability to visualize the results of model checking to direct the modification of the UML model. These steps would require modifications to the Hydra translator, and a system on the back end of the formal language analysis tools to capture and use the analysis output.

The process of generating VHDL and especially Promela is not very different from generating a program in a language like Java or C++. If an alternate mapping for a common production programming language could be constructed so that the semantics matched the mapping to the formal target language used for development and analysis, the entire system would consist of a graphical editor whose diagrams could be run directly in simulation, analyzed in a model checker, and finally, the hypothetical system could generate production code from the final diagrams. The key to success for this system would be a careful matching of the semantics obtained for simulation and analysis with the semantics obtained in the final program. This step is essential to ensure the results of the analysis are valid. We must point out, however, that semantics matching is already a problem in conventional design and analysis. How does the analyst know that the Promela model she has hand-coded and verified matches the C-coded program constructed from the same specifications?

Finally, we have been able to form a unified class/dynamic metamodel for a single homomorphic mapping to a target language. There should be no reason to not include other UML diagrams in the unified metamodel. In particular, use case and/or sequence diagrams would provide useful verification information. If detailed sequence diagrams were also part of the mapping, the resulting specification would be constrained by the sequence diagram information so that analysis would be greatly enhanced. In effect, test cases to verify behavior would be assembled with the class and dynamic model. All that would remain would be to formulate appropriate temporal logic expressions for the assembled system.

Bibliography

[1] Bruce Powell Douglass. *Real–Time UML.* Addison–Wesley, 1998.

[2] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*, chapter 1. P T R Prentice Hall, 1994.

[3] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Loresensen. *Object Oriented Modeling and Design.* Prentice Hall, 1991.

[4] Grady Booch. *Object-Oriented Analysis and Design With Applications.* Addison–Wesley, 1994.

[5] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-Time Object Oriented Modeling.* John Wiley & Sons, 1994.

[6] Rational Software Corporation, Santa Clara, CA 95051-0951. *UML Notation Guide*, 1.0 edition, January 1997.

[7] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addision-Wesley, 1999.

[8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addision-Wesley, 1999.

[9] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–22, sep 1990.

[10] Ivar Jacobson. Is the Object Technology Software's Industrial Platform? *IEEE Software*, 10(1):24–42, 1993.

[11] Peter J Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, Inc., 1996.

[12] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems.* McGraw-Hill, Inc., New York, 1993.

[13] D. Perry. *VHDL.* McGraw-Hill, New York, 1991.

[14] IEEE. *IEEE Standard VHDL Language Reference Manual*, June 1994. ANSI/IEEE Std 1076-1993.

[15] Gerald J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), may 1997.

[16] Gerald J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[17] Enoch Y. Wang and Betty H. C. Cheng. A Rigorous Object-Oriented Design Process. *Proc. of International Conference on Software Process, Naperville, IL*, 1998.

[18] Robert H. Bourdeau and Betty H. C. Cheng. A formal semantics of object models. *IEEE Trans. on Software Engineering*, 21(10):799–821, October 1995.

[19] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proc. of IEEE International Conference on Software Engineering (ICSE97)*, Boston, MA, May 1997.

[20] Enoch Y. Wang and Betty H. C. Cheng. Formalizing and integrating the functional model into object-oriented design. In *Proc. of International Conference on Software Engineering and Knowledge Engineering*, June 1998. Nominated for Best Paper.

[21] J-M. Bruel and R. B. France. Transforming UML models to formal specifications. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.

[22] Malcolm Shroff and Robert B France. Towards a Formalization of UML Class Structures in Z. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE Comput. Soc, Los Alamitos, CA, USA, aug 1997.

[23] A. S. Evans, R. B. France, K. C. Lano, and B. Rumpe. The UML as a formal modelling notation. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.

[24] Rational Software, Microsoft, Hewlett Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, PTech Taskon, Reich Technologies, Softem. *UML Semantics*, ad/97-08-04 edition, September 1997. available on http://www.rational.com.

[25] D. Harel. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[26] Rational Software, Microsoft, Hewlett Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, PTech Taskon, Reich Technologies, Softem. *UML Semantics*, ad/97-08-04 edition, September 1997. section 2, page 6–9, available on http://www.rational.com.

[27] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, (8):666–677, August 1978.

[28] 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.

[29] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *cacm*, 18(8):453–457, 1975.

[30] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[31] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[32] Martin D. Fraser, Kulderp Kumar, and Vijay K. Vaishnavi. Strategies for incorporating formal specifications. *Communications of the ACM*, (10):74–86, October 1994.

[33] Enoch Y. Wang and Betty H. C. Cheng. A rigorous object-oriented design process. In *Proc. of International Conference on Software Process*, Naperville, Illinos, June 1998.

[34] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, 1994.

[35] Yile Enoch Wang. *Integrating Informal and Formal Approaches to Object-Oriented Analysis and Design.* PhD thesis, Michigan State University, mar 1998.

[36] Bruce Powell Douglass. *Real–Time UML*, chapter 2. Addison–Wesley, 1998.

[37] John R. Ellis. *Objectifying Real-Time Systems.* SIGS books, New York, 1994.

[38] Paul T Ward and Stephen J Mellor. *Structured Development for Real-time Systems*, volume 2. Yourdon Press, 1985.

[39] Derek J Hatley and Imtiaz A Pirbhai. *Strategies for Real-time System Specification.* Dorset House Publishing, 353 West 12th Street New York, NY, 1987.

[40] Paul T Ward and Stephen J Mellor. *Structured Development for Real-Time Systems*, volume 2, chapter 2. Yourdon Press, 1985.

[41] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*, chapter 2, page 53. Addision-Wesley, 1999.

[42] Tony Torre. Private Communication, 1998. Private communcation of industrial methods while working on projects for Eaton Corp.

[43] Tom Demarco. *Structured Analysis and Sytem Specification.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[44] George A Miller. The Magical Number Seven, Plus of Minus Two. *The Psychological Review*, 63(2), mar 1956.

[45] Bruce Powel Douglass. *Real–Time UML*, chapter 1. Addison–Wesley, 1998.

[46] Roger S. Pressman. *Software Engineering A Practitioneer's Approach.* McGraw-Hill, Inc., 3rd edition, 1992.

[47] W. S. Davis. *Tools and Techniques for Structured Systems Analysis and Design.* Addison–Wesley, 1983.

[48] John E. Hopcraft and Jeffery D. Ullman. *Introduction to Automata Theory, and Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[49] David Harel and Amnon Naamad. The StateMate Semantics of Statecharts. *ACM Trans. Soft. Eng. Method.*, 1996.

[50] Rational Software, Microsoft, Hewlett Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, PTech Taskon, Reich Technologies, Softem. *UML Notation Guide*, ad/97-08-04 edition, September 1997. section 9.3.3, page 107, available on http://www.rational.com.

[51] Daniel D. Gajski, Frank Vahid, and Sanjiv Narayan. Speccharts: A VHDL Front-End for Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, June 1995.

[52] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[53] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop, Ilkley*, electronic Workshops in Computing. Springer-Verlag, 1998.

[54] Kevin C. Lano. Z++, an Object Oriented Extension to Z. In John E Nicholls, editor, *Z User Workshop*, pages 151–172, 1991.

[55] Rational Software, Microsoft, Hewlett Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, PTech Taskon, Reich Technologies, Softem. *Object Constraint Language Specification*, ad/97-08-08 edition, September 1997. available on http://www.rational.com.

[56] Luis Mandel and Maria Victoria Cengarle. On the Expressive Power of the Object Constraint Language OCL. Technical report, Forschungsinstitut für Angewandte Software Technologie, feb 1999.

[57] E. F. Codd. A relational model of data for large shared data banks. *acm*, 13(6):377–387, 1970.

[58] P. C. Kanellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B Formal Models and Semantics*, volume B, chapter 17, pages 1075–1156. The MIT Press, 1990.

[59] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development the FUSION Method.* Prentice Hall, 1994.

[60] Ruth Malan, Reed Letsinger, and Derek Coleman. *Object-Oriented Development at Work: FUSION in the real world.* Prentice Hall, 1996.

[61] Ruth Malan, Reed Letsinger, and Derek Coleman. *Object-Oriented Development at Work: FUSION in the real world*, chapter 12, pages 343–354. Prentice Hall, 1996.

[62] Ralf Jungclaus, Gunter Saake, Thorsten Hartman, and Chirstina Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[63] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[64] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999.* Kluwer, 1999.

[65] Samuel P. Harbison and Guy L. Steele Jr. *"C" A Reference Manual.* Prentice-Hall, 1987.

[66] Leslie Lamport. *LaTeX User's Guide and Reference Manual.* Addison–Wesley Publishing Company, 1994.

[67] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. Technical Report DEC SRC 15, Digital Systems Research Center, December 1986.

[68] Honeywell Corproration. *DOME Guide*, version 5.2.1 edition, 1999. DOME is available on the WEB at www.htc.honeywell.com/dome.

[69] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reily & Associates, 1995.

APPENDICES

Appendix A

VHDL for Furnace Example

This appendix gives the entire text of the VHDL generated by Hydra with a VHDL LSCL for the furnace example found in Section 5.3. This code executes correctly as written, but shows the incorrect behavior of the system that allows heating and cooling to be active simultaneously. This use case is described in more detail in Section 5.3. The driver object that instantiates the rest of the model and provides events to the model is found at the end of the VHDL listing below. Since the driver object references parts of the rest of the model, VHDL rules require it to be last in the source file.

```
- This VHDL was generated by HYDRA. The comments were inserted by hand.
use std.textio.all;
package easyio is
    procedure say (constant str : in string);
end package easyio;

package body easyio is
    procedure say (constant str : in string) is
        variable myline : line;
    begin
        write(myline, now);
        write(myline, " : ");
        write(myline, str);
        writeline(output, myline);
    end;
end package body easyio;
------------------------------

package common is
- This is an enumeration of the signal 'demand'
    type demandtype is (none,off,heat,cool,fanon,auto,enough);
```

```
- Similarly for signal 'mode'
    type modetype is (none,off,heat,cool,fanon,auto);
end package common;
- This is the package associated with the object
package pk_top is
- All states in the object are enumerated here.
    type st is (none, CCSTATE1, seasonswitch, SSHeat, SSOff, SSCool,
    fanswitch, FSAuto, FSOn, furnrelay, FROff, FROn, AC_relay, ACOff,
    ACOn, fanrelay, RAuto, ROn);
    type st_a is array (natural range <>) of st;
    function resolve_st (v:  in st_a) return st;
    subtype rs_st is resolve_st st;
end package pk_top;
- This is the package containing the resolution function
package body pk_top is
    function resolve_st
        (v :  in st_a) return st is
        variable i :  natural;
    begin
        for i in v'range loop
            if v(i) /= st'(none) then
                return v(i);
            end if;
        end loop;
        return st'(none);
    end function;
end package body;
-----------------------------
- This is the seasonswitch composite state
- The 'use' statements pick up declarations required
- by the folowing code
use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity cs_seasonswitch is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        instate:  out st;
        ins_fanswitch:  in st;
        ins_furnrelay:  in st;
        ins_AC_relay:  in st;
        ins_fanrelay:  in st);
end entity;

architecture abstract of cs_seasonswitch is
begin
    instate <= state when state/=st'(none);

- A typical state. The state hangs until state=SSHeat, then is falls thru
- to the event wait, below.
    s_SSHeat:  process
    begin
        wait until state=st'(SSHeat);
        say("In state SSHeat");
- Waiting for an event, mode=cool or mode=off
        loop
            wait until mode=modetype'(cool)
                or mode=modetype'(off);
            if mode=modetype'(cool) then
- A typical transition. Write a new value to state; SSCool in this case
                state <= st'(SSCool), null after 1 fs;
                exit;
            elsif mode=modetype'(off) then
                state <= st'(SSOff), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_SSOff:  process
    begin
        wait until state=st'(SSOff);
        say("In state SSOff");
        loop
            wait until mode=modetype'(heat)
                or mode=modetype'(cool);
            if mode=modetype'(heat) then
                state <= st'(SSHeat), null after 1 fs;
                exit;
            elsif mode=modetype'(cool) then
                state <= st'(SSCool), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_SSCool:  process
    begin
        wait until state=st'(SSCool);
        say("In state SSCool");
        loop
            wait until mode=modetype'(off)
                or mode=modetype'(heat);
            if mode=modetype'(off) then
                state <= st'(SSOff), null after 1 fs;
                exit;
            elsif mode=modetype'(heat) then
                state <= st'(SSHeat), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(seasonswitch);
        state <= st'(SSOff), null after 1 fs;
    end process;
end abstract;
————————————————————
- This is the fanswitch composite state
use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity cs_fanswitch is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        ins_seasonswitch:  in st;
        instate:  out st;
        ins_furnrelay:  in st;
        ins_AC_relay:  in st;
        ins_fanrelay:  in st);
end entity;

architecture abstract of cs_fanswitch is
begin
    instate <= state when state/=st'(none);
- state FSAuto. All states start with s_NAME
    s_FSAuto:  process
    begin
        wait until state=st'(FSAuto);
        say("In state FSAuto");
        loop
            wait until mode=modetype'(fanon);
            if mode=modetype'(fanon) then
                state <= st'(FSOn), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_FSOn:  process
    begin
        wait until state=st'(FSOn);
        say("In state FSOn");
        loop
            wait until mode=modetype'(auto);
            if mode=modetype'(auto) then
                state <= st'(FSAuto), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(fanswitch);
        state <= st'(FSAuto), null after 1 fs;
    end process;
end abstract;
----------------------------------

use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity cs_furnrelay is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        ins_seasonswitch:  in st;
        ins_fanswitch:  in st;
        instate:  out st;
        ins_AC_relay:  in st;
        ins_fanrelay:  in st);
end entity;

architecture abstract of cs_furnrelay is
begin
    instate <= state when state/=st'(none);

    s_FROff:  process
    begin
        wait until state=st'(FROff);
        say("In state FROff");
        loop
            wait until demand=demandtype'(heat);
            if demand=demandtype'(heat) and ins_seasonswitch=st'(SSHeat)
            then
                state <= st'(FROn), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_FROn:  process
    begin
        wait until state=st'(FROn);
        say("In state FROn");
        loop
```

```
            wait until demand=demandtype'(enough);
            if demand=demandtype'(enough) then
                state <= st'(FROff), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(furnrelay);
        state <= st'(FROff), null after 1 fs;
    end process;
end abstract;
--------------------

use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity cs_AC_relay is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        ins_seasonswitch:  in st;
        ins_fanswitch:  in st;
        ins_furnrelay:  in st;
        instate:  out st;
        ins_fanrelay:  in st);
end entity;

architecture abstract of cs_AC_relay is
begin
    instate <= state when state/=st'(none);

    s_ACOff:  process
    begin
        wait until state=st'(ACOff);
        say("In state ACOff");
        loop
            wait until demand=demandtype'(cool);
            if demand=demandtype'(cool) and ins_seasonswitch=st'(SSCool)
            then
                state <= st'(ACOn), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_ACOn:  process
    begin
        wait until state=st'(ACOn);
        say("In state ACOn");
        loop
            wait until demand=demandtype'(enough);
            if demand=demandtype'(enough) then
                state <= st'(ACOff), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(AC_relay);
        state <= st'(ACOff), null after 1 fs;
    end process;
end abstract;
```

```
––––––––––––––––––––––

use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity cs_fanrelay is
    port(state:  inout rs_st;
        mode:  inout modetype;
        demand:  inout demandtype;
        ins_seasonswitch:  in st;
        ins_fanswitch:  in st;
        ins_furnrelay:  in st;
        ins_AC_relay:  in st;
        instate:  out st);
end entity;

architecture abstract of cs_fanrelay is
begin
    instate <= state when state/=st'(none);

    s_RAuto:  process
    begin
        wait until state=st'(RAuto);
        say("In state RAuto");
        loop
            wait until ins_fanswitch=st'(FSOn);
            if ins_fanswitch=st'(FSOn) and (ins_seasonswitch=st'(SSCool)
            or ins_seasonswitch=st'(SSHeat)) then
                state <= st'(ROn), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_ROn:  process
    begin
        wait until state=st'(ROn);
        say("In state ROn");
        loop
            wait until ins_seasonswitch=st'(SSOff);
            if ins_seasonswitch=st'(SSOff) then
                state <= st'(RAuto), null after 1 fs;
                exit;
            end if;
        end loop;
    end process;

    s_init:  process
    begin
        wait until state=st'(fanrelay);
        state <= st'(RAuto), null after 1 fs;
    end process;
end abstract;
––––––––––––––––––––––
- This is the top level object container for the control module.
- It is last because it refers to all other components in the system
- and VHDL required pre-declaring names.
use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity obj_top is
    port(mode:  inout modetype;
        demand:  inout demandtype);
end entity;

architecture abstract of obj_top is
```

```
    signal state :  rs_st;
- signals from CCSTATE1
    signal ss_seasonswitch:  rs_st;
    signal ss_fanswitch:  rs_st;
    signal ss_furnrelay:  rs_st;
    signal ss_AC_relay:  rs_st;
    signal ss_fanrelay:  rs_st;
    signal ins_seasonswitch:  st;
    signal ins_fanswitch:  st;
    signal ins_furnrelay:  st;
    signal ins_AC_relay:  st;
    signal ins_fanrelay:  st;
- signals from
begin
- These are instantiation statements generated by Hydra to start
- the concurrent states.
- CCState output
    l1:  entity cs_seasonswitch(abstract)
- This is a typical mapping of ports into a concurrent component
        port map(state=>ss_seasonswitch, mode=>mode, demand=>demand,
        instate=>ins_seasonswitch, ins_fanswitch=>ins_fanswitch,
        ins_furnrelay=>ins_furnrelay, ins_AC_relay=>ins_AC_relay,
        ins_fanrelay=>ins_fanrelay);
    ss_seasonswitch <= st'(seasonswitch), null after 1 fs when
    state=st'(seasonswitch) or state=st'(fanswitch) or
    state=st'(furnrelay) or state=st'(AC_relay) or state=st'(fanrelay);
    l2:  entity cs_fanswitch(abstract)
        port map(state=>ss_fanswitch, mode=>mode, demand=>demand,
        ins_seasonswitch=>ins_seasonswitch, instate=>ins_fanswitch,
        ins_furnrelay=>ins_furnrelay, ins_AC_relay=>ins_AC_relay,
        ins_fanrelay=>ins_fanrelay);
    ss_fanswitch <= st'(fanswitch), null after 1 fs when
    state=st'(seasonswitch) or state=st'(fanswitch) or
    state=st'(furnrelay) or state=st'(AC_relay) or state=st'(fanrelay);
    l3:  entity cs_furnrelay(abstract)
        port map(state=>ss_furnrelay, mode=>mode, demand=>demand,
        ins_seasonswitch=>ins_seasonswitch,
        ins_fanswitch=>ins_fanswitch, instate=>ins_furnrelay,
        ins_AC_relay=>ins_AC_relay, ins_fanrelay=>ins_fanrelay);
    ss_furnrelay <= st'(furnrelay), null after 1 fs when
    state=st'(seasonswitch) or state=st'(fanswitch) or
    state=st'(furnrelay) or state=st'(AC_relay) or state=st'(fanrelay);
    l4:  entity cs_AC_relay(abstract)
        port map(state=>ss_AC_relay, mode=>mode, demand=>demand,
        ins_seasonswitch=>ins_seasonswitch,
        ins_fanswitch=>ins_fanswitch, ins_furnrelay=>ins_furnrelay,
        instate=>ins_AC_relay, ins_fanrelay=>ins_fanrelay);
    ss_AC_relay <= st'(AC_relay), null after 1 fs when
    state=st'(seasonswitch) or state=st'(fanswitch) or
    state=st'(furnrelay) or state=st'(AC_relay) or state=st'(fanrelay);
    l5:  entity cs_fanrelay(abstract)
        port map(state=>ss_fanrelay, mode=>mode, demand=>demand,
        ins_seasonswitch=>ins_seasonswitch,
        ins_fanswitch=>ins_fanswitch, ins_furnrelay=>ins_furnrelay,
        ins_AC_relay=>ins_AC_relay, instate=>ins_fanrelay);
    ss_fanrelay <= st'(fanrelay), null after 1 fs when
    state=st'(seasonswitch) or state=st'(fanswitch) or
    state=st'(furnrelay) or state=st'(AC_relay) or state=st'(fanrelay);

    s_init:  process
    begin
        state <= st'(seasonswitch), null after 1 fs;
        wait;
    end process;
end abstract;
use std.textio.all;
use work.easyio.all;
use work.common.all;
use work.pk_top.all;
entity context is
end entity;

architecture abstract of context is
    signal demand :  demandtype;
    signal mode :  modetype;

begin
– This driver is hand coded and included into Hydra at translation time.
– It exercises the scenario of demanding heat, then cool, which causes
– the A/C to turn on, incorrectly.
– Driver system in heat and cool at same time
– mode to heat
– demand heat
– modem to cool
– demand cool

    controller1:  entity obj_top(abstract)
        port map(demand=>demand, mode=>mode);

    driver1:  process
    begin
        wait for 3 ns;
        say("setting mode to Heat");
        mode <= modetype'(heat), modetype'(none) after 1 ns;
        wait for 5 ns;
        say("Demanding heat");
        demand <= demandtype'(heat), demandtype'(none) after 1 fs;
        wait for 5 ns;
        say("setting mode to Cool");
        mode <= modetype'(cool), modetype'(none) after 1 fs;
wait for 5 ns;
say("Demanding cool");
demand <= demandtype'(cool), demandtype'(none) after 1 fs;
        wait;
    end process;
end abstract;
```

Appendix B

Hydra Language Input for the Smart Cruise Case Study

This appendix contains the entire Hydra language translated output from the diagrams contained in Chapter 9, Figures 9.8, 9.12, 9.13, 9.15, and 9.17. The Promela generated from this specification is contained in Appendix C.

