DATE DUE	DATE DUE	DATE DUE

ENABLIN
OF O
EMPHAS

Enabling Integrative Analyses and Refinement of Object-Oriented Models with Special Emphasis on High-Assurance Embedded Systems

By

Laura Anne Campbell

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 Engineering

2004

Abstract

Enabling Integrative Analyses and Refinement of Object-Oriented Models with Special Emphasis on High-Assurance Embedded Systems

By

Laura Anne Campbell

Requirements modeling and analysis is one of the most difficult tasks in the software development process. Hardware constraints and potentially complex control logic exacerbate this problem for embedded systems development. While requirements errors can be costly for software systems in general, they can be especially costly for high-assurance or safety-critical embedded systems where failure can have dire consequences. Therefore, methods for modeling and rigorously analyzing embedded systems requirements have value for developers.

The *ad hoc* development approaches currently used in embedded systems lack systematic methods for both modeling and analyzing requirements. Although the embedded systems community has expressed interest in exploring how object-oriented modeling, specifically the UML, can be used for embedded systems development, UML lacks a formal semantics, thus precluding rigorous analysis of requirements expressed as UML models. While formalization of UML enables rigorous analysis of formal models derived from UML diagrams, formalization itself is not sufficient to broaden the community of embedded systems developers who can use formal methods to rigorously analyze requirements. To enable developers to model and analyze requirements in UML without having to know details of formal models requires a framework and process that takes advantage of a UML formalization yet insulates developers from the formal models produced by such a formalization.

This research presents an approach to validating embedded systems requirements

modeled as both UML diagrams and LTL properties using formal verification (*e.g.*, model checking) techniques. We describe a model development and analysis framework that insulates the developer from formal models and outputs of tools, and an overall model development and analysis process.

To Mother

"Never was there a better daughter, kinder sister, or truer friend."

– Jane Austen, *Emma*

ACKNOWLEDGMENTS

It is not possible in this space to list every person who assisted me along this journey, but I especially wish to thank my committee members: Dr. Betty H. C. Cheng (chairperson), Dr. R. E. Kurt Stirewalt (co-advisor), Dr. Anthony Wojcik, and Dr. Bryan Pijanowski for their time and guidance, and constructive comments. Without their support and encouragement, this work would not have been possible. I would also like to thank Dr. Abdol-Hossein Esfahanian, Dr. Eric Torng, and Graduate Secretary Linda Moore for providing a listening ear when I needed it.

In many ways this work has been a synthesis of previous explorations integrating informal and formal methods. I especially wish to thank the following people from the Software Engineering and Network Systems (SENS) research group at Michigan State University: Dr. Y. Enoch Wang (OMT-to-LOTOS formalization), Dr. William E. McUmber (UML-to-Promela formalization and general formalization framework), Min Deng (rewriting Hydra to use the Visitor design pattern), Sascha Konrad (requirements patterns), Ali Ebnenasir (domain knowledge of fault-tolerance), and Ryan Stephenson (coding contributions on MINERVA and Hydra).

This work has been supported in part by the following grants: NSF EIA-0000433, EIA-0130724, CDA-9700732, CDA-9617310, CCR-9633391, CCR-9901017, CCR-9984727; DARPA grant No. F30602-96-1-0298, managed by Air Force's Rome Laboratories; Department of the Navy, and Office of Naval Research under Grant No. N00014-01-1-0744; a Motorola doctoral fellowship; and in cooperation with Eaton Corporation, Siemens Automotive, and Detroit Diesel Corporation.

Finally, I must thank my husband Paul, family, and friends who were there for me throughout this process.

Thank you.

LIST OF FI

Table of Contents

List of Figures

D1 Control

D7 Control

E1 Lateral

E2 State

E7 Spatial

E8 Spatial
Stat

E9 Spatial
Tra

E4 Spatial
UM

I1 UML

I2 State

I3 UML
Tra

Chapter 1

Introduction

As software has become increasingly used in *critical systems* where failure can have dire or even catastrophic consequences [6], the need to have high assurance in its correctness has increased. Many approaches to software development include at least the following two phases early in the software life cycle: *requirements analysis* (what the software will do) and *design* (how the software will accomplish its tasks) [7, 8]. Studies have shown that requirements errors are between 10 and 100 times more costly to correct at later phases of the software life cycle than at the requirements analysis phase itself [9]. Thus, requirements errors can have a significant impact on the reliability, cost, and safety of a system.

One approach to this problem is to document software requirements and design using a *formal language*, a language with a rigorously defined syntax and semantics [10]. This approach is one of the basic elements of the software engineering disciplines referred to as *formal methods*, which are characterized by a formal language and a set of rules governing the manipulation of expressions in that language [10]. The advantages to using formal methods are significant, including the use of notations that are precise, verifiable, and facilitate automated processing. A *formal model*, a formal-language representation of a software system's requirements and design, can

be rigorously manipulated to allow a developer to assess the consistency, completeness, and robustness of a design before it is implemented. However, attempting to construct a formal model directly from an informal, high-level (*e.g.*, prose) requirements document can be challenging. Additionally, formal methods by their nature are rigorous and many require a steep learning curve. Thus far, formal methods are not widely used in industry except for extremely critical software [11].

Other approaches to requirements analysis and design include numerous object-oriented techniques [12, 13, 14, 15]. These *semi-formal methods* enable the rapid construction of a system model using intuitive graphical notations with well-defined syntax, and user-friendly languages, including Object Modeling Technique (OMT) [14] and Unified Modeling Language (UML) [16, 17, 18]. While such techniques have proved to be useful, as measured by their popularity in industry [19, 20], the graphical notations used with these methods are often (semantically) ambiguous, resulting in diagrams that are easily misinterpreted. The lack of formal semantics for the diagrams or their integration prevents rigorous and/or automated analysis of them. Furthermore, without rigorous development processes, users can potentially create diagrams that contain inconsistencies, ambiguities, and other types of errors that cannot be detected by syntax-oriented editing or analysis utilities [19, 21]. The potential result is a collection of erroneous diagrams that are used to guide the development of a computer-based system.

In an effort to leverage the benefits and address the shortcomings of both formal and semi-formal approaches to requirements analysis and design, many projects [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35] have proposed techniques for associating formal semantics with semi-formal modeling notations, including UML [26, 28, 29, 30, 31, 32, 33, 35]. The interest in UML is largely due to the fact that it has become a *de facto* standard in object-oriented software modeling and development.

1.1 Prob

1.1 Problem Description

Errors introduced during the analysis process are commonly the most expensive to fix, as they usually require subsequent changes to design and code [36]. While requirements errors can be costly for software systems in general [9], they can be especially costly for *embedded systems*, *i.e.*, systems with embedded microprocessors, not only for safety reasons but also in terms of hardware fabrication costs. Given the potentially critical nature of embedded systems (*e.g.*, X-by-wire, medical devices, *etc.*) in which faulty behavior of a system could lead to significant loss, methods for modeling and developing embedded systems and rigorously analyzing behavior before starting the design phase and committing to code are increasingly important. However, currently much of the embedded systems industry uses *ad hoc* development approaches [37, 38] that lack systematic methods for both modeling and analyzing requirements, thus potentially leading to unreliable and unsafe systems. Although the embedded systems community has expressed interest in exploring how object-oriented modeling, specifically the UML, can be used for embedded systems development [37, 38, 39], UML lacks a formal semantics, thus precluding rigorous analysis of requirements expressed as UML models. An approach that integrates both semi-formal and formal techniques enables rigorous analysis early in the software life cycle. This research applies UML formalization to enable developers, who are not experts in formal methods, to use automated tools to *validate* (*i.e.*, check their understanding of) embedded-systems requirements specified as UML models.

While formalization of UML enables rigorous analysis of formal models derived from UML diagrams, formalization itself is not sufficient to broaden the community of embedded systems developers who can use formal methods to rigorously analyze requirements. To enable developers to model and analyze requirements in UML without having to know details of formal models requires a framework and process that takes advantage of a UML formalization yet insulates developers from the formal

models produced by such a formalization.

This research presents an approach to validating embedded systems requirements modeled as both UML diagrams and linear time temporal logic (LTL) properties using formal verification (*e.g.*, model checking) techniques. We describe a model development and analysis framework that insulates the developer from formal models and outputs of tools, and an overall model development and analysis process that systematically applies the following steps.

1. **Model creation.** The overall process provides developers with model creation guidelines to both take advantage of a UML formalization and accelerate the initial development of requirements models.

2. **Automated generation of formal models.** Our approach complements and extends an existing general UML formalization framework for embedded systems development [31, 33] that supports the automated generation of formal models.

3. **Automated translation of UML-based LTL specifications into target-language-based LTL specifications.** Our approach enables embedded systems developers to model requirements in UML and check the UML models for adherence to developer-specified LTL properties. The developer specifies LTL properties in terms of UML attributes, in-state predicates, and message reception; these UML concepts are translated to their formalized representations and checked against the formal model derived from the UML model.

4. **Model analysis.** In order for developers to make effective use of formal analysis tools without being experts, the overall process provides guidelines for using the different analyses available, including using them in an integrated fashion.

5. **Visual interpretation of analysis results.** The overall process incorporates visual-based feedback techniques relating sometimes cryptic analysis re-

sults back to diagrams in order to insulate developers from the output of formal analysis tools.

1.2 Thesis

Our approach to validating embedded systems requirements incorporates the following enabling techniques, described in the previous section:

- Model creation [40, 41, 42],
- automated generation of formal models [31, 33],
- automated translation of UML-based LTL specifications into Promela-based LTL specifications [40],
- model analysis [40, 41, 42], and
- visual interpretation of analysis results [40, 41, 42].

Thesis Statement: *By using a combination of graphical model construction, automated translation of models and critical properties to formal specifications, automated analysis of specifications, and visual interpretation of analysis results, a developer can make use of formal-verification tools to support model validation while being insulated from the formal models used by these tools.*

1.3 Contributions

This research had several objectives:

1. To leverage existing object-oriented notations, formal languages, formal analysis techniques, and formal language analysis tools.

2. To keep as much of the underlying formal method(s) as possible transparent to the user while providing user guidance for instantiating requirements-based properties.

3. To explore the tradeoffs between different analysis techniques to determine which technique is most appropriate at different stages of modeling.

4. To leverage visualization techniques for displaying the results of well-formedness error detection (*i.e.*, consistency checking) and formal analyses.

An overarching objective of this research was to facilitate technology transfer of rigorous software engineering techniques to developers of embedded systems. To achieve these objectives, we developed a model development and analysis framework that leverages, integrates, and encapsulates a previously developed formalization framework for object-oriented notations [31, 33], thus enabling embedded systems developers to use formal-verification tools to support requirements validation while insulating the developers from the formal models used by such tools. We described an instantiation of these integrated frameworks with an existing object-oriented notation (UML), previously developed mapping rules from UML to an existing formal language (Promela), and an existing formal language analysis tool (Spin). We described and demonstrated a model development and analysis process that applies light-weight (*e.g.*, consistency checking) and heavy-duty (*e.g.* model checking) analyses at different modeling stages, and leverages specification patterns by Dwyer *et al.* [43] to provide user guidance for instantiating requirements-based properties to be checked against a formal model. Results of structural and behavioral analyses are visualized in terms of UML diagrams in order to insulate users from the output of formal analysis tools.

In summary, this research makes several contributions [1, 40, 41, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56]:

- **Bifurcated approach to analysis [40, 47, 51].** Preliminary investigations [44, 45, 46] indicated that formal analysis tools are not well-suited to detecting structural problems within the formal models generated from diagrams; that is, diagram well-formedness should be checked prior to formal model generation. Therefore, we describe and demonstrate a bifurcated approach to analysis that incorporates both structural (diagram level) and behavioral (formal model level) analyses.

- **Visualization of analysis results to guide diagram refinements [40, 41, 47, 50].** We developed visualization techniques for results of both structural analyses of diagrams and behavioral analyses of automatically generated formal models. Visualizations include highlighting portions of diagrams, animating diagrams, and generating new diagrams.

- **Framework and process for development and analysis of (formalized) UML models [40, 41, 49, 56].** We developed a model development and analysis framework (Chapter 4) that leverages, integrates, and encapsulates a previously developed formalization framework for object-oriented notations [31, 33], including feedback to diagrams from formal analysis tools in order to insulate users from the outputs of such tools. We also developed an iterative and incremental model development and analysis process (Chapter 4) that comprises steps for (1) model construction, (2) structural analyses, (3) behavioral analyses, and (4) refinements based on feedback from both types of analyses.

- **Structural and behavioral patterns for modeling fault handling.** High-assurance systems must often remain operational even in the presence of faults. We developed structural and behavioral patterns for modeling the fault-tolerance concepts of *detectors* and *correctors* [57] in UML to provide guidance for modeling and analyzing fault handling requirements.

1.4 Org

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we present background material regarding two formalizations of semi-formal object-oriented graphical notations: Wang's OMT formalization work [27] and McUmber's general framework for formalizing UML for embedded systems [31, 33]. Wang's formalization provides the context for our preliminary investigations, while we make direct use of McUmber's formalization framework in our approach. Chapter 3 describes preliminary investigations with Wang's formalization and general distributed systems [44, 45, 46] exploring tradeoffs between different analysis techniques to see what analysis technique is most appropriate at different stages of modeling. The lessons learned in these investigations motivated our subsequent research with analyses and visualizations at different modeling stages as applied to McUmber's UML formalization and embedded systems, described in the remainder of the dissertation. Additionally, our model development and analysis framework, introduced in Chapter 4, leverages and extends the specification generator architecture used by both of these formalization approaches. We then instantiate this framework with tools to support McUmber's UML-to-Promela formalization [31, 33], and we present a model development and analysis process based on this instantiation. This model development and analysis process serves as a road map to the remainder of the dissertation. Chapter 5 discusses both structural and behavioral analyses enabled by the formalization of UML, while Chapter 6 describes visualization techniques we have developed to relate the results of both structural and behavioral analyses back to the formalized UML diagrams from which models were generated. To validate our approach, we performed several case studies [40, 41, 42, 53, 54, 55] obtained from the automotive industry, including an Adaptive Cruise Control system [40] that uses radar to avoid collisions, an Anti-Lock Braking System [53] with redundant brake sensors, a self-cleaning Diesel Filter System [41, 42] that removes soot from diesel truck exhaust, and an Electronically

Controlled Steering system [54, 55] that provides variable-assistance power steering; we present results from the **Adaptive Cruise Control** case study [40] in Chapter 7. In Chapter 8, we assess the cost of using instrumentation to enable visualizations in terms of the increase in the size of the state space (an important concern for model checking). We also discuss the applicability of the overall model development and analysis approach to other application domains. In Chapter 9 we describe how our model development and analysis framework can be applied to fault-tolerant systems, including the development of structural and behavioral UML patterns, as well as critical properties. In Chapter 10 we review the related literature. Finally, Chapter 11 presents concluding remarks and outlines potential future investigations.

Chapter 2

Background

This chapter discusses two previous works that integrate semi-formal graphical modeling notations and formal languages: the general UML formalization framework [31, 33] extended in this dissertation, and its predecessor, a formalization [27, 58, 59, 60] of *Object Modeling Technique* (OMT) [14] diagrams in terms of LOTOS[1] [61] models. The latter was the focus of our preliminary investigations [44, 45, 46] of integrative analysis techniques for formalized object-oriented diagrams. Additionally, in Chapter 3 we use examples from these investigations to illustrate both the need for consistency checking of diagrams prior to formal model generation, and the benefits derived from visualizing analysis results.

2.1 Formalization of OMT

Wang *et al.* [27, 58, 59, 60] described a set of rules and a process to enable the semi-automated generation of LOTOS [61] models from OMT [14] diagrams. Our preliminary investigations [45, 46] examined the types of analyses afforded by this formalization using existing LOTOS analysis tools. We provide a brief overview of OMT, LOTOS, and two LOTOS analysis tools, followed by a high-level overview of

[1] Language of Temporal Ordering Specification

Wang's OMT-to-LOTOS formalization rules [27].

2.1.1 OMT Overview

The *Object Modeling Technique* (OMT) [14] uses three simple graphical notations in a complementary manner to express structural, behavioral, and functional characteristics of a software system. The *object model* depicts static structural information; that is, the objects comprising a system and the relationships between those objects. Furthermore, this model provides the context for the other two models, the dynamic and functional. While the *dynamic model* portrays the behavior of the system and its objects, the *functional model* captures the data and the services of the system and its objects. The three orthogonal models are briefly discussed in the following text.

Object Model. An object model depicts structural information about a system. The object model of a system consists of a *class diagram*; that is, a diagram consisting of boxes and lines. Boxes represent the classes or objects comprising the system, and lines denote the relationships or *associations* between them. Each class has a name and may be adorned with text denoting a list of attributes and their data types, or a list of operation signatures. Associations may have a filled or hollow circle at their endpoints to indicate multiplicities of *many* (zero or more) or *optional* (zero or one), respectively. Multiplicity of *exactly one* has no special adornment at the endpoint, while *one or more* may be indicated with the text "1+". A triangle or diamond at one endpoint denotes sub-typing or aggregation, respectively. The *Unified Modeling Language* (UML) [16, 17, 18], discussed in Section 2.2.2, has incorporated most of the OMT-style class diagram notation for its class diagram.

Dynamic Model. The dynamic model of a system describes the behavior of the system and its objects via a collection of *state transition diagrams*. Each state diagram depicts the permissible sequence of states and events for one class. The notation

Functional M

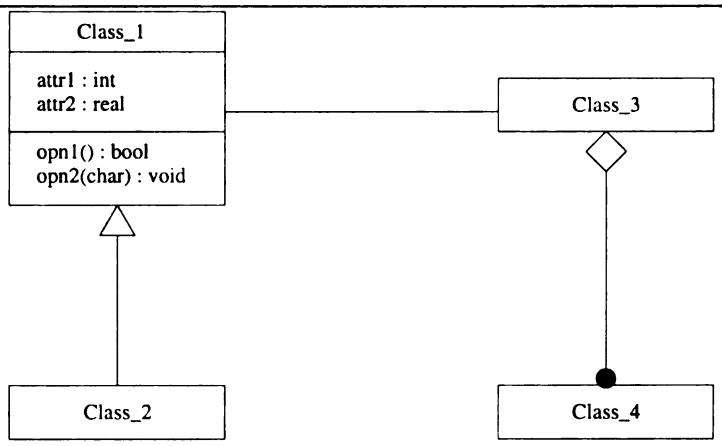


Figure 2.1: Example OMT class diagram

is based on Harel's Statecharts [62] with slight variations. Rounded rectangles annotated with names represent states, and labeled directed arcs represent state transitions. States may be simple (flat), composite (containing nested states), or concurrent (having partitions each containing a state transition diagram). The label on a transition may contain the name of the event that triggers the transition, a guard condition, or an action such as sending an event to another object. The UML has also adopted most of the OMT-style state diagram notation for its state diagram.

Functional Model. A collection of *data flow diagrams* portraying the data and services of the system and its objects comprises the functional model of a system. In data flow diagrams (see Figure 2.3), ovals annotated with names correspond to processes, and directed arcs correspond to data flows. Arcs are labeled with the data type and optionally the data name. The UML does not include data flow diagrams.

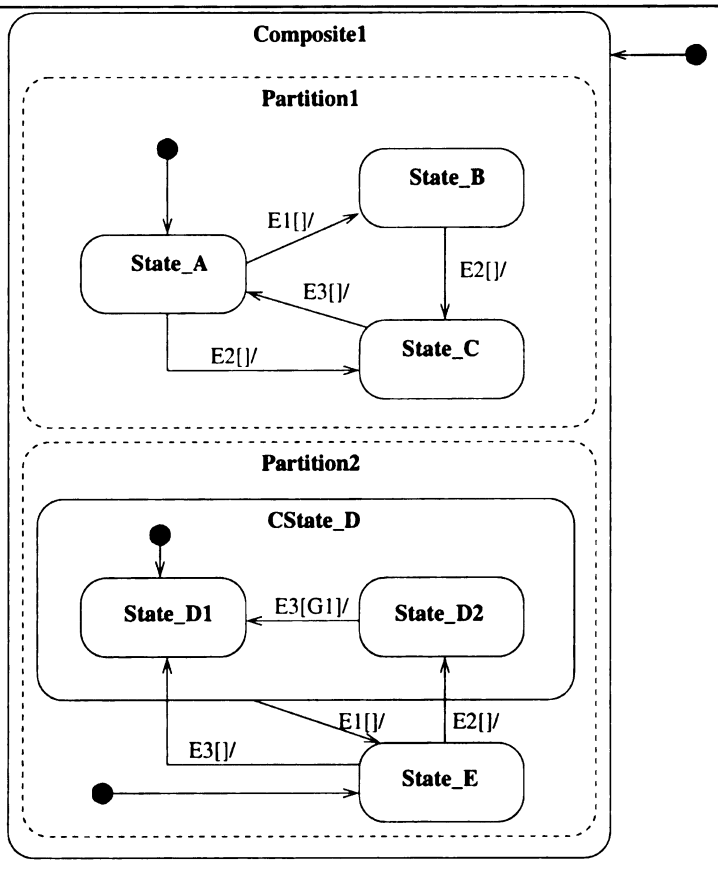


Figure 2.2: Example OMT state transition diagram

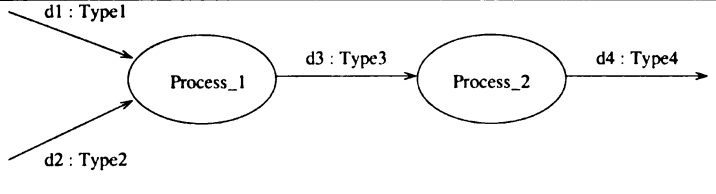


Figure 2.3: Example OMT data flow diagram

2.1.2 LOTOS Overview

LOTOS [61] originates from the telecommunications industry as a specification language for networking and telecommunications protocols. Basic LOTOS is built upon a *process algebra*, enabling specifiers to model externally observable behaviors, or *events*. A process algebra is an algebraic theory that formalizes the notion of concurrent computation [63]. Full LOTOS includes an *algebraic specification* language, ACT ONE [64], to enable specification of abstract data types (ADTs) and modeling of process communication via data exchange. Algebraic specification can be used to define relationships between ADTs in terms of operations on them [65].