The **SYSTEM** class driver file called "scdriver" in line 2 of the Hydra language listing is shown below. The diagram below corresponds directly with the state machine in Figure 9.15.

SYSTEM class driver file Promela specifications

```
/*
#define p control_q??[target]
#define q radar_q??[ackcontrol]
*/
#define p control_q??[brakes]
#define q control[controlpid]@idle

#define min(x,y) (x<y->x:y)
#define max(x,y) (x>y->x:y)

int controlpid;
int radarpid;
init
{

atomic {radarpid = run radar();
controlpid = run control(); run car();}
timeout -> control_q!set;
printf("model started");
/* if
:: atomic {radar_V.x > 2000 && radar_V.x < 3700 ->
control_q!brakes; printf("sent brakes")}
fi */
}
```

Hydra language input for the SmartCruise model

```
Model Smartcruise {
DriverFile scdriver;

Object control {
# Radar Signals
   Signal target;
   Signal dist(int);
   Signal lost;
   Signal ackradar;
   Signal brakes;

# Car signals
   Signal carspeed(int);
   Signal ackcar;

# User signals
   Signal set;

   InstanceVar int x1; /* last x read */
   InstanceVar int x2; /* x just read, copied to x1 */
   InstanceVar int tinc := 1; /* time increment.  1 second */
   InstanceVar int vc; /* v of car */
   InstanceVar int vt; /* v of target */
   InstanceVar int v; /* closing v.  + = closing */
   InstanceVar int tmin := 2; /* minimum spacing in secs at vt */
   InstanceVar int z1; /* closest allowable distance.  10% under tmin */
   InstanceVar int z2; /* calculated tmin dist */
   InstanceVar int xhit; /* distance to coast to hit targ */
   InstanceVar int xcoast; /* disatnce to close to targ */
   InstanceVar int a := 15; /* deceleration speed of car ft/sec*sec */
   InstanceVar int setspd; /* speed car set to maintain */
   InstanceVar bool closing; /* true while at throttle, closing */

   Initial idle;
   State idle {
      Transition "set^car.getspeed" to gotit;
   }
   State gotit { /* wait for car speed */
      Transition "carspeed(setspd)^car.setspeed(setspd)" to setit0;
   }
   State setit0 { /* wait for radar on and set car speed */
      Transition "ackcar ^ radar.on" to setit1;
   }
   State setit1 {
      Transition "ackradar" to maintain;
   }
   State maintain { /* wait for target acquisition */
      Transition "target^radar.ackcontrol" to getx1;
      Transition "brakes^radar.off^car.unset" to caroff;
   }
   State ackcar0 {
```

```
            Transition "ackcar^radar.ackcontrol" to maintain;
         }
         State getx1 { /* wait for radar x dist or lost sig */
            Transition "dist(x1)",
      "/xcoast := 0;",
      "closing := 0",
      "^ radar.ackcontrol" to getspd;
            Transition "lost ^ car.setspeed(setspd)" to ackcar0; /* lost tracking */
            Transition "brakes^radar.off^car.unset" to caroff;
         }
      # Begin control sequence

         State getspd { /* get car speed */
            Transition "dist(x2)^car.getspeed" to calc;
            Transition "lost^car.setspeed(setspd)" to ackcar0;
            Transition "brakes^radar.off^car.unset" to caroff;
         }
         State calc { /* perform all the calculations */
            Transition "carspeed(vc)",
      "/ v:= (x1-x2)/tinc;",
      "vt := vc - v;",
      "z2 := vt*tmin;",
      "z1 := z2 - z2/10;",
      "x1 := x2;",
      "print('v=%d, x1=%d, z1=%d z2=%d',",
      " v,x1,z1,z2)" to getxc;
            Transition "brakes^radar.off^car.unset" to caroff;
         }
         State getxc {
            Transition "[ closing]",
      "/xhit := (v*v)/(2*a);",
      "xcoast := xhit+z2 + tinc*v;",
      "print('xcoast=%d, xhit=%d', xcoast, xhit)" to alarm;
            Transition "[closing]" to alarm;
         }
         State alarm { /* Make sure we're not too close */
            Transition "[x1 >= z1]" to warn;
            Transition "[x1 < z1]", /* This is real bad */
            "/print('WARNING: TOO CLOSE');",
            "print('x1=%d, z1=%d', x1, z1)",
      "^car.unset" to alloff;
         }
         State warn { /* check to see if we are going to hit him */
            Transition "[xhit <= x1]" to close; /* we wont hit him...  */
            Transition "[xhit > x1]" to sendwarn; /* looks like we will hit him */
         }
         State sendwarn {
            Transition "/print('WARNING: GOING TO HIT TARGET')" to close;
         }
         State close {
            Transition "[closing]^radar.ackcontrol" to getspd;
            Transition "[ closing]" to waiting;
         }
         State waiting {
            Transition "[x1 > xcoast]^radar.ackcontrol" to getspd;
            Transition "[x1 <= xcoast]/closing := 1 ^ car.setspeed(vt)" to ac;
         }
         State ac {
            Transition "ackcar^radar.ackcontrol" to getspd;
            Transition "brakes^radar.off^car.unset" to caroff;
         }
         State alloff {
            Transition "ackcar ^ radar.off" to idle;
            Transition "brakes" to alloff;
         }
         State caroff {
            Transition "ackcar" to idle;
            Transition "dist(x1)" to caroff;
      Transition "lost" to caroff;
      Transition "target" to caroff;
      Transition "brakes" to caroff;
      Transition "carspeed(vc)" to caroff;
   }
}

Object radar {
   Signal carv(int);
   Signal ackcontrol;
   Signal on;
   Signal off;
   Signal brakes;

   /* IMPORTANT: Initialize vt to target speed, x to target distance */
   /* initial target speed */
   InstanceVar int vt := 900;
   InstanceVar int x := 4500;
   InstanceVar int vc;
   InstanceVar int v;
   InstanceVar bool tmode := 0;
   InstanceVar bool brks := 0;

   Initial r_off;

   State r_off {
      Transition "on ^ control.ackradar" to r0;
      Transition "off" to r_off;
   }
   State r0 {
      Transition "^car.getv" to r1;
   }
   State r1 {
      Transition "carv(vc)/v := vc-vt; x := x-v;",
"print('x is %d', x)" to r3;
   }
   State r3 {
      Transition "[ tmode & x <= 4000]",
"/tmode := 1; print('Firing target')",
"^ control.target" to r4;
      Transition "[ tmode & x > 4000]" to r0;
      Transition "[tmode & x > 4000] ^ control.lost" to r_ack;
      Transition "[tmode & x <= 4000]",
"/print('Sending x=%d to control', x)",
"^ control.dist(x)" to r_ack;
      /* The next statement non-deterministially applies brakes */
      /* Transition "[tmode & x <= 4000]^control.brakes" to r_ack; */

   }
   State r4 {
      Transition "ackcontrol ^ control.dist(x)" to r_ack;
      Transition "off/tmode := 0" to r_off;
   }
   State r_ack {
      Transition "ackcontrol" to r0;
      Transition "off/tmode := 0" to r_off;
   }
}

Object car {
   /* initial car speed */
   InstanceVar int setv := 1100;
   InstanceVar int realv := 1100;

   Signal setspeed(int);
   Signal getspeed(int);
   Signal getv;
   Signal unset;
```

```

     Initial updatex;

     ConcurrentState cc2 {
        CompositeState updatex {
  Initial car1;
  State car1 {
  /* Transition "getv^radar.carv(realv)" to car1; */
  Transition "getv[realv < setv]",
  "/realv := realv + 15;",
  "realv := min(realv, setv);",
  "print('realv=%d setv=%d', realv,setv)",
  "^radar.carv(realv)"
  to car1;
  Transition "getv[realv > setv]",
  "/realv := realv - 15;",
  "realv := max(realv, setv);",
  "print('realv=%d setv=%d', realv,setv)",
  "^radar.carv(realv)"
  to car1;
  Transition "getv[realv == setv]",
  "/print('realv=%d setv=%d', realv,setv)",
  "^radar.carv(realv)"
  to car1;
  }
        }

        CompositeState updatespd {
  Initial car3;
  State car3 {
  Transition "setspeed(setv)",
  "/print('car speed set to %d',setv)",
  "^control.ackcar" to car3;
  }
        }

        CompositeState dogetspd {
  Initial car4;
  State car4 {
  Transition "getspeed^control.carspeed(realv)" to car4;
  }
        }

        CompositeState dounset {
  Initial car5;
  State car5 {
  Transition "unset ^ control.ackcar" to car5;
  }
        }
     }
  }

  }


```

Appendix C

Promela Specifications for Smart Cruise

This appendix contains the entire final Promela specification generated by Hydra from the Hydra language input shown in Appendix B for the Smart Cruise case study. How various UML structures become Promela statements seen here is explained in Chapter 6. The design and testing of the model this specification represents is contained in Chapter 9. The **SYSTEM** class behavior, and hence the contents of the driver file, are seen in lines 49–73. The commented out lines 69–72 contain an experiment to provide al alternate means of generating a *brakes* signal.

The final contents of the driver file `scdriver` are shown below, followed by the Promela for the entire model.

scdriver file contents

```
/*
#define p control_q??[target]
#define q radar_q??[ackcontrol]
*/
#define p control_q??[brakes]
#define q control[controlpid]@idle

#define min(x,y) (x<y->x:y)
#define max(x,y) (x>y->x:y)

int controlpid;
int radarpid;
init
{

atomic {radarpid = run radar();
controlpid = run control(); run car();}
timeout -> control_q!set;
printf("model started\n");
/* if
:: atomic {radar_V.x > 2000 && radar_V.x < 3700 ->
control_q!brakes; printf("sent brakes\n")}
fi  */
}
```

Promela specifications for the Smart Cruise model

```
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};
mtype={lost, ackcar, carspeed, target, set, carv, unset, none,
        ackcontrol, getspeed, dist, getv, brakes, ackradar, off, on,
        setspeed};
typedef control_T {
        int a;
        int xcoast;
        int setspd;
        int v;
        int tmin;
        int x1;
        int vc;
        int x2;
        int vt;
        int z1;
        int z2;
        int xhit;
        int tinc;
```

```
                        bool closing;
                        }
                control_T control_V;
                chan control_q=[5] of {mtype};
                chan dist_p1=[5] of {int};
                chan carspeed_p1=[5] of {int};
                typedef radar_T {
                        int x;
                        bool tmode;
                        int vc;
                        bool brks;
                        int vt;
                        int v;
                        }
                radar_T radar_V;
                chan radar_q=[5] of {mtype};
                chan carv_p1=[5] of {int};
                typedef car_T {
                        int realv;
                        int setv;
                        }
                car_T car_V;
                chan car_q=[5] of {mtype};
                chan getspeed_p1=[5] of {int};
                chan setspeed_p1=[5] of {int};
                chan t=[1] of {mtype};
                mtype={free};

                /* User specified driver file */

                #define p control_q??[target]
                #define q radar_q??[ackcontrol]
                /*
                #define p control_q??[brakes]
                #define q control[controlpid]idle
                */
                #define min(x,y) (x<y->x:y)
                #define max(x,y) (x>y->x:y)

                int controlpid;
                int radarpid;
                init
                {

                atomic {radarpid = run radar();
                controlpid = run control(); run car();}
                timeout -> control_q!set;
                printf("model started");
                /* if
                :: atomic {radar_V.x > 2000 && radar_V.x < 3700 ->
                control_q!brakes; printf("sent brakes")}
                fi */
                }




                proctype control()
                {atomic{
                mtype m;
                int dummy;
                        control_V.a = 15;
                        control_V.tmin = 2;
                        control_V.tinc = 1;
                /* Init state */
                        goto idle;
                /* State idle */
idle:                 skip;
```

```
                       printf("in state control.idle");
in_idle:
                       if
                       :: control_q?set -> car_q!getspeed; goto gotit
                       fi;
              /* State gotit */
gotit:              skip;
                       printf("in state control.gotit");
in_gotit:
                       if
                       :: control_q?carspeed -> carspeed_p1?control_V.setspd ->
                          setit0
                       fi;
              /* State setit0 */
setit0:             skip;
                       printf("in state control.setit0");
in_setit0:
                       if
                       :: control_q?ackcar -> radar_q!on; goto setit1
                       fi;
              /* State setit1 */
setit1:             skip;
                       printf("in state control.setit1");
in_setit1:
                       if
                       :: control_q?ackradar -> goto maintain
                       fi;
              /* State maintain */
maintain:           skip;
                       printf("in state control.maintain");
in_maintain:
                       if
                       :: control_q?brakes -> radar_q!off;car_q!unset; goto caroff
                       :: control_q?target -> radar_q!ackcontrol; goto getx1
                       fi;
              /* State ackcar0 */
ackcar0:            skip;
                       printf("in state control.ackcar0");
in_ackcar0:
                       if
                       :: control_q?ackcar -> radar_q!ackcontrol; goto maintain
                       fi;
              /* State getx1 */
getx1:              skip;
                       printf("in state control.getx1");
in_getx1:
                       if
                       :: control_q?brakes -> radar_q!off;car_q!unset; goto caroff
                       :: control_q?lost -> atomic{setspeed_p1!control_V.setspd;
                          car_q!setspeed}; goto ackcar0
                       :: control_q?dist -> dist_p1?control_V.x1 ->
                          control_V.xcoast = 0; control_V.closing = 0;
                          radar_q!ackcontrol; goto getspd
                       fi;
              /* State getspd */
getspd:             skip;
                       printf("in state control.getspd");
in_getspd:
                       if
                       :: control_q?brakes -> radar_q!off;car_q!unset; goto caroff
                       :: control_q?lost -> atomic{setspeed_p1!control_V.setspd;
                          car_q!setspeed}; goto ackcar0
                       :: control_q?dist -> dist_p1?control_V.x2 -> car_q!getspeed;
                          goto calc
                       fi;
              /* State calc */
calc:               skip;
                       printf("in state control.calc");
```

```
in_calc:
                       if
                       ::  control_q?brakes -> radar_q!off;car_q!unset; goto caroff
                       ::  control_q?carspeed -> carspeed_p1?control_V.vc ->
                          control_V.v = (control_V.x1-control_V.x2)/control_V.tinc;
                          control_V.vt = control_V.vc-control_V.v; control_V.z2 =
                          control_V.vt*control_V.tmin; control_V.z1 =
                          control_V.z2-control_V.z2/10; control_V.x1 = control_V.x2;
                          printf("v=%d, x1=%d, z1=%d z2=%d",control_V.v,
                          control_V.x1, control_V.z1, control_V.z2); goto getxc
                       fi;
              /* State getxc */
getxc:              skip;
                       printf("in state control.getxc");
in_getxc:
                       if
                       ::  1 -> if
                          ::  !control_V.closing -> control_V.xhit =
                             (control_V.v*control_V.v)/(2*control_V.a);
                             control_V.xcoast =
                             control_V.xhit+control_V.z2+control_V.tinc*control_V.v;
                             printf("xcoast=%d, xhit=%d",control_V.xcoast,
                             control_V.xhit); goto alarm
                          ::  control_V.closing -> goto alarm
                          ::  else -> 0
                          fi
                       fi;
              /* State alarm */
alarm:              skip;
                       printf("in state control.alarm");
in_alarm:
                       if
                       ::  1 -> if
                          ::  control_V.x1>=control_V.z1 -> goto warn
                          ::  control_V.x1<control_V.z1 ->
                             printf("WARNING: TOO CLOSE");;
                             printf("x1=%d, z1=%d",control_V.x1, control_V.z1);
                             car_q!unset; goto alloff
                          ::  else -> 0
                          fi
                       fi;
              /* State warn */
warn:               skip;
                       printf("in state control.warn");
in_warn:
                       if
                       ::  1 -> if
                          ::  control_V.xhit<=control_V.x1 -> goto close
                          ::  control_V.xhit>control_V.x1 -> goto sendwarn
                          ::  else -> 0
                          fi
                       fi;
              /* State sendwarn */
sendwarn:           skip;
                       printf("in state control.sendwarn");
in_sendwarn:
                       if
                       ::  1 -> printf("WARNING: GOING TO HIT TARGET");; goto close
                       fi;
              /* State close */
close:              skip;
                       printf("in state control.close");
in_close:
                       if
                       ::  1 -> if
                          ::  control_V.closing -> radar_q!ackcontrol; goto getspd
                          ::  !control_V.closing -> goto waiting
                          ::  else -> 0
                          fi
                       fi;
              /* State waiting */
waiting:            skip;
                       printf("in state control.waiting");
in_waiting:
                       if
                       ::  1 -> if
                          ::  control_V.x1>control_V.xcoast -> radar_q!ackcontrol;
                             goto getspd
                          ::  control_V.x1<=control_V.xcoast -> control_V.closing
                             = 1; atomic{setspeed_p1!control_V.vt; car_q!setspeed};
                             goto ac
                          ::  else -> 0
                          fi
                       fi;
              /* State ac */
ac:                 skip;
                       printf("in state control.ac");
in_ac:
                       if
                       ::  control_q?brakes -> radar_q!off;car_q!unset; goto caroff
                       ::  control_q?ackcar -> radar_q!ackcontrol; goto getspd
                       fi;
              /* State alloff */
alloff:             skip;
                       printf("in state control.alloff");
in_alloff:
                       if
                       ::  control_q?brakes -> goto alloff
                       ::  control_q?ackcar -> radar_q!off; goto idle
                       fi;
              /* State caroff */
caroff:             skip;
                       printf("in state control.caroff");
in_caroff:
                       if
                       ::  control_q?brakes -> goto caroff
                       ::  control_q?lost -> goto caroff
                       ::  control_q?dist -> dist_p1?control_V.x1 -> goto caroff
                       ::  control_q?ackcar -> goto idle
                       ::  control_q?carspeed -> carspeed_p1?control_V.vc -> goto
                          caroff
                       ::  control_q?target -> goto caroff
                       fi;
exit:               skip
              }}


              proctype radar()
              {atomic{
              mtype m;
              int dummy;
                       radar_V.x = 4500;
                       radar_V.vt = 900;
              /* Init state */
                       goto r_off;
              /* State r_off */
r_off:              skip;
                       printf("in state radar.r_off");
in_r_off:
                       if
                       ::  radar_q?off -> goto r_off
                       ::  radar_q?on -> control_q!ackradar; goto r0
                       fi;
              /* State r0 */
r0:                 skip;
                       printf("in state radar.r0");
```

```
in_r0:
                       if
                       ::  1 -> car_q!getv; goto r1
                       fi;
              /* State r1 */
r1:                 skip;
                       printf("in state radar.r1");
in_r1:
                       if
                       ::  radar_q?carv -> carv_p1?radar_V.vc -> radar_V.v =
                          radar_V.vc-radar_V.vt; radar_V.x = radar_V.x-radar_V.v;
                          printf("x is %d",radar_V.x); goto r3
                       fi;
              /* State r3 */
r3:                 skip;
                       printf("in state radar.r3");
in_r3:
                       if
                       ::  1 -> if
                          ::  !radar_V.tmode && radar_V.x<=4000 -> radar_V.tmode =
                             1; printf("Firing target");; control_q!target; goto
                             r4
                          ::  !radar_V.tmode && radar_V.x>4000 -> goto r0
                          ::  radar_V.tmode && radar_V.x>4000 -> control_q!lost;
                             goto r_ack
                          ::  radar_V.tmode && radar_V.x<=4000 ->
                             printf("Sending x=%d to control",radar_V.x);
                             atomic{dist_p1!radar_V.x; control_q!dist}; goto r_ack
                          ::  else -> 0
                          fi
                       fi;
              /* State r4 */
r4:                 skip;
                       printf("in state radar.r4");
in_r4:
                       if
                       ::  radar_q?ackcontrol -> atomic{dist_p1!radar_V.x;
                          control_q!dist}; goto r_ack
                       ::  radar_q?off -> radar_V.tmode = 0; goto r_off
                       fi;
              /* State r_ack */
r_ack:              skip;
                       printf("in state radar.r_ack");
in_r_ack:
                       if
                       ::  radar_q?ackcontrol -> goto r0
                       ::  radar_q?off -> radar_V.tmode = 0; goto r_off
                       fi;
exit:               skip
              }}


              proctype car()
              {atomic{
              mtype updatex_code, updatespd_code, dogetspd_code, dounset_code;
              int updatex_pid, updatespd_pid, dogetspd_pid, dounset_pid;
              mtype m;
              int dummy;
                       car_V.realv = 1100;
                       car_V.setv = 1100;
              /* Init state */
                       goto to_updatex;
to_updatex:
to_updatespd:
to_dogetspd:
to_dounset:
                       atomic {
                       updatex_pid = run updatex(none);
```

```
                  updatespd_pid = run updatespd(none);
                  dogetspd_pid = run dogetspd(none);
                  dounset_pid = run dounset(none);
                  }
                  wait??eval(updatex_pid),updatex_code;
                  wait??eval(updatespd_pid),updatespd_code;
                  wait??eval(dogetspd_pid),dogetspd_code;
                  wait??eval(dounset_pid),dounset_code;
                  assert(0);
exit:         skip
}}

proctype updatex(mtype state)
{atomic{
mtype m;
int dummy;
/* Init state */
                  goto car1;
/* State car1 */
car1:         skip;
                  printf("in state car.car1");
in_car1:

                  if
                  :: car_q?getv -> if
                     :: car_V.realv<car_V.setv -> car_V.realv =
                        car_V.realv+15; car_V.realv = min(car_V.realv,
                        car_V.setv); printf("realv=%d setv=%d",car_V.realv,
                        car_V.setv); atomic{carv_p1!car_V.realv; radar_q!carv};
                        goto car1
                     :: car_V.realv>car_V.setv -> car_V.realv =
                        car_V.realv-15; car_V.realv = max(car_V.realv,
                        car_V.setv); printf("realv=%d setv=%d",car_V.realv,
                        car_V.setv); atomic{carv_p1!car_V.realv; radar_q!carv};
                        goto car1
                     :: car_V.realv==car_V.setv ->
                        printf("realv=%d setv=%d",car_V.realv, car_V.setv);
                        atomic{carv_p1!car_V.realv; radar_q!carv}; goto car1
                     :: else -> goto in_car1
                     fi
                  fi;
exit:         skip
}}


proctype updatespd(mtype state)
{atomic{
mtype m;
int dummy;
/* Init state */
                  goto car3;
/* State car3 */
car3:         skip;
                  printf("in state car.car3");
in_car3:

                  if
                  :: car_q?setspeed -> setspeed_p1?car_V.setv ->
                     printf("car speed set to %d",car_V.setv);
                     control_q!ackcar; goto car3
                  fi;
exit:         skip
}}


proctype dogetspd(mtype state)
{atomic{
mtype m;
int dummy;
/* Init state */
```

```
                        goto car4;
        /* State car4 */
car4:             skip;
                        printf("in state car.car4");
in_car4:
                        if
                        ::  car_q?getspeed -> atomic{carspeed_p1!car_V.realv;
                            control_q!carspeed}; goto car4
                        fi;
exit:             skip
        }}


        proctype dounset(mtype state)
        {atomic{
        mtype m;
        int dummy;
        /* Init state */
                        goto car5;
        /* State car5 */
car5:             skip;
                        printf("in state car.car5");
in_car5:
                        if
                        ::  car_q?unset -> control_q!ackcar; goto car5
                        fi;
exit:             skip
        }}




        /* This is the universal event dispatcher routine */
        proctype event(mtype msg)
        {
        mtype type;
        int pid;

        atomic {
        do
        ::  evq??[eval(msg),pid] ->
        evq??eval(msg),pid;
        evt!msg,pid;
        do
        ::  if
        ::  evq??[type,eval(pid)] -> evq??type,eval(pid)
        ::  else break;
        fi
        od
        ::  else -> break
        od}
exit:   skip
        }
```