A full LOTOS model may include definitions of abstract data types in addition to process descriptions. Each process in LOTOS has an identifier, a formal gate list, and a description of its functionality including *behavior expressions*. Each behavior expression is built from primitive *actions* or *events* and predefined operations such as *action prefix* (`;`) for sequentiality, *choice* (`[]`) for selection between alternative behaviors, and *parallel composition* (`|||`) for concurrency. A full LOTOS event consists of its name (known as the *gate*) and a parameter list. Processes may exchange data if they *synchronize* on a gate; that is, one process *offers* (`!`) a value at the gate and another process *accepts* (`?`) that value into a variable of compatible data type. For example, in the LOTOS code fragments shown in Figure 2.4, process A offers the values contained in its variables `x` and `y` at gate `foo`, while process B waits at gate `foo` for an offer of two values, one of type `Int` and one of type `String`, respectively. If `x` is of type `Int` and `y` is of type `String`, then processes A and B may synchronize on gate `foo`, with process B's variables `w` and `z` taking on the values of process A's variables `x` and `y`, respectively. This type of synchronization is known as *value passing*.

The behavior of a LOTOS model can be represented graphically by a *Labeled Transition System* (LTS). An LTS is a tree-like structure with anonymous nodes and labeled arcs. Labels on arcs represent events, or occurrences of actions. The root of

endproc

```
process A (sender)   :  foo !  x !  y;
process B (receiver) :  foo ?  w:  Int ?  z:  String;
```

Figure 2.4: LOTOS synchronization with data exchange

```
process Soda [75c, PepsiButton, DispensePepsi,
              CokeButton, DispenseCoke] := noexit
    75c;
    (
        (PepsiButton;
            (DispensePepsi;
                Soda [75c, PepsiButton, DispensePepsi,
                    CokeButton, DispenseCoke]))
    []
        (CokeButton;
            (DispenseCoke;
                Soda [75c, PepsiButton, DispensePepsi,
                    CokeButton, DispenseCoke]))
    )
endproc
```

Figure 2.5: LOTOS model of recursive soda vending machine

the tree appears at the top of the structure.

As an example, consider a soda[2] vending machine modeled as a single LOTOS process (Figure 2.5). In this model, the vending machine first accepts 75 cents from the customer. Next, the customer may choose either Pepsi or Coke by pressing the appropriate button, causing the vending machine to dispense the desired beverage. The vending machine then awaits the next customer. Note that this model is an extremely simplistic abstraction and so does not take into account details such as making change or indicating when a particular brand of soda is not available.

As shown in Figure 2.5, the **choice** construct selects between two alternative be-

[2]Pepsi and Coke are registered trademarks of their respective companies.

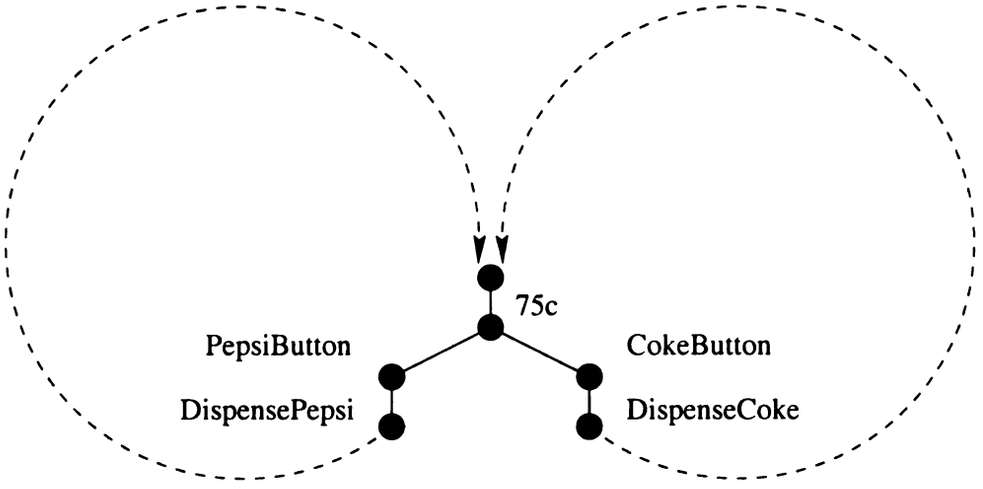


Figure 2.6: LTS of recursive soda vending machine

haviors (purchasing Pepsi or purchasing Coke) based on the next event (pressing the appropriate button). Modeling a choice between two behavior expressions B_1 and B_2, the expression $B_1[]B_2$ behaves like $B_1(B_2)$ if the next event is the first event of behavior $B_1(B_2)$. The **noexit** operator is used for defining recursive process instantiations. This construct, together with process instantiation, allows modeling of repetitive or even infinite behaviors. In Figure 2.5, after dispensing a beverage the vending machine awaits the next customer.

A choice can be shown in an LTS as a split or branch from the node representing the choice operation, with the behavior tree for $B_1(B_2)$ appended to the left (right) branch. Repetitive behavior can be shown as a dashed arc looping back to the start of the behavior. Figure 2.6 illustrates both alternatives offered by the vending machine (purchasing Pepsi or purchasing Coke) and its return to awaiting the next customer.

2.1.3 An

TOPO LOL

CADP Ove

2.1.4 For

2.1.3 Analysis Tools for LOTOS

TOPO/LOLA Overview. TOPO[3]/LOLA[4] [66], a LOTOS tool suite developed by the University of Madrid, offers syntax and semantics checking of basic and full LOTOS models. TOPO can handle "incomplete" models; that is, not every operation needs to be fully defined with algebraic equations. LOLA, a transformational and state exploration tool built atop TOPO, provides several types of expansion (state space generation), interactive simulation, and test composition.

CADP Overview. CADP[5] [67], developed by INRIA and Verimag, also offers syntax and semantics checking of LOTOS models, as well as interactive simulation and test composition. However, its additional features for viewing, manipulating, and comparing LTSs do not have a counterpart in TOPO/LOLA. Furthermore, CADP supports automated comparison of the LTSs that represent a design before and after refinement, the generation of counterexamples for deadlock cases, and the verification of temporal logic properties against LTSs.

2.1.4 Formalization Overview

Wang *et al.* [27, 58, 59, 60] formalized the OMT models in terms of full LOTOS syntax that allows the information expressed in the graphical object, dynamic, and functional models to be captured in terms of algebraic specifications and process algebras. The commonalities between the ACT ONE algebraic specifications and the basic LOTOS process algebras induced by the formalization rules make it possible to integrate the object, dynamic, and functional models into a full LOTOS model. Figure 2.7 contains a high-level overview of the OMT models and their corresponding LOTOS formal models. Overall, instances of classes or *objects* are represented in LOTOS

[3]Toolset to support product realization from LOTOS specifications
[4]LOtos LAboratory
[5]Caesar/Aldebaran Development Package

by a c

Object M

Dynamic M

Functional M

F

2.2 For

by a combination of abstract data types and a process instantiation. Each state becomes a subprocess, and each state transition an instantiation of the destination state subprocess. Object *services*, or the events to which objects respond, are realized as synchronization and data exchange between processes.

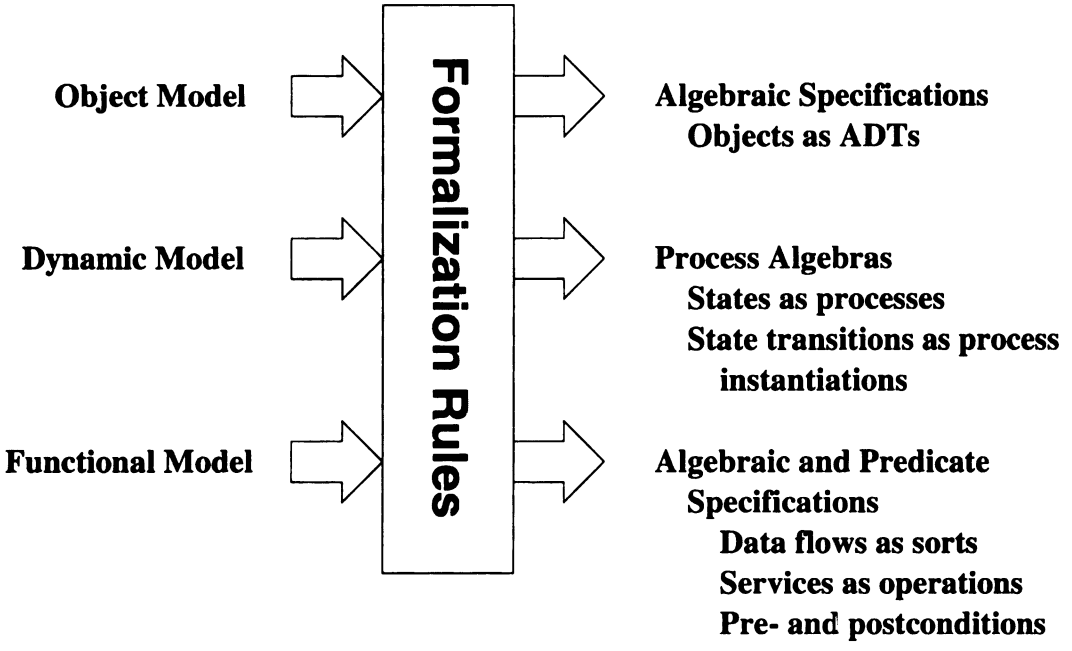


Figure 2.7: Overview of OMT-to-LOTOS formalization rules

2.2 Formalization of UML

McUmber *et al.* [30, 31, 33] built on Wang *et al.*'s approach by describing a general framework to attach formal semantics to any semi-formal diagrammatic notation(s). They then formalized a subset of the *Unified Modeling Language* (UML) [16, 17, 18] relevant to embedded systems development in terms of both VHDL[6] [68] and

[6]VHSIC Hardware Description Language

Promela[7] [69]. Our research extends this general formalization framework by providing a graphical editing and visualization environment [50, 51] both for drawing the UML diagrams comprising a system and for visualizing analysis results. Our efforts to date have focused on McUmber's UML-to-Promela formalization [31, 33]. Therefore, we include a description of not only UML diagrams affected by the formalization (*e.g.*, class and state) but also UML diagrams used in displaying analysis results (*e.g.*, sequence and collaboration), and we briefly overview Promela, the input language for the model checker Spin [69]. We then provide a high-level overview of Spin analysis tool suite and McUmber's UML-to-Promela formalization rules [31, 33].

2.2.1 General Formalization Framework

Wang's approach [27], as shown in Figure 2.8, creates mapping rules from a semi-formal language model (OMT notation) to a formal language model (LOTOS). However, his approach does not provide any guidance regarding where, how, and why to create mapping rules. McUmber's approach [31, 33], on the other hand, provides a general framework to guide the creation of mapping rules. As shown in Figure 2.9, his approach is based on mappings between *metamodels*, where a metamodel is a class diagram that describes the constructs of a modeling language and the relationships between the constructs. Relationships between metamodel elements in the source (*i.e.*, UML) constrain the mapping rules to realize the semantics of each major construct in the semi-formal language metamodel in terms of the formal language.

2.2.2 UML Overview

The *Unified Modeling Language* (UML) [16, 17, 18] comprises a collection of diagrammatic notations that depict an object-oriented software system. A product of the Object Management Group (OMG) under guidance from Booch, Rumbaugh, and Jacob-

[7]Process or Protocol Meta Language

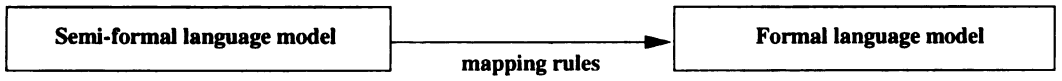


Figure 2.8: Unconstrained mapping from semi-formal to formal model

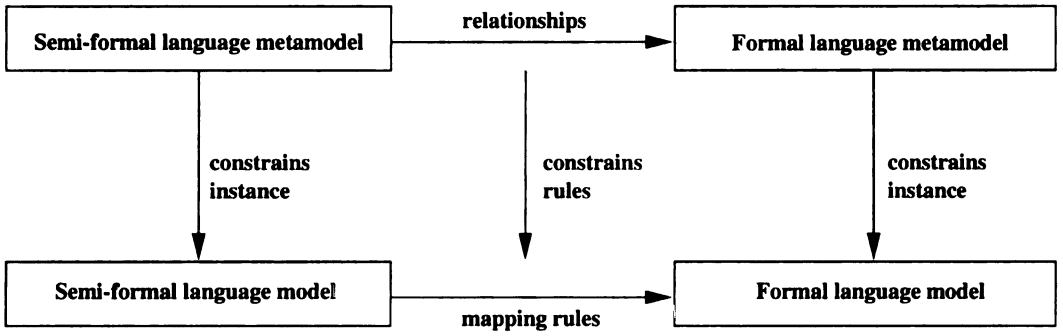


Figure 2.9: Constrained mapping from semi-formal to formal model

son, the UML incorporates and extends several modeling languages, most notably the Use Case Diagram [15], Object Modeling Technique (OMT) [14] and Statecharts [62]. The UML contains several distinct types of diagrams to depict the structure (Class Diagrams, Object Diagrams, Component Diagrams, Deployment Diagrams) and behavior (Use Case Diagrams, Activity Diagrams, State Diagrams, Sequence Diagrams, Collaboration Diagrams) of a system. This research focuses primarily on the class, state, sequence, and collaboration diagrams. As noted in Section 2.1.1, UML class and state diagrams use notation similar to that of OMT's object and dynamic models.

Class Diagram. As in the class diagram of the OMT object model, the UML *class diagram* depicts the classes in the system and the relationships between the

classes. Classes are drawn as boxes. Each class is labeled with its name and may additionally contain a list of attributes and their data types, and a list of operation signatures (including optional parameters and return types). Relationships between classes are drawn as lines, possibly with adorned endpoints, between them. There are four types of relationships between classes. The most general, *association*, is a binary relationship between two classes. Three additional relationships are *subtype, aggregation*, and *composition*, which are drawn as a small hollow triangle at the superclass end, a hollow diamond at the aggregate end, and a filled diamond at the aggregate end, respectively. Multiplicity on an endpoint is indicated numerically, with the number applying to instances at that end of the line. An optional instance is denoted by "0..1", "*" indicates 0 or more, and "1..*" denotes one or many.

State Diagram. Similar to the state diagram of OMT's dynamic model, the UML *state diagram* notation uses many Statechart [62] conventions to describe the dynamic behavior of objects. States are drawn as rounded rectangles with transitions between states depicted as directed arcs between the rectangles, indicating the direction of the transition. Transitions are labeled with an event followed optionally by a guard (enclosed in brackets), a forward slash, an action list (separated by semicolons), and a message list (separated by carets). Composite states may contain further sets of state diagrams. Concurrency of composite state machines is indicated by separating the composite states into partitions with dotted lines.

Sequence Diagram. A UML *sequence diagram* depicts objects and the ordering of messages that are exchanged between them within the context of a behavior scenario. As shown in Figure 2.10, objects are depicted as named boxes along the x axis, each with a line, the object's *life-line*, extending beneath it parallel to the y axis. The life-line may be a single dashed line, or in the case of concurrent objects, a rectangle. Messages exchanged between objects are drawn as labeled arrows between the life-

Collaboratio

Other Diag

lines of the sender and receiver objects. The messages are ordered in increasing time along the y axis.

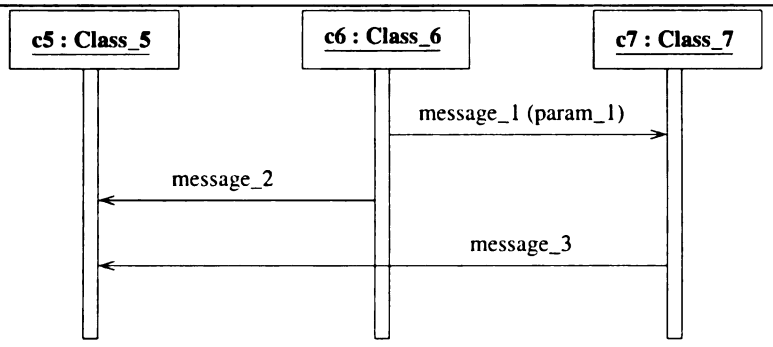


Figure 2.10: Example UML sequence diagram

Collaboration Diagram. Isomorphic to a sequence diagram, a UML *collaboration diagram* also portrays objects and the messages that pass between them within the context of a behavior scenario. A collaboration diagram, however, emphasizes the objects' structural organization rather than the temporal ordering of the messages they exchange. As shown in Figure 2.11, objects are depicted as named boxes, while the paths of communication, or *links*, between objects that exchange messages are drawn as lines between the boxes. These lines are adorned with labeled arrows that indicate both the messages themselves and the direction of the messages. Messages may be prefixed with a sequence number to indicate temporal ordering. The box depicting an object may also include information about values that may change over time, such as the object's state or attributes of interest.

Other Diagrams. The UML also contains other diagrams depicting a software system's structure (Object Diagrams, Component Diagrams, and Deployment Diagrams) and behavior (Use Case Diagrams, and Activity Diagrams). However, work

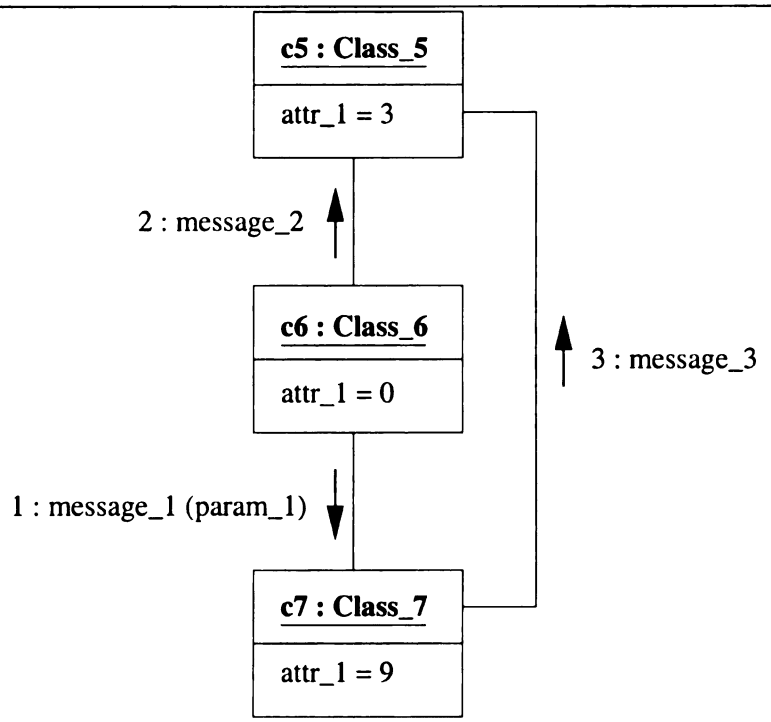


Figure 2.11: Example UML collaboration diagram

to date has focused on the Class, State, Sequence, and Collaboration Diagrams as relevant for modeling embedded systems and visualizing analysis results.

2.2.3 Promela and LTL Overview

Promela[8] [69] is a C-based language originally developed by Bell Labs to specify telecommunications protocols, but it has gained increasing use in other industrial domains involving distributed systems. Promela models consist of *processes*, *channels*, and *variables*. Processes are global objects, created dynamically, that execute asynchronously. Channels and variables may be either local or global. Processes communicate via unbuffered or buffered channels.

[8]PROcess MEta LAnguage

2.2.4 Analysis Tools for Promela

A model written in Promela can execute in a simulation environment, or model checking techniques can be applied to verify a number of temporal properties, including safety and liveness. Spin [69] (used in model-checking mode) verifies claims written in *linear time temporal logic* (LTL) [5] against a model. LTL extends propositional logic with temporal operators that apply to a sequence of states, such as the unary operators *henceforth* ($\Box$) and *eventually* ($\Diamond$). $\Box$ means that the operand is true at *every* state in the sequence, while $\Diamond$ means that the operand is true at *some* state in the sequence. Some variants of LTL include a unary *next* ($\bigcirc$) operator, meaning that the operand is true in the *next* state in the sequence; however, the default in Spin is to disallow this operator as it exacerbates the *state explosion problem.*[9] Thus, we do not consider properties containing the next operator.

LTL in terms of UML. As a convenience, McUmber developed a utility that enables users to specify LTL claims in terms of class, state, attribute, and signal names from UML diagrams, and then translates them into the Promela variables and syntax required by Spin. Acceptable terms include `className.attributeName` to reference the value of a particular attribute of a given class, `in(className.stateName)` to test the condition that a given class has currently reached a particular (UML) state during its execution,[10] and `sent(className.messageName)` to test the condition that a given class has been sent a particular message. LTL operators henceforth ($\Box$) and eventually ($\Diamond$) are given to the utility as `[]` and `<>`, respectively.

[9]Because of what is known as the state explosion problem, model checkers in practice must use some technique, or combination of techniques, to reduce the state space explored. Spin in particular uses the partial order reduction to limit the number of interleaving sequences considered [69, 70].

[10]A state predicate term in an LTL claim causes Hydra to generate an extra boolean variable that becomes true when Promela execution enters the part of code corresponding to the UML state, and false when execution leaves that part of code.

Specification Patterns. We have found the specification pattern taxonomy by Dwyer *et al.* [43] to be useful when attempting to write formal temporal logic properties (*i.e.*, LTL claims) based on English natural language requirements. The specification pattern repository organizes common specification patterns into a taxonomy based on order (*e.g.*, response, precedence) and occurrence (*e.g.*, universality, absence, existence). Specification patterns are expressed in a number of temporal formalisms, including LTL, and include suggestions for typical use. While most of the properties that we have explored to date fall into the globally-applicable format of their two most popular general categories, response or leads-to (*e.g.*, $\Box(p \rightarrow \Diamond q)$, meaning that it is always the case that when p is true, q eventually becomes true) and universality or always (*e.g.* $\Box p$, meaning that it is always the case that p is true), Dwyer *et al.*'s addition of non-global scopes to the property patterns makes it easier for a developer to define complex properties that apply to only a portion of system execution.

2.2.5 Formalization Overview

Figure 2.12 contains a high-level overview of the UML class and state diagrams and their corresponding Promela specifications. McUmber *et al.* [31] formalized the diagrams according to the following diagram integration conventions: (1) Each class that has a modeled behavior has a corresponding state diagram that expresses that behavior. We say that this state diagram *belongs to* or is *owned by* the class. (2) Events to which a class reacts may appear as events on transitions in the state diagram belonging to the class, but must appear in the operations list of the class. (3) Variables used in a state diagram must appear in the attributes list of the class that owns the state diagram.

The UML-to-Promela formalization rules are based on a constrained mapping between the integrated UML class and state diagram source metamodels and the target language Promela metamodel. Classes are mapped to Promela type definitions

Class Diagra

State Diagr

(*typedefs*), with class attributes mapping to variable declarations within a *typedef*. A class's behavior as represented by state diagram is mapped to a process description (*proctype*) with simple states as labels and transitions as gotos. Composite or concurrent states are mapped to separate subprocesses. Events on transitions become message receptions while actions become either assignment statements or message sends. The formalization rules provide a queueing mechanism between objects for handling inter-object communication.

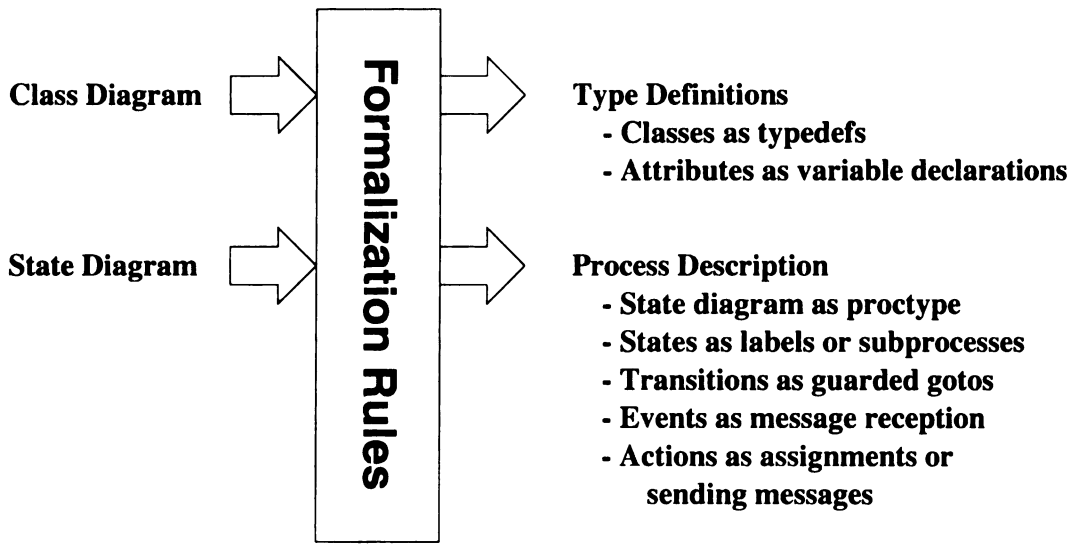


Figure 2.12: Overview of UML-to-Promela formalization rules

Chapter 3

Preliminary Investigations: Highlights and Lessons Learned

Using Wang's approach [27], we performed preliminary investigations of integrative analyses and visualization-based feedback for OMT diagram refinement in two case studies [45, 46]. In the first study [45], we examined how two different LOTOS tool suites (TOPO/LOLA [66] and CADP [67]) can be used in tandem to analyze formalized OMT diagrams, using the *Environmental Information System* (ENFORMS) [71], a multimedia distributed decision support system, as a running example. The second study [46] explored in more detail the types of analyses available with the CADP tool suite, including how they can be used together, and illustrated the visualization capabilities of the tool suite. We used a variation of the *Teleservices and Remote Medical Care System* (TRMCS), the suggested project [72] for the Tenth International Workshop on Software Specification and Design (IWSSD-10), as a running example. This chapter presents highlights from the case studies and discusses additional lessons learned that have motivated our later work with McUmber's UML-to-Promela formalization [31] and the Spin [69] analysis tools.

3.1 Hi

3.1.1 A

3.1 Highlights

In [45, 46], we developed an analysis process to be used in tandem with Wang's iterative design process [60] (see Appendix A, page 234, for a high-level overview of Wang's process). We also investigated using two different LOTOS analysis tool suites (TOPO/LOLA [66] and CADP [67]) in an integrated fashion, and explored the output options of each of the analysis tool suites. This section overviews our analysis process, integrative analyses, and example output from the tool suites.

3.1.1 Analysis Process

Our analysis process (Figure 3.1) supports analyses of OMT diagrams created or refined in each iteration of Wang's design process [60] via the LOTOS model derived from them according to Wang's mapping rules [27]. *Structural* and *behavioral analyses* are shown as boxes above and below the horizontal dashed line, respectively, in the process flow diagram in Figure 3.1. First, structural analyses (syntax and semantics checks) analyze intra- and inter-diagram consistency. Further, they check algebraic equations (manually) added to the formal model by the developer, if any, for LOTOS syntax or semantics errors. Structural analyses must be performed each time the LOTOS model changes; that is, each time the OMT diagrams from which it is derived are modified (causing re-generation of the LOTOS model), or each time the developer adds or refines an algebraic equation. All structural errors in a set of diagrams or its derived LOTOS model must be corrected before behavioral analyses (simulation and state-based exploration) are enabled. Simulation can be used to validate requirements, to increase confidence that a system or object in question behaves as desired, or to debug erroneous behavior. Once simulation has established a reasonable level of confidence in the model, state-based exploration can be used to find more subtle flaws. Techniques include concurrency analyses (*e.g.*. deadlock detection), test com-

3.1.2

position, and refinement checking as shown in Figure 3.1. Deadlock usually indicates an error in inter-object communication. Composing a *test process*, a LOTOS process describing a desired sequence of actions, with a LOTOS model detects whether that sequence of actions is possible in the model. (Searching the LTS[1] graph corresponding to a given LOTOS model for a pattern describing a desired sequence of actions is a similar technique not available in all LOTOS analysis tool suites.) Finally, refinement checking the LOTOS model derived from a set of OMT diagrams against the LOTOS model derived from a refinement of the same set of OMT diagrams can test whether the behavior of the refined model is *equivalent*[2] to the behavior of the original model. (Refinement checking is not enabled until the model has undergone at least one iteration of refinement in Wang's design process [60], indicated by the dashed arc in Figure 3.1.) As illustrated by the solid feedback arcs to the topmost process box in Figure 3.1, flaws found with any behavioral analysis technique or combination of techniques must be corrected in the diagram(s) before beginning the next iteration of Wang's design process [60]. The diagram corrections are checked by re-generating the LOTOS model, applying and passing structural analyses, and applying and passing the appropriate behavioral analyses.

3.1.2 Integrative Analyses

The analysis process mandates performing structural analyses before behavioral analyses, and suggests performing behavioral analyses in order of increasing effort needed, either in terms of time/space required to perform the analysis or on the part of the developer. Of the behavioral analyses available with either TOPO/LOLA or CADP (some analyses are available in one tool suite but not the other; see Chapter 2, Section 2.1.3, page 17 for a brief comparison of the tool suites), we found simulation,

[1]Labeled Transition System. See Chapter 2, Section 2.1.2, page 14 for a description.

[2]Depending on the LOTOS tool suite used, there may be several options available for the equivalence relation between the behaviors of the original and refined models, including *strong bisimulation* [61, 73], *observational equivalence* [73], and *safety equivalence* [74].

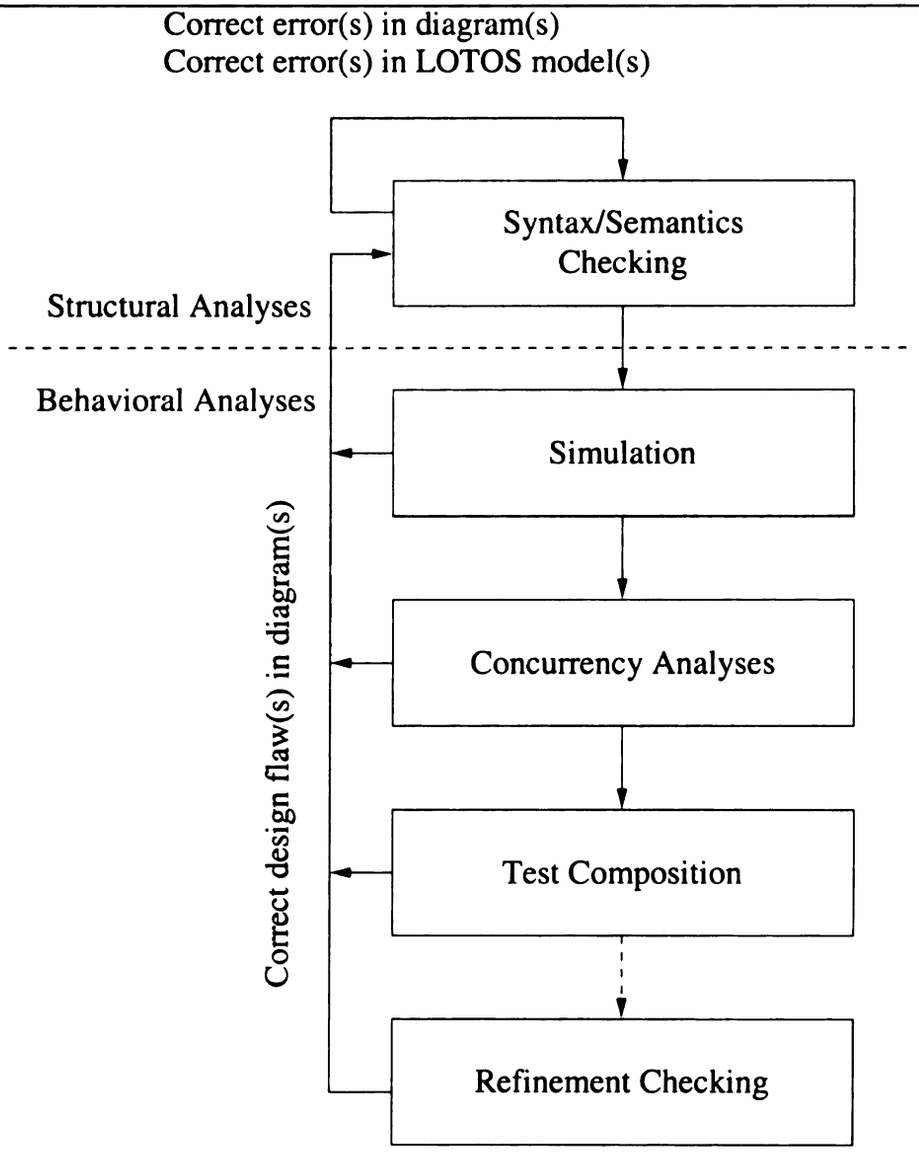


Figure 3.1: Analysis process for LOTOS models derived from OMT diagrams

concurrency analyses (*e.g.*, deadlock detection, finding the shortest path to a deadlock state), test composition or searching for patterns in the LTS graph, and refinement checking to be the most useful. Simulation can be performed very early in the modeling process, exploring a single path through the LTS graph at a time and guiding the developer towards correcting more obvious flaws. State-based exploration techniques are more expensive than simulation in terms of time/space or developer effort required, so they are delayed until simulation establishes a reasonable level of confidence in the design. Global concurrency analyses (*e.g.*, deadlock detection) can be used to check for errors in inter-object communication, but otherwise do not require anything else from the developer. On the other hand, test composition or searching for patterns in the LTS graph requires the developer to provide either a test process to be composed in parallel with the model, or a pattern describing a desired sequence of actions for which to search the LTS graph. Finally, refinement checking requires a baseline model and a refinement of that model for comparison against each other. In addition to being used on their own, each analysis technique can take advantage of the less expensive techniques; for example, once a deadlock has been detected, simulation can be used to demonstrate the sequence(s) of actions leading to that deadlock, facilitating understanding of the cause of the problem.

For illustrative purposes, we focus our discussion of integrative analyses on a few examples from the *Teleservices and Remote Medical Care System* (TRMCS) case study [46], in which we examined how various features of the CADP tool suite can be used in an integrated fashion to analyze OMT diagrams via the LOTOS model generated from them according to Wang's mapping rules [27]. In particular, we discuss test composition, exploration of paths that lead to deadlock, searching for patterns in the LTS graph, and refinement checking. (See [45, 46] for discussion of other analyses possible with LOTOS analysis tool suites.)

TRMCS Example

The *Teleservices and Remote Medical Care System* (TRMCS) is a distributed software system intended to offer access to a large volume of distributed patient data. At the most abstract level, the system should allow a user to browse a set of indices relevant to patient data in order to select the correct patient identity, retrieve a patient's record based on such an identity key, and enter a diagnosis into the patient's profile. The system-level object model, object functional model (OFM), and dynamic model shown in Figures 3.2, 3.3, and 3.4, respectively, depict the TRMCS at a high level of abstraction. (See Chapter 2, Section 2.1.1, page 11 for an overview of OMT diagram syntax.)

In essence, the TRMCS waits for the user to select a patient identity key. Afterwards, the user may continue to browse and select other keys, or may transmit the request for a patient's record. If the patient profile is found, then it becomes available for entering a diagnosis. Otherwise, the user must select a new patient identity key. Figures 3.5 and 3.6 show the LOTOS model derived from the OMT diagrams in Figures 3.2, 3.3, and 3.4 according to Wang's formalization rules [27] (the specification has been split across two figures to comply with printing restrictions). Sorts and operations (lines 8-38, Figure 3.5) are derived from the object and object functional models, while processes (lines 40-77, Figure 3.6) reflect the behavior described in the dynamic model.

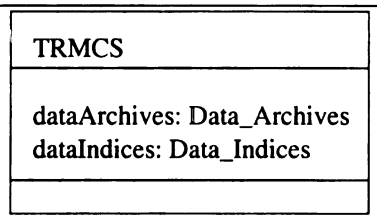


Figure 3.2: System-level object model of the TRMCS

User_I

Patient

Diagn

F

TRMCS

Select_Patient
Select_Patie

Select_P
Select_F

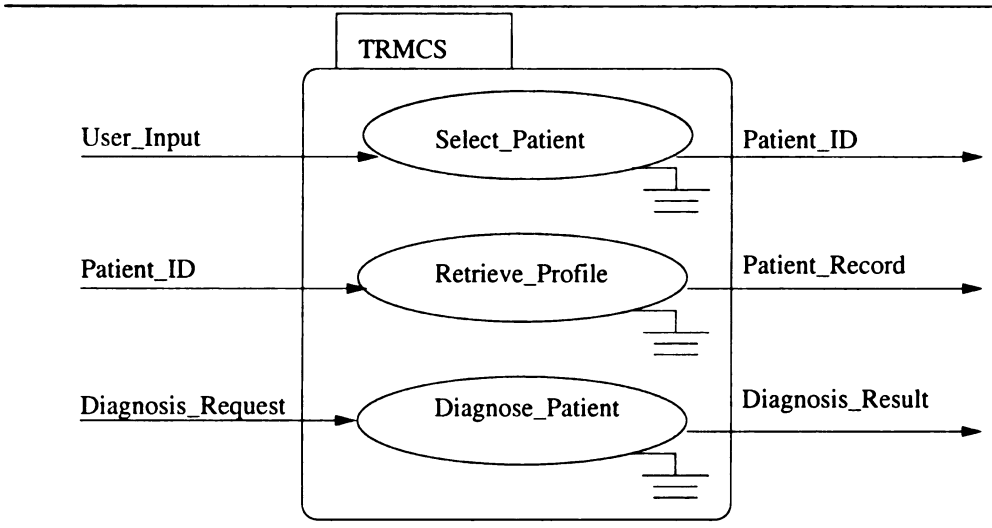


Figure 3.3: System-level object functional model of the TRMCS

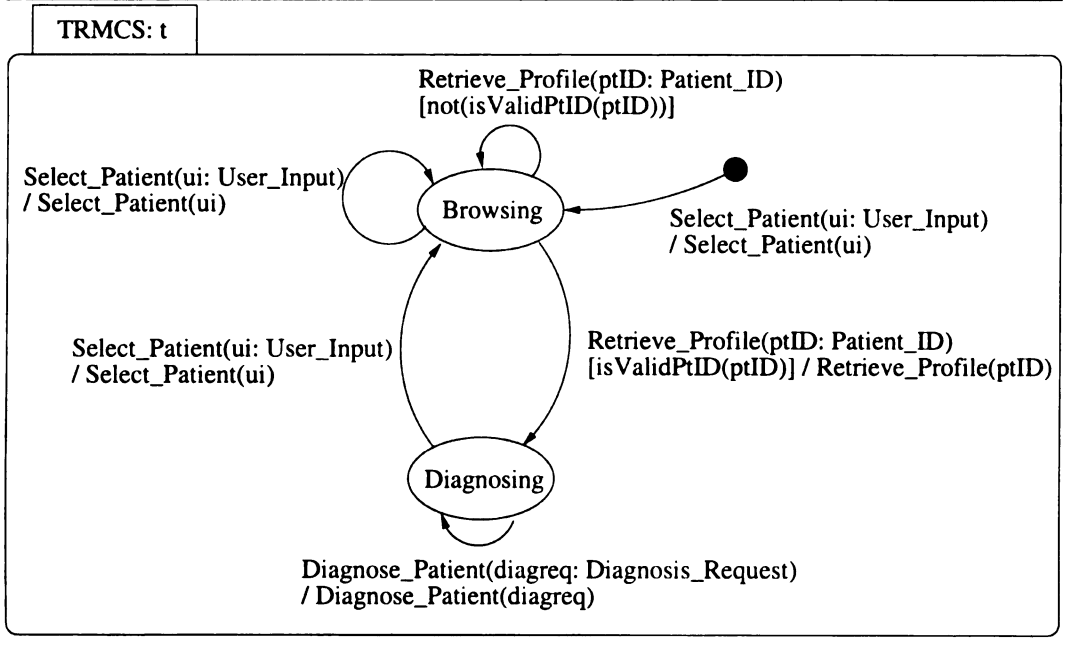


Figure 3.4: System-level dynamic model of the TRMCS

specifi

libra
Bl
endli

type
sc

op

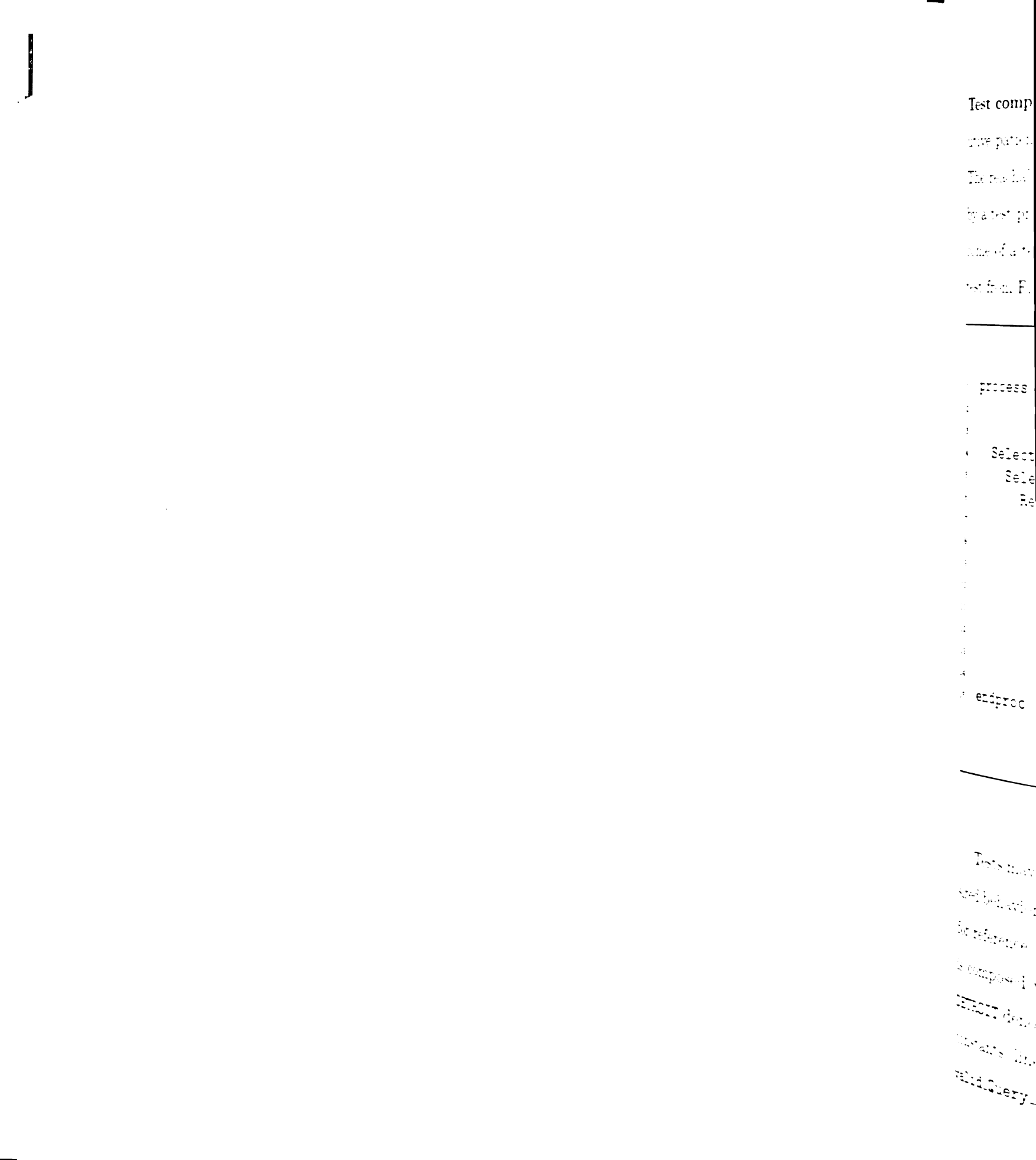
endtyp

```
specification TRMCS [Select_Patient, Retrieve_Profile,
                     Diagnose_Patient] (t : TRMCS) : noexit

   library
      BOOLEAN
   endlib

   type TRMCS is Boolean
      sorts
           Data_Archives, Data_Indices, User_Input,
           Patient_ID, Patient_Record,
           Diagnosis_Request, Diagnosis_Result, TRMCS
      opns
           undef_Data_Archives : -> Data_Archives
           _eq_ : Data_Archives, Data_Archives -> Bool
           undef_Data_Indices : -> Data_Indices
           _eq_ : Data_Indices, Data_Indices -> Bool
           undef_User_Input : -> User_Input
           _eq_ : User_Input, User_Input -> Bool
           undef_Patient_ID : -> Patient_ID
           _eq_ : Patient_ID, Patient_ID -> Bool
           undef_Patient_Record : -> Patient_Record
           _eq_ : Patient_Record, Patient_Record -> Bool
           undef_Diagnosis_Request : -> Diagnosis_Request
           _eq_ : Diagnosis_Request, Diagnosis_Request -> Bool
           undef_Diagnosis_Result : -> Diagnosis_Result
           _eq_ : Diagnosis_Result, Diagnosis_Result -> Bool
           undef_TRMCS : -> TRMCS
           _eq_ : TRMCS, TRMCS -> Bool
           Select_Patient : User_Input -> Patient_ID
           Retrieve_Profile : Patient_ID -> Patient_Record
           Diagnose_Patient : Diagnosis_Request -> Diagnosis_Result
           isValidPtID : Patient_ID -> Bool
           isValidRec : Patient_Record -> Bool
           make_TRMCS : Data_Archives, Data_Indices -> TRMCS
           getArchives : TRMCS -> Data_Archives
           getIndices : TRMCS -> Data_Indices
   endtype
```