Appendix D

Yacc Grammar for the Hydra Language

Yacc [69], standing for "Yet another compiler compiler", is a tool to help designers write compilers. Yacc takes as input a grammar representing the language to be parsed and outputs routines suitable for inclusion in compiler-like programs. Hydra uses Yacc to parse the Hydra language. This appendix contains the grammar for the Hydra language in Yacc form.

Tokens are strings pre-processed by the lexical analyzer for the parser. The tokens are defined as follows:

- MODEL: The word "Model"
- ID: Any string identifer starting with a letter.
- NUM: A number
- QSTR: A double quoted string including its contents.
- OBJECT: The word "Object"
- ASSOC: The word "Association"

- TYPE: The word "Type"
- CCSTATE: The word "ConcurrentState"
- CSTATE: The word "CompositeState"
- TO: The word "to"
- ENUM: The word "Enum"
- DRIVERFILE: The word "DriverFile"
- STATE: The word "State"
- TRANS: The word "Transition"
- INIT: The word "Initial"
- JOIN: The word "Join"
- HISTORY: The word "History"
- FROM: The word "from"
- INSTVAR: The word "InstanceVar"
- SIGNAL: The word "Signal"

```
%token MODEL,ID,NUM,QSTR,OBJECT,ASSOC,TYPE,CCSTATE,CSTATE,TO,ENUM
%token DRIVERFILE,STATE,TRANS,INIT,FORMAL,JOIN,HISTORY,FROM,INSTVAR
%token SIGNAL

%%
model:  MODEL ID '{' modelbody '}'
        ;
modelbody:      modelstmt
        |       modelbody modelstmt
        ;
modelstmt:      ENUM ID list ';'
        |       DRIVERFILE ID ';'
        |       OBJECT ID '{' objectbody '}'
        ;
objectbody:     objectstmt
        |       objectbody objectstmt
        ;
objectstmt:     signal
        |       cstate
        |       ccstate
        |       init
        |       join
        |       history
        |       state
        |       instvar
        ;
signal: SIGNAL ID ';'
        | SIGNAL ID '(' ID ')' ';'
        ;
association: ASSOC ID ';'
             | ASSOC ID '(' ID ID ')' ';'
             ;
ccstatebody:    ccstatestmt
        |       ccstatebody ccstatestmt
        ;
ccstatestmt:    state
        |       cstate
        ;
cstatebody:     cstatestmt
        |       cstatebody cstatestmt
        ;
cstate:         CSTATE ID '{' cstatebody '}'
        ;
cstatestmt:     state
        |       init
        |       cstate
        |       join
        |       history
        |       ccstate
        ;
state:          STATE ID '{' statebody '}'
statebody:      statestmt
        |       statebody statestmt
        ;
statestmt:      transition
        ;
init:           INIT ID ';'
        ;
history:        HISTORY ID ';'
;
join:           JOIN ID FROM ID TO ID ';'
;
transition:     TRANS qstrlist TO ID ';'
                | TRANS TO ID ';'
;
ccstate:        CCSTATE ID '{' ccstatebody '}'
;
instvar:        INSTVAR ID ID ';'
                | INSTVAR ID ID ':' '=' numid ';'
                ;
list:   '(' alist ')'
        ;
alist:  ID
        |       alist ',' ID
        ;
numid:  ID
        | NUM
        ;
qstrlist:       QSTR
                | qstrlist ',' QSTR
                ;
```

Appendix E

UML to VHDL Mapping Rules

This appendix provides a concise listing of the UML to VHDL mapping rules. Explanation and sample application of the rules is contained in Chapter 5.

Rule VHDL 1
Formalize each object of class **CLASS** as an entity/architecture pair according the following template:

```
entity obj_CLASS is
    port(state:  inout rs_st;
        sig_1:  inout sig_1type;
        .
        .
        .
        sig_n inout sig_ntype;
        instate:  out st;
        ins_state_1:  in st;
        .
        .
        .
        ins_state_n:  in st);
end entity;

architecture abstract of obj_CLASS is
begin
    instate <= state when state/=st'(none);
- Statements for states and composite states go here
end abtract;
```

`sig_1` through `sig_n` are inter-object signals. `instate` is used for the $IN()$ state predicate to convey to other entity architecture pairs (that represent various other mapped templates) the simple states this entity/architecture is in. `ins_state_1` through `ins_state_n` are incoming VHDL signals that simple states of this object can test for the $IN()$ predicate.
Relationship to Homomorphism
This rule satisfies the mapping of **Class** to **Class**. The *source* and *target* relationships are preserved by including `signals` in the VHDL class definition.
End of Rule VHDL 1.

Rule VHDL 2
Create a class **package** and **package body** named pk_*classname* to contain declarations for instance variables, the list of state names, and the resolution function required for the bus signal `state` (introduced in Rule VHDL 5). The **package** and **package body** are constructed according to the following template:

```
package pk_CLASS is
- State signal names
    type st is (none, state_1, ...  , state_n);
- An array required for the resolution function
    type st_a is array (natural range <>) of st;
- Instance variables
    shared variable var_1 <type>;
    .
    .
    .
    shared variable var_n <type>;
end package pk_CLASS;
- Resolution function for state bus
package body pk_CLASS is
    function resolve_st
        (v : in st_a) return st is
        variable i : natural;
    begin
        for i in v'range loop
            if v(i) /= st'(none) then
                return v(i);
            end if;
        end loop;
        return st'(none);
    end function;
end package body;
```

Where **CLASS** is the name of the class, and `var_i` is an instance variable. Precede each entity/architecture with a **use work.pk_CLASS.all** statement.
Relationship to Homomorphism
This rule complies with the mapping of **InstanceVariable** to **InstanceVariable**. The aggregation is preserved by including the **package** above the definition of the class, thus causing inclusion of the instance variables in the class.
End of Rule VHDL 2.

Rule VHDL 3
Class specialization is formalized as duplication of the parent class components, with appropriate renaming of the components that are named by the class, followed by modification, as required.
Relationship to Homomorphism
This rule preserves the mapping of **Generalization** to **Generalization**.
End of Rule VHDL 3.

Rule VHDL 4
Class aggregation is formalized as follows: In the collection **architecture** body, instantiate as many instances of the part entity as required by the instantiation of a particular model. The **port map** on the instantiation must match the **port** declaration for the subclass **entity**. A unique label must be provided on each entity instantiation statement.
Relationship to Homomorphism
This rule compiles with the mapping of Aggregation to Aggregation.
End of Rule VHDL 4.

Rule VHDL 5
For state named *statename*, events E_1, E_2, ... , E_n guarded by guards GD_1, GD_2, ... , GD_n, eventless transitions guarded by guards EGD_1, EGD_2, ... , EGD_n, and destination states $nextstate_1$, $nextstate_2$, ... , $nextstate_n$, formalize each simple state as a **process** according to the following template:

```
s_statename: process
begin
      wait until state=statename;
      { entry: processing statements }
      if $EGD_1$ then
         state <= $nextstate_1$;
      elsif $EGD_2$ then
         state <= $nextstate_2$;
         ⋮
      elsif $EGD_n$ then
         state <= $nextstate_n$;
      endif
      loop
         wait until $E_1$ or $E_2$ or ... $E_n$;
         if $E_1$ and $GD_1$ then
              state <= $nextstate_1$, null after 1 fs;
         { Other transition actions }
         exit;
         elsif $E_2$ and $GD_2$ then
              state <= $nextstate_2$, null after 1 fs;
         { Other transition actions }
         exit;
         end if;
      end loop;
      {exit: processing statements }
end process;
```

If there are no event-less transitions from this state then the **if** checking guards EGD_i above the **loop** can be eliminated.
Relationship to Homomorphism
This rule defines the **StateProcess** template, which is the homomorphic target of the **SimpleState** class. The **StateProcess** is a *State* and *StateVertex*. **StateProcess** has relationships with **ActionSequence** and **Transition** (through the *StateVertex* class). These relationships are preserved through the constructs **if** E_n for events and guards. Action sequences are preserved within the same **if** statement.
End of Rule VHDL 5.

Rule VHDL 6
Formalize each single threaded composite state named **CSTATE** according to the following template:

```
entity cs_CSTATE is
    port(state:  inout rs_st;
        sig_1:  inout <type>;
        .
        .
        .
        sig_n:  inout <type>;
        lsig_1:  inout <type>;
        .
        .
        .
        lsig_n:  inout <type>;
        ins_CS1:  in st;
        .
        .
        .
        ins_CS2:  in st;
        instate:  out st);
end entity;

architecture abstract of cs_CSTATE is
begin
- the body of the composite state follows. Contents may
- include simple states, composite states, concurrent composite
- states, and pseudo-states.
end abstract;
```

where `state` is the state signal per Rule VHDL 5, `sig_i` are the inter-object signals per Rule VHDL 1, `lsig_i` are intra-object signals, `ins_CS_i` are the signals for the $IN()$ predicate, and `instate` is the signal conveying this entity/architecture's current state to the rest of the object.
Relationship to Homomorphism
This rule complies with the mapping **CompositeState** to **Ent/Arch**. The template defined responds to state transitions and therefore is a *State* and a *StateVertex*. Other **StateVertex** components are included below the **begin** statement in the template in compliance with the substate aggregation. The **signal** declarations provide the links for included **StateProcesses**, which are **StateVertices**.
End of Rule VHDL 6.

Rule VHDL 7
In the parent composite state or object, instantiate the composite state as follows:

```
l1:  entity cs_CSTATE(abstract)
    port map(state=>state, sig_1=>sig_1, ..., sig_n=>sig_n,
            lsig_1=>lsig_1, ..., lsig_n=>lsig_n,
            ins_CS1=>ins_CS1,
            .
            .
            .
            ins_CSTATE=>instate,
            .
            .
            .
            ins_CS_n=>ins_CS_n);
```

where the signals are as in Rule VHDL 6. Note that signal `in_CSTATE` connects to the child `instate` signal.
Relationship to Homomorphism
This rules specifies how to satisfy the outgoing and incoming relationships between *StateVertex* and **Transition** and the *StateVertex* is a composite state.
End of Rule VHDL 7.

Rule VHDL 8
Given composite concurrent child states $child_1$, $child_2$, ... , $child_n$, separated by dotted lines in the composite state, instantiate each component as a child composite state. Each child *state* signal is connected to a unique state signal (instead of *state* in the parent) in the **port map**. All concurrent states are started in parallel by assigning the initial concurrent state name to the unique signal created. Use the following template for instantiating the child concurrent states:

```
architecture abstract of cs_state is
    signal state : rs_st ;
    signal ss_child1 : rs_st ;
    signal ss_child2 : rs_st ;
        ⋮
    signal ss_childn : rs_st ;
begin
    I1: entity cs_child1(abstract)
        port map(state=> ss_child1, ... )
    I2: entity cs_child2(abstract)
        port map(state=> ss_child2, ... )
            ⋮
    In: entity cs_childn(abstract)
        port map(state=> ss_childn, ... )
    ss_child1 <= child1,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
    ss_child2 <= child2,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
    ss_childn <= childn,null after 1 fs
        when state=child1 or state=child2 ... or state=childn
```

where ss_child_i is the temporary state signals holding cs_child_i's thread.
Relationship to Homomorphism
This rule complies with the **ConcurrentComposite** to **ConcurrentComposite** class mapping. The rule further defines how the outgoing ang incoming relationships on class *StateVertex* are maintained.
End of Rule VHDL 8.

Rule VHDL 9
In the following template $csname_n$ are the names of the concurrent composite state components, join is the name of the join state, and *newstate* is the name of the state entered after the join operation. A *join* pseudostate is formalized with one concurrent signal assignment per inbound transition arc and a join process. The signal assignments and the process take the form:

```
join_csname1 <= join when ss_csname1=join;
join_csname2 <= join when ss_csname2=join;
⋮
join_csnamen <= join when ss_csnamen=join;
process join; begin
    wait until join_csname1 = join
            and join_csname2 = join
            ⋮
            and join_csnamen = join;
    join_csname1 <= none;
    join_csname2 <= none;
    ⋮
    join_csnamen <= none;
    state <= newstate, null after 1 fs;
end process;
```

Each join_$csname_n$ must be declared in the enclosing **architecture** as an enumerated signal type matching *state* in the current component. The name of the join process must be unique.
Relationship to Homomorphism
This rule complies with the mapping **Join** to **J-Process**. A **J-Process** is a kind of *StateVertex*, thus the rules defines how the outgoing and incoming relationships are maintained.
End of Rule VHDL 9.

Rule VHDL 10
Map default initial states to a state process named *s_initial* according to the following template:

```
s_init: process
    begin
        wait until state=st'(CSTATE);
        state <= st'(DEFAULT), null after 1 fs;
    end process;
```

where CSTATE is the name of the enclosing composite state or object, and DEFAULT is the name of the deafult initial state.
Relationship to Homomorphism
This rules complies with the **Initial** to **I-Process** mapping.
End of Rule VHDL 10.

Rule VHDL 11
Map history states to an initial state named *s_initial* according to the following template:

```
s_init: process
    begin
        wait until state=st'(CSTATE);
        state <= st'(DEFAULT), null after 1 fs;
    end process;
```

where CSTATE is the name of the enclosing composite state or object, and DEFAULT is the name of the default initial state. In addition, add the following statement between the **architecture** and **begin** statement for the composite state:
signal history: st := st'(DEFAULT);
Relationship to Homomorphism
This rule complies with the **History** to **H-Process** mapping.
End of Rule VHDL 11.

Rule VHDL 12
Map a final state to a state process named *s_final* whose body contains a single **wait** with no parameters.
Relationship to Homomorphism
This rules complies with the mapping of **Final** to **S-Process**
End of Rule VHDL 12.

Appendix F

UML to Promela Mapping Rules

This appendix provides a concise listing of the UML to Promela mapping rules. Explanation of the rules in contained in Chapter 6.

Rule Promela 1
Every model mapped from UML to Promela contains the following channel definitions and the dispatching proctype, called event(), to handle intra-object transitions:

```
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};

/*
        The body of the model goes here.  This section will contain
        proctypes for objects and composite states
*/

/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
mtype type;
int pid;

atomic {
do
::  evq??[eval(msg),pid] ->
evq??eval(msg),pid;
evt!msg,pid;
do
::  if
::  evq??[type,eval(pid)] -> evq??type,eval(pid)
::  else break;
fi
od
::  else -> break
od}
exit:   skip
}
```

Relationship to Homomorphism
This rule provides details to complete the mapping from **Model** to **Model**
End of Rule Promela 1.

Rule Promela 2
Each class is formalized in Promela as a `proctype` definition. The body of the `proctype` is enclosed in an `atomic` statement. The `proctype` accepts no parameters. The following template shows the class container:

```
proctype CLASS()
{atomic{
        mtype m;
        /* Statements for the top level of the object
           go here */
exit:   skip
}
```

Relationship to Homomorphism
This rule follows from the **Model** to **ObjectProctype** mapping. The aggregation in **Model** is preserved because each class is contained within the Promela model file.
End of Rule Promela 2.

Rule Promela 3
Class specialization is formalized as duplication of the parent class components, followed by modification, as required.
Relationship to Homomorphism
This rule follows directly from the depiction of the metamodels. The type relationship with *Relationships* is preserved.
End of Rule Promela 3.

Rule Promela 4
Aggregation is formalized as an instantiation of the "part" class as many times are required or specified. The aggregate object and part objects communicate with messages.
Relationship to Homomorphism
This rule follows directly from the depiction of the metamodels. The type relationship with *Relationships* is preserved.
End of Rule Promela 4.

Rule Promela 5
Instance variables are formalized as members of a `typedef` structure statement named uniquely for the class according the the following template:

```
typedef CLASS_T {
          <type> var_1;
          .
          .
          .
          <type> var_n;
}

CLASS_T CLASS_V;
```

where `CLASS` is the name of the class. The mapped instance variable declarations are placed at the beginning of the Promela specifications before any `proctypes`.
Relationship to Homomorphism
The template listed above forms the **InstanceVariable** class in the Promela metamodel. The aggregation is preserved by naming conventions and the generation of Promela statements for the class, which reference the `typdef` and `CLASS_V` declaration described above.
End of Rule Promela 5.

Rule Promela 6
Formalize inter-object relationships as a set of Promela channels. For class CLASS, signal types SIG_1 through SIG_n, where each signal type has a number of variables, use the template below. In the template, assume SIG_1 has m variables, SIG_2 has n variables, and SIG_k has o variables. <type> represents the type of the variable passed in the message.

```
chan CLASS_q=[5] of {mtype}
chan SIG_1_p1=[5] of <type>;
        .
        .
        .
chan SIG_1_pm=[5] of <type>;

chan SIG_2_p1=[5] of <type>;
        .
        .
        .
chan SIG_2_pn=[5] of <type>;

chan SIG_k_p1=[5] of <type>;
        .
        .
        .
chan SIG_k_po=[5] of <type>;
```

The result of this template is placed at the beginning of the Promela specifications.
Relationship to Homomorphism
The above template forms the **Relationships** class in the Promela metamodel.
End of Rule Promela 6.