Figure 3.5: High-level LOTOS model of the TRMCS, data part

```
39
40 behavior
41
42    Select_Patient ? ui : User_Input;
43       Select_Patient ! Select_Patient (ui);
44          Browsing [Select_Patient, Retrieve_Profile, Diagnose_Patient] (t)
45 where
46    process Browsing [Select_Patient, Retrieve_Profile,
47                      Diagnose_Patient] (t : TRMCS) : noexit :=
48       Select_Patient ? ui : User_Input;
49          Select_Patient ! Select_Patient (ui);
50             Browsing [Select_Patient, Retrieve_Profile,
51                       Diagnose_Patient] (t)
52       []
53       Retrieve_Profile ? ptID : Patient_ID;
54          (
55             [isValidPtID (ptID)] ->
56                Retrieve_Profile ! Retrieve_Profile (ptID);
57                   Diagnosing [Select_Patient, Retrieve_Profile,
58                               Diagnose_Patient] (t)
59             []
60             [not (isValidPtID (ptID))] ->
61                Browsing [Select_Patient, Retrieve_Profile,
62                          Diagnose_Patient] (t)
63          )
64    endproc
65
66    process Diagnosing [Select_Patient, Retrieve_Profile,
67                        Diagnose_Patient] (t : TRMCS) : noexit :=
68       Select_Patient ? ui : User_Input;
69          Select_Patient ! Select_Patient (ui);
70             Browsing [Select_Patient, Retrieve_Profile,
71                       Diagnose_Patient] (t)
72       []
73       Diagnose_Patient ? diagreq : Diagnosis_Request;
74          Diagnose_Patient ! Diagnose_Patient (diagreq);
75             Diagnosing [Select_Patient, Retrieve_Profile,
76                         Diagnose_Patient] (t)
77    endproc
78
79 endspec
```

Figure 3.6: High-level LOTOS model of the TRMCS, behavior part



Test composition. The simple test process we use (Figure 3.7) asks if two consecutive patient selections and valid patient record retrievals can be made successfully. The reachability of a `Success` event (line 13 in Figure 3.7) in the behavior described by a test process composed with the LOTOS model of a system determines the outcome of a test. Figure 3.8 shows, in text-based format, the (successful) result of the test from Figure 3.7 composed with the LOTOS model from Figures 3.5 and 3.6.

```
1  process Test [Select_Patient, Retrieve_Profile,
2                Diagnose_Patient, Success] : noexit :=
3
4    Select_Patient ! valid_User_Input;
5      Select_Patient ? ptID: Patient_ID;
6        Retrieve_Profile ! valid_Patient_ID;
7          Retrieve_Profile ? ptRec: Patient_Record;
8            ([isValidRec(ptRec)] ->
9              Select_Patient ! valid_User_Input;
10               Select_Patient ? ptID: Patient_ID;
11                 Retrieve_Profile ! valid_Patient_ID;
12                   Retrieve_Profile ? ptRec: Patient_Record;
13                     ([isValidRec(ptRec)] -> Success; stop))
14
15 endproc
```

Figure 3.7: Test for high-level TRMCS model

Tests may be repeated after refinement to ensure that the system still performs desired behaviors. After refinement (refined OMT diagrams for the TRMCS are included for reference in Appendix B, page 236), the LOTOS model derived for the TRMCS is composed with a revised test process updated to reflect constants `LANSING` and `DETROIT` denoting specific Data Repository instantiations. The two `valid_Patient_ID` constants (lines 6 and 11 in Figure 3.7) are replaced with tuples `Patient_ID(LANSING, valid_Query_Request)` and `Patient_ID(DETROIT, valid_Query_Request)`, intro-

des (0,
0, "SEL
1, "SEL
2, "RET
3, "RET
4, "SEL
5, "SEL
6, "RET
7, "RET
8, SUCCE

```
des (0, 9, 10)
(0, "SELECT_PATIENT !VALID_USER_INPUT", 1)
(1, "SELECT_PATIENT !VALID_PATIENT_ID", 2)
(2, "RETRIEVE_PROFILE !VALID_PATIENT_ID", 3)
(3, "RETRIEVE_PROFILE !VALID_PATIENT_RECORD", 4)
(4, "SELECT_PATIENT !VALID_USER_INPUT", 5)
(5, "SELECT_PATIENT !VALID_PATIENT_ID", 6)
(6, "RETRIEVE_PROFILE !VALID_PATIENT_ID", 7)
(7, "RETRIEVE_PROFILE !VALID_PATIENT_RECORD", 8)
(8, SUCCESS, 9)
```

Figure 3.8: Results of high-level TRMCS model under test

duced during refinement of the high-level TRMCS *Retrieve Profile* service (see [46] for details or Appendix B for an overview of this service refinement). Thus the revised test asks if it is possible to query the LANSING Data Repository, receive a valid patient record, and then query the DETROIT Data Repository, also receiving a valid patient record in response. If all paths lead to success, then we can say that the original high-level LOTOS model (derived from the original OMT diagrams modeling the system) and the more detailed refinement LOTOS model (derived from refined diagrams) are testing equivalent (with respect to this simple test).

Information regarding the results of the refined model under test is shown in Figure 3.9. CADP indicates six *deadlock states* (*i.e.*, sinks) in the resulting LTS graph.[3] By examining the labeled arcs leading to these deadlock states, we determine that only the paths ending at states 185 and 186 are *successful* (*i.e.*, terminate with arcs labeled with the **Success** event). Paths ending at any of the other deadlock states represent the test's *failure*; that is, either the first query (**LANSING**) results in an undefined patient record, or the first query passes but the second query (**DETROIT**)

[3]CADP uniquely numbers the nodes in an LTS graph.

refined_tr
137 state
27 distin
Branching

refined_tr

121 122 12

Paths lea

results in an undefined patient record. Thus the original model and its refinement are not testing equivalent with respect to the consecutive query test.

```
refined_trmcs.aut:
187 states, 282 transitions , 6 taus transitions
27 distinct labels
Branching factor : 1.50802

refined_trmcs.aut is deterministic

120 122 123 173 185 186 are deadlock states
```

Figure 3.9: Information about refined TRMCS model under test

Paths leading to deadlock. Further investigation is warranted to determine the nature of the failed queries. CADP offers several options for eliciting examples of such queries. The most straightforward one simply explores the LTS graph and finds the shortest path from the initial state to a given state, in this case one of the deadlock states 120, 122, 123, or 173 listed in Figure 3.9 (recall that paths ending at deadlock states 185 and 186 are successful). The shortest path gives the developer a minimal example depicting a problem. Ideally the developer will gain insight into the nature of the problem either by examining the example itself or by stepping through the example with interactive simulation. Figure 3.10 shows the shortest path to state 120, indicating that the first *Retrieve Profile* query (`LANSING`) results in an undefined patient record. Similar results are obtained for states 122 and 123, while the shortest path to state 173 reveals a path where the first query passes but the second query (`DETROIT`) results in an undefined patient record.

des (0, 12
0,"SELECT
1,"NAME_MO
6,"SELECT
15,"NAME_
32,"NAME_
93,"RETRI
69,"CLIEN
75,"NAME_M
84,"NAME_
91,"NAME_
99,"CLIEN
110,"RETR

Figure 3.1
Text

Searching f

Repository

Data Repos

```
des (0, 12, 121)
(0,"SELECT_PATIENT !VALID_USER_INPUT",1)
(1,NAME_MONITOR_GETTABLE,6)
(6,"SELECT_PATIENT !SELECT_PATIENT (VALID_USER_INPUT, DATAINDICESLIST)",15)
(15,"NAME_MONITOR_GETTABLE !EMPTY",32)
(32,"NAME_MONITOR_REGISTER !DETROIT !5678",53)
(53,"RETRIEVE_PROFILE !TUPLE_PATIENT_ID (LANSING, VALID_QUERY_REQUEST)",63)
(63,"CLIENT_QUERY !LANSING !VALID_QUERY_REQUEST",75)
(75,NAME_MONITOR_GETTABLE,84)
(84,"NAME_MONITOR_GETTABLE !INSERT (EMPTY, DETROIT, 5678)",91)
(91,"NAME_MONITOR_REGISTER !LANSING !1234",99)
(99,"CLIENT_QUERY !UNDEF_QUERY_RESULT",110)
(110,"RETRIEVE_PROFILE !UNDEF_PATIENT_RECORD",120)
```

Figure 3.10: Example of shortest path to state 120 in refined TRMCS model under test

Searching for patterns. Initial examination of these four shortest path examples suggests that failed queries can be attributed to the timing of Data Repository registration. That is, if the Client attempts to query a Data Repository before that Data Repository has registered with the Name Monitor, then the Client will not be able to make the connection and must return an undefined patient record. To further confirm our suspicions, we may use another CADP feature that allows us to search for patterns in the LTS graph. In this case, we are interested in paths leading to the return of an undefined patient record in response to a query (this pattern is shown in Figure 3.11). Either breadth-first or depth-first search may be used; we opt for depth-first search and obtain 43 examples at varying depths, only 14 of which pass the first query (*i.e.*, receiving a valid patient record from the LANSING Data Repository). One such example is shown in Figure 3.12. Examining these additional paths leads us to conclude that the undefined patient records are indeed the result of a delayed Data Repository registration with the Name Monitor.

<empty>

```
<until> [SELECT_PATIENT !VALID.*]
<until> [RETRIEVE_PROFILE !UNDEF.*]
```

Figure 3.11: Pattern: undefined patient records in response to valid queries

```
*** sequence found at depth 25

<initial state>
"SELECT_PATIENT !VALID_USER_INPUT"
"SELECT_PATIENT !SELECT_PATIENT (VALID_USER_INPUT, DATAINDICESLIST)"
"RETRIEVE_PROFILE !TUPLE_PATIENT_ID (LANSING, VALID_QUERY_REQUEST)"
"NAME_MONITOR_GETTABLE"
"NAME_MONITOR_GETTABLE !EMPTY"
"NAME_MONITOR_REGISTER !LANSING !1234"
"CLIENT_QUERY !LANSING !VALID_QUERY_REQUEST"
"NAME_MONITOR_GETTABLE"
"NAME_MONITOR_GETTABLE !INSERT (EMPTY, LANSING, 1234)"
"DATA_REPOSITORY_SEARCH !VALID_QUERY_REQUEST !1234"
"DATA_REPOSITORY_S1234 !VALID_QUERY_REQUEST !1234"
"i" (i)
"DATA_REPOSITORY_S1234 !VALID_QUERY_RESULT"
"DATA_REPOSITORY_SEARCH !VALID_QUERY_RESULT"
"CLIENT_QUERY !VALID_QUERY_RESULT"
"RETRIEVE_PROFILE !VALID_PATIENT_RECORD"
"SELECT_PATIENT !VALID_USER_INPUT"
"SELECT_PATIENT !SELECT_PATIENT (VALID_USER_INPUT, DATAINDICESLIST)"
"RETRIEVE_PROFILE !TUPLE_PATIENT_ID (DETROIT, VALID_QUERY_REQUEST)"
"CLIENT_QUERY !DETROIT !VALID_QUERY_REQUEST"
"NAME_MONITOR_GETTABLE"
"NAME_MONITOR_GETTABLE !INSERT (EMPTY, LANSING, 1234)"
"NAME_MONITOR_REGISTER !DETROIT !5678"
"CLIENT_QUERY !UNDEF_QUERY_RESULT"
"RETRIEVE_PROFILE !UNDEF_PATIENT_RECORD"
<goal state>
```

Figure 3.12: Failure on second query (DETROIT) in refined TRMCS model under test

Refinement checking. If the methods previously described do not yield conclusive results, then CADP offers an automated way to check refinement by comparing the LTS graphs of a model before and after refinement. Several bisimulation equivalences [61, 73] are available, as well as observational [73] and safety relations [74]. The comparisons offer a diagnostic counterexample at the first indication that the LTSs differ. For illustrative purposes we compare the model under test before and after refinement, subject to *minimization* under the safety relation and *hiding* of the aggregate objects' services (to make the LTS graphs amenable to visual inspection).

By visually comparing the LTS graphs in Figure 3.13 (the high-level LOTOS model derived from the original OMT diagrams, under test) and Figure 3.14 (the LOTOS model derived from the refined OMT diagrams, under test), we immediately see that the refined model has two opportunities to fail the test (in Figure 3.14, the arcs from node 4 and node 7 to node 9). The high-level model (Figure 3.13) treats the *Retrieve Profile* service as a function, so that valid queries (arcs from node 2 to node 3 and from node 6 to node 7) always result in valid patient records (arcs from node 3 to node 4 and from node 7 to node 8). However, the refined model realizes the high-level *Retrieve Profile* service as a collaboration of aggregate objects' services. If we attribute the failed queries to the nature of distributed computation, that is, the possibility in our model that the Client may pose a query before the desired Data Repository has registered, then, in this context, the refined TRMCS realizes the *Retrieve Profile* service transparently. If, however, the system requires that all Data Repositories be online and accessible all the time, then the current design is not satisfactory.

3.1.3 Analysis Tool Output

Overall, CADP has much more user-friendly output than TOPO/LOLA. During the case studies [45, 46], we were able to output both text-based paths through the LTS

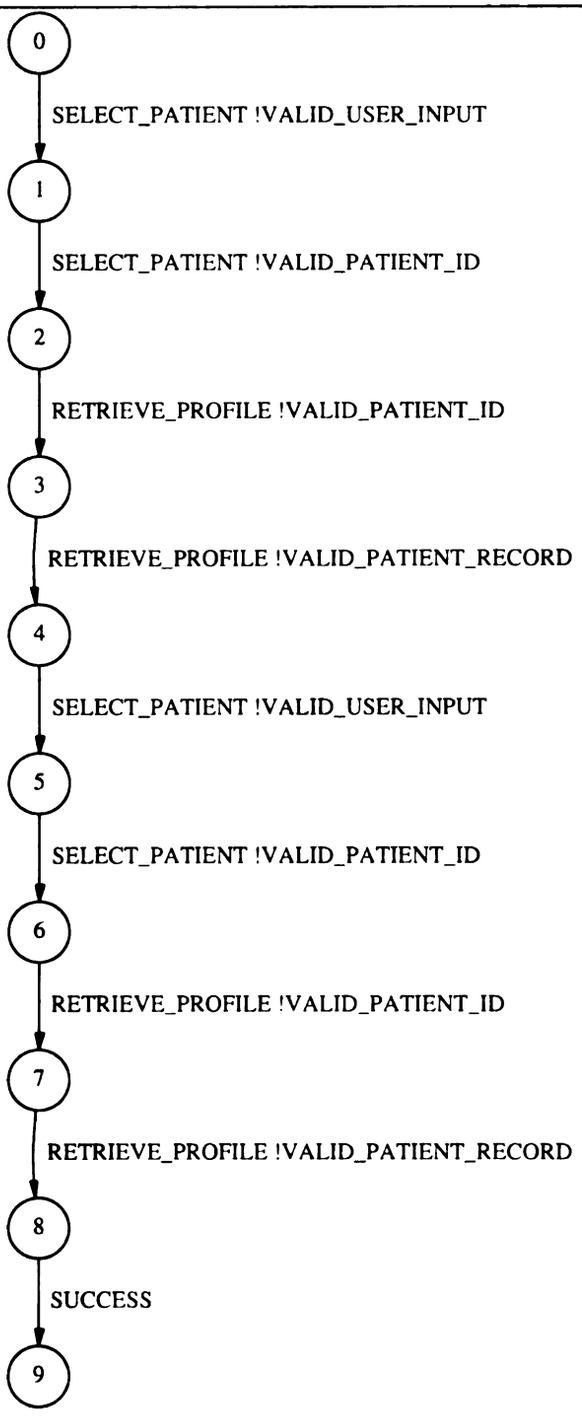


Figure 3.13: Original TRMCS model under test

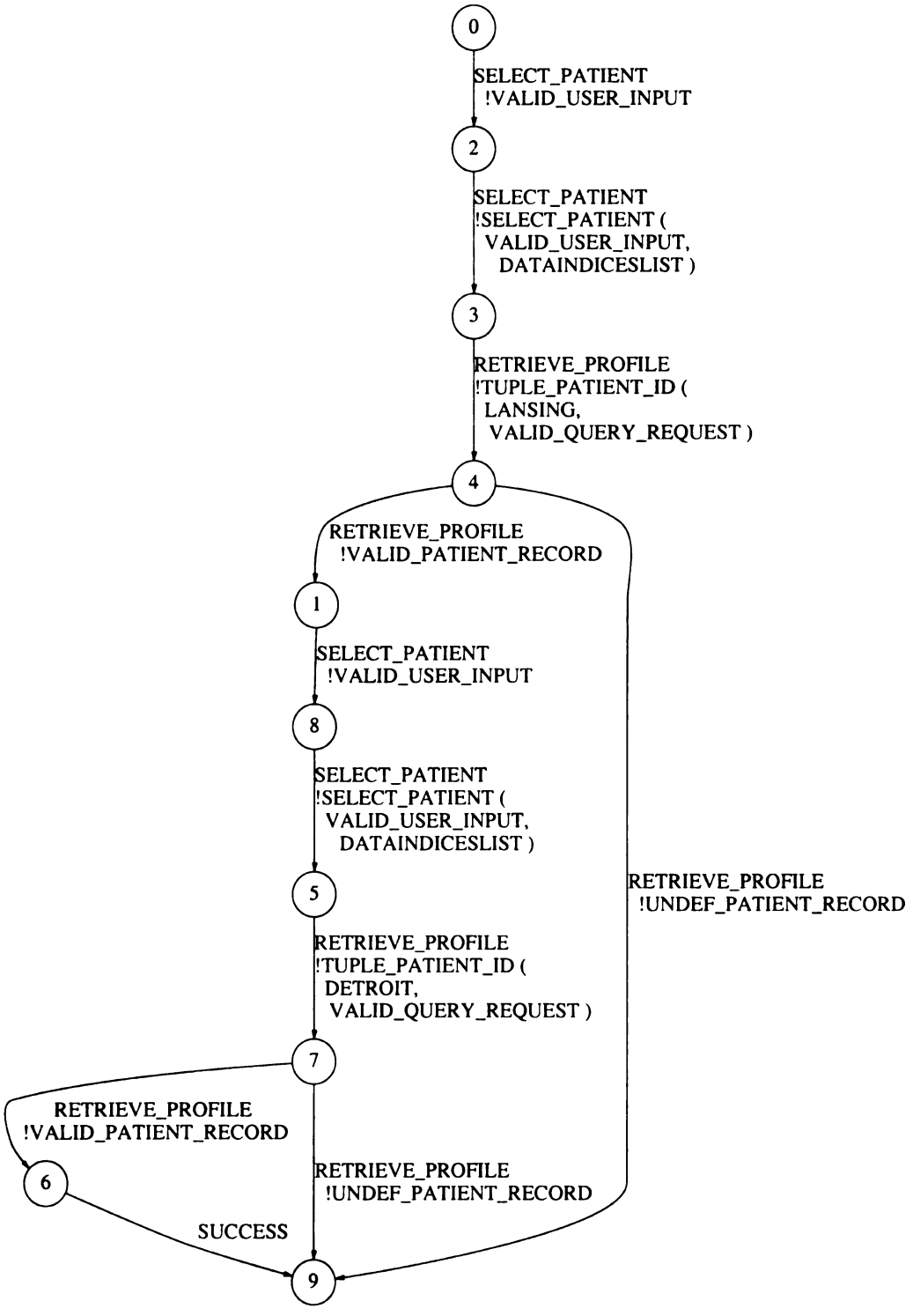


Figure 3.14: Refined TRMCS model under test, safety-reduced

graph matching certain criteria (such as the shortest path to a particular state or matching a given pattern of actions), and graphical depictions of minimized LTS graphs (a feature not available in TOPO/LOLA). CADP's graphical depiction of an LTS graph can act as a "road map" for simulations and facilitate understanding of failed test cases (Figure 3.14), while its text-based output is easily read by humans.

On the other hand, TOPO/LOLA's cryptic text-based output informs the developer that deadlocks exist, but makes the developer search for the deadlocks in a cumbersome log file and manually back-trace the sequence of actions that led to them. For example, in [45] TOPO/LOLA found four deadlocks in initial refinements of the requirements model for ENFORMS [71] that, after examining the log file (see Figure 3.15 for excerpts), we attributed to the Client being unable to submit a query with an undefined address. We first had to search the log file for the keyword **stop**, indicating a deadlock, and then manually back-trace each process's instantiation to figure out the sequence of actions leading to the deadlock. Adding a guarding condition to the dynamic model of the Client to check for undefined Archive Server addresses resolved these deadlocks.

3.2 Lessons Learned

In addition to the highlights, we learned two lessons from these case studies [45, 46] that have motivated our later work with McUmber's UML-to-Promela formalization [31] and the Spin [69] analysis tools. First, structural analyses (*i.e.*, diagram consistency checking) should be performed prior to formal model generation because formal model analysis tools do not always detect diagram inconsistencies. Second, in order to be useful, analysis results should be related back to the diagrams. This section briefly overviews these lessons and how they have affected our later work.

```
......
process duplicate3 [format_request,retrieve_data,analyze_data,client_query,
        name_server_gettable,archive_server_query,name_server_register,
        archive_server_q4899,archive_server_q5699] : noexit :=
        name_server_register  ! storet ! 5699;
   stop
endproc
......
process duplicate4 [format_request,retrieve_data,analyze_data,client_query,
        name_server_gettable,archive_server_query,name_server_register,
        archive_server_q4899,archive_server_q5699] : noexit :=
        name_server_register  ! pcs ! 4899;
   stop
endproc
......
process duplicate9 [format_request,retrieve_data,analyze_data,client_query,
        name_server_gettable,archive_server_query,name_server_register,
        archive_server_q4899,archive_server_q5699] (rr_92:retrieve_request)
        : noexit := archive_server_query  ! retreq_getqueryrequest(rr_92)
        ! undef_address;
   stop
endproc
......
process duplicate17 [format_request,retrieve_data,analyze_data,client_query,
        name_server_gettable,archive_server_query,name_server_register,
        archive_server_q4899,archive_server_q5699] (rr_92:retrieve_request)
        : noexit := archive_server_query  ! retreq_getqueryrequest(rr_92)
        ! undef_address;
   stop
endproc
......
```

Figure 3.15: TOPO/LOLA log file indicating deadlock in ENFORMS

Structur

FORMS [T

II. W

Structural Analysis of Diagrams. In [45], in order to validate Wang's approach [27] we applied Wang's mapping rules to object-oriented diagrams for ENFORMS [71]. During this process, we discovered (by visual inspection), but did not at first correct, several diagram inconsistencies. In many cases, the signature of a given message send in one state diagram did not match the signature of a corresponding event, the given message's reception, in another state diagram (*e.g.*, parameters were transposed, *etc.*). However, we found that the TOPO analysis tool was incapable of detecting these errors in the resulting formal model (*i.e.*, syntax and semantics checking passed).

In Wang's formalization of OMT diagrams into full LOTOS models [27], a message, possibly with parameters, from the state diagram of a sender object is received by a target object as an event on a transition in the target object's state diagram. The variables in the event's parameter list (representing attributes of the target object) take on the values passed in the message. An example of inconsistent state diagrams is shown in Figure 3.16. The diagrams are inconsistent because the signature for message `foo`'s reception on the transition in Object B's state diagram differs from the signature for sending message `foo` to Object B on the transition in Object A's state diagram (x is an `Int` and y is a `String`). However, in terms of LOTOS, which does not require that processes must synchronize, there is nothing syntactically or semantically wrong with the model fragments shown in Figure 3.17 derived from the diagrams in Figure 3.16, even though the processes will not synchronize and exchange data as intended.

This example underscores the motivation for consistency checking at the diagram level, as well as for automated instead of manual translation of diagrams to models. Consistency checking at the diagram level is necessary to detect inconsistencies that might otherwise pass target language syntax and semantics checking and introduce subtle errors into the model. Chapter 5 discusses how we use this approach in our

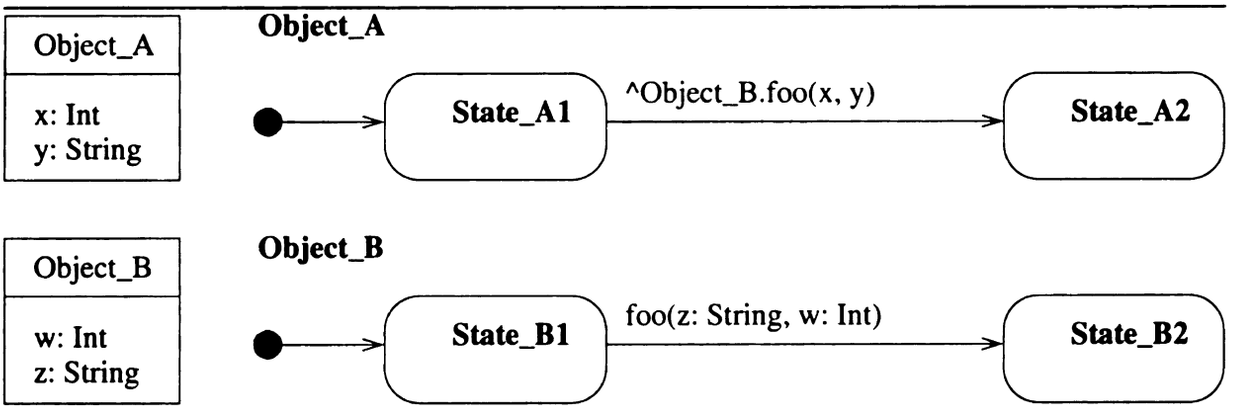


Figure 3.16: Inconsistent OMT state diagrams

```
process A (sender)   :  foo !  x !  y;
process B (receiver) :  foo ?  z:  String ?  w:  Int;
```

Figure 3.17: LOTOS model fragments from inconsistent diagrams

work with McUmber's UML-to-Promela formalization [31].

Visualization of Analysis Results in Terms of Diagrams. Our experiences demonstrate that the visualized LTS graph produced by CADP can act as a "road map" for simulations and facilitate understanding of failed test cases, much more so than TOPO/LOLA's cryptic text-based output. However, Wang's formalization does not include information other than event names and parameters on arc labels in the LTS, and thus analysis results cannot be automatically mapped back to state diagrams (the same event may appear on multiple transitions so a simple name lookup scheme fails). Instead, the developer must infer the relationship between the analysis results, including the visualized graph, and the original diagrams, based in part on knowledge of Wang's mapping rules. One approach to this problem is to push diagram-specific knowledge into the formal model so that such information can be recovered from analyses of the formal model. Chapter 6 discusses how we use this approach in our work with McUmber's UML-to-Promela formalization [31].

Chapter 4

Model Development and Analysis: Framework and Process

This chapter describes a model development and analysis framework that addresses model creation, model analysis, and model refinement based on analysis results. This framework extends the formal model generator architecture suggested by Wang's [27] and McUmber's [31] approaches, including well-formedness checks and visual feedback to diagrams (Chapter 3, Section 3.2). To provide context for the following chapters, we instantiate this framework with concrete tools that support McUmber's UML-to-Promela mapping rules [31], and describe a model development and analysis process that pertains to this instantiation of the framework. This process encapsulates the steps of model creation, model analysis, and model refinement, and it serves as a roadmap for the analyses discussed in Chapter 5 and the visualizations of analysis results discussed in Chapter 6.

Formal Model Generator Architecture. Both Wang's [27] and McUmber's [31] approaches suggest an architecture similar to what is shown in Figure 4.1 to realize a formal model generator based on mapping rules from a (source) semi-formal

language to a (target) formal language.[1] Each mapping approach uses a graphical notation (*e.g.*, OMT, UML) as the semi-formal language, which implies the use of a **Graphical Editor** for that particular notation. The user draws graphical diagrams for an object-oriented system based on **(prose) requirements**. A **Translator** that incorporates knowledge of the **mapping rules** from a given graphical notation to a particular formal language (*e.g.*, OMT-to-LOTOS, UML-to-Promela) generates the formal model corresponding to the (intermediate representation of) graphical diagrams drawn by the user. Together, the **Graphical Editor** and **Translator** realize a formal model generator. However, formal model generation is not an end in itself. As shown in Figure 4.1, the user then performs analyses on a generated formal model with appropriate **Formal Language Analysis Tools** (*e.g.*, TOPO/LOLA, Spin). Such tools may require interaction from the user, as illustrated by the "commands" data flow. If the formal language allows, the user may optionally supply **properties** (as indicated by the dashed arcs in Figure 4.1) for the formal language analysis tool(s) to check against the formal model (*e.g.*, a test process to compose with a LOTOS model, an LTL property to check against a Promela model). As indicated by the dash-dotted arc in Figure 4.1, the user must interpret (raw) analysis results obtained from the formal language analysis tool(s).

Problems with Suggested Architecture. Both approaches assume the well-formedness of diagrams, including diagram integration constraints. Therefore, neither addresses the problems that ensue from attempting to generate formal models from ill-formed diagrams, other than expecting that syntax/semantics checking or other analyses provided by formal language analysis tool(s) will detect such errors at the formal model level. However, as discussed in Chapter 3, some diagram ill-formedness errors are not readily detectable at the formal model level.

Additionally, as represented by the dash-dotted arc in Figure 4.1, both approaches

[1] Appendix C illustrates the architectures realized in these approaches.

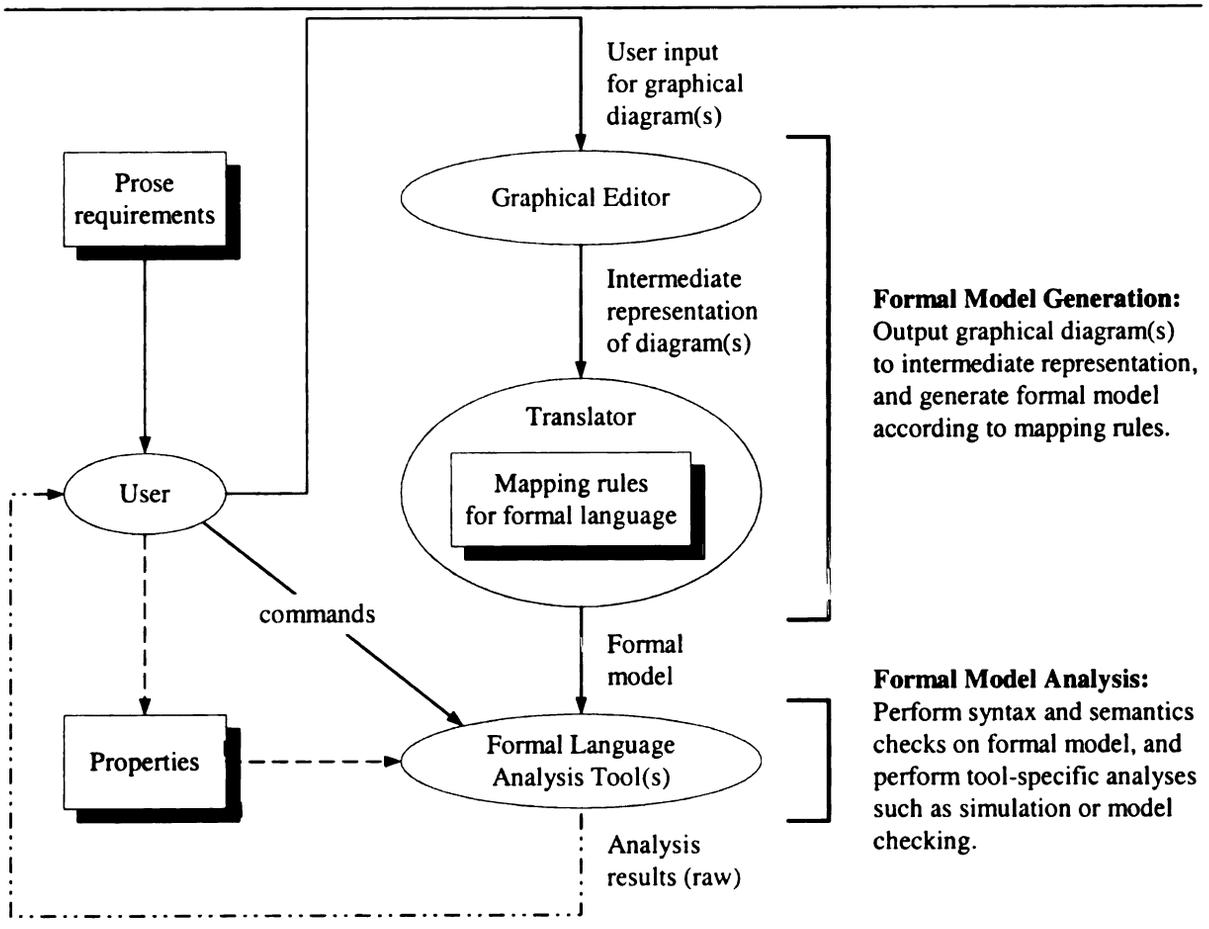


Figure 4.1: Suggested architecture for a formal model generator

expect the user to interpret raw analysis results emanating from formal language analysis tool(s). As noted in Chapter 3, sometimes these results are quite cryptic. No mechanism is provided for relating analysis results back to the original diagrams in order to guide diagram refinement.

Model Development and Analysis Framework. Our model development and analysis framework, shown in Figure 4.2, extends the formal model generator architecture from Figure 4.1 in two ways. First, although we still split the task of analysis into two parts, structural and behavioral, we apply structural analyses to the *diagrams* (*i.e.*, prior to formal model generation) to ensure intra- and inter-diagram consistency, thus placing this burden on tools that are more suited to the task than formal language analysis tools. Formal language analysis tools then perform behavioral analyses on formal models generated from well-formed diagrams.

Second, as represented by the bold arcs to the **Graphical Editor & Visualization Environment** oval in Figure 4.2, results from both types of analyses are visualized in terms of graphical diagrams. Structural analyses are visualized in terms of the original diagrams in order for the user to locate and correct diagram inconsistencies. Raw analysis results from behavioral analyses are processed by an **Analysis Result Processor** (shown as a bold oval) and then visualized in terms of either the original diagrams or complementary ones, thus relating analysis results from the formal model level back to the diagram level to guide diagram refinements. (Raw analysis results from behavioral analyses may also be processed into a human-readable report to complement visualizations, as represented by the bold dash-dotted arc to the **User** oval in Figure 4.2.) Therefore the graphical editor of Figure 4.1 becomes a **Graphical Editor & Visualization Environment** (shown as a bold oval in Figure 4.2).

While current work [40, 41, 42, 53] has focused on a subset of the UML graphical notation appropriate to embedded systems, generated Promela models (according to

McUmber's UML-to-Promela mapping rules [31]), and the Promela analysis tools of Spin [69], future work may include extending the framework in a third dimension to incorporate other formal languages and tools, such as SMV[2] [75].

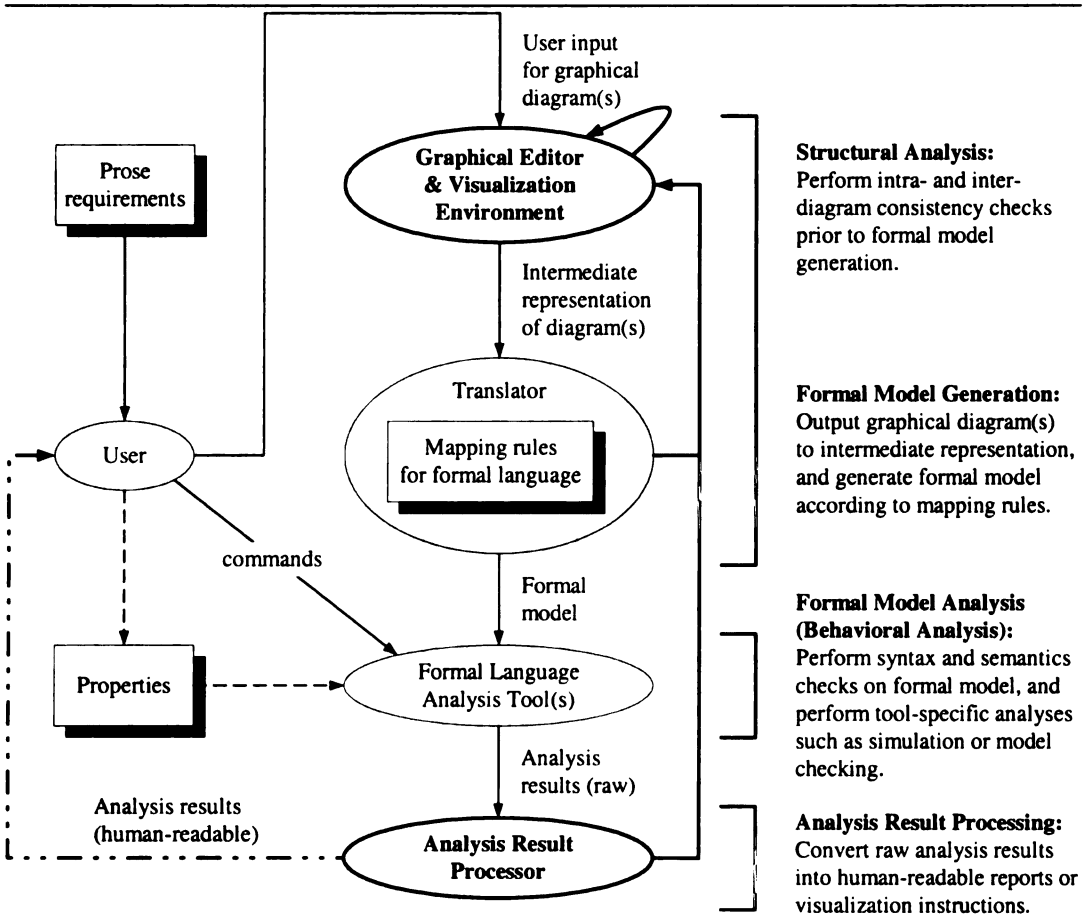


Figure 4.2: Model development and analysis framework

Instantiated Framework. The formalization of UML enables various types of analyses of the UML diagrams and corresponding formal model of an object-oriented software system, depending on the mapping rules given and the formal language analysis tools(s) available for the chosen target formal language. To provide the context

[2]Symbolic Model Verifier

for the follow

framework wi

rules [3]. Bu

model-level p

UML diagram

Figure 4.3

Editor & Vi

5], [2], et. al. t

in UML-to-[

MINERVA et

UML class

a step to

analyses of

Formal Lan

If using

property

as in Figu

fication pa

have four

formal pro

Section 2.2

Finally,

ERVA in P

of Petri Ne

Figure 4.2

for the following chapters, we first instantiate the model development and analysis framework with concrete tools that support McUmber's UML-to-Promela mapping rules [31]. Based on this instantiation, we then describe an iterative and incremental model development and analysis process to create, analyze, and refine (formalized) UML diagrams. This process serves as a roadmap for subsequent chapters.

Figure 4.3 refines the framework from Figure 4.2, instantiating the **Graphical Editor & Visualization Environment** with our (UML) tool MINERVA [40, 47, 50, 51, 52], and the **Translator** with McUmber's tool Hydra [31, 48] that incorporates his UML-to-Promela mapping rules [31]. Parts A and B, respectively, indicate that MINERVA and Hydra can be used cooperatively both to perform structural analyses on UML (class and state) diagrams drawn by the user, and to generate formal models in a (supported) target language (in this case, Promela). The user performs behavioral analyses of generated Promela models with Spin [69] (Part C), which instantiates the **Formal Language Analysis Tool(s)** process oval of Figure 4.2.

If using Spin in model-checking mode, the user may supply Spin with an **LTL property** (Part D) to check against the Promela model. As indicated by the dashed arc in Figure 4.3, we recommend but do not require the use of Dwyer *et al.*'s **specification patterns** [43] to guide the creation of LTL properties [40, 41, 42, 53]. We have found the specification pattern taxonomy to be useful when attempting to write formal properties based on English natural language requirements (see Chapter 2, Section 2.2.4 for a brief overview of specification patterns).

Finally, as represented by the bold arcs from MINERVA, Hydra, and Spin to MINERVA in Part E, a combination of plug-in functions within MINERVA and a collection of Perl scripts handle the responsibilities of the **Analysis Result Processor** from Figure 4.2.