Rule Promela 7
Simple states are formalized as a *State Block*, which is constructed from the following template:

```
state-name:  printf(''in state object-name.state-name'');
          H_composite-state-name = st_statename;
          evt!<event-name 1>,_pid;
          evt!<event-name 2>,_pid;
             .
             .
             .
          evt!<event-name n>,_pid;
          if
          ::  <transition event expression 1> -> <guard list 1>
             <action list 1>
             <send list 1>
             goto nextstate
          ::  <transition event expression 2> -> <guard list 2>
             <action list 2>
             <send list 2>
             goto nextstate
             .
             .
             .
          ::  <transition event expression n> -> <guard list n>
             <action list n>
             <send list n>
             goto nextstate
          fi;
```

In this template, `state-name` is the name of the state, `object-name` is the name of the enclosing object, and `composite-state-name` is the name of the enclosing composite state (or object if this is the highest level). The meta terms are defined as: `<transition event expression i>` are channel receive expressions for `<event-name i>` as described in Rules Promela 10 and Promela 11, `<guard list i>` is the construction for transition guards as described in Rule Promela 16, `<action list i>` is a list of statements for the action on the transition, and `<send list i>` is a list of channel send operations for messages sent by the transition.
The second line of the template is present only if a history state is present in the enclosing composite state.
Relationship to Homomorphism
This rule provides the **StateBlock** class in the Promela metamodel. The template above describes event dispatching and action sequences that preserve the `outgoing`, `incoming`, `entry`, and `exit` relationships. **StateBlock** is a *State* since it handles events and maintains the state of the object.
End of Rule Promela 7.

Rule Promela 8
A composite state named `composite-state-name` is formalized as a `proctype` with a formal parameter of type `mtype`. A `proctype` representing a composite state is activated in the parent composite state or object with a `run(first-state-name)` statement that passes the name of the state to begin, or `none` when the transition is to the boundary of the composite state. Use the following template for the composite state `proctype`:

```
proctype composite-state-name(mtype state)
{atomic{
<initial state sequence>
<state sequences>
exit:     skip
}}
```

The meta sequences are defined as: `<initial state sequence>` is either a history state construct or an initial state construct, as required. See Rules Promela 19 and Promela 20. The body of the simple states and transfers to composite states are contained in `<state sequences>`.
Relationship to Homomorphism
Class **Proctype** holds simple states and holds other composite states through references (calling sequences) to the `proctypes` that represent the composite states, hence **Proctype** is a *State* and an aggregation of *StateVertex*.
End of Rule Promela 8.

Rule Promela 9
Concurrent composite states are formalized as composite states per Rule Promela 8, but are activated in parallel using the template below. Let cp1,cp2... cpn be the names of n concurrent composite state components. Then, in the parent composite state that activates child concurrent composite states, declare variables int cp1_pid, ..., int cpn_pid to hold the child PIDs. Similarly, declare mtype variables, mtype cp1_code,...,cpn_code for each component. The template for the declares is as follows:

```
int cp1_pid, cp2_pid, ..., cpn_pid;
mtype cp1_code, cp2_code, ..., cpn_code;
```

The declarations are placed at the beginning of the parent proctype. Then, the following template is used:

```
/*
        Compound labels for a transfer to the concurrent
        state component boundary.
*/
to_cp1:
to_cp2:
.
.
.
to_cpn:
cp1_pid = run cp1(none);
cp2_pid = run cp2(none);
.
.
.
cpn_pid = run cpn(none);
wait??eval(cp1_pid),cp1_code;
wait??eval(cp2_pid),cp2_code;
.
.
.
wait??eval(cpn_pid),cpn_code;
/* The join sequences are optional */
<join sequence 1>
.
.
.
<join sequence n>
```

The meta term <join sequence> is either assert(0) if no join construct has been specified, or one or more join sequences as described in Rule Promela 18. The concurrent composite state is activated by transferring control to any of the labels to_cp1 through to_cpn.
Relationship to Homomorphism
A **ConcurrentProctype** is a type of **Proctype** as described above. This rule details the specialization for the **ConcurrentProctype** and also preserves the join relationship by setting up the mechanisms required to map **join** to **Wait-join**.
End of Rule Promela 9.

Rule Promela 10
An intra-object transition is formalized as a send operation sending a message containing the event name and PID to channel evq to signify the transition's readiness, and a receive statement that waits for the same message to occur on channel `evt`. The following template shows the Promela construction:

```
evq!e1,_pid;
evq!e2,_pid;
.
.
.
evq!e1,_pid;
if
::  evt??e1,eval(_pid) -> <transition actions>
::  evt??e2,eval(_pid) -> <transition actions>
.
.
.
::  evt??e1,eval(_pid) -> <transition actions>
.
.
.
fi;
```

Events are dispatched by running the model-wide procedure `event()`.
Relationship to Homomorphism
This rule complies with the **Transition** to **Transition** mapping and complies with the `trigger` relationships between *Event* and **Transition**. As the metamodel depicts, there are two subtypes of **Event-Dispatch**. This rule also covers the **Run-Event()** subclass.
End of Rule Promela 10.

Rule Promela 11
An inter-object transition event using queued semantics is formalized as a receive operation on the set of channels corresponding to the event and its parameters generated by Rule Promela 6. If signal parameters are present, then pass each in a channel named for the signal per Rule Promela 6. Each parameter channel is declared as a single entry message queue of the same type as the signal parameter. An inter-object signal send is formalized as a set of channel write statements to send the parameters and signal to the channels associated with the destination object. For class **CLASS**, and inter-object signal E that receives message values into instance variables `var1, var2, ..., varn`, the following template shows the message send operation:
`E_p1!parm1; E_p2!parm2; ... E_pn!parmn; CLASS_q!E;`
and the corresponding receive and transition statements are:

```
if
::  CLASS_q?E -> E_p1?CLASS_V.var1;
                 E_p2?CLASS_V.var2;
                 .
                 .
                 .
                 E_pn?CLASS_V.varn -> <transition actions>
.
.
.
fi;
```

Relationship to Homomorphism
This rule complies with the **Transition** mapping and also reflects the superclass–subclass structure of **Event-Dispatch** in the Promela metamodel. The `incoming` relationship between **Transition** and *State Vertex* is preserved by the Promela constructs specified above.
End of Rule Promela 11.

Rule Promela 12
Formalize a missing event on a transition as "1", or true.
Relationship to Homomorphism
This rule complies with the **Transition** mapping and the *Event* aggregation. As noted on the metamodel, a transition does not require an event. When absent, this rule provides semantics for the missing event.
End of Rule Promela 12.

Rule Promela 13
An intra-composite-state transition is formalized as a `goto` transferring control to the label on the *State Block* representing the destination state.
Relationship to Homomorphism
This rule complies with the **Transition** mapping and the relationships with *StateVertex*. This rule also reflects the superclass–subclass relationship for **Event-Dispatch** in the Promela metamodel.
End of Rule Promela 13.

Rule Promela 14
A Transition that crosses the composite-state boundary downward in the state hierarchy is formalized as a `run()` statement that starts the child composite state `proctype` that either contains the destination state, or is on the transition path to the destination state. If the destination state is not the immediate child composite-state boundary, then the `run()` statement contains a parameter naming the destination state. Assume the next composite state downward on the path to the destination is `CSTATE` and the destination state is named `Dest`, then this portion of the mapping generates the statement:

```
run CSTATE(Dest);
```

Relationship to Homomorphism
This rule further describes the **Transition** mapping and preserves the relationships with *StateVertex*, specifically, down to the **Proctype** Promela metamodel class.
End of Rule Promela 14.

Rule Promela 15
A transition that crosses a composite-state boundary upward in the state hierarchy is formalized as a channel send operation followed by an exit from the composite state. The message sent consists of the current processes PID and the name of the destination state. The statement template for the transition to state `Dest` is:

```
wait!_pid,Dest; goto exit;
```

Relationship to Homomorphism
The comments for Rule Promela 14 are applicable to this rule.
End of Rule Promela 15.

Rule Promela 16
Formalize a guard on a transition as an `if-fi` block located immediately after the statements representing the event reception. Gather all like events into one wait statement. For one or more transitions on event `E` guarded by `G1` through `Gn`, and wait expression `Q` (`Q` is some form of a wait on a channel per Rule Promela 10, Rule Promela 11, or Rule Promela 12) the following template applies:

```
if
::  Q?E -> if
        ::  G1 -> <transition 1>
        ::  G2 -> <transition 2>
        .
        .
        .
        ::  Gn -> <transition n>
        ::  else goto in_STATE;
        fi
.
.
.
```

For eventless transitions guarded as above, replace the final `else` with `else 0`.
Relationship to Homomorphism
This rule complies with the **Guard** to **IFGuard** mapping and preserves the aggregation between **Guard** and **Transition** in the UML metamodel.
End of Rule Promela 16.

Rule Promela 17
Timed events are mapped to uniquely named events following the naming convention $TE1, TE2, \ldots, TEn$. The modeler is responsible for generating these events elsewhere in the model.
Relationship to Homomorphism
Class**TimeEvent** is mapped to **Event-Dispatch** because there are no constructs in Promela for time-constrained events. Therefore, to complete the mapping, timeouts and interval waits are mapped to user-generated events.
End of Rule Promela 17.

Rule Promela 18

Each *Join* pseudo-state is formalized as a `if-fi` block. For each *Join*, construct a guard consisting of a conjunct testing each concurrent component's return code for a transition to that *Join*. The *Join* template is as follows:

```
          if
          :: cp1_code == st_join1 && cp2_code == st_join1 ...  && cpn_code == st_join1
               -> goto dest1
          :: cp1_code == st_join2 && cp2_code == st_join2 ...  && cpn_code == st_join2
               -> goto dest2
          .
          .
          .
          :: cp1_code == st_joinn && cp2_code == st_joinn ...  && cpn_code == st_joinn
               -> goto destn
          fi
```

Where `cpn_code` is the return code from the n-th process indicating the join state, `st_joinn` is the name of the n-th join, and `destn` is the destination from the n-th join.

Relationship to Homomorphism

This rule complies with the mapping of **Join** to **Wait-join**. In Promela, the `join` construct occupies the same relative position as a state, and has a destination label.Therefore, it qualifies as a subtype of *Pseudostate*.

End of Rule Promela 18.

Rule Promela 19

Each *Initial* pseudo-state is formalized as a goto to the initial state. The goto is located at the start of the Promela procedure associated with the composite state or object.

Relationship to Homomorphism

Init-goto is mapped from **Start**. The Promela implementing `init` fits within a composite state or object envelope, and has an outbound transition. Therefore, it qualifies as a subtype of the supertype *Pseudostate*.

End of Rule Promela 19.

Rule Promela 20
Formalize each *History* pseudo-state as an `if-fi` block at the top of the Promela procedure representing the composite state. The `if-fi` block tests the value of a variable holding the name of the last state. When a match is found, the `if-fi` transitions to the state with a goto as in other transitions. The `if-fi` block follows the following template:

```
if
:: H_composite-state-name == st_state1 -> goto state1;
:: H_composite-state-name == st_state2 -> goto state2;
.
.
.
:: H_composite-state-name == st_staten -> goto staten;
fi;
```

Where `composite-state-name` is the name of the composite state (objects may not contain history pseudo-states at their top level), and `statei` is the name of the i-th state in the composite state. When a *History* state is present, line 2 in the template in Rule Promela 7 must be present and the variable `H_composite-state-name` is declared as type `mtype` in the global declarations. The initial value of `H_composite-state-name` is the default initial state.
Relationship to Homomorphism
The comments for Rules Promela 18 and Promela 19 apply to this rule, which maps **Histroy** to **History-goto**.
End of Rule Promela 20.

Rule Promela 21
Formalize each *Final* pseudo-state as a transfer (`goto`) to label `exit:` at the end of the containing Promela procedure.
Relationship to Homomorphism
This rule complies with the mapping of **Final** to **Goto-exit**, and the previous comments for Pseudostates apply.
End of Rule Promela 21.