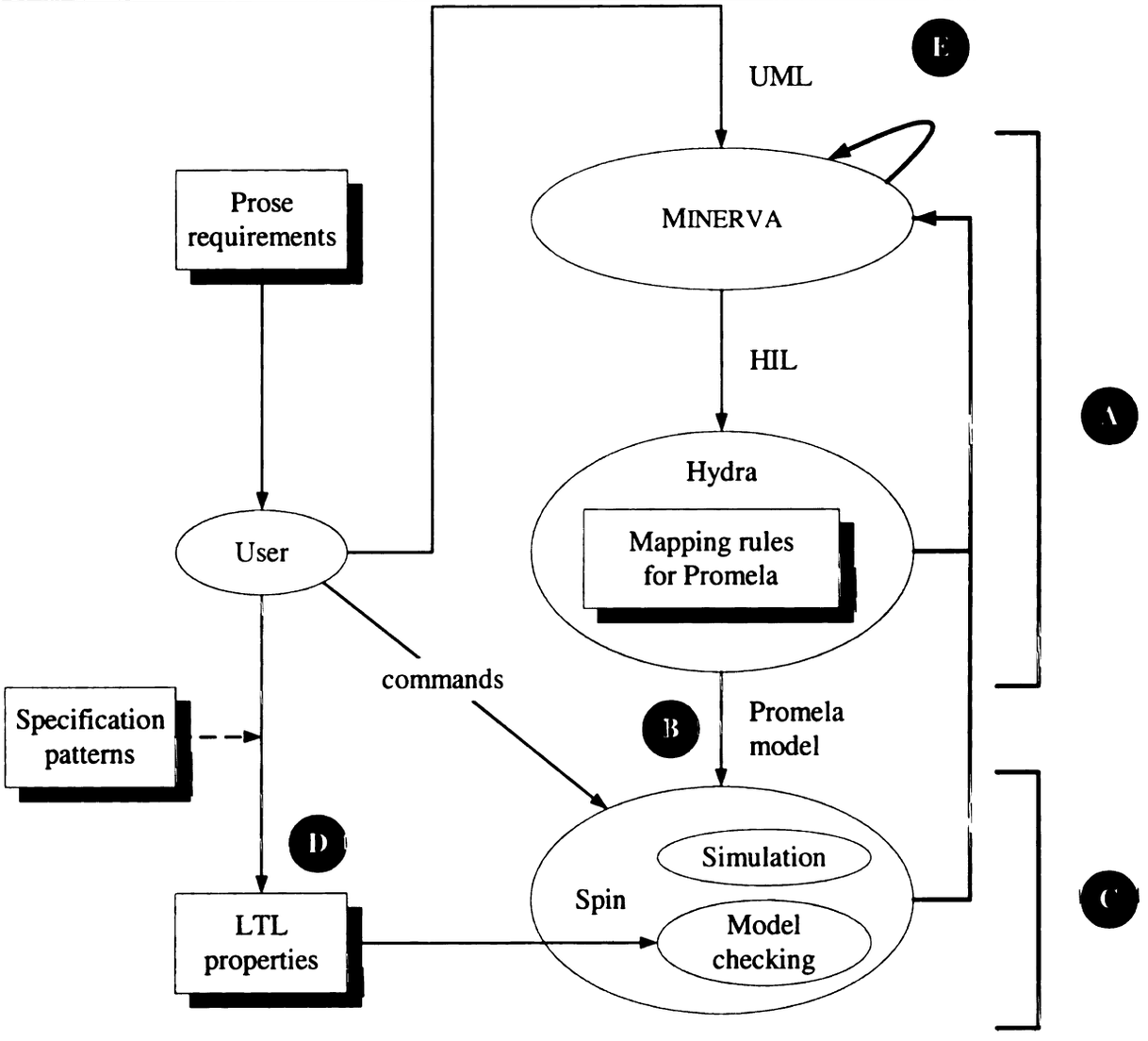


Figure 4.3: Framework from Figure 4.2 instantiated with tools

Model Deve

Figure 4.4. In

Step B

Model Development and Analysis Process. The model development and analysis process is iterative and incremental. As shown in the process flow diagram in Figure 4.4, in Step A, the user begins by drawing UML diagrams in MINERVA's graphical editors for the class and state diagrams. The class diagram is required, and then additional state diagrams may be associated with individual classes. Assuming no errors are introduced at the diagram level, in Step C MINERVA produces an intermediate representation of the diagrams, and in Step E, Hydra translates the intermediate representation into a Promela model (according to UML-to-Promela mapping rules). Spin can then be used either in simulation mode (Step F) to explore behavior such as requirements or usage scenarios, or in model checking mode (Step G) to check requirements-based properties.

Steps B and D shown as bold rectangles (and Feedback lines 1 and 2 shown as bold dotted arcs) in Figure 4.4 were not present in Wang's and McUmber's approaches. In general, it was left to the formal model analysis tool(s) to detect syntax and semantics errors at the formal model level with no direct feedback to the diagrams to indicate the source of the errors. We argue that diagram inconsistencies are better detected, and visual feedback of these errors is more easily achieved, at the diagram level (that is, prior to formal model generation). Chapter 5, Section 5.1, describes the structural analyses available. Discussion of how visual feedback from structural analyses (Feedback lines 1 and 2) is accomplished is deferred until Chapter 6.

Analysis feedback support as depicted by Feedback lines 3 and 4, shown as bold dotted arcs in Figure 4.4, were also not available in Wang's and McUmber's approaches. Chapter 5, Section 5.2, discusses the behavioral analyses afforded by Spin, namely simulation and model checking. In Chapter 6 we show how both UML model simulation and simulation of counterexamples[3] with Spin (Steps F and H, respectively) can produce trace data that can be displayed directly in the context of the

[3]A counterexample produced by the Spin model checker can be retraced with Spin in guided simulation mode in order to produce trace output, normally suppressed in model checking mode.

Structural Analyses
(diagram level)

Behavioral Analyses

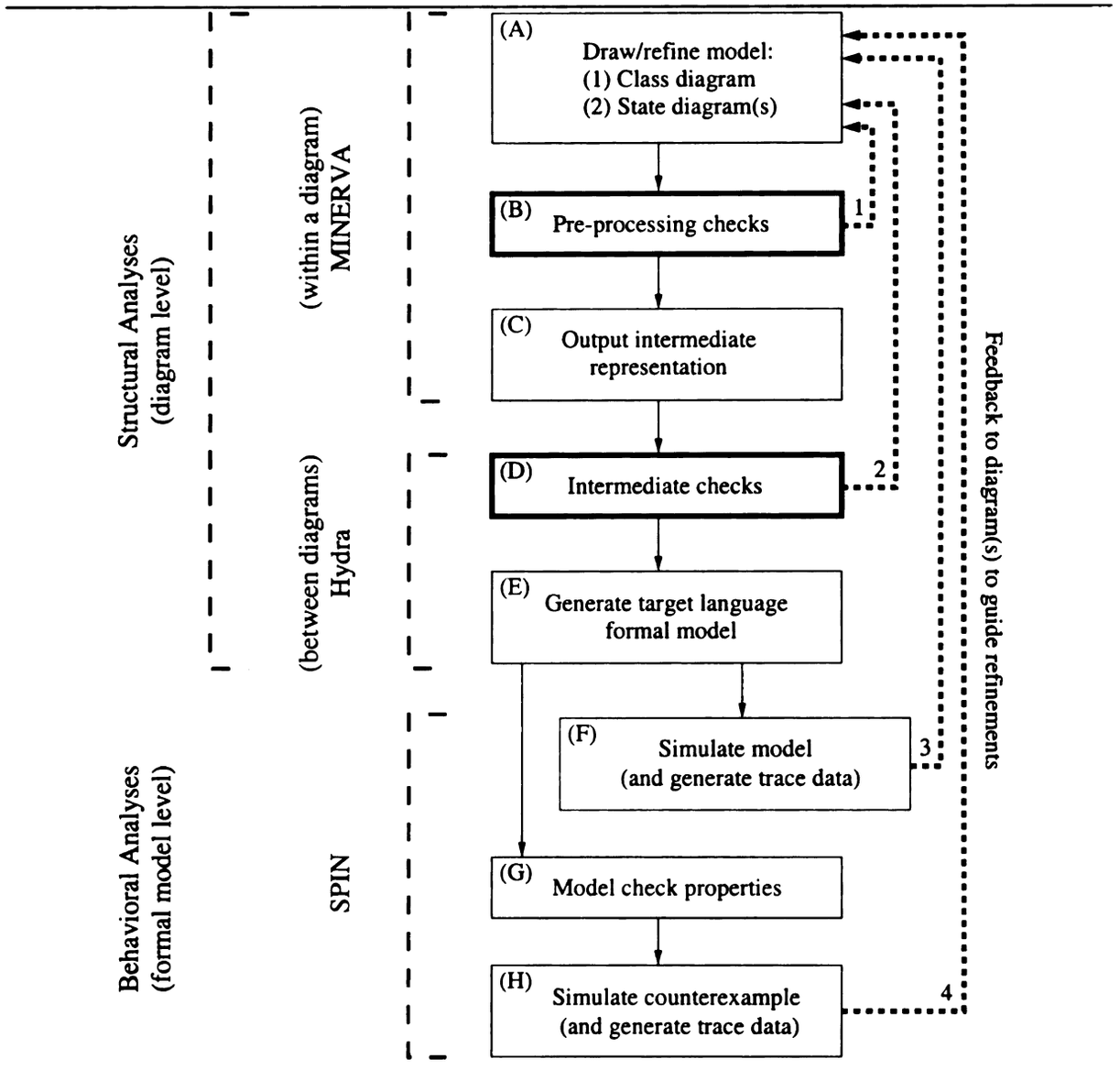


Figure 4.4: Iterative and incremental model development and analysis process

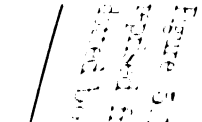
diagrams using MINERVA (Feedback lines 3 and 4) to aid in the debugging and refinement of the UML diagrams.

Chapter 5

Analyses

This chapter discusses the types of analyses enabled by the formalization of semi-formal object-oriented graphical modeling notations such as OMT or UML. For illustration purposes, we focus primarily on the analyses enabled by McUmber's [31] formalization of UML diagrams with Promela, the input language for the analysis tool Spin [69]; thus, the model development and analysis process described in Chapter 4 for use with the model development and analysis framework instantiated in Chapter 4 is reproduced in Figure 5.1 for convenience to the reader. This chapter discusses the bolded and dash-dotted boxes in the process flow diagram.

As shown in Figure 5.1, analyses are divided into two categories, structural and behavioral. (Structural analyses are indicated with bolded boxes, and behavioral analyses are indicated with dash-dotted boxes.) Structural analyses, discussed in Section 5.1, pertain to intra- and inter-diagram consistency checking with respect to several criteria, while behavioral analyses available, discussed in Section 5.2, depend on the target formal language and formal language analysis tool(s) chosen. For example, as shown in Figure 5.1, McUmber's UML-to-Promela formalization [31] affords both simulation (Step F) and model checking (Step G) capabilities via Spin [69]. In contrast, as discussed in Chapter 3 and [45, 46], Wang's OMT-to-LOTOS formaliza-



Behavioral Analyses

Structural Analyses
(deemphasized)

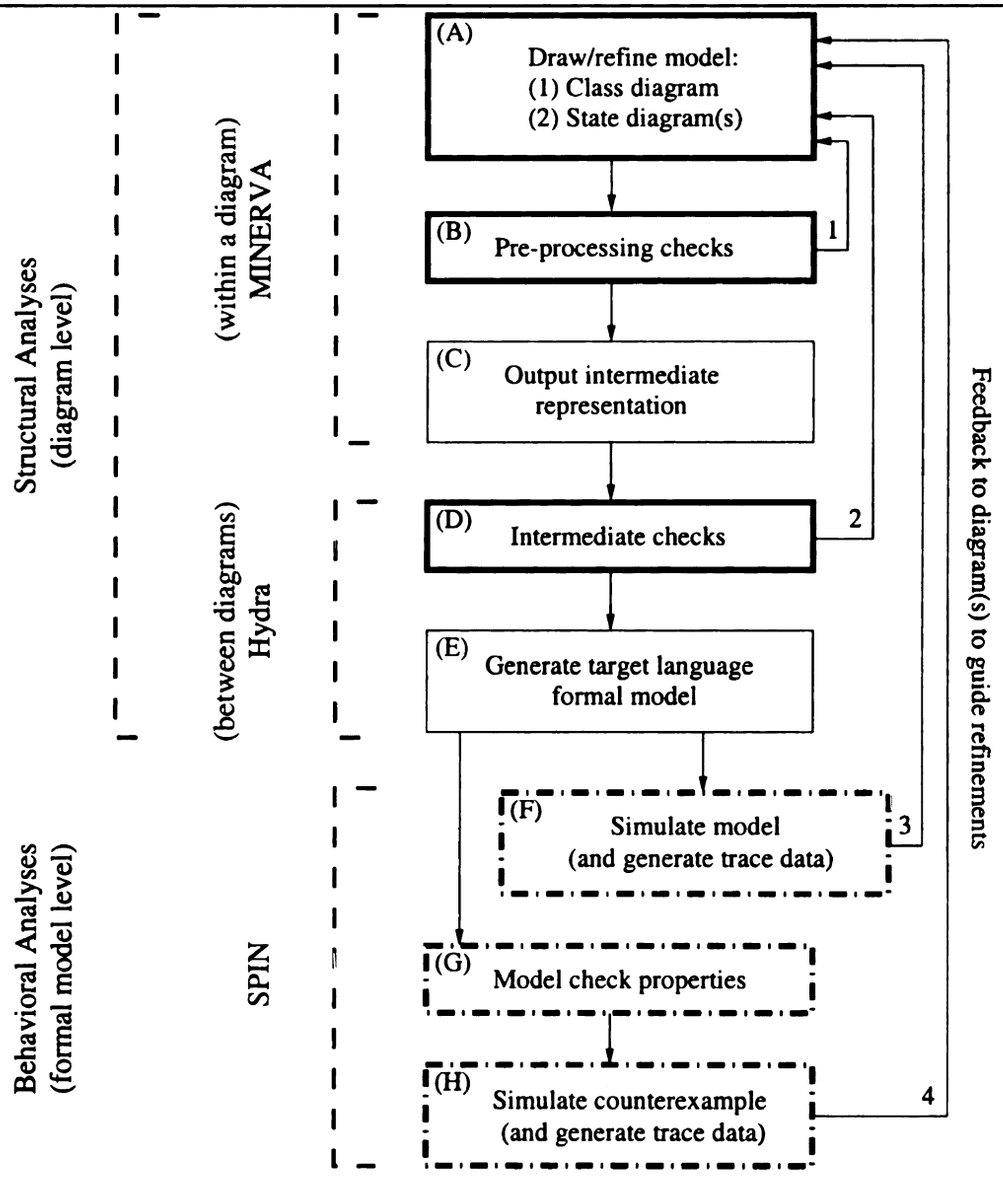


Figure 5.1: Iterative and incremental model development and analysis process, reprised from Figure 4.4. Bold boxes (A, B, D) represent structural analyses; dash-dotted boxes (F, G, H) represent behavioral analyses.

5.1 Str

tion [27] enables simulation, concurrency analyses, test composition, and refinement checking via TOPO/LOLA [66] and CADP [67].

5.1 Structural Analyses

Structural analysis of object-oriented graphical diagrams refers to consistency checking at the diagram level. Consistency checks cover graphical and textual syntax inherent to the object-oriented graphical modeling notation used (*e.g.*, OMT, UML), well-formedness assumptions and integration conventions inherent to the notation used or imposed by the formalization rules, and constraints relative to the intended target formal language.

For ease of exposition, we group the discussion of structural analyses into three categories, which correspond to Steps A, B, and D of Figure 5.1, respectively. Section 5.1.1 describes to what extent the graphical editing environment can prevent certain types of graphical syntax violations from being drawn (Step A), or guide the user towards creating consistent diagrams. Section 5.1.2 presents many types of checks performed within individual class and state diagrams (Step B), while Section 5.1.3 enumerates several integration checks performed between the class diagram and each state diagram, and among all state diagrams (Step D). We use the term *problem* to denote an aspect of a diagram that warrants user attention. The severity of a problem and its potential consequences dictate whether it is classified as a *warning* or as an *error*. Problems categorized as warnings do not have adverse effects on the behavior of the generated formal model (*e.g.*, defining an attribute in a class in the class diagram but not using it in the corresponding state diagram). On the other hand, problems categorized as errors either indicate syntactically ill-formed diagrams (*e.g.*, giving two states the same name in a state diagram) or have the potential to invalidate a generated formal model (*e.g.*, a state diagram expecting an event that is

never sent as

potential to

therefore m

errors. Error

5.1.1 Pr

The graph

derives fro

We prior

Prevention

posed by a

syntax, thus

Note that

a syntactic

State discr

Hard block

learning Sea

syntactic

the transp

A generic

and thereby

mean accu

diagram

ing hidden

"Over sp

sequences

for an

never sent as a message from any state diagram in the model, an error that has the potential to cause deadlock). We use a conservative approach to classify problems; therefore, most problems described in the remainder of this section have been deemed errors. Errors must be corrected prior to formal model generation.

5.1.1 Preventing or Containing Diagram Errors

The graphical editing environment can prevent certain types of graphical syntax violations from being drawn, or guide the user towards creating consistent diagrams. We refer to these strategies as *prevention* and *containment*, respectively.

Prevention. Consistency with a graphical syntax can, to a large extent, be imposed by a graphical editing tool that has been enhanced by knowledge of the correct syntax, thus preventing many graphical syntactical errors from ever being drawn. Note that in an ordinary drawing application such as `xfig` [76], it is possible to draw a syntactically incorrect UML state diagram[1] like the one shown in Figure 5.2. The state diagram shows a transition (directed arc) from an initial pseudostate (small filled black circle) to a state (rounded rectangle labeled `State_A`), and a transition leaving `State_A` (directed arc labeled with the event *event1*). This state diagram is syntactically incorrect with respect to the graphical syntax of state diagrams because the transition labeled *event1* has a source state (`State_A`) but no destination state. A generic drawing application like `xfig` has no knowledge of state diagram syntax, and therefore does not impose any relationships (or constraints on relationships) between filled circles, rounded rectangles, and directed arcs. Further, although the state diagram example in Figure 5.2 uses only pseudostates, states, and transitions, nothing inherent in a generic drawing application like `xfig` would prevent the user from

[1]Our example pertains to the UML state diagram, but the general concept also applies to other diagrams, such as the UML class diagram, or other graphical notations, such as OMT. See Chapter 2 for an overview of OMT and UML syntax.

drawing sq...

of a state di...

Figure 5.2.

However...

X, S1, S2...

constructs a...

corresponding...

state diagram...

with a gray...

diagram gr...

state diagr...

the state d...

Draw. Py...

boolean p...

Figure 5

Contain

Each forc...

Draw a...

Boolean...

drawing squares, triangles, and undirected lines, none of which is a valid component of a state diagram.

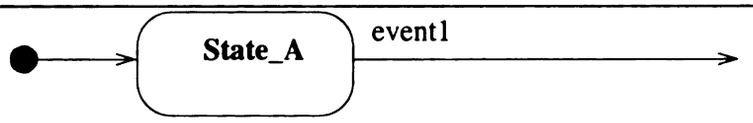


Figure 5.2: Syntactically incorrect state diagram, with respect to graphical syntax

However, a metamodel[2]-based graphical editing tool, such as MINERVA [40, 47, 50, 51, 52] which supports both UML class and state diagrams, can prevent the construction of diagram components that are inconsistent with the metamodel for the corresponding type of diagram. For example, as shown in Figure 5.3, a transition in a state diagram must have a source and destination state and cannot be drawn otherwise with a graphical editing tool such as MINERVA that comprehends and enforces state diagram graphical syntax. Further, unlike a generic drawing application, MINERVA's state diagram editor allows only those diagram components that are consistent with the state diagram metamodel, that is, pseudostates, states, and transitions, to be drawn. Preventing such well-formedness errors decreases the consistency-checking burden placed on other tools.

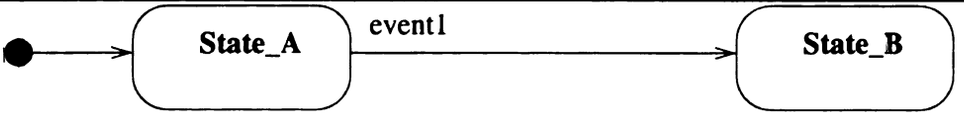


Figure 5.3: Syntactically correct state diagram, with respect to graphical syntax

Containment. In addition to a graphical syntax, textual annotations are a common feature of many graphical notations for object-oriented systems. OMT and

[2]Recall from Chapter 2 that a *metamodel* is a class diagram that describes the constructs of a modeling language and the relationships between the constructs.

UML in par

operation sh

diagrams to

between sta

syntax to d

for a transfo

and Hydra

placed with

provided by

is used in

UML in particular use text in class diagrams to label classes, to list attributes and operation/signal signatures, and to label relationships. They also use text in state diagrams to label states, and to describe events, guards, and actions on transitions between states. Many kinds of textual annotations in UML have their own suggested syntax to a certain extent. For example, the suggested textual annotation syntax for a transition label in a state diagram is described in Expression (5.1), where the *actionExpression* may comprise a sequence of actions, some of which may be message sends [18]. The transition syntax currently supported by MINERVA [40, 47, 50, 51, 52] and Hydra [31, 48], described in Expression (5.2), allows message sends to be displayed either as actions (*e.g.*, `send(targetObjectName.signalName)`) or separately, preceded by a caret (*e.g.*, ^`targetObjectName.signalName`). This transition syntax is used in the following examples.

$$eventSignature\ [\ guardCondition\ ]\ /\ actionExpression \tag{5.1}$$

eventSignature	$eventName$
	$eventName(param_1, \ldots, param_i)$
guardCondition	$booleanExpression$
actionExpression	$actionList\ messageList$
actionList	$action_1; \ldots; action_j$
messageList	$\hat{}message_1\hat{} \ldots \hat{}message_k$

(5.2)

In a graphical editing tool that enforces the graphical syntax of state diagrams but places no restrictions on the entry of textual annotations such as transition labels, it would be possible to draw the state diagram shown in Figure 5.4. This state diagram, while obeying graphical syntax, has a nonsensical string as its transition label.

Between the metamodel for overall diagram syntax and the user interface for a diagram editing tool, it is possible to offer the user some structured guidance for

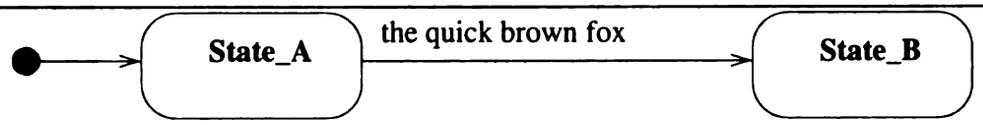


Figure 5.4: Nonsensical transition label

entering textual annotations. In MINERVA, for example, we present the user with separate text entry areas for the event name, parameter list, guarding condition, list of actions, and list of messages in a state diagram transition label. MINERVA combines all the information with the proper delimiting characters into one label for display. Enforcing the general syntactical structure of transition labels may also assist a parser or other consistency checking utility in pinpointing errors, for example in the event name, guarding condition, *etc.*.