Appendix G

VHDL LCS Files

The listings below contain the Perl source code for Hydra for the VHDL translations. The Hydra Class Library consists of the files (marked in the listing) CCState.pm, Context.pm, Init.pm, Object.pm, CState.pm, History.pm, Join.pm, and State.pm.

```
#!/usr/bin/perl

# The main VHDL Hydra routine. The main program passes the AST to
# Model. Sub Model is the beginning of the AST Walker, which is then
# contained in the balance of this source file. The Class Library
# routines are listed as separate files below.

{
    use Yacc;
    use CState;
    use CCState;
    use State;
    use Object;
    use Context;
    use History;
    use Join;
    use Init;

    $tree = Yacc->Parse($ARGV[0]);
    print "parse complete\n";
    Context->OutFile('pan');
    Model($tree);
    Context->Output;
}
```

```
#  Start of the AST Walker routines.

sub Model
{
    my ($tree) = @_;

    print "model name=$$tree{ID}\n";
    my $stmt = $$tree{modelbody}{modelstmtlist};
    my $ent;
    foreach $ent (@$stmt) {   # Each one of these is a hash
        if ($$ent{type} eq 'Enum') {
            print "Enum $$ent{ID}=$$ent{list}\n";
            my @list;
            $$ent{list} =~ s/^\((.+)\)$/$1/;
            @list = split(/,/, $$ent{list});
            Context->Enum($$ent{ID}, @list);
        }
        elsif ($$ent{type} eq 'Driverfile') {
            print "Driverfile=$$ent{ID}\n";
            Context->DriverFile($$ent{ID});
        }
        elsif ($$ent{type} eq 'Object') {
            print "Object $$ent{ID}\n";
            Object($ent);
        }
        else {die "bad type $$ent{type}"}
    }
}

sub Object
{
    my ($tree) = @_;

    $$tree{type} eq 'Object' || die "this is not an Object node";

    print "In Object. Object Name=$$tree{ID}\n";
    my $objref = new Object($$tree{ID});

    my $body = $$tree{objectbody}{objectstmtlist};
    my $ent;
    my %nametoref;
    my @joinlist;
    foreach $ent (@$body) {
        if ($$ent{type} eq 'Assoc') {
            print "Assoc $$ent{ID} type $$ent{assoctype}\n";
            $objref->Assoc($$ent{ID}, $$ent{assoctype});
        }
        elsif ($$ent{type} eq 'CState') {
            $nametoref{$$ent{ID}} = CState($ent, $objref);
        }
        elsif ($$ent{type} eq 'CCState') {
             my $ref = CCState($ent, $objref);
             $nametoref{$$ent{ID}} = $ref;
             my @names;
```

```
                my @members;
                @members = $ref->GetMembers;
                my $thing;
                print "***** CStates in $$ent{ID}\n";
                print "ref is a ",ref($ref),"\n";
                foreach $thing (@members) {
                    my $name = $thing->GetName;
                    $nametoref{$name} = $thing;
                    print "$name is a ",ref($nametoref{$name}),"\n";
                }
               print "CCState $$ent{ID}\n";
        }
        elsif ($$ent{type} eq 'State') {
            $nametoref{$$ent{ID}} = State($ent, $objref);
        }
        elsif ($$ent{type} eq 'Init') {
            print "Init state jumps to $$ent{ID}\n";
            if ($init) {
                print "ERROR: $$tree{id} has more than one init state\n";
            }
            $init = $$ent{ID};
        }
        elsif ($$ent{type} eq 'Join') {
            push(@joinlist, [$$ent{ID},$$ent{from},$$ent{to}]);
        }
        else {die "bad type $$ent{type}"}
    }

    print "Object ref keys\n";
    foreach $ent (keys %nametoref) {
        print "$ent is a ",ref($nametoref{$ent}),"\n";
    }

    foreach $ent (@joinlist) {
        print "Making Join $$ent[0] from $$ent[1] to $$ent[2]\n";
        new Join($objref, $$ent[0], $nametoref{$$ent[1]}, $$ent[2]);
    }

    if ($init) {new Init($objref, $nametoref{$init})}
}

sub CState
{
    my ($tree, $parent) = @_;

    $$tree{type} eq 'CState' || die "node is not a CState";

    print "In CState. CState name=$$tree{ID}\n";
    my $cstateref = new CState($parent, $$tree{ID});

    my $body = $$tree{body}{cstatestmtlist}; # List of stmts
    my %nametoref;
    my ($ent, $init, $history);
    foreach $ent (@$body) {
        print "-- New node type $$ent{type}\n";
        if ($$ent{type} eq 'State') {
            my $r = State($ent, $cstateref);
            $nametoref{$$ent{ID}} = $r;     # save ref by name
        }
        elsif($$ent{type} eq 'CState') {
            my $r = CState($ent, $cstateref);
            $nametoref{$$ent{ID}} = $r;     # save ref by name
        }
        elsif ($$ent{type} eq 'Init') {
            print "Init state jumps to $$ent{ID}\n";
            if ($init) {
                print "ERROR: $$tree{id} has more than one init state\n";
            }
            $init = $$ent{ID};
        }
        elsif ($$ent{type} eq 'History') {
            print "History state jumps to $$ent{ID}\n";
            if ($history) {
                print "ERROR: $$tree{id} has more than one History state\n";
            }
            $history = $$ent{ID};
        }
        elsif ($$ent{type} eq 'CCState') {
            my %names;
            %names = CCState($ent, $objref);
            my $key;
            foreach $key (keys %names) {$nametoref{$key} = $name{$key}}
            print "CCState $$ent{ID}\n";
        }
        else {die "bad type $$ent{type}"}
    }
# Now to Init states
    print "CState ref keys\n";
    foreach $ent (keys %nametoref) {
        print "$ent is a ",ref($nametoref{$ent}), "\n";
    }
    if ($history) {
        print "-- calling history. Parent is ",$cstateref->GetName,"\n";
        new History($cstateref, $nametoref{$history});
    }
    if ($init) {
        print "-- calling init\n";
        new Init($cstateref, $nametoref{$init});
    }
    else {"Warning: There is not INIT state for $$tree{ID}\n"}
    print "-- Return from CState\n";
    return $cstateref;  # return the ref to this CState
}

sub CCState
{
# The only legal statements for this container is CState. Transitions (such
# as inits) may want to transition to one of our components, which starts the
# threads running. Therefore, we pass back a hash of CState refs indexed by
# CState name.
    my ($tree, $parent) = @_;

    print "In CCState. CCState name=$$tree{ID}\n";
    my $ccstate = new CCState($parent);

    my $body = $$tree{body}{ccstatestmtlist}; # List of stmts
    my $ent;
    foreach $ent (@$body) {
        if ($$ent{type} eq 'State') {
            State($ent, $ccstate);
        }
        elsif($$ent{type} eq 'CState') {
            CState($ent, $ccstate);
        }
        else {die "bad type $$ent{type}"}
    }
    return $ccstate;
}

sub State
{
    my ($tree, $parent) = @_;

    $$tree{type} eq 'State' || die 'Node is not a State node';
    print "State name=$$tree{ID}\n";
    my $stateobj = new State($parent, $$tree{ID});

    my $body = $$tree{statebody}{statestmtlist};
    my $ent;
    foreach $ent (@$body) {
        if ($$ent{type} ne 'Trans') {
            die "something other than Trans in a state";
        }
        print "Transition $$ent{tran} to state $$ent{dest}\n";
        my $tran;
        ($tran) = $$ent{tran} =~ /^"(.+)"$/;
        $stateobj->Tran($tran, $$ent{dest});
    }
    return $stateobj;   # return the ref to this state
}
#!/usr/bin/perl
package CCState;

# Instance Vars
# members: list of refs to member cstates
# parent: who owns me (where I get included)

# Need a CCState to hold CStates, which run concurrently. A CCState is a kind
# of  container that holds CStates, that hold states.
# CCState has no output method because it only generates instantiation
# stmts within the thing (CState, probably) that it is in. It has LclOutput
# because it needs to write statements into the thing it is in.
# The idea is to create a new state signal for each parallel component
# and instantiate a ent/arch for each one. We also listen for any of the
# CState names and start the threads if we get one. This means transitions
# into CCStates is not allowed.

# You have to call RecordState to record a new CState that is within this
# CCState. This is equivalent to calling State within CState.
sub new
{
    my ($class, $parent) = @_;
    $obj = {};
    bless $obj;
    $COUNT++;
    $$obj{name} = "CCSTATE$COUNT";
    $$obj{parent} = $parent;
    $parent->AddState($obj);      # add to instantiation list of parent
    $$obj{objref} = $parent->GetObjRef; # get ref to object
    Context->Depend($obj, $parent);     # Register dependency
# determine dominant (highest) CCState. CState returns what it's parent
# does and Object returns null, so a null means we are highest. Also record
# if this is the dominant ccstate. This is used for 'in' state predicate.
    $$obj{domcc} = $parent->HiCCState;
    if (!$$obj{domcc}) {
        print "I AM the domniant CCSTATE\n";
        $$obj{domcc} = $obj;
        $$obj{medomcc} = 1;
    }
    else {$$obj{medomcc} = 0}
    return $obj;
}

sub HiCCState
# return dominant ccstate, perhaps me.
{
    $obj = shift;
    return $$obj{domcc};
}

sub RecordCState
# record a new CState added somewhere within this CCState context
{
    my ($obj, $cs) = @_;

    print "CState ",$cs->GetName," recorded in CCstate\n";
    push(@{$$obj{cslist}}, $cs);
}

sub GetCStateList
{
    return @{$$obj{cslist}};
}


sub GetObjRef
```

```
{
    my $obj = shift;
    return $$obj{objref};
}

sub AddState
{
# $state is ref to state-obj added to this CCState
# It had better be a CState.
    my ($obj, $state) = @_;

    push(@{$$obj{members}}, $state);
    my $name;
# CCStates need not have their name recorded because there is no official
# (given by user) name. Also, we don't want a state transition for
# CCStates
    if (ref($state) eq 'CCState') {return}
    $name = $state->GetName;
# add the name of this state to the containing object
    if ($name) {$$obj{objref}->AddStateName($name)}
}

sub GetName {return $$obj{name};}

sub LclOutput
{
    my $obj = shift;

    Put Context 0,"-- CCState output\n";
    @ports = $$obj{objref}->GetPortNames;
    my $when = '';
    my @names;
    foreach $cs (@{$$obj{members}}) {
        my $name = $cs->GetName;
        push(@names, $name);
        if ($when) {$when .= " or state=st'($name)"}
        else {$when = "when state=st'($name)"}
    }
# now run thru list of names and make instantiations
    my $cs;
    foreach $name (@names) {
        $label = Context->GetLabel;
        Put Context 1,"$label: entity cs_$name(abstract)\n";
        my $port = "port map(state=>ss_$name";
        foreach $a (@ports) {
            $port .= ", $a=>$a";
        }
# do the ports for the instate predicate
        foreach $cs ($$obj{domcc}->GetCStateList) {
            my $csname = $cs->GetName;
            if ($name eq $csname) {$port .= ", instate=>ins_$name"}
            else {$port .= ", ins_$csname=>ins_$csname"}
        }
        Put Context 2, "$port);\n";
        Put Context 1,"ss_$name <= st'($name), null after 1 fs $when;\n";
    }
}

sub OutputSignals
{
    my $obj = shift;

    foreach $state (@{$$obj{members}}) {
        my $name = $state->GetName;
        Put Context 1,"signal ss_$name: rs_st;\n";
    }
    if ($$obj{medomcc}) {
        my $cs;
        foreach $cs (@{$$obj{cslist}}) {
            my $name = $cs->GetName;
            Put Context 1,"signal ins_$name: st;\n";
        }
    }
}

sub GetStateNameList
{
# return a list of CStates that lie within the CCState. Called by Join.
# works by running thru the list of members calling each's GetName method.
    my $obj = shift;
    my $cs;
    my @names = ();
    foreach $cs (@{$$obj{members}}) {
        my $name = $cs->GetName;
        push(@names, $name);
    }
    return @names;
}

sub GetMembers
{
# Return a list of members of this CCState
    my $obj = shift;
    return @{$$obj{members}};
}
1;
#!/usr/bin/perl
package CState;
use State;

# name: my name
# objname: name of object owning this state
# objref: reference to object owning this state
# state: list of names of states to instantiate within my arch body
# parent: ref to who owns me

sub new
{
```

```
# Make a new Composite state with parent ref to enclosing object, state, etc
    my ($class, $parent, $name) = @_;

    print "My parent is ",$parent->GetName,"\n";
    my $obj = {};
    bless $obj;
    $$obj{name} = $name;
    $$obj{parent} = $parent;
    $$obj{objref} = $parent->GetObjRef;
    $parent->AddState($obj);
    Context->Depend($obj, $parent);
# next find highest (most containing) CCState, if any. If there is one,
# add me to his list. This is for 'in' predicate processing
    $$obj{domcc} = $parent->HiCCState;
    print "CState got a ",ref($$obj{domcc}), " back for HiCC\n";
    if ($$obj{domcc}) {
        print "There IS a dominant CCState\n";
        $$obj{domcc}->RecordCState($obj);  # if exists, add me.
    }
    else {print "there is no dominant CCState\n"}
    return $obj;
}

sub HiCCState
{
# This just passes the call thru. See CCState for details of what's going
# on.
    my $obj = shift;
    return $$obj{parent}->HiCCState;
}

sub GetName
{
    my $obj = shift;
    return $$obj{name};
}

sub GetObjRef
{
# return a reference to the Object (highest level)
    my $obj = shift;
    return $$obj{objref};
}

sub AddState
{
# called by simple state process to include state in my list of states
    my ($obj, $state) = @_;

    push(@{$$obj{states}}, $state);
    my $name;
    $name = $state->GetName;
    print "=== CState name $$obj{name} adding state $name\n";
    if ($name) {
        $$obj{objref}->AddStateName($name);
    }
}

sub GetStateNameList
{
    my $obj = shift;

    my ($state,$name,@list);
    foreach $state (@{$$obj{states}}) {
        $name = $state->GetName;
        if ($name) {push(@list, $name)}
    }
    return @list;
}

sub OutputSignals {return;}

sub LclOutput
{
    my $obj = shift;

    my $label = Context->GetLabel;
    my @ports;
    @ports = $$obj{objref}->GetPortNames;
    Put Context 1,"$label: entity cs_$$obj{name}(abstract)\n";
    my $port = 'port map(state=>state';
    foreach $var (@ports) {
        $port .= ", $var=>$var";
    }
# do the ports for the instate predicate
    if ($$obj{domcc}) {
        my $cs;
        my $parentname = $$obj{parent}->GetName;
        foreach $cs ($$obj{domcc}->GetCStateList) {
            my $csname = $cs->GetName;
            if ($$obj{name} eq $csname) {$port .= ", instate=>ins_$$obj{name}"}
            elsif ($parentname eq $csname) {$port .= ", ins_$csname=>instate"}
            else {$port .= ", ins_$csname=>ins_$csname"}
        }
    }
    Put Context 2,"$port);\n";
}

sub Output
{
    my $obj = shift;

    my $state;
    Put Context 0,"use std.textio.all;\n";
    Put Context 0,"use work.easyio.all;\n";
    Put Context 0,"use work.common.all;\n";
    my $objname = $$obj{objref}->GetName;
    Put Context 0,"use work.pk_$objname.all;\n";
    Put Context 0,"entity cs_$$obj{name} is\n";
    my @ports = $$obj{objref}->GetPortNames;
    my @porttype = $$obj{objref}->GetPortTypes;
    my $i;
    if (@ports == 0) {Put Context 1,"port(state: inout rs_st);\n"}
    else {Put Context 1,"port(state: inout rs_st;\n"}
    for ($i=0; $i<@ports; $i++) {
        if ($i > 0) {Put Context 0,";\n"}
        Put Context 2,"$ports[$i]: inout $porttype[$i]";
    }
    if ($$obj{domcc}) {
        my @insports = $$obj{domcc}->GetCStateList;
        my $csobj;
        my $csname;
        foreach $csobj (@insports) {
            $csname = $csobj->GetName;
            Put Context 0,";\n";
            if ($csname eq $$obj{name}) {Put Context 2,"instate: out st"}
            else {Put Context 2,"ins_$csname: in st"}
        }
    }
    Put Context 0,");\n";
    Put Context 0,"end entity;\n\n";
    Put Context 0,"architecture abstract of cs_$$obj{name} is\n";
# Add signals for states
    foreach $state (@{$$obj{states}}) {
        $state->OutputSignals;
    }
    Put Context 0,"begin\n";

# If there are 'in' predicates, need this stmt to set instate.
    if ($$obj{domcc}) {
        Put Context 1,"instate <= state when state/=st'(none);\n";
    }

    foreach $state (@{$$obj{states}}) {
        $state->LclOutput;
    }
    Put Context 0,"end abstract;\n";
    Put Context 0,"-----------------------------------------------------\n\n";
}
1;
#!/usr/bin/perl
package Context;
use FileHandle;

sub BEGIN
{
    @ObjectList = ();
}

sub Enum
{
# Handle ENUM types. These will go into package common
```

```
    my ($class, $name, @list) = @_;

    $Enum{$name} = [@list];
}

sub OutFile
{
    my ($class, $filename) = @_;
    open(OUT, ">$filename.vhdl");
}

sub Put
{
# Produce output. indent for looks. First parm is ident spaces X 4. Value
# of -1 ==> write out first entry as is.
    my ($class, $indent, @a) = @_;

    if ($indent eq -1) {
        print OUT $a[0];
        return;
    }
    $indent *= 4;
    my $text = join('', @a);
    while (length($text)+$indent > 72) {
        for ($i=72-$indent; $i; $i--) {
            if (substr($text,$i,1) eq ' ') {last}
        }
        print OUT ' ' x $indent, substr($text,0,$i),"\n";
        $text = substr($text,$i+1);
    }
    print OUT ' ' x $indent, $text;
}

sub GetLabel
{
    $Count++;
    return "l$Count";
}

sub AddObj
{
    my $class = shift;
    my $obj = shift;

    push(@ObjectList, $obj);
}

sub Initial
{
    my $text = <<"EOF";
use std.textio.all;
package easyio is
    procedure say (constant str : in string);
end package easyio;
```

```
package body easyio is
    procedure say (constant str : in string) is
        variable myline : line;
    begin
        write(myline, now);
        write(myline, " : ");
        write(myline, str);
        writeline(output, myline);
    end;
end package body easyio;
---------------------------------------------------------------------------

package common is
EOF

    Put Context -1,$text;
    my $x;
    foreach $x (keys %Enum) {
        Put Context 1,"type $x is (", join(',',@{$Enum{$x}}), ");\n";
    }
    Put Context 0, "end package common;\n\n";
    foreach $x (@ObjectList) {
        print "context doing initial for object ",$x->GetName,"\n";
        $x->Initial;
    }
}

sub Depend
{
# b depends on a, that is, a should occur first in the output. This
# methods keeps track of each CState and CCState and keeps them in order.
    my ($class, $a, $b) = @_;

    if (!$b) {
        push(@Depend, $a);
        return;
    }
    for ($i=0; $i,@Depend; $i++) {
        if ($Depend[$i] == $b) {
            splice(@Depend,$i,0,$a);
            return;
        }
    }
    push(@Depend, $a, $b);
}

sub PutState2CS
{
# Save name of parent state under this state name so we can find out
# to whom a strate belongs.

    my ($class, $state, $cs) = @_;
    $State2CS{$state} = $cs;
```

```
}

sub State2CS
{
# Return to whom a state (by name) belongs (name of context)

    my ($class, $state) = @_;
    return $State2CS{$state};
}

sub DriverFile
{
# The filename of the driver is specified here. All we need is an
#architecture abstract of context, declare the signals needed,
# instantiate the top object abd begin driving.

    my ($class,$filename) = @_;
    $DriverFile = $filename;
}

sub Output
{
# This is the main output routine. It cycles thru the list of CStates and
# CCStates, which are in dependency order, and calls it's output routine.
    Context->Initial;
    print "Entities order:\n";
    my ($name,$ent);
    foreach $ent (@Depend) {
        if (ref($ent) eq 'CCState') {next}
        $name = $ent->GetName;
        print "$name\n";
        $ent->Output;
    }
    Put Context 0,"use std.textio.all;\n";
    Put Context 0,"use work.easyio.all;\n";
    Put Context 0,"use work.common.all;\n";
    foreach $obj (@ObjectList) {
        $name = $obj->GetName;
        Put Context 0,"use work.pk_$name.all;\n";
    }
    Put Context 0,"entity context is\n";
    Put Context 0,"end entity;\n\n";
    open(IN, $DriverFile) || die "No driver file specified\n";
    while ($l = <IN>) {
        Put Context 0,$l;
    }
    close (IN);
}
1;
#!/usr/bin/perl
package History;

sub new
{
```

```
# $parent is an obj ref, $dest is a ref to dest state (default start)
    my ($class, $parent, $dest) = @_;

    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    $$obj{dest} = $dest;
    $parent->AddState($obj);
    return $obj;
}

sub GetName {return '';}

sub OutputSignals
{
    my $obj = shift;

    my $name = $$obj{dest}->GetName;
    Put Context 1,"signal history: st := st'($$obj{dest});\n";
}

sub LclOutput
{
    my $obj = shift;

    my @list;
    my $when = '';
    my $name;
    @list = $$obj{parent}->GetStateNameList;
    foreach $name (@list) {
        if ($when eq '') {$when = "state=st'($name)"}
        else {$when .= " or state=st'($name)"}
    }
    Put Context 0, "\n";
    Put Context 1,"history <= state when $when;\n";

    Put Context 1,"s_init: process\n";
    Put Context 1,"begin\n";
    Put Context 2,"say(\"In state $$obj{name} history\");\n";
    Put Context 2, "wait until state=st'(", $$obj{parent}->GetName, ");\n";
    Put Context 2,"state <= history, null after 1 fs;\n";
    Put Context 1,"end process;\n";
}
1;
#!/usr/bin/perl
package Init;

# Create an Init state. We can handle transitions to the CState because
# we build waits that listen for the CState name.

sub new
# parent is the contining entitiy (like CState), $dest is ref to start
# state which may be a CState or a State. NOTE: dest is a REF.
{
    my ($class, $parent, $dest) = @_;

    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    $$obj{dest} = $dest;
    $parent->AddState($obj);
    return $obj;
}

sub GetName {return '';}

sub OutputSignals {return;}

sub LclOutput
{
    my $obj = shift;

    my $object = $$obj{parent}->GetObjRef;
    Put Context 0,"\n";
    Put Context 1,"s_init: process\n";
    Put Context 1,"begin\n";

# If Init is not contained in Object (top level), then we have to listen
# for our parent's name, and I must be in a CState (CCState's don't have
# official names. If I AM at the Object level, just initialize the first
# state and begin
    if ($object != $$obj{parent}) {
        my $name = $$obj{parent}->GetName;
        Put Context 2,"wait until state=st'($name);\n";
    }
    Put Context 2,"state <= st'(",
    $$obj{dest}->GetName, "), null after 1 fs;\n";
    if ($object == $$obj{parent}) {Put Context 2,"wait;\n"}
    Put Context 1,"end process;\n";
}
1;
#!/usr/bin/perl
package Join;

# This object builds a join construct inside a composite state of some
# sort. It creates a signal for each parallel state in the CCstate
# then waits for a transition to it's name from that CCstate component.
# This is not complete: If a thread just stops, the join will never
# complete, but then we don't have a 'stop' yet. It is equivalent to
# just go to the join.
# Join gets it's list of CStates from it's parent CCState, so the join can
# only be one level above the CCState.

sub new
{
# new Join(parent, my_name, CCstate, dest)
# $parent is ref to where to put the join construct.
# my_name is the state name where transitions go to join; join name(literal)
# $CCstate is the concurrent state ref from which the join is coming
# $parent, $ccstate are obj refs
# $name and $dest are literals
    my ($class, $parent, $name, $ccstate, $dest) = @_;

    print "   JOIN $name to $dest\n";
    my $obj = {};
    bless $obj;
    $$obj{ccstate} = $ccstate;
    $$obj{dest} = $dest;
    $$obj{name} = $name;
    $$obj{parent} = $parent;
    $parent->AddState($obj);
   return $obj;
}

sub GetName
{
    my $obj = shift;
    return $$obj{name};
}

sub OutputSignals
{
    my $obj = shift;

    @states = $$obj{ccstate}->GetStateNameList;
    foreach $state (@states) {
        Put Context 1,"signal $$obj{name}_$state: st;\n";
    }
}

sub LclOutput
{
    my $obj = shift;
    my @states;
    @states = $$obj{ccstate}->GetStateNameList;
    Put Context 0,"\n";
    my $and;
# build static signals from CCState list
    foreach $cs (@states) {
        Put Context 1,"$$obj{name}_$cs <= ss_$cs" .
            " when ss_$cs=st'($$obj{name});\n";
        if ($and) {$and .= " and "}
        $and .= "$$obj{name}_$cs=st'($$obj{name})";
    }
    Put Context 1,"s_$$obj{name}: process\n";
    Put Context 1,"begin\n";
    Put Context 2,"wait until $and;\n";
    Put Context 2,"say(\"$$obj{name} join complete\");\n";
    Put Context 2,"state <= st'($$obj{dest}), null after 1 fs;\n";
    foreach $cs (@states) {
        Put Context 2,"$$obj{name}_$cs <= st'(none);\n";
    }
    Put Context 1,"end process;\n\n";
}

sub GetSignals
{
    my $obj = shift;

    my $state;
    foreach $state ($$obj{ccstate}->GetStateNameList) {
        Put Context 1,"signal $$obj{name}_$cs: st;\n";
    }
}
1;
#!/usr/bin/perl
package Object;
use State;
use CCState;
# name: my name
# states: list of refs of states to instantiate
# statenames: list of names of states for state type dcl
# portnames: list of port variables for entity
# porttypes: corresponding type of ports

#LclOutput vs. Output:
# Method Output produces and Ent/Arch of some type.
# Methods LclOutput produces content intended to be inside an arch,
# so State,History,Init have LclOutput's only.
#    CState, CCState, have both (one to make the Ent/arch, another
#    for the parent to build stmts to call it.)

sub new
{
    my ($class, $name) = @_;

    $obj = {};
    bless $obj;
    $$obj{name} = $name;
    $$obj{ordername}[0] = $name;
    $$obj{orderref}[0] = $obj;
    Context->AddObj($obj);  # add this object to list of objects
    Context->Depend($obj);
    return $obj;
}

sub HiCCState {return '';}

sub Assoc
{
    my ($obj, $var, $type) = @_;

    push(@{$$obj{portnames}}, $var);
    push(@{$$obj{porttypes}}, $type);
}
```

```
sub GetPortNames
{
    $obj = shift;
    return @{$$obj{portnames}};
}

sub GetPortTypes
{
    $obj = shift;
    return @{$$obj{porttypes}};
}

sub GetName
{
    my $obj = shift;
    return $$obj{name};
}

sub GetObjRef
{
# return a reference to myself, since I am the Object.
    my $obj = shift;
    return $obj;
}

sub AddState
{
# called by simple state process to include state in my list of states
    my ($obj, $state) = @_;

    push(@{$$obj{states}}, $state);
    my $name;
    $name = $state->GetName;
    $obj->AddStateName($name);
}

sub AddStateName
{
    my ($obj, $name) = @_;

    if ($name ne '') {push(@{$$obj{statenames}}, $name)}
}

sub Output
{
    my $obj = shift;

    my $state;
    Put Context 0,"use std.textio.all;\n";
    Put Context 0,"use work.easyio.all;\n";
    Put Context 0,"use work.common.all;\n";
    Put Context 0,"use work.pk_$$obj{name}.all;\n";
    Put Context 0,"entity obj_$$obj{name} is\n";
    my @ports = $obj->GetPortNames;
    my @porttype = $obj->GetPortTypes;
    my $i;
    Put Context 1,"port($ports[0]: inout $porttype[0]";
    for ($i=1; $i<@ports; $i++) {
        Put Context 0,";\n";
        Put Context 2,"$ports[$i]: inout $porttype[$i]";
    }
    Put Context 0,");\n";
    Put Context 0,"end entity;\n\n";
    Put Context 0,"architecture abstract of obj_$$obj{name} is\n";
# Add signals for states
    Put Context 1,"signal state :  rs_st;\n";
    foreach $state (@{$$obj{states}}) {
        Put Context 0,"-- signals from ",$state->GetName,"\n";
        $state->OutputSignals;
    }
    Put Context 0,"begin\n";

    foreach $state (@{$$obj{states}}) {
        $state->LclOutput;
    }
    Put Context 0,"end abstract;\n";
}

sub Dcl
{
# Just hold on to a bunch of object level declares. Used for assoc port
# signals.
    my ($obj, $var) = @_;

    push(@{$$obj{dcls}}, $var);
}

sub Initial
{
    my $obj = shift;

    Put Context 0,"package pk_$$obj{name} is\n";
    my $dcl;
    foreach $dcl (@{$$obj{dcls}}) {
        Put Context 1,"$dcl\n";
    }

#    Put Context 1,"type st is (none",
#    my $s;
    my $s = join(', ', @{$$obj{statenames}});
    Put Context 1,"type st is (none, $s);\n";
#    foreach $s (keys %{$$obj{statenames}}) {
#        Put Context 0, ", $s";
#    }
#    Put Context 0, ");\n";
    Put Context -1,<<"EOF";
    type st_a is array (natural range <>) of st;
    function resolve_st (v: in st_a) return st;
    subtype rs_st is resolve_st st;
end package pk_$$obj{name};

package body pk_$$obj{name} is
    function resolve_st
        (v : in st_a) return st is
        variable i : natural;
    begin
        for i in v'range loop
            if v(i) /= st'(none) then
                return v(i);
            end if;
        end loop;
        return st'(none);
    end function;
end package body;
------------------------------------------------------------------------
EOF
}
1;
#!/usr/bin/perl
package State;
use Context;
use CState;
# name: name of state
# parent: my owing composite state (or object)
#    (the next 3 are sequenced together)
# event: list of events
# guard: list of guards; null entry=no guard
# dest: list of dest states
#   (the next 2 sequenced together)
# noeventguard: list of guard without events
# noeventdest:  matching dests for trans without guards

sub new
{
    my ($class, $parent, $name) = @_;

    my $obj = {};
    bless $obj;
    $$obj{event} = [];
    $$obj{guard} = [];
    $$obj{dest} = [];
    $$obj{noeventguard} = [];
    $$obj{noeventdest} = [];
    $$obj{parent} = $parent;
    $$obj{name} = $name;
    $parent->AddState($obj);  # add myself to parent's list
# This is a kinda a hack to find the CState from the state name
    Context->PutState2CS($name, $parent->GetName);
    return $obj;
}

sub GetName
```

```
{
    my $obj = shift;
    return $$obj{name};
}

sub Tran
{
    my ($obj, $tran, $dest) = @_;

    my ($gd, $action, $msg, $subgd);
# For event[gd]/action^msg, strip of stuff after event leaving raw event.
# Since these expressions have regex chars in them, the $subgd statements
# are escaping these so the tran =~ s/... works right
    $tran .= ';';
    if ($tran =~ /\[ ([^\];]+) \]/x) {  # Look for guard
        $subgd = $gd = $1;
        $subgd =~ s/([()\[\]])/\\$1/g;  # escape regex
        $tran =~ s/\[$subgd\]//;        # get rid of guard
    }
    if ($tran =~ /\/ ([^\^;]+) /x) {    # Look for action
        $subaction = $action = $1;
        $subaction =~ s/([()\[\]])/\\$1/g;  # escape regex
        $tran =~ s{\/$action}{};            # delete action from tran
    }
    if ($tran =~ /\^([^;]+)/) {   # Look for msg
        $msg = $1;
        $tran =~ s{\^$msg}{};     # delete action from tran
    }
    chop $tran;                   # get rid of ';'
    my $event = $tran;

# next 4 stmts take care of in(...) predicate and add quote to enums, like
#  type'(value). Enums count on form: type(value)=value
    $event = InTransform($event);
    $event =~ s/([A-z]+)=([A-z]+)\(([A-z]+)\)/$1=$2\'($3)/g;
    $gd = InTransform($gd);
    $gd =~ s/([A-z]+)=([A-z]+)\(([A-z]+)\)/$1=$2\'($3)/g;


# event driven transition stored here:
    if ($event) {
        print "TRAN  saving $event,$gd -> $dest\n";
        push(@{$$obj{event}}, $event);
        push(@{$$obj{guard}}, $gd);
        push(@{$$obj{dest}}, $dest);
    }
# no event (guard only) transitions
    else {
        if (!$gd) {
            die "No event or guard for transition $tran for state $$obj{name}";
        }

        push(@{$$obj{noeventguard}}, InTransform($gd));
        push(@{$$obj{noeventdest}}, $dest);
    }
}

# Local procedure called by Tran. Replaces in(x) predicate with correct
# form of signal test.
sub InTransform
{
    my $in = shift;

    while ($in =~ /in\(([A-z0-9]+)\)/) {
        my $ins = $1;
        my $cs = Context->State2CS($ins);
        my $instr = "ins_$cs=st'($ins)";
        $in =~ s/in\(([A-z0-9]+)\)/$instr/;
    }
    return $in;
}

sub OutputSignals {}

sub LclOutput
{
    my $obj = shift;

    Put Context 0,"\n";
    Put Context 1,"s_$$obj{name}: process\n";
    Put Context 1,"begin\n";
    Put Context 2,"wait until state=st'($$obj{name});\n";
    Put Context 2,"say(\"In state $$obj{name}\");\n";
    if (@{$$obj{noeventguard}} == 0 && @{$$obj{event}} == 0) {
        print "WARNING No transitions from state $$obj{name}\n";
    }
    my $i;
    if (@{$$obj{noeventguard}} > 0) {
# process event-less transitions
        my $first = 1;
        for ($i=0; $i<@{$$obj{noeventguard}}; $i++) {
            if ($first) {Put Context 2,"if $$obj{noeventguard}[$i] then\n"}
            else {Put Context 2,"elsif $$obj{noeventguard}[$i] then\n"}
            $first = 0;
            Put Context 3,
            "state <= st'($$obj{noeventdest}[$i]), null after 1 fs;\n";
# Any message sending goes here....
        }
        Put Context 2,"end if;\n";
    }

# ====> now do regular event driven transitions  <=====
    if (@{$$obj{event}} > 0) {
        Put Context 2,"loop\n";
# generate wait statement for transitions
        Put Context 3,"wait until $$obj{event}[0]";
        if (@{$$obj{event}} > 1) {Put Context 0,"\n"}
#  add -OR- and next event
```

```
        my $i;
        for ($i=1; $i<@{$$obj{event}}; $i++) {
            if ($i>1) {Put Context 0,"\n"}
            Put Context 4,"or $$obj{event}[$i]";
        }
        Put Context 0,";\n";

# Now select which event occurred with an IF stmt. Guards are here too.
        if ($$obj{guard}[0] ne '') {
            Put Context 3,"if $$obj{event}[0] and $$obj{guard}[0] then\n";
        }
        else {
            Put Context 3, "if $$obj{event}[0] then\n";
        }
        Put Context 4,"state <= st'($$obj{dest}[0]), null after 1 fs;\n";
        Put Context 4,"exit;\n";
        for ($i=1; $i<@{$$obj{event}}; $i++) {
            my $gc;
            if ($$obj{guard}[$i]) {$gc = "and $$obj{guard}[$i]"}
            else {$gc = ''}
            Put Context 3,"elsif $$obj{event}[$i] $gc then\n";
            Put Context 4,"state <= st'($$obj{dest}[$i]), null after 1 fs;\n";
            Put Context 4,"exit;\n";
        }
        Put Context 3,"end if;\n";
        Put Context 2,"end loop;\n";
    }
    Put Context 1,"end process;\n";
}

1;
```