Even with such guidance, constructing structured nonsense may still be possible (although not very likely since we assume users generally have a valid transition label in mind), as shown in Figure 5.5. To prevent the nonsense transition label in this example, additional guidance could be provided to the user in terms of pop-up menus for state transition label components populated with information gleaned from the class diagram. However, diagram integration rules would need to be coded into the diagram editing environment, making it needlessly complex to maintain and slow to execute. Additionally, any item desired to be displayed in a transition label would first have to be entered in the class diagram, forcing the user to switch back and forth between diagrams. Bookkeeping would also need to be performed to invalidate transition labels that use information subsequently deleted from the class diagram. Text entry for items such as class, attribute, operation/signal, and state names could not be restricted, so parsing and consistency checking would not be eliminated. We have found that using structural guidance to alleviate most textual syntactical errors, parsing to detect any remaining syntactical errors, and consistency checking to detect

definition/usage and other discrepancies, including graphical syntax anomalies such as a missing start state, effectively distributes the responsibilities for consistency checking between the diagram editing environment and other utilities.

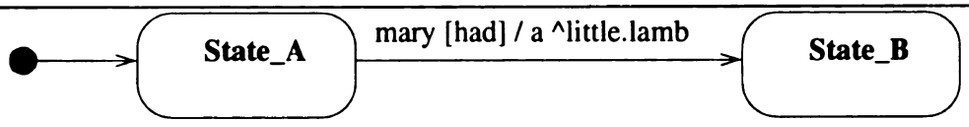


Figure 5.5: Structured nonsense on a transition

5.1.2 Problems Within a Diagram

This section discusses problems within individual diagrams; that is, the class diagram and each state diagram. A diagram and its elements are checked for graphical and textual syntax problems and violations of well-formedness assumptions that could not be prevented by the graphical editing environment, such as a missing start state in a state diagram or an invalid class, operation/signal, or attribute name in a class diagram. Diagram elements may also be checked for consistency with the target formal language in terms of reserved word usage. Figures 5.6 and 5.7 list **class diagram well-formedness assumptions** (overall diagram assumptions and assumptions about the various parts of the class diagram). Figure 5.8 lists **state diagram well-formedness assumptions**.

5.1.3 Problems Between Diagrams

This section discusses problems between diagrams; that is, between the class diagram and each state diagram, and between state diagrams. The class diagram provides the context for other diagrams; therefore, an important type of consistency check between the class diagram and each state diagram is definition/usage check. Such

- Naming
  - A valid name depends on the formalization rules and target language constraints (such as no whitespace, no special characters, must start with a particular letter, *etc.*)
  - Name uniqueness among classes
  - Name uniqueness among attributes within a class
  - Name uniqueness among operations/signals within a class
  - No reserved words from target language used as names
- Data Types
  - A valid data type depends on the formalization rules and target language constraints (for example, Promela does not support real numbers as a data type)
- Inheritance
  - No circular inheritance (formal model generation cannot handle cycles)
  - No multiple inheritance

Figure 5.6: Overall class diagram well-formedness assumptions

- Classes
  - A class must have a name
- Attributes
  - An attribute must have a name
  - An attribute must have a data type
  - An attribute may have a default value
  - A default value must match the attribute's data type
- Operation/Signals
  - An operation/signal must have a name
  - An operation/signal may have parameters (depends on formalization rules and target language constraints)
  - An operation/signal parameter must have a data type
  - An operation/signal may have a return data type (depends on formalization rules and target language constraints; McUmber's UML-to-Promela formalization uses only asynchronous signals)

Figure 5.7: Class diagram parts well-formedness assumptions

- Nam
  -
  -
- Hor
  -
  -
- Sta
  -
  -
  -
  -
- Tra
  -
- A
  -

- Naming
  - Name uniqueness among all states
  - No reserved words from target language used as names
- Hierarchy
  - State diagrams are *hierarchical* and as such may have nested *levels*
  - The topmost level of a state diagram must have exactly one start state
- States
  - A simple state, composite state, concurrent-composite state, or concurrent region must have a name
  - A simple state may have entry or exit actions
  - A concurrent-composite state must contain more than one concurrent region
  - A concurrent region must have exactly one start state
- Transitions
  - A transition may have associated actions
- Actions
  - An action may be an assignment statement or a message send

Figure 5.8: Overall state diagram well-formedness assumptions

a thick ex

transitions

authors m

within the s

Usage of wa

classificati

usage is fi

in a class w

during the

introduced

more dev

a class ad

check to

that you

difficult

model is

an error.

a check ensures, for example, that all the variables used in actions and messages on transitions or entry/exit actions in states within a state diagram have been defined as attributes in the owning class in the class diagram, and that events used on transitions within the state diagram have been defined as operations/signals in the owning class. Usage of variables or events in a state diagram without their having been defined in the class diagram will lead to syntax errors in the target language formal model; thus, such usage is flagged as an error. However, definition of attributes or operations/signals in a class without their use in the corresponding state diagram is a common practice during the evolution of a system's model. Attributes and operations/signals may be introduced in a class with the intent of being used at some point in the future when more detail is added to the model. Therefore, this type of inconsistency between a class and its corresponding state diagram is flagged as a warning only. Another check between state diagrams alerts the user when a state diagram expects a message that no object sends. This situation is an inconsistency among the state diagrams contained in the entire model that may cause a deadlock when the behavior of the model is explored with the target-language analysis tool. It is therefore considered an error.

Integration assumptions between the class and state diagrams are listed in Figure 5.9. Problems between the class diagram and a state diagram are listed in Figure 5.10, while problems between state diagrams are summarized in Figure 5.11. Both Figures 5.10 and 5.11 include a brief problem description, a classification of the problem as a warning or an error, and the potential consequence if the problem is not detected and addressed prior to specification generation.

- State diagram ownership
  - A class may have zero or one state diagrams associated with it; we say that the class *owns* the state diagram
  - A state diagram must be owned by a class
- Attributes
  - Variables used on the left hand side of assignment statements in actions within states or on transitions in a state diagram must be defined as attributes in the owning class
  - Variables used in event and message parameter lists, guarding conditions, or expressions on the right hand side of assignment statements must be defined as attributes in the owning class
- Operations/Signals
  - The events handled by a state diagram must be defined as operations/signals in the owning class
- Messages
  - The recipient of a message must be defined as a class in the class diagram
  - Messages sent must be defined as operations/signals in the owning class of the recipient state diagram
  - The recipient class of a message must have a state diagram associated with it
  - For each message sent to a recipient state diagram, there should be at least one transition in the recipient state diagram that handles the message.

Figure 5.9: Integration assumptions between class and state diagrams

Problem	Classification	Consequence
Use of a variable in a state diagram without it being defined as an attribute in the owning class in the class diagram	Error	Target language syntax error
Defining an attribute in a class in the class diagram but not using it as a variable in the corresponding state diagram	Warning	No adverse effect on target language specification, except in the case of some model checkers may contribute to state space explosion
Use of an event in a state diagram without it being defined as an operation/signal in the owning class in the class diagram	Error	Target language syntax error
Defining an operation/signal in a class in the class diagram but not using it as an event in the corresponding state diagram, or as a message in any state diagram	Warning	No adverse effect on target language specification

Figure 5.10: Problems between class and state diagrams

Problem
Use of a
in a n
diagram
that ev
Use of
in a n
diagram
this st
the ou
less s
Expe
never s
day s

Problem	Classification	Consequence
Use of an operation/signal in a message send to a state diagram that never handles that event	Error	Potential deadlock: If the message send takes place, then the recipient state diagram will be unable to handle the incoming event and will deadlock.
Use of an operation/signal in a message send to a state diagram when the operation/signal is not defined in the owning class of the recipient state diagram	Error	Target language syntax error
Expecting an event that is never sent as a message from any state diagram	Error	Potential deadlock: If an object enters a state where the only transitions out of that state depend upon the reception of messages which will clearly never be sent, then the object will deadlock in that state.

Figure 5.11: Problems among state diagrams

5.2 B

5.2.1 S

5.2 Behavioral Analyses

After addressing any problems detected with structural analyses and then successfully generating the target language formal model for the model of the software system, behavioral analyses can be performed with the appropriate target language analysis tools. In the case of Promela models, we use Spin [69], a Promela model analysis tool that offers simulation and model checking capabilities. Simulation is useful for validating behavior of specific paths of execution through a given system. The analysis is not complete in the sense that it is not possible to validate every possible scenario. Once simulation has been used to validate the "common" scenarios and critical "exception cases", then model checking can be used to help check that the entire model satisfies specific critical properties. These analyses, corresponding to Steps F, G, and H of the process depicted in Figure 5.1 (page 60) respectively, are described in the remainder of this chapter.

5.2.1 Simulation

Simulation (Step F) enables validation of behavioral requirements or scenarios, and debugging of a system model. Hydra can be used to automatically generate Promela models from UML diagrams in order to use Spin's simulation utilities to "execute" the UML diagrams. Simulation reveals whether a system model executes at all, and if it does, how closely it conforms to expected behavior. Because the effort (constructing temporal claims) and resources (memory and time) required to perform model checking are more demanding than that needed for simulation, we find that using an iterative process of simulation and visualization of results within UML diagrams, refinement of UML diagrams, and automatic regeneration of a formal model enables us to gain a better intuitive understanding of a system and to correct many flaws prior to using model checking.

5.2.2 N

Model-checki
Each simul
but more t
other hand
of execut
this state
invariants a
communica
may, for ex
a certain ra
always hap
model-chec
that demon
themselves
errors.

In our
detect conc
for counter
correctness
the Spin
State space
a counter ex

Because of
state explo
based appr

5.2.2 Model Checking

Model checking [69, 70] (Step G) can be used to check for violation of global properties. Each simulation run interleaves the execution steps of various concurrent components, but, nonetheless, follows only one particular execution path. Model checking, on the other hand, expands a restricted set of execution paths from all possible interleavings of execution steps into one large graph, or *state space*.[3] By exhaustively exploring this state space, model checking can automatically detect deadlocks, test system invariants against a model, and check temporal claims. Deadlock usually indicates a communication protocol error between objects in a system model. System invariants may, for example, check that the value of an instance variable does not fall outside a certain range. Temporal claims usually test properties such as "something good always happens," or "something bad never happens." If a claim is violated, then model checking produces a *counterexample*, which is a sequence of execution steps that demonstrates how the claim was violated. Counterexample traces, which can themselves be simulated (Step H), can be extremely useful in tracing the source of errors.

In our approach, we use requirements-based properties and model checking to detect counterexamples in order to validate UML models. That is, we are looking for counterexamples (indicating flaws in a UML model) rather that trying to verify correctness of a model. In this dissertation, the phrase "verified successfully" means that Spin did not produce a counterexample in an exhaustive search of the generated state space, while "failed" means that the property did not hold, and Spin did produce a counterexample.

[3]Because of what is known as the *state explosion problem*, model checkers in practice must use some technique, or combination of techniques, to reduce the state space. Spin in particular uses the partial order reduction to limit the number of interleaving sequences considered [69, 70].

Chapter 6

Visualizations

This chapter discusses visualization techniques that we have developed for results of both structural analyses of diagrams and behavioral analyses of generated formal models. For illustration purposes, we focus primarily on visualization of results from the analyses enabled by McUmber's UML-to-Promela formalization [31]; thus, the model development and analysis process described in Chapter 4 for use with the model development and analysis framework instantiated in Chapter 4 is reproduced in Figure 6.1 for convenience to the reader. The bold directed arcs 1–4 represent feedback of analysis results to diagrams, discussed in this chapter. We visualize results in terms of both original and newly generated, complementary, diagrams in order to guide the user in refining the original diagrams modeling a system.

The following sections overview visualizations of results of both structural and behavioral analyses, including generation of new UML diagrams from behavioral analysis results. We introduce a small example for illustration purposes in Section 6.1. Feedback lines 1–4 in Figure 6.1 represent feedback of analysis results to diagrams. Feedback lines 1 and 2, discussed in Section 6.2, represent feedback of results from structural analyses of diagrams, while Feedback lines 3 and 4, discussed in Section 6.3, represent feedback of results from behavioral analyses of a generated Promela model

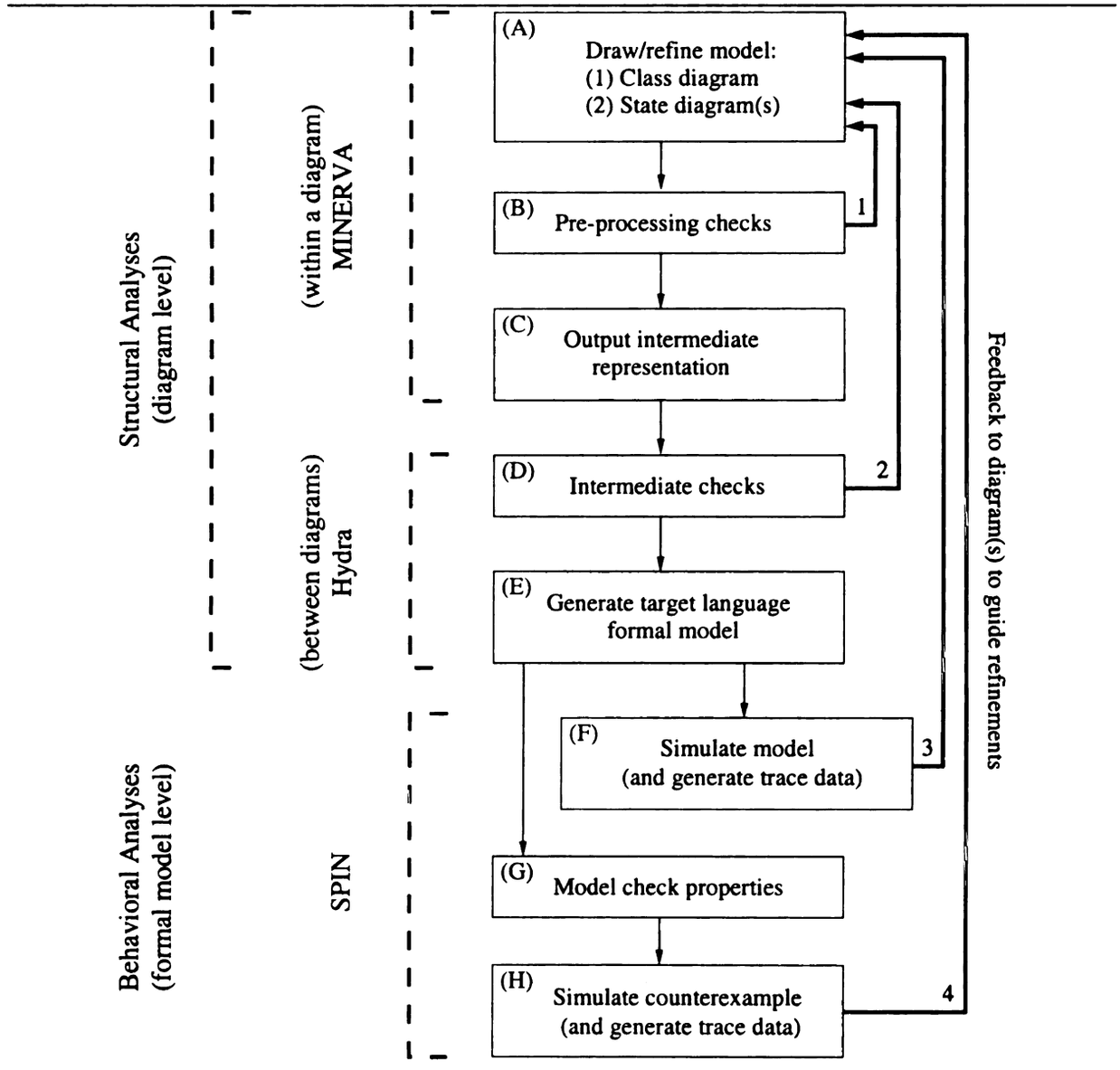


Figure 6.1: Iterative and incremental model development and analysis process, reprised from Figure 4.4. Bold arcs 1–4 represent feedback of analysis results to diagrams.

(specifically, from simulation or simulation of counterexamples with Spin). Because screen shots of the MINERVA GUI do not reproduce well, in this chapter we simulate what the user would see for each visualization example by providing either textual output or drawings of diagrams. "Highlighting in color" is simulated by drawing the affected portion(s) of the diagram(s) in bold.

6.1 *Producer-Consumer* Example

To illustrate structural and behavioral visualizations in the following sections, we introduce a small *Producer-Consumer* model. While not specific to the embedded systems domain, the *Producer-Consumer* problem has the advantages of being both well-known and small enough to illustrate several visualizations effectively, and it enables us to demonstrate a key issue in embedded systems design, namely, the importance of coordinated inter-object communication.

The class diagram for the model is shown in Figure 6.2. Each class has a state diagram associated with it that describes its behavior. Additionally, the system is represented by a special class, the _SYSTEMCLASS_, that is an aggregate of classes representing its main components, the Producer and Consumer classes. We model one Producer and one Consumer as indicated by the (default) multiplicity of one on the association between the _SYSTEMCLASS_ and each of the Producer and Consumer classes. This variation of *Producer-Consumer* follows a very simple supply-and-demand philosophy: the Producer waits for the Consumer to demand an item before supplying one. We assume that the Consumer does not make another demand until the Producer has handled the previous demand. Additionally, the Producer supplies only a limited number of items (five, as indicated by the default value[1] of the

[1]If a default value is not given, then Spin [69] automatically sets a variable's initial value to zero; however, not all target languages behave this way. For example, in the absence of a default value, SMV [75] non-deterministically chooses a variable's initial value from the variable's range of possible values.

attribute

in the arr

P
limited
num_
deman

The

64. res

SYSTE

strike

Adverse

has no

the Pro

ducer

The Con

Have_Mo

attribute `limited_ed`). The Producer keeps track of how many items it has supplied in the attribute **num_made**.

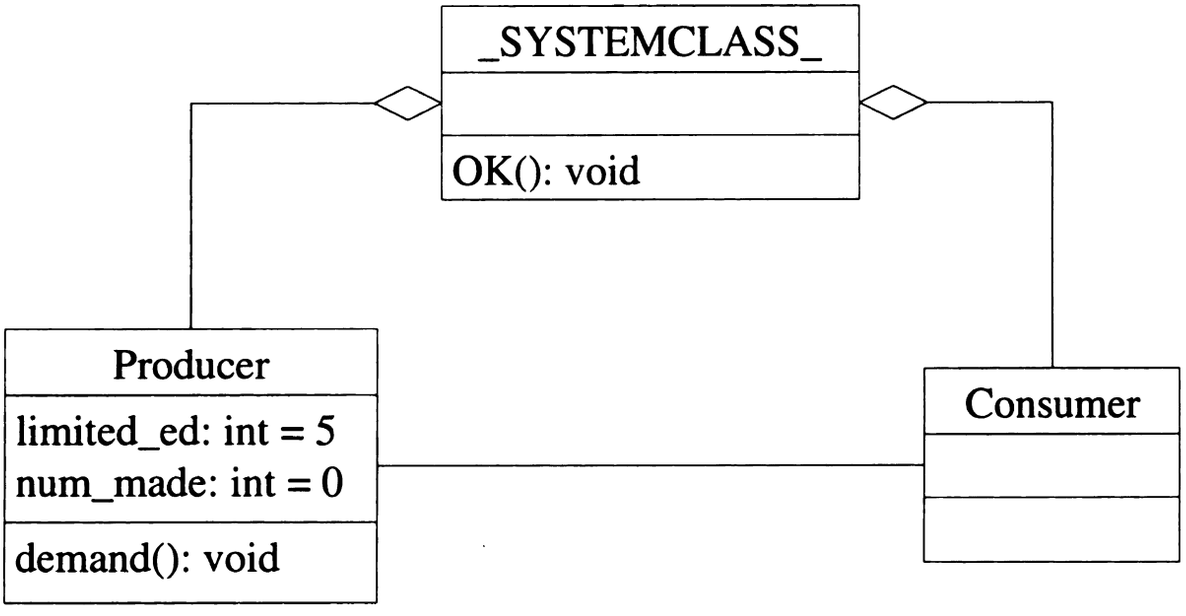


Figure 6.2: UML class diagram for *Producer-Consumer* model

The state diagrams for the Producer and Consumer are shown in Figures 6.3 and 6.4, respectively. Upon instantiation, the Producer sends an *OK* message to the _SYSTEMCLASS_ and enters its `Waiting_For_Demand` state. Until it has supplied the entire limited run of items, the Producer responds to *demands* from the Consumer, increments its counter that tracks the number of items it has supplied, and enters its `Advertise` state. The transition from state `Advertise` to state `Waiting_For_Demand` has no explicit event or guard. Instead, it has an implicit event *done* that causes the Producer to return immediately to state `Waiting_For_Demand`. Once the Producer has supplied the entire limited run of items, it enters the `Sorry_Sold_Out` state. The Consumer starts in its `Have_Money_Will_Spend` state. The transition from state `Have_Money_Will_Spend` to state `Waiting_For_Supply` has no explicit event or guard.

Instead, it
a-by-to-s
the trans-
immedia-

^_SYSTE

[]/

Instead, it has an implicit event *done* that causes the Consumer to transition immediately to state `Waiting_For_Supply` after sending a *request* to the Producer. Likewise the transition from state `Waiting_For_Supply` to state `Have_Money_Will_Spend` fires immediately.