Appendix H

SPIN LCS Files

The listings below contain the Perl source code for Hydra for the Promela translations. The Hydra Class Library consists of the files (marked in the listing) CCState.pm, Context.pm, Init.pm, Log.pm, State.pm, CState.pm, History.pm, Join.pm, and Object.pm.

```
#!/usr/bin/perl
{
    use Yacc;
    use CState;
    use CCState;
    use State;
    use Object;
    use Context;
    use History;
    use Join;
    use Init;
    use Log;

    $| = 1;
#    Register Log(State);
    $tree = Yacc->Parse("$ARGV[0].xyz");
    print "parse complete\n";
    Context->OutFile("$ARGV[0].pr");
    Model($tree);
    Context->Output;
}

sub Model
{
    my ($tree) = @_;
    my $stmt = $$tree{modelbody}{modelstmtlist};
    my $ent;
    foreach $ent (@$stmt) {   # Each one of these is a hash
        if ($$ent{type} eq 'Enum') {
            my @list;
            $$ent{list} =~ s/^\((.+)\)$/$1/;
            @list = split(/,/, $$ent{list});
            Context->Enum($$ent{ID}, @list);
        }
        elsif ($$ent{type} eq 'Driverfile') {
            Context->DriverFile($$ent{ID});
        }
        elsif ($$ent{type} eq 'Object') {
            Object($ent);
        }
        else {die "bad type $$ent{type}"}
    }
}

sub Object
{
    my ($tree) = @_;

    $$tree{type} eq 'Object' || die "this is not an Object node";

    my $objref = new Object($$tree{ID});
    my $init;

    my $body = $$tree{objectbody}{objectstmtlist};
    my $ent;
    my %nametoref;
    my @joinlist;
    foreach $ent (@$body) {
        if ($$ent{type} eq 'Signal') {
            $objref->Signal($$ent{name},$$ent{sigtype});
        }
        elsif ($$ent{type} eq 'CState') {
            $nametoref{$$ent{ID}} = CState($ent, $objref);
        }
        elsif ($$ent{type} eq 'CCState') {
             my $ref = CCState($ent, $objref);
# This used to save a ref to the CCState, but it needs to save refs to
# the CStates in the CCState, hence the following code. Fetch members,
# then get their names. We still need a ref to the CCState container.
             $nametoref{$$ent{ID}} = $ref;
             my @members;
             @members = $ref->GetMembers;
             my $thing;
             foreach $thing (@members) {
                 my $name = $thing->GetName;
                 $nametoref{$name} = $thing;
             }
        }
        elsif ($$ent{type} eq 'State') {
            $nametoref{$$ent{ID}} = State($ent, $objref);
        }
        elsif ($$ent{type} eq 'Init') {
            if ($init) {
                print "**** $$tree{id} has more than one init state\n";
            }
            $init = $$ent{ID};
        }
        elsif ($$ent{type} eq 'Join') {
            push(@joinlist, [$$ent{ID},$$ent{from},$$ent{to}]);
        }
        elsif ($$ent{type} eq 'InstVar') {
            $objref->InstVar($$ent{vtype}, $$ent{var}, $$ent{initval});
        }
        else {die "bad type $$ent{type}"}
    }

    print "Object ref keys\n";
    foreach $ent (keys %nametoref) {
        print "$ent is a ",ref($nametoref{$ent}),"\n";
    }

    foreach $ent (@joinlist) {
        print "Making Join $$ent[0] from $$ent[1] to $$ent[2]\n";
        new Join($objref, $$ent[0],
                 $nametoref{$$ent[1]}, $nametoref{$$ent[2]});
    }

    Put Log "Object $objref->Name init is $init";
    if ($init) {new Init($objref, $nametoref{$init})}
}

sub CState
{
    my ($tree, $parent) = @_;

    $$tree{type} eq 'CState' || die "node is not a CState";

    my $cstateref = new CState($parent, $$tree{ID});
    my $myname = $$tree{ID};

    my $body = $$tree{body}{cstatestmtlist}; # List of stmts
    my %nametoref;
    my ($ent, $init, $history);
    foreach $ent (@$body) {
        if ($$ent{type} eq 'State') {
            my $r = State($ent, $cstateref);
            $nametoref{$$ent{ID}} = $r;     # save ref by name
        }
        elsif($$ent{type} eq 'CState') {
            my $r = CState($ent, $cstateref);
            $nametoref{$$ent{ID}} = $r;     # save ref by name
        }
        elsif ($$ent{type} eq 'Init') {
            if ($init) {
                print "**** $$tree{id} has more than one init state\n";
            }
            $init = $$ent{ID};
        }
        elsif ($$ent{type} eq 'History') {
            if ($history) {
                print "**** $$tree{id} has more than one History state\n";
            }
            $history = $$ent{ID};
        }
        elsif ($$ent{type} eq 'CCState') {
            my %names;
            %names = CCState($ent, $objref);
            my $key;
            foreach $key (keys %names) {$nametoref{$key} = $name{$key}}
        }
        else {die "bad type $$ent{type}"}
    }
# Now to Init states
    print "CState ref keys\n";
    foreach $ent (keys %nametoref) {
        print "$ent is a ",ref($nametoref{$ent}), "\n";
    }
    if ($history) {
# This differs from VHDL, where we pass state name instead of state ref
        new History($cstateref, $nametoref{$history});
    }
    if ($init) {
        if (!exists($nametoref{$init})) {
            print "For CState ",$cstateref->GetName, " $init not found\n";
            exit(1);
        }
        new Init($cstateref, $nametoref{$init});
    }
    else {"Warning: There is no INIT state for $$tree{ID}\n"}
    return $cstateref;  # return the ref to this CState
}

sub CCState
{
# The only legal statements for this container is CState. Transitions (such
# as inits) may want to transition to one of our components, which starts the
# threads running. Therefore, we pass back a hash of CState refs indexed by
# CState name.
    my ($tree, $parent) = @_;

    my $ccstate = new CCState($parent, $$tree{ID});

    my $body = $$tree{body}{ccstatestmtlist}; # List of stmts
    my $ent;
    foreach $ent (@$body) {
        if ($$ent{type} eq 'State') {
            State($ent, $ccstate);
        }
        elsif($$ent{type} eq 'CState') {
            CState($ent, $ccstate);
        }
        else {die "bad type $$ent{type}"}
    }
    return $ccstate;
}

sub State
{
    my ($tree, $parent) = @_;

    $$tree{type} eq 'State' || die 'Node is not a State node';
    my $stateobj = new State($parent, $$tree{ID});

    my $body = $$tree{statebody}{statestmtlist};
    my $ent;
    foreach $ent (@$body) {
        if ($$ent{type} ne 'Trans') {
            die "something other than Trans in a state";
        }
        my $tran;
        ($tran) = $$ent{tran} =~ /^"(.+)"$/;
        $stateobj->Tran($tran, $$ent{dest});
    }
    return $stateobj;   # return the ref to this state
}
#!/usr/bin/perl
package CCState;

# This object cooperates with Join closely. There has to be at least
# one Join to match each CCState. There can be more than one Join.

sub new
{
    my ($class, $parent, $name) = @_;

    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    $$obj{name} = $name;
    $$obj{objref} = $parent->GetObjRef;
    $parent->AddState($obj);
    $$obj{states} = [];
    return $obj;
}

sub GetObjRef
{
# return a ref to the object. I don't know where it is but my parent does.
# Eventually, the 'parent' IS the object.
    my $obj = shift;
    return $$obj{objref}
}

sub GoDown
{
# Parents call this routine looking for a dest state. We don't own any
# states except member CStates, we delegate down right away.

    my ($obj, $dest) = @_;

# Look down. Everything here has to be a CState
    foreach $ent (@{$$obj{states}}) {
        if ($ent->GoDown($dest)) {return 1}  # one of my children has it
    }
# Looking up makes no sense because parent called me in the first place...
    return 0;
}

sub GoUp
{
# My children call me here looking for dest state. We have no local simple
# states. We look down, then up as before.


    my ($obj, $dest, $caller) = @_;
# Look at our local CStates. No runtab required because Join will pick
# this up. The choices of designated next states from the children
# are limited (to Joins, to be precise)
    if ($$obj{owndest}{$dest}) {
        return ['lclcs'];   # Someone lower than my child will get this
    }

# Look down, cycle thru children
    foreach $ent (@{$$obj{states}}) {
        if ($caller == $ent) {next}
        if ($ent->GoDown($dest)) {return 1}
    }

# Look at parent.
    if ($$obj{parent}->GoUp($dest, $obj)) {
        return ['up'];   # return up... still no runtab needed
    }

# not found anywhere....
    return ();
}

sub AddState
{
    my ($obj, $state) = @_;

    push(@{$$obj{states}}, $state);
    my $name = $state->GetName;
    $$obj{owndest}{$dest} = 'lclcs';
```

```
}

sub AddJoin
{
    my ($obj, $join) = @_;

    Put Log $join->GetName;
    push(@{$$obj{joins}}, $join);
}

sub GetMembers
{
    my ($obj) = @_;

    return @{$$obj{states}};
}

sub GetName
{
    my ($obj) = @_;

    my $ent;
    my @list;
    foreach $ent (@{$$obj{states}}) {push(@list, $ent->GetName)}
    return @list;
}

sub PreProcess
{
    my ($obj) = @_;

    my $ent;
    Context->AddEnum($$obj{objref}->GetName, 'none');
    foreach $ent (@{$$obj{states}}) {$ent->PreProcess}
}

sub LclOutput
{
    my ($obj) = @_;

    my $ent;
    my @names;
    Put Log "LclOutput called";
    foreach $ent (@{$$obj{states}}) {
        my $name = $ent->GetName;
        push(@names, $name);
        Put Context 0,"to_$name:";
    }
    Put Context 1,"atomic {";
    foreach $ent (@names) {Put Context 1,"${ent}_pid = run $ent(none);"}
    Put Context 1,"}";
#  wait??eval(cp1_pid),cp1_code;
    foreach $ent (@names) {
        Put Context 1,"wait??eval(${ent}_pid),${ent}_code;";
    }
# Time to do Join stuff.
    if (@{$$obj{joins}}) {
        Put Context 1,"do";
        foreach $ent (@{$$obj{joins}}) {
            $ent->LclOutput;   # write out the correct if - dispatch
        }
        Put Context 1,"od;";
    }
    else {
        print "**** Warning: no join for CCState $$obj{name}\n";
        Put Context 1,"assert(0);";
    }
# That's it. We're done.
}

sub Output
{
    my ($obj) = @_;

    my $ent;
# I have nothing to write out, but my CStates must be written.
    foreach $ent (@{$$obj{states}}) {
        Put Log  "calling output for ", $ent->GetName, "\n";
        $ent->Output;
    }
}
1;
#!/usr/bin/perl
package CState;
use Object;

# Resolve dests, dispatch tables and run tables:
# Each simple state has to know where it's destination state is for each
# transition. There are four possibilities:
# 1. Local transition. goto state; is the contruct
# 2. Local transition to composite state. goto to_cstate; is contruct
# 3. Transition down into composite state. m=state; goto to_cstate;
# 4. Transition up to parent composite state. wait??eval(pid),m is construct

# In addition, we need essentially a routing from a src to a dest. So each
# CState or Object must keep a route (lcl, lclcs, up, down) for each dest
# that might pass thru.

# This is done by passing control to each simple state at 'Resolve'
# where it calls it's parent 'Find'. Find looks locally for simple states and
# composite states (returns 'lcl' and 'lclcs'), then looks down into it's
# composite states via 'GoDown'. A hit here returns 'down' and the name
# of the CState down. Finally, it looks up via GoUp. A hit here returns
# 'up'. Now the simple states know all the proper constructs to get to the
# dest.

# CStates and Object must also keep track of what it returned for the dest.
# For an 'up', the 'run cstate' construct higher up is going to get a return
# state for the next dest, so a table (runtab) is required to dispatch the
# next dest properly. For 'down', the receiving CState must have an initial
# dispatch tab so control can be passed as required. The choices are as
# before, locally, locally-to-cstate, up (some more), or down (some more).
# For up and down, any given cstate may only be a passthru for a dest several
# levels deep.

# The 'run cstate' construct is contructed by LclOutput, so it has to know
# about what to expect back. The CState itself must know about incoming
# states in Output, because that is where the body is contructed.

# So State must tell it's parent LclOutput about possible returns, and
# it' child Output about dispatch table entries. State calls BldTab of parent
# with 'up', or calls child 'down'. Each CState propogates the call according
# it's route tab entry until 'lcl' or 'lclcs' is found.

sub BEGIN
{
    %CS = (CState=>1, CCState=>1);
}


sub new { my ($class, $parent, $name) = @_;

    my $obj = {};
    bless $obj;
    $$obj{name} = $name;
    $$obj{parent} = $parent;
    $$obj{objref} = $parent->GetObjRef; # See GetObjRef below
    $$obj{objref}->AllStates($obj);     # add to list of states
    $$obj{states} = [];   # reference to states we own
    $$obj{owndest} = {};  # text names of states we own
    $$obj{inittab} = {};  # list of state info we'll get called with
    $$obj{runtab} = {};   # list of states that may come back
    $parent->AddState($obj); # add myself to parent's list
    return $obj;
}

sub GetName
{
    my ($obj) = @_;

    return $$obj{name};
}

sub HistoryPresent
{
    my ($obj) = @_;
    return exists($$obj{historypresent});
}


sub GetObjRef
```

```
{
# return a ref to the object. I don't know where it is but my parent does.
# Eventually, the 'parent' IS the object.
    my $obj = shift;
    return $$obj{objref}
}

sub AddState
{
# called by simple state process to include state in my list of states
    my ($obj, $state) = @_;

    push(@{$$obj{states}}, $state);
    my $name;
    $name = $state->GetName;
# lcl means simple local state, lclcs means composite state of some kind
    $$obj{owndest}{$name} = ref($state) eq 'State' ? 'lcl' : 'lclcs';
    if (ref($state) eq 'History') {$$obj{historypresent} = $state}
}

sub GetMembers
{
# This method produces a list of refs of the states we own.
    my ($obj) = @_;

    return @{$$obj{states}};
}

sub Find
{
# This is called my the simple state we own to resolve a dest
    my ($obj, $dest) = @_;

    Put Log "Find for cstate $$obj{name} called. Looking for $dest";
# Look locally first.
    if (exists($$obj{owndest}{$dest})) {return $$obj{owndest}{$dest}}

# Next, look down thru CState nodes
    foreach $ent (@{$$obj{states}}) {
        if (!$CS{ref($ent)}) {next}  # Skip all non-CStates
        if ($ent->GoDown($dest)) {  # parent -> child call
            $$obj{runtab}{$dest} = ['down', $ent->GetName];
            Put Log "Find in $$obj{name} $dest is down";
            return ('down', $ent->GetName);
        }
    }

# Finally look up a level; child -> parent
    if ($$obj{parent}->GoUp($dest, $obj)) {
        $$obj{runtab}{$dest} = ['up'];
        Put Log "Find in $$obj{name} $dest is up";
        return ('up');
    }
    die "can't find $dest";
```

```
}

# GoUp returns the context in which it finds the dest, but for LclOutput
# in States and CStates, this is how the label will look becuase that is
# where the LclOuput code will be... in the parent. Only GoUp calls GoUp,
# GoUp also records the dest context the call to the parent returns.

# GoDown just returns true if it finds the label down, but the 'called'
# GoDown should expect this dest coming into the code built by Output...
# the body of the CState. So GoDown records inittab entries

# The process starts with a call to GoUp from a simple State coming from
# ResolveDest.

sub GoDown
{
# Parents call this routine looking for a dest state.
    my ($obj, $dest) = @_;

    Put Log "GoDown of $$obj{name} called, looking for $dest";

# First, look locally
    if (exists($$obj{owndest}{$dest})) {
        $$obj{inittab}{$dest} = [$$obj{owndest}{$dest}];
        return 1;
    }
# Look down
    foreach $ent (@{$$obj{states}}) {
        if (!$CS{ref($ent)}) {next}
        if ($ent->GoDown($dest)) {
            my $csname = $ent->GetName;
            $$obj{inittab}{$dest} = ['down', $csname];
            return 1;
        }
    }
# Looking up makes no sense because parent called me in the first place...
    return 0;
}

sub GoUp
{
# My children call me here looking for dest state. I will not call
# the child that called me, hence the $child parm.
    my ($obj, $dest, $child) = @_;

    Put Log "GoUp of cstate $$obj{name} called, looking for $dest";

# Look locally. Can't be me; no way to write this.
    if (exists($$obj{owndest}{$dest})) {
        Put Log "in $$obj{name} $dest is $$obj{owndest}{$dest}";
        return [$$obj{owndest}{$dest}];
    }
# Look down, cycle thru children
    my $ent;
    foreach $ent (@{$$obj{states}}) {
        if ($ent == $child) {next}  # don't loop back down to same object
        if (!$CS{ref($ent)}) {next}
        if ($ent->GoDown($dest)) {
            my $csname = $ent->GetName;
            $$obj{runtab}{$dest} = ['down', $csname];
            Put Log "in $$obj{name} dest is down";
            return ['down', $csname];
        }
    }

# Look up a level to my parent. Return value gives context parent see
# dest (if any), so we save this because this is where our LclOutput code
# will be constructed.

    if ($ref = $$obj{parent}->GoUp($dest, $obj)) {
        Put Log "$$obj{name} got back $$ref[0], $$ref[1]";
        $$obj{runtab}{$dest} = $ref;
        Put Log "got back $$ref[0], $$ref[1] from ",$$obj{parent}->GetName;
        return ['up'];
    }
# not found anywhere....
    return ();
}

sub PreProcess
{
    my ($obj) = @_;
# Our parent calls us here when all the input is parsed. We need to do
# any final setups, like resolving dests, making CCState members appear
# local, etc. This is called before any output routine.

# First, we need to make children of CCStates we own look local to us.
# CCState->GetName returns a list of CStates it owns.
    my $init;
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'Init') {$init = 1}
        if (ref($ent) eq 'CCState') {
            foreach $name ($ent->GetName) {$$obj{owndest}{$name} = 'lclcs'}
        }
        $ent->PreProcess;
    }
    if (!$init) {
        print "**** Warning CState $$obj{name} has no initial state\n";
    }
    my $key;
    Put Log "Here is $$obj{name} owndest:";
    foreach $key (keys %{$$obj{owndest}}) {
        Put Log "state=$key, value=$$obj{owndest}{$key}";
    }
}

sub LclOutput
{
    my ($obj) = @_;
    my $name = $$obj{name};

    Put Context 0, "/* Link to composite state $name */";
    Put Context 0, "to_$name:${name}_pid = run $name(m);";
    Put Context 1,"wait??eval(${name}_pid),m;";
# Now do runtab
    Put Log "building runtab for ",$ent->GetName;
    Put Context 1,"if";
    foreach $ent (keys %{$$obj{runtab}}) {
        Put Log "dest=$ent $$obj{runtab}{$ent}[0], $$obj{runtab}{$ent}[1]";
        PutNR Context 1,":: m == st_$ent -> ";
        if ($$obj{runtab}{$ent}[0] eq 'lcl') {
            Put Context 0,"goto $ent;";
        }
        elsif ($$obj{runtab}{$ent}[0] eq 'lclcs') {
            Put Context 0,"goto to_$ent;";
        }
        elsif ($$obj{runtab}{$ent}[0] eq 'up') {
            Put Context 0,"wait!_pid,st_$ent; goto exit;";
        }
        elsif ($$obj{runtab}{$ent}[0] eq 'down') {
            Put Context 0,"m = st_$ent; goto to_$$ent[2];";
        }
    }
    Put Context 1,"fi;";
}

sub Output
{
    my $obj = shift;

# ----- Initial  stuff ------
# Categorize states we own.
    my @csnames;
    my @ccnames;
    my $init;
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'CState') {push(@csnames, $ent->GetName)}
        elsif (ref($ent) eq 'CCState') {push(@ccnames, $ent->GetName)}
        elsif (ref($ent) eq 'History' || ref($ent) eq 'Init') {$init = $ent}
    }

    Put Context 0,"proctype $$obj{name}(mtype state)";
    Put Context 0, "{";
# declare pid vars for runs and waits.
    my $dcl;
    if (@csnames > 0) {foreach $ent (@csnames) {$int .= "${ent}_pid,"}}

# declare pid vars for CCState runs
    if (@ccnames > 0) {
        foreach $ent (@ccnames) {
            $int .= "${ent}_pid,";
            $mtype .= "${ent}_code,";
        }
        chop $mtype;
        Put Context 0,"mtype={$mtype};";
    }

    Put Log "int=$int";
    if ($int) {
        $int =~ s/^(.+),$/$1/;
        Put Context 0,"int $int;";
    }
    Put Context 0,"mtype m;";
    Put Context 0,"int dummy;";

# build initial dispatch table
    if (keys %{$$obj{inittab}}) {
        Put Context 1,"if";
        foreach $ent (keys %{$$obj{inittab}}) {
            if ($$obj{inittab}{$ent}[0] eq 'lcl') {
                Put Context 1,":: state == st_$ent -> goto $ent";
            }
            elsif ($$obj{inittab}{$ent}[0] eq 'lclcs') {
                Put Context 1,":: state == st_$ent -> goto to_$ent";
            }
            elsif ($$obj{inittab}{$ent}[0] eq 'down') {
                Put Context 1,
                ":: state == st_$ent -> goto to_$$obj{inittab}{$ent}[1]";
            }
            else {die "bad initab entry"}
        }
# Picks up default initial state, or dies (to catch translation errors)
        if ($init) {Put Context 1,":: else -> skip  /* drop to init state */"}
        else {Put Context 1, ":: else -> assert(0) /* no init state - " .
              "die if bad state */"}
        Put Context 1,"fi;";
    }
# Do init state or history setup
    if ($init) {$init->LclOutput}

# now put out code for states and cstates.
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'Join') {next} # Skip Joins. CCState will handle
# We already did init and history
        if (ref($ent) eq 'Init' || ref($ent) eq 'History') {next}
        $ent -> LclOutput;
    }
# Wind up this CState
    Put Context 0,"exit:skip";
    Put Context 0,"}";
    Put Context 0;
    Put Context 0;

# Output body of composite states we own
    foreach $ent (@{$$obj{states}}) {$ent->Output}
}
```

```
1;
#!/usr/bin/perl
package Context;
use Object;

sub OutFile
{
# Save the name of the output file.
    my ($class, $file) = @_;
    $OUTFILE = $file;
}

sub DriverFile
{
# save the driver file name.
    my ($class, $file) = @_;

    $DRIVERFILE = $file;
}

sub Enum
{
# I think I can ignore Enums. We get this info from transitions
    return;
}

sub AddEnum
{
# save the enums we need globally. This is the defintion of mtype. The
# objname is passed just so we can do error checking.

   my ($classname, $objname, $enum) = @_;

   my ($class,$f,$line) = caller;
   Put Log "enum $enum added. $class($line) called me";
   if (exists($enums{$enum}) && $enums{$enum} ne $objname) {
       print "**** Object $objname adding Enum $enum already added by ",
       "$enums{$enum}\n";
       return;
   }
   $enums{$enum} = $objname
}

sub AddObj
{
# keep a list of objects in the model
    my ($class, $obj) = @_;
    push(@Objects, $obj);
}

sub Open
{
    my ($class, $file) = @_;
    open(OUT, ">$file");
```

```
}

sub Output
{
# This is where the good stuff starts. We need to do the following things:
# 1. write out global mtypes enums
# 2. write out global mtypes for associations
# 3. grab the driverfile and put if out.
# 4. cycle thru the objects in the model calling each's Output
    open(OUT, ">$OUTFILE");

    Put Context 0,"chan evq=[10] of {mtype,int};";
    Put Context 0,"chan evt=[10] of {mtype,int};";
    Put Context 0,"chan wait=[10] of {int,mtype};";

    my $ent;
    foreach $ent (@Objects) {
        $ent->PreProcess;
    }

# Enums
    my $list = join(', ', keys(%enums));
    Put Context 0, "mtype=\{$list\};";

    foreach $ent (@Objects) {
        $ent->GlobalOutput;
    }
    Put Context 0,"chan t=[1] of {mtype};";
    Put Context 0,"mtype={free};";

# Driverfile:
    open(DRV, "$DRIVERFILE.pr") || die "can't open driverfile";
    Put Context 0;
    Put Context 0, "/* User specified driver file  */";
    while ($l = <DRV>) {
        chop $l;
        Put Context 0,$l}
    Put Context 0;
    close(DRV);

# Cycle thru objects and get output

    foreach $ent (@Objects) {
        $ent->Output;
    }
    print OUT <<"EOF";

/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
        mtype type;
        int pid;

        atomic {
```

```
        do
        :: evq??[eval(msg),pid] ->
           evq??eval(msg),pid;
           evt!msg,pid;
           do
           :: if
              :: evq??[type,eval(pid)] -> evq??type,eval(pid)
              :: else break;
              fi
           od
        :: else -> break
        od}
exit:   skip
}
EOF
    close(OUT);
}

sub MakeOut
{
    my ($lvl, $txt) = @_;

    my ($out,$margin);
    my ($label, $rest);
    ($label,$rest) = $txt =~ /^([A-z0-9]+:)(.*)/s;
    $margin = length($label) > $LEFTMARGIN ? length($label)+1 : $LEFTMARGIN;
    if ($label) {$out = $label . ' ' x ($margin-length($label)) . $rest}
    else {$out = ' ' x $tab[$lvl] . $txt}
    return $out;
}

sub cut
{
    my ($str, $len) = @_;


    my ($i,$p,$lasti,$ret,$rest,$inq);
    for ($i=0; $i<length($str); $i++) {
        $inq = substr($str,$i,1) eq '"' ? ++$inq%2 : $inq;
        if ($inq) {next}
        if (substr($str,$i,1) eq ' ') {
            if ($i+1 > $len) {
                $p = $lasti ? $lasti : $i;
                $ret = substr($str,0,$p);
                $rest = substr($str,$p+1);
                return ($ret, $rest);
            }
            else {$lasti = $i}
        }
    }
    if ($lasti) {return (substr($str,0,$lasti), substr($str,$lasti+1))}
    else {return ($str, '')}
}
```

```
sub Put
{
# Put (lvl, @list) write a string out indented according to level. If the
# string has a label, is is placed at the left. If the string is over
# 72 chars, it is split to a next line indented at lvl+1.

# If Put, lvl=0 follows a PutNR, the string is placed on the right
# end of the previous PutNR and if folded, folded at the PutNR lvl+1

# If multiple Puts are req'd, do this: PutNR lvl,text for the first with
# lvl set to what is req'd. The do PutNR 0,text as many times as needed,
# then Put 0,text.
    my ($class, $lvl, @list) = @_;

    my $out = MakeOut($lvl, join('', @list));
    $out = $STR ? $STR . $out : $out;
    $lvl = !$lvl && $LASTLVL ? $LASTLVL : $lvl;
    my $lvl2 = $lvl;
    my $x;
    if (length($out) > 72) {
        if ($out =~ /^ +(.+)/) {$out = $1}
        while (length($out) > 72-$tab[$lvl2]) {
            ($x,$out) = cut($out, 72-$tab[$lvl2]);
            print OUT ' ' x $tab[$lvl2], "$x\n";
            $lvl2 = $lvl+1;
        }
        if ($out) {print OUT ' ' x $tab[$lvl2], "$out\n"}
    }
    else {print OUT "$out\n"}
    $STR = '';
    $LASTLVL = 0;
}

sub PutNR
{
    my ($class, $lvl, @list) = @_;

    if (!$STR) {
        $LASTLVL = $lvl;
        $STR = MakeOut($lvl, join('', @list));
    }
    else {$STR .= MakeOut(0, join('', @list))}
    return;
}

sub Seq{return $SEQ++}

sub BEGIN
{
    $LEFTMARGIN = 10;
    $INC = 3;
    $tab[0] = 0;
    $tab[1] = $LEFTMARGIN;
    my $i = $LEFTMARGIN + $INC;
    my $j = 2;
    while ($i < 72) {
        $tab[$j++] = $i;
        $i += $INC;
    }
    $SEQ = 1;
}
1;
#!/usr/bin/perl
package History;

sub new
{
# Both parent and start are refs.
    my ($class, $parent, $start) = @_;

    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    $$obj{start} = $start;
    $parent->AddState($obj);
    $$obj{objref} = $parent->GetObjRef;
    $$obj{objref}->AllStates($obj);  # We want a call to ResolveDest later
    return $obj;
}

sub GetName {return 'History'}

sub PreProcess
{
    my ($obj) = @_;

    my $ent;
    foreach $ent ($$obj{parent}->GetMembers) {
        if ($ent == $obj) {next}  # skip self
        Context->AddEnum($$obj{objref}->GetName, "st_" . $ent->GetName);
    }
# declare the global var to hold history.
    my $pn = $$obj{parent}->GetName;
    $$obj{objref}->AddMtype("H_$pn", "st_" . $$obj{start}->GetName);
}


sub Output {return}

sub ResolveDest
{
# This is called while states are resolving dests. We use the call to
# tell the object to declare the history mtype globally
    my ($obj) = @_;

    my $name = $$obj{parent}->GetName;
    my $dest = $$obj{start}->GetName;
    if (ref($$obj{start}) eq 'CState' ||ref($$obj{start}) eq 'CCState') {
        $$obj{objref}->AddMtype("H_$name", "to_$dest");
    }
    else {$$obj{objref}->AddMtype("H_$name", $dest)}
}

sub LclOutput
{
    my ($obj) = @_;

# If the intial state is CState, we need a 'to_state' type label.
    my $name = $$obj{start}->GetName;
    my @states = $$obj{parent}->GetMembers;
    my $csname = $$obj{parent}->GetName;
    my $name;
    Put Context 0,"/* History pseduostate construct  */";
    Put Context 1,"if";
    foreach $ent (@states) {
        if ($ent == $obj) {next}  # But not me....
        $name = $ent->GetName;
        if (ref($ent) eq 'CState') {
            Put Context 1,":: H_$csname == st_$name -> goto to_$name;";
        }
        else {Put Context 1,":: H_$csname == st_$name -> goto $name;"}
    }
    Put Context 1,"fi;";
}
1;
#!/usr/bin/perl
package Init;

sub new
{
# Both parent and start are refs.
    my ($class, $parent, $start) = @_;

    Put Log "parent isa ",ref($parent), " start isa ",ref($start);
    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    if (ref($start) ne 'State' && ref($start) ne 'CState' &&
        ref($start) ne 'CCState') {
        print "**** Init for ",$parent->GetName," is undefined\n";
        exit(1);
    }
    $$obj{start} = $start;
    $parent->AddState($obj);
    return $obj;
}

sub GetName {return 'Init'}

sub PreProcess {return}

sub Output {return}
sub LclOutput
{
    my ($obj) = @_;

    Put Log "Doing Init for ", $$obj{parent}->GetName;
# If the intial state is CState, we need a 'to_state' type label.
    my $name = $$obj{start}->GetName;
    Put Context 0, "/*  Init state      */";
    if (ref($$obj{start}) eq 'CState') {
        Put Context 1,"goto to_$name;";
    }
    else {Put Context 1,"goto $name;"}
}
1;
#!/usr/bin/perl
package Join;

use CCState;

sub new
{
# All of these are refs except $name, which is my join name (equiv to
# state name)
    my ($class, $parent, $name, $ccstate, $dest) = @_;

    my $obj = {};
    bless $obj;
    $$obj{name} = $name;
    $$obj{dest} = $dest;
    $$obj{parent} = $parent;
    $$obj{objref} = $parent->GetObjRef;
    $$obj{objref}->AllStates($obj);   # tell our object we're here.
    $parent->AddState($obj);
    $$obj{ccstate} = $ccstate;
    $ccstate->AddJoin($obj);         # tell CCState about our join.
    return $obj;
}

sub GetName
{
    my ($obj) = @_;

    return $$obj{name};
}

sub ResolveDest
{
# Need to resolve the dest here just like we do for states, except there is
# only one.
    my ($obj) = @_;

    Put Log "Resolve called for join $$obj{name}";
    my ($dir, $cstate) = $$obj{parent}->Find($$obj{dest}->GetName);
    Put Log "$$obj{name} got back $dir, $cstate";
    if (!$dir) {die "Join can't find dest $$obj{dest}"}
    $$obj{desttype} = $dir;
    $$obj{cstate} = $cstate;  # except for 'down', this is null
}

sub PreProcess {return}

sub Output {return}

sub LclOutput
{
    my ($obj) = @_;
    ($class, $f, $line) = caller;
    Put Log "I was called by $class($line)";
# Get a list of all the CStates in the CCState
    my @csnames = $$obj{ccstate}->GetName;
# Here is what I am building:
#        :: cp1_code == st_join && cp2_code == st_join -> goto state
    PutNR Context 1,":: ";
    my $dest = $$obj{dest}->GetName;
    Put Log "dest=$dest, type=",ref($$obj{dest});
    my $first = 1;
    foreach $ent (@csnames) {
        if ($first) {
            PutNR Context 0, "${ent}_code == st_$$obj{name}";
            $first = 0;
        }
        else {PutNR Context 0," && ${ent}_code == st_$$obj{name}"}
    }
    PutNR Context 0, " -> ";

# Now determine what kind of dispatch we need.
    write_tran($$obj{desttype}, $dest, '');
}

sub write_tran
{
    my ($desttype, $dest, $cstate) = @_;
# This is a local routine to write out the right kind of transition.

    Put Log " WriteTran desttype=$desttype, dest=$dest, cstate=$cstate";
# local state transition, use goto
    if ($desttype eq 'lcl') {
        Put Context 0,"goto $dest;";
    }
    elsif ($desttype eq 'lclcs') {
        Put Context 0,"goto to_$dest;";
    }
# transition to state up somewhere (higher level)
    elsif ($desttype eq 'up') {
        Put Context 0,"wait!_pid,st_$dest; goto quit;";
    }
# transition to lower CState
```

```
    elsif ($desttype eq 'down') {
        Put Context 0,"m = st_$dest; ",
        "goto to_$cstate;";
    }
# whoops! we have a problem here....
    else {die "destination <$desttype> not typed"}
}
1;
#!/usr/bin/perl
package Log;

sub Put
{
    my($class, @m) = @_;

    ($class, $f, $line) = caller;
    if ($track{$class}) {
        print "$class($line) ". join(' ', @m) . "\n";
    }
}

# Track Log(pkg, pkg, ...)
# This routine accepts package names for log output.
# If Register is not called with the package name, it is not logged

sub Register
{
    my($class, @names) = @_;

    my $name;
    foreach $name (@names) {
        $track{$name} = 1;
    }
}

1;
#!/usr/bin/perl
package Object;

sub BEGIN
{
    %CS = (CState=>1, CCState=>1);
}

sub new
{
    my ($class, $name) = @_;

    my $obj = {};
    $$obj{allstates} = [];   # all states in object at any level
    $$obj{states} = [];      # My immediate local states
    $$obj{enums} = {};       # global enums we're asked to declare
    $$obj{name} = $name;
    Context->AddObj($obj);  # add me to the context (model)
    bless $obj;
    return $obj;
}

sub GetName
{
    my ($obj) = @_;
    return $$obj{name};
}

sub GetObjRef
{
# I am the top (the object) so this is the reference everyone wants
    my $obj = shift;
    return $obj;
}

sub Signal
{
# Save signal we expect from other objects. Signal may have int parm.
# Signals are named objname_signame, but this is done in the output
# routines.
    my ($obj, $name, $type) = @_;
# mark this event as a signal and save parm var if there is one.
    $$obj{signal}{$name} = $type;
    Put Log "registering signal $name type $type";
    Context->AddEnum($$obj{name}, $name);  # add the signal name
}

sub IsASignal
{
# Determine is an event is a object->object signal
    my ($obj, $event) = @_;

    return exists($$obj{signal}{$event});
}

sub GetSignalVar
{
# detetmine is obj->obj signal has a parm
    my ($obj, $event) = @_;

    return $$obj{signal}{$event};
}

sub InstVar
{
    my ($obj, $type, $var, $initval) = @_;

    $$obj{instvar}{$var} = [$type, $initval];
}

sub HistoryPresent
{
    my ($obj) = @_;
    return exists($$obj{historypresent});
}

sub AllStates
{
# Every simple state everywhere is recorded here. We cycle thru them later to
# resolve dests on trans. Composite states don't have dests
    my ($obj, $state) = @_;

    Put Log "AllStates called by ",$state->GetName;
    if (ref($state) ne 'State' && ref($state) ne 'Join') {
        Put Log $state->GetName, " is not simple state or join";
        return;
    }
    push(@{$$obj{allstates}}, $state);
    return $obj;
}

sub AddState
{
# called by state process to include state in MY list of states
# This differs from AllStates by only having the states in the top
# level, where the former has ALL STATES in the entire object.
    my ($obj, $state) = @_;

    push(@{$$obj{states}}, $state);
    my $name;
    $name = $state->GetName;
# lcl means simple local state, lclcs means composite state of some kind
    $$obj{owndest}{$name} = ref($state) eq 'State' ? 'lcl' : 'lclcs';
    if (ref($state) eq 'History') {$$obj{historypresent} = $state}
}

sub GetMembers
{
# This method produces a list of refs of the states we own.
    my ($obj) = @_;

    return @{$$obj{states}};
}

sub Find
{
    my ($obj, $dest) = @_;

    Put Log "Find for object $$obj{name} called. Looking for $dest";
# Look locally first.
    if (exists($$obj{owndest}{$dest})) {return ($$obj{owndest}{$dest}, '')}
    Put Log "$dest not local";
# Next, look down thru CState nodes
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) ne 'CState') {next}  # Skip all non-CStates
        if ($ent->GoDown($dest)) {  # parent -> child call
```

```
            return ('down', $ent->GetName);
        }
    }
# If not found by now, then it doesn't exist
    die "object $$obj{name} can't find dest $dest";
}

# No GoDown method because I don't have any parents that can call me.
# See CState for description of GoUp and GoDown.

sub GoUp
{
# My children call me here looking for dest state.

    my ($obj, $dest, $caller) = @_;

    Put Log "GoUp of object $$obj{name} called, looking for $dest";

# Look locally
    if (exists($$obj{owndest}{$dest})) { # we own this simple state
        return [$$obj{owndest}{$dest}];  # return our context for $dest
    }

# Look down, cycle thru children
    foreach $ent (@{$$obj{states}}) {
        if ($caller == $ent) {next} # Don't loop back to caller object
        if (!$CS{ref($ent)}) {next}
        if ($ent->GoDown($$obj{$dest})) {
            return ['down', $ent->GetName];
        }
    }
# If not found by now, then it doesn't exist in this object
    print "**** ERROR: object $$obj{name} can't find dest $dest";
    exit(1);
}

sub Assoc
{
# All events flow thru evq and evt channels for now. Since they'll appear
# as events, we can ignore them here. This will work until we need to
# pass values on assoc. Then we'll need legitimate channels.
    return;
}

sub Dcl
{
    my ($obj, $var, $type) = @_;
# Save declared instance var.
    $$obj{ivars}{$var} = $type;
}


sub AddMtype
{
```

```
# This routine allows global delcaration of mtype variables; not definition.
# See AddEnum above for defintion of mtype value.
    my ($obj, $var, $initial) = @_;
    $$obj{mtypes}{$var} = $initial;
}

sub AddWhen
{
    my ($obj, $name, $clause) = @_;

    $$obj{when}{$name} = $clause;
}

sub INTarget
{
    my ($obj, $name) = @_;
# We are being told of a state name that is the target of an IN predicate
# somewhere in the object. We need to make en enum, an enum variable, and
# save state name so it can be queried later.

    Put Log "$name added as IN target";
    if (!exists($$obj{IN})) {
        $obj->AddMtype($$obj{name} . '_state');
        $$obj{IN} = 1;
    }
    Context->AddEnum($$obj{name}, "st_$name");# make sure state name declared
    $$obj{INtarget}{$name} = 1;
}

sub IsAINTarget
{
    my ($obj, $name) = @_;
# return true if $name is a target of an IN predicate
    return exists($$obj{INtarget}{$name});
}

sub PreProcess
{
    my ($obj) = @_;
# Context calls us here when all the input is parsed. We need to do
# any final setups, like resolving dests, making CCState members appear
# local, etc. This is called before any output routine.

# First, we need to make children of CCStates we own look local to us.
# CCState->GetName returns a list of CStates it owns. We also call other
# PreProcess methods.
    my $ent;
    my $init;
    my $name;
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'Init') {$init = 1}
        $ent->PreProcess;
        if (ref($ent) eq 'CCState') {
            foreach $name ($ent->GetName) {$$obj{owndest}{$name} = 'lclcs'}
        }
    }
    my $key;
    Put Log "Here is $$obj{name} owndest:";
    foreach $key (keys %{$$obj{owndest}}) {
        Put Log "state=$key, value=$$obj{owndest}{$key}";
    }

# See if there is an Init state
    if (!$init) {
        print "**** Warning: Object $$obj{name} has no Init state\n";
    }

# This is a good time to resolve all destinations
    my $i = @{$$obj{allstates}};
    Put Log "PreProcess called for $$obj{name}. $i states";
    foreach $ent (@{$$obj{allstates}}) {$ent->ResolveDest}
    Put Log "$$obj{name} dests all resolved";
}

sub GlobalOutput
{
    my ($obj) = @_;
# This is the first output routine called. All global declares need to
# written at this time. Exception: Mtype dcls are at the context level
# because objects can share signals (no... move it back here. prefix
# signal with object name.

    my $ent;

# Write out global mtype variable dcls. Some have initializers (history)
    foreach $ent (keys %{$$obj{mtypes}}) {
        if ($$obj{mtypes}{$ent}) {
            Put Context 0, "mtype $ent=$$obj{mtypes}{$ent};";
        }
        else {Put Context 0,"mtype $ent;"}
    }

# write out my instance vars
    Put Log "starting to write instvars in $$obj{name}";
    if (keys %{$$obj{instvar}}) {
        Put Context 0, "typedef $$obj{name}_T {";
        foreach $ent (keys %{$$obj{instvar}}) {
            Put Context 1, "$$obj{instvar}{$ent}[0] $ent;";
        }
        Put Context 1, "}";
        Put Context 0, "$$obj{name}_T $$obj{name}_V;";
    }

# Handle signals
    if (keys %{$$obj{signal}}) {
        Put Context 0,"chan $$obj{name}_q=[5] of {mtype};";
    }
# shared variables for signals...
# WARNING: shared variables may get overwritten.
#     foreach $ent (%{$$obj{signal}}) {
#        if ($$obj{signal}{$ent}) {
#            Put Context 0,"$$obj{signal}{$ent} ${ent}v;"
#            }
#     }

# Parameter channel for signals. Replacement for shared variable.
    foreach $ent (%{$$obj{signal}}) {
        if ($$obj{signal}{$ent}) {
            Put Context 0,"chan ${ent}_p1=[5] of {$$obj{signal}{$ent}};";
        }
    }

    Put Log "global output done in $$obj{name}";
}

sub Output
{
    my $obj = shift;

# ----- Initial  stuff ------
# Categorize states we own.
    my @csnames;
    my @ccnames;
    my ($int, $mtype);
    my $init;
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'CState') {push(@csnames, $ent->GetName)}
        elsif (ref($ent) eq 'CCState') {push(@ccnames, $ent->GetName)}
        elsif (ref($ent) eq 'History' || ref($ent) eq 'Init') {$init = $ent}
    }

    Put Context 0,"proctype $$obj{name}()";
    Put Context 0, "{";
# declare pid vars for runs and waits.
    my $dcl;
    if (@csnames > 0) {
        foreach $ent (@csnames) {$int .= "${ent}_pid,"}
    }

# declare pid vars for CCState runs
    if (@ccnames > 0) {
        if ($int) {$int .= ', '}
        foreach $ent (@ccnames) {
            $int .= "${ent}_pid, ";
            $mtype .= "${ent}_code, ";
        }
        Put Log "mtype is <$mtype>";
        $mtype =~ s/, $//;
        Put Context 0,"mtype $mtype;";
    }
# from csname code above
    if ($int) {
        $int =~ s/, $//;
        Put Context 0,"int $int;";
    }
    Put Context 0,"mtype m;";
    Put Context 0,"int dummy;";

# Debugging output to list to see enums
    foreach $ent (keys %{$$obj{enums}}) {
        Put Context 1,"printf(\"$ent=%d\\n\", $ent);";
    }

# Instance var initializers
    foreach $ent (keys %{$$obj{instvar}}) {
        if ($$obj{instvar}{$ent}[1]) {
            Put Context 1, "$$obj{name}_V.$ent = $$obj{instvar}{$ent}[1];";
        }
    }

# Start when procs
    foreach $ent (keys %{$$obj{when}}) {
        Put Context 1,"run proc$ent();";
    }

# Do init state or history setup
    if ($init) {$init->LclOutput}

# now put out code for states and cstates.
    foreach $ent (@{$$obj{states}}) {
        if (ref($ent) eq 'Join') {next} # Skip Joins. CCState will handle
        if (ref($ent) eq 'History' || ref($ent) eq 'Init') {next}
        $ent -> LclOutput;
    }

# Wind up this Object
    Put Context 0,"exit:skip";
    Put Context 0,"}";
    Put Context 0;

# Output body of composite states we own
    foreach $ent (@{$$obj{states}}) {
        Put Log "object name $$obj{name} outputing type ", ref($ent);
        $ent->Output;
    }
    Put Log "composite states done";

# Output when clause monitors
    foreach $ent (keys %{$$obj{when}}) {
        Put Context 0;
        Put Context 0,"proctype proc$ent()";
        Put Context 0,"{";
        Put Context 1, "do";
        Put Context 2, ":: $$obj{when}{$ent} -> run event($ent)";
        Put Context 1, "od";
        Put Context 0, "}";
    }
    Put Context 0;
}

1;
#!/usr/bin/perl
package State;

use TranYacc;
sub BEGIN
{
    $EVENTCNT = 1;   # This counts no-event transitions without guards.
}

sub new
{
    my ($class, $parent, $name) = @_;

    my $obj = {};
    bless $obj;
    $$obj{parent} = $parent;
    $$obj{name} = $name;
    $$obj{objref} = $parent->GetObjRef;  # get a ref to object we're in.
    $$obj{dest} = [];
    $parent->AddState($obj);
    $$obj{objref}->AllStates($obj);   # tell our object we're here.
    return $obj;
}

sub GetName
{
    my ($obj) = @_;

    return $$obj{name};
}

sub Tran
{
# We parse the transaction. This produces a delimited string of event,guard
# action,msg. Save each in a hash by it's delimiter type. If event=null
# grab a number for a placeholder on the event hash.
    my ($obj, $tran, $dest) = @_;

    my $ret;
    if ($tran) {
        TranYacc->SetObjName($$obj{objref}->GetName); # tell parser object name
        Put Log "transition before parse <$tran>";
        $ret = TranYacc->Parse($tran);
        if (!$ret) {
            print "Bad transition expression: $tran\n";
            exit;
        }
    }
    else {$ret = ''}
    Put Log "tranret=$ret";
    my @list = split(/\#/, $ret);
    my %tab;
    my ($type,$thing);
    while (@list) {
        $type = shift(@list);
        $thing = shift(@list);
        $tab{$type} = $thing;
    }
# Check to see if there IS an event.
    if (!exists($tab{E}) || $tab{E} eq 'NULL') {
        $tab{E} = 'NOEVENT'}   # No event, placeholder.
# Check for 'when' clause and register is there is.
    if (exists($tab{E}) && $tab{E} =~ /^when\((.+)\)$/) {
        my $exp = $1;
        $tab{E} = 'when' . Context->Seq;
        $$obj{objref}->AddWhen($tab{E}, $exp);   # register when clause
    }
# This pattern is unique to IN predicate because of parser. Tell object
# who is the target of the IN predicate. That state needs to build tracking
# code.
    if (exists($tab{G}) && $tab{G} =~ /state == st_([^ ]+)/) {
        my $var = $1;
        $$obj{objref}->INTarget($var);
    }
    push(@{$$obj{event}{$tab{E}}}, [$tab{G}, $tab{A}, $tab{M}, $dest]);
    $$obj{sigvar}{$tab{E}} = $tab{V};
    Put Log "event=$tab{E}";
    Put Log "var=$tab{V}";
    Put Log "guard=$tab{G}";
    Put Log "action=$tab{A}";
    Put Log "msg=$tab{M}";
    Put Log "dest=$dest";
}


sub GetParent
{
# return my parent composite thingy (CState or Object)
    return $$obj{objref};
}



# Ok, here's what we do below: The object knows about every state. At output
# time, the object cycles thru the states, and each state resolves where
# each dest is relative to the owner of this state. Results can be 'up',
# 'down', or 'lcl'. If down, we need the cstate to go down to.

sub ResolveDest
{
    my $obj = shift;

    Put Log "Resolve called for state $$obj{name}";
    my $parent = $$obj{parent};
    my ($ent,$dir,$csname, $ref);
    foreach $e (keys %{$$obj{event}}) {      # this gets each event
        foreach $g (@{$$obj{event}{$e}}) {  # this gets each guard/action
            $dest = $$g[3];                 # get the dest
# This handles transition to 'exit' state
            if ($dest eq 'exit') {
                $$obj{desttype}{$dest} = 'lcl';
                next;
            }
            $ref = $$obj{parent}->GoUp($dest, $obj);  # resolve direction
            Put Log "Got back:$$ref[0], $$ref[1]";
            if (!$$ref[0]) {die "can't find dest $dest"}
            $$obj{desttype}{$dest} = $$ref[0];   # save direction
            $$obj{cstate}{$dest} = $$ref[1];     # save ref to CState, if req'd
# Setup enum for state name
            if ($$ref[0] eq 'up' || $$ref[0] eq 'down') {
                Context->AddEnum($$obj{objref}->GetName, "st_$dest");
            }
        }
    }
}

sub PreProcess
{
    my $obj = shift;
# The only thing we do is scan all events and register enums for
# those that are NOT signals. 'Object' already registered signals.
# NOTE: we continue to save signals by their name without the OBJ_ prefix.

    my $e;
    my $objref = $$obj{objref};
    foreach $e (keys %{$$obj{event}}) {
        if ($e ne 'NOEVENT' && !$objref->IsASignal($e)) {
            Context->AddEnum($objref->GetName, $e)}
        if (grep($$_[0], @{$$obj{event}{$e}}) > 0) {$$obj{needlabel} = 1}
    }
}

sub Output {return}

sub LclOutput
{
    my $obj = shift;


    Put Log "LclOutput for state $$obj{name}";

    my $objref = $$obj{objref}; # get objrect ref to save writing.
    my $objname = $objref->GetName;  # get name of object

    Put Context 0, "/* State $$obj{name}   */";
    Put Context 0,"$$obj{name}:printf(\"in state $objname.$$obj{name}\\n\");";
# Do we need to save state for history
    if ($$obj{parent}->HistoryPresent) {
        Put Context 1,"H_", $$obj{parent}->GetName, " = st_$$obj{name};",
        "  /* Save state for history */";
    }

# Do we need guarded transition label?
# See below.
    if ($$obj{needlabel}) {Put Context 0,"$$obj{name}\_G:"}

# Set up events we look for. Incoming signals are prefixed with objname
    foreach $e (keys %{$$obj{event}}) {
        Put Log "setting up event $e in state $objname";
        if ($e eq 'NOEVENT') {next}  # a no event transition
# queued semantics
        if (!$objref->IsASignal($e)) {Put Context 1,"evq!$e,_pid;"}
# not queued semantics
#       if ($objref->IsASignal($e)) {Put Context 1,"evq!$e,_pid;"}
    }

# If we are target of IN predicate, need assignment stmt
    if ($objref->IsAINTarget($$obj{name})) {
        Put Context 1,"${objname}_state = st_$$obj{name};";
    }

    Put Context 1, "atomic {if :: !t?[free] -> t!free :: else skip fi;}";
# Wait for events, and build transitions, actions, send msgs.
# If event is signal, and there is a parm, save it.
# Keys are events, saved as [guard, action, msg, dest, var]
# First check if there are guards (the grep)
    my $dest;
    Put Context 1,"if";
    foreach $e (keys %{$$obj{event}}) {
        Put Log "working on event $e in state $$obj{name}";
        if ($e eq 'NOEVENT') {PutNR Context 1, ":: 1 -> "}  # no event
        elsif ($objref->IsASignal($e)) {
            Put Log "$e is a signal var=$$obj{sigvar}{$e}";
            if ($$obj{sigvar}{$e}) {
# queued semantics
                PutNR Context 1, ":: atomic{${objname}_q?$e -> ",
                "${e}_p1?$$obj{sigvar}{$e}} -> ";
# The next statement uses a temp variable: WARNING there is nothing
# to guarrantee that this variable won't get overwritten.
#               "$$obj{sigvar}{$e} = ${e}v} -> ";
# non queued semantics
#               PutNR Context 1, ":: atomic{evt??$e,eval(_pid) -> ",
#               "$$obj{sigvar}{$e} = ${e}v} -> ";
            }
            else {
# queued semantics
                PutNR Context 1, ":: ${objname}_q?$e -> ";
# non-queued
#               PutNR Context 1, ":: evt??$e,eval(_pid) -> ";
            }
        }
        else {PutNR Context 1, ":: evt??$e,eval(_pid) -> "}
        PutNR Context 0,"t?free; ";
# If IN predicate, we need to clear variable
        if ($objref->IsAINTarget($$obj{name})) {
            PutNR Context 0,"${objname}_state = none; ";
        }

# Now for the code that follows the event.
# 0=guard, 1=action 2=msg 3=dest 4=var
# First we check to see if there are any guards. If so, we need an 'if'
# construct. No condition is '1'
        if (grep($$_[0], @{$$obj{event}{$e}})) { # any guards?
            Put Log "guards present";
            Put Context 0,"if";                # 'if' for guard
            foreach $g (@{$$obj{event}{$e}}) {  # enumerate conditions
                Put Log "gd=$$g[0], action=$$g[1], msg=$$g[2] ",
                "dest=$$g[3], var=$$g[4]";

# check for guard. No guard gets a '1'
                if ($$g[0]) {PutNR Context 2,":: $$g[0] -> "}
                else {PutNR Context 2,":: 1 -> "}
                codeseq($g, $obj);   # write out code seq
            }
# Need this to flush event if guard is false.
            Put Context 2,":: else -> goto $$obj{name}\_G";
            Put Context 2,"fi";
        }
# We are here is there are no guards, and hence one set of code for this event
        else {
            Put Log "no guards";
            codeseq($$obj{event}{$e}[0], $obj); # there is only 1 item on list
        }
    }
    Put Context 1,"fi;";  # if that closes off event dispatch
}

sub codeseq
{
# This routine writes assumes there is an action or msg
    my ($g, $obj) = @_;

# check for action
    if ($$g[1]) {PutNR Context 0, " $$g[1]"}

# check for msg
    if ($$g[2]) {PutNR Context 0, " $$g[2]"}
# Finally, write dest expression
    write_tran($$obj{desttype}{$$g[3]}, $$g[3], $$obj{cstate}{$$g[3]});
    Put Context 0;
}

sub write_tran
{
    my ($desttype, $dest, $cstate) = @_;
```

```
# This is a local routine to write out the right kind of transition.

    Put Log " WriteTran desttype=$desttype, dest=$dest, cstate=$cstate";
# local state transition, use goto
    if ($desttype eq 'lcl') {
        PutNR Context 0," goto $dest";
    }
    elsif ($desttype eq 'lclcs') {
        PutNR Context 0," goto to_$dest";
    }
# transition to state up somewhere (higher level)
    elsif ($desttype eq 'up') {
        PutNR Context 0," wait!_pid,st_$dest; goto exit";
    }
# transition to lower CState
    elsif ($desttype eq 'down') {
        PutNR Context 0," m = st_$dest; ",
        "goto to_$cstate";
    }
# whoops! we have a problem here....
    else {die "destination <$desttype> not typed"}
}
1;
```