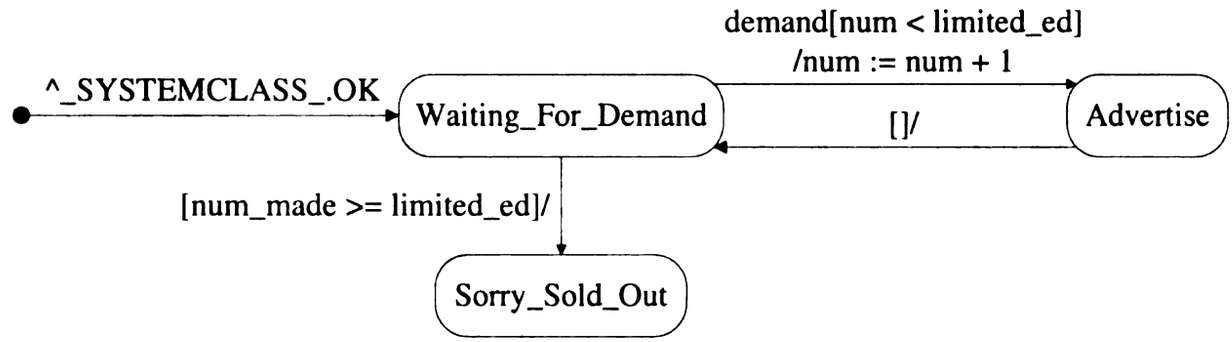


Figure 6.3: UML state diagram for Producer

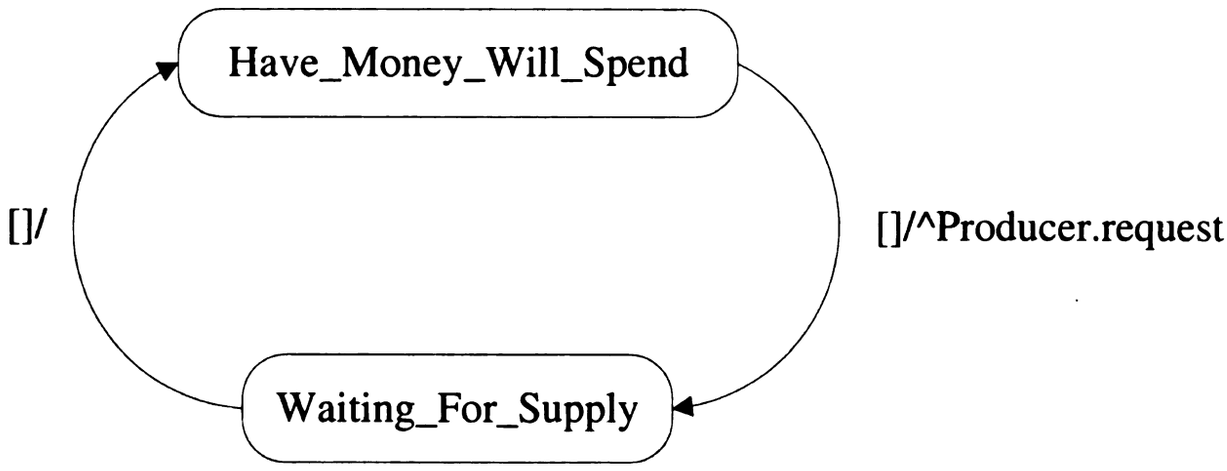


Figure 6.4: UML state diagram for Consumer

6.2 S

Structural

of problem

are visual

diagrams

6.2.1 V

Verification

and using

Graphical

are simple

Petri-net

and state

between

Petri-net.

6.2.2 S

First, the

generated

lines as

"****"

5.7.9, at

the State

that the

detected.

6.2 Structural Visualizations

Structural visualizations, as described in the next subsection, include visualization of problems within and between diagrams. Problems within an individual diagram are visualized within that diagram, while the different parts of a problem between diagrams are visualized within the affected diagrams.

6.2.1 Within and Between Diagrams

Visualizations of structural analysis results include displaying text-based messages and using color to highlight the location of problems in UML diagrams within the graphical editing environment (*e.g.*, MINERVA). Problems within individual diagrams are simply highlighted within each diagram, while problems between diagrams may be presented in a specific ordering. For example, definition/usage problems between class and state diagrams may show the class diagram first, while send/receive problems between state diagrams may show the sender first (if there is one) and then the receiver.

6.2.2 Structural Analyses Applied to the *Producer-Consumer* Example, with Visualizations

First, the diagrams are drawn in MINERVA. The intermediate representation is then generated and given to Hydra for structural analysis, which reveals several problems as shown in Figure 6.5. Each problem reported is prefixed with four asterisks ("****") and a classification of warning or error. For example, the errors on lines 3, 5, 7, 9, and 11 in Figure 6.5 indicate inconsistencies between the class diagram and the state diagram for the **Producer**. In each case, the problem stems from the fact that the variable **num** has been used in the **Producer** state diagram but has not been declared. The error corresponding to these messages is visualized by highlighting the

appropriate transition(s) in the Producer state diagram, and then highlighting the Producer class in the class diagram, as simulated in Figures 6.6 and 6.7, respectively, to indicate that a variable has been used, but has not been declared. The actual error (determined by the user after evaluating the aforementioned error messages together with the corresponding visualization) is that the variable num has been used instead of the declared variable num_made. To correct the error, we revise the appropriate transition(s) in the Producer state diagram to use the declared variable num_made. The revised Producer state diagram is shown in Figure 6.8.

```
1  ****    Starting...Hydra V1.4 (4/18/2002)
2  parse complete
3  **** ERROR Class: [Producer] State: [Waiting_For_Demand]
4       Variable num is undeclared
5  **** ERROR Class: [Producer] State: [Waiting_For_Demand]
6       Variable num is undeclared
7  **** Error: variable num undefined in transition
8       demand[num < limited_ed]/num := num + 1
9  **** ERROR Class: [Producer] State: [Waiting_For_Demand]
10      Variable num is undeclared
11 **** Error: variable num undefined in transition
12      demand[num < limited_ed]/num := num + 1
13 **** Warning Class: [Producer]
14      Instance variable 'limited_ed' is declared but unused
15 **** Warning Class: [Producer]
16      Instance variable 'num_made' is declared but unused
17 **** Warning Class: [Consumer]  No Initial state
18 **** ERROR Class: [Consumer] State: [Have_Money_Will_Spend]
19      Signal 'request', sent to class Producer, is not declared
20 **** Warning Class: [_SYSTEMCLASS_] State: [Done]
21      State can never be exited (no outbound transitions)
22 **** Warning Class: [Producer] State: [Sorry_Sold_Out]
23      State can never be exited (no outbound transitions)
```

Figure 6.5: Structural problems reported by Hydra

Line 18 in Figure 6.5 reports that the signal *request* is not declared for class Producer in the class diagram, although it is used in the Consumer state diagram to send a message to Producer. This error represents another inconsistency between

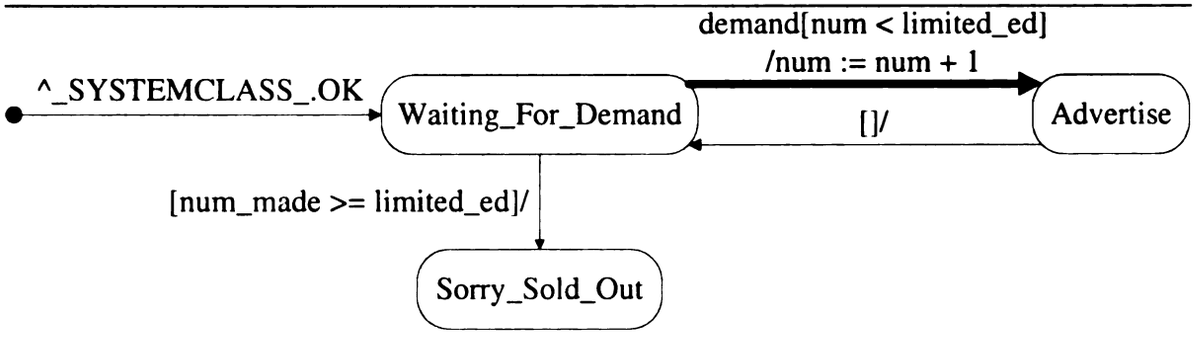


Figure 6.6: UML state diagram for Producer showing highlighted transition from state `Waiting_For_Demand` to state `Advertise` that uses the undeclared attribute `num`

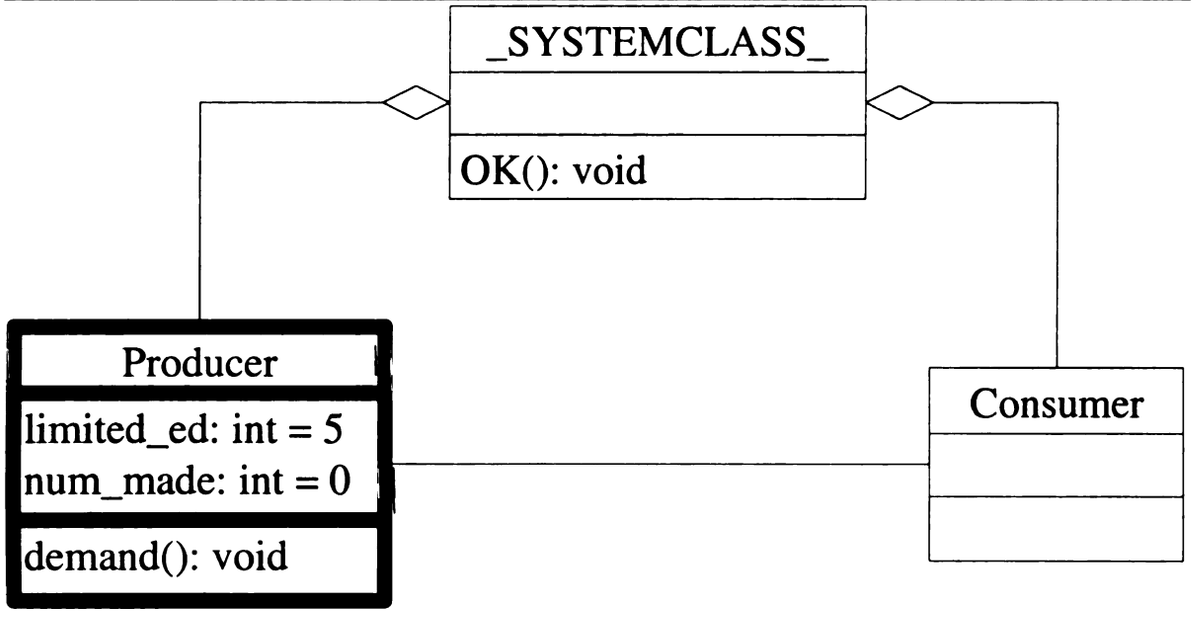


Figure 6.7: UML class diagram for *Producer-Consumer* model showing highlighted class Producer that has not declared the attribute `num` nor the signal *request*

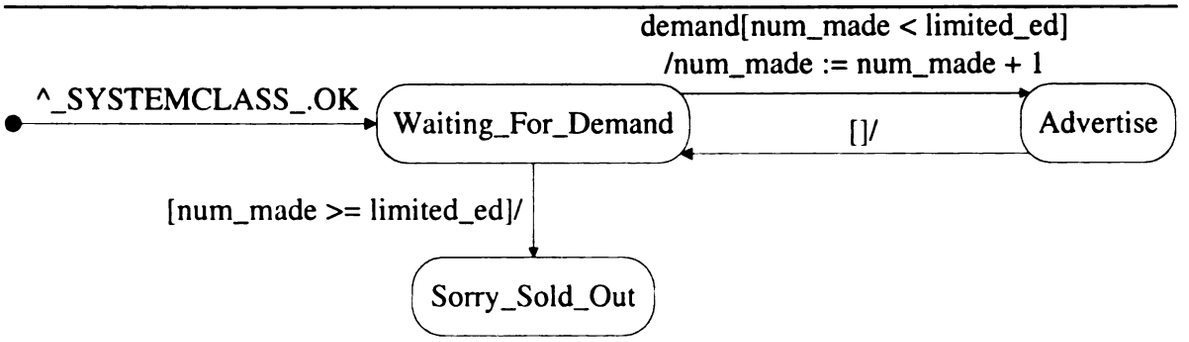


Figure 6.8: UML state diagram for Producer after fixing structural problems. (Used declared attribute `num_made` instead of undeclared `num` on transition from state `Waiting_For_Demand` to state `Advertise`.)

diagrams and is visualized by first displaying the highlighted transition in question in the Consumer state diagram as simulated in Figure 6.9 and then the highlighted Producer class in the class diagram as (previously) simulated in Figure 6.7. The actual error is that the signal *request* has been used instead of the declared signal *demand*. To correct the error, we revise the message on the transition in question in the Consumer state diagram to use the declared signal *demand*. (Alternately, we could have revised the declarations in the Producer class in the class diagram to declare the signal *request*.)

Line 17 in Figure 6.5 (page 82) describes a well-formedness warning specific to the Consumer state diagram. Because the problem is a *missing* start state, only the state diagram for the Consumer itself can be shown, along with a textual message indicating the problem. The developer must decide which is the initial state (`Have_Money_Will_Spend` or `Waiting_For_Supply`) and connect a *start* pseudostate to it in order for any of the Promela code representing the Consumer state diagram in the generated formal model to be reachable. To correct this error, we add a start state and an initial transition to state `Have_Money_Will_Spend`, indicating that state `Have_Money_Will_Spend` is the initial state of the Consumer state diagram. The re-

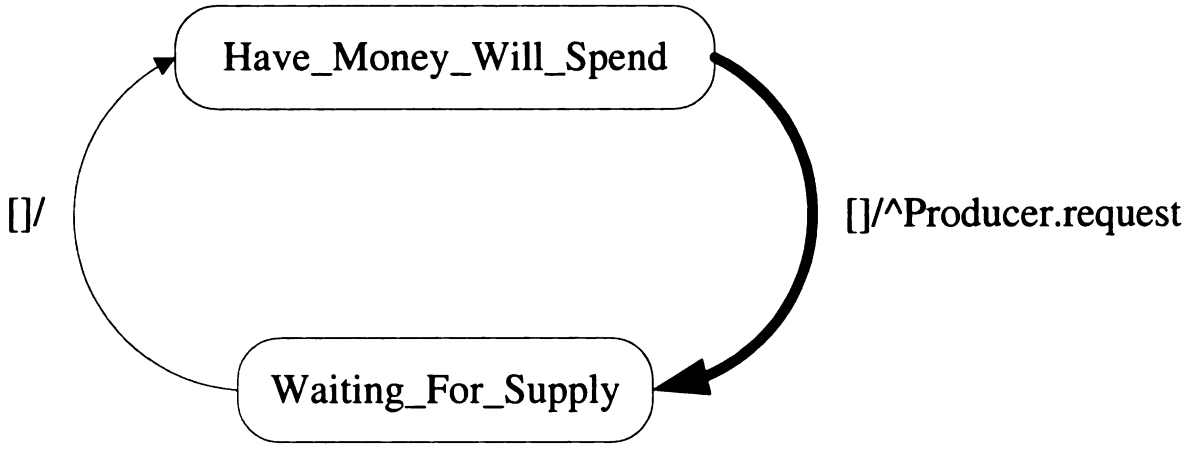


Figure 6.9: UML state diagram for Consumer showing highlighted transition from state `Have_Money_Will_Spend` to state `Waiting_For_Supply` that uses the undeclared signal *request*

vised Consumer state diagram, handling the errors on both Line 17 and Line 18 in Figure 6.5 (page 82), is shown in Figure 6.10.

For completeness, Figure 6.11 shows the class diagram after structural revisions (no changes were needed). We apply structural analyses to the intermediate representation generated from the revised diagrams and find no errors. Now that structural problems have been addressed, Promela models can be generated from the diagrams and behavioral analyses applied. The next section discusses the types of visualizations possible from Spin's behavioral analysis results and then demonstrates them in terms of the *Producer-Consumer* example.

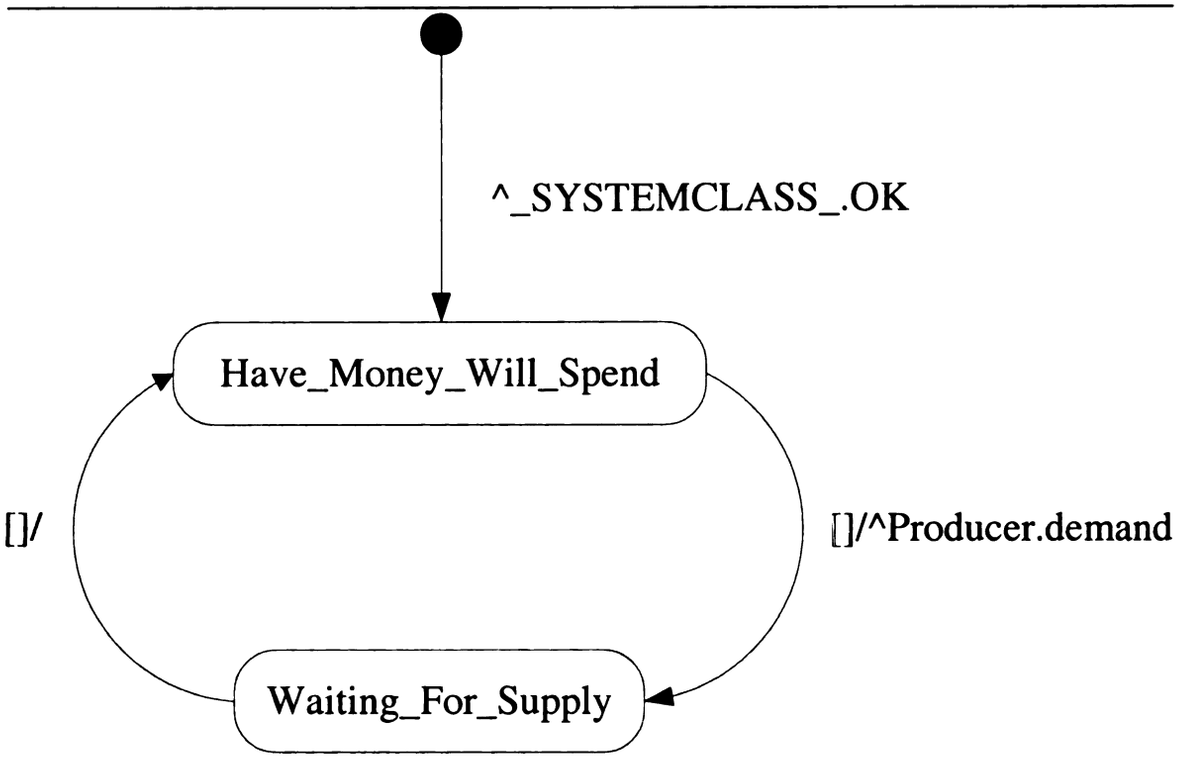


Figure 6.10: UML state diagram for Consumer after fixing structural problems. (Used declared signal *demand* instead of undeclared *request* on transition from state `Have_Money_Will_Spend` to state `Waiting_For_Supply`. Added initial transition to state `Have_Money_Will_Spend`.)

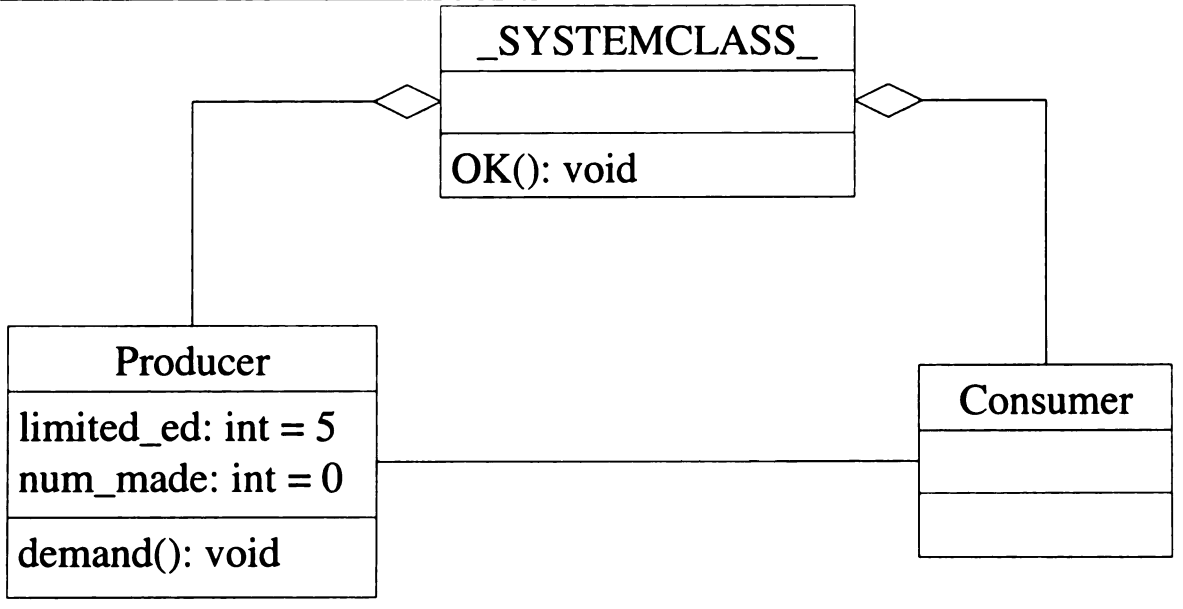


Figure 6.11: UML class diagram for *Producer-Consumer* model after fixing structural problems. (No changes were made.)

6.3 Behavioral Visualizations

Behavioral visualizations, as described in the following subsections, include state diagram animation, sequence diagram generation, and both collaboration diagram generation and animation. Refinements shown to the *Producer-Consumer* diagrams in this section will be described step-by-step in Section 6.3.4. To enable feedback to existing state diagrams, MINERVA offers the user three ways to augment the intermediate textual representation that will later be translated into Promela. These choices are: "States Only", "Transitions Only", and "States and Transitions" (the default generation of the intermediate textual representation uses none of these options). The choice selected affects the type of print statements[2] added to the intermediate representation, and thus the final formal model. These print statements, combined with several options for Spin runtime flags, affect the level of detail present in the trace output from Spin and thus the type of visualizations that can be achieved. The default generation does not add any extra statements to the intermediate representation. Currently the formalization to Promela does not incorporate identity tags for graphical elements, so with no extra statements, no automatic feedback to the original diagrams can be generated other than possibly highlighting states based on a unique naming scheme (depending on the Spin flags chosen, it may still be feasible to generate sequence and/or collaboration diagrams). MINERVA's user interface offers the choices "States Only" and "Transitions Only" as mechanisms to add special *print entry actions* to states or special *print actions* to transitions, respectively, while the choice "States and Transitions" does both. These special print actions, while not explicitly available to the user at the UML diagram level, are added transparently to the intermediate representation that is then given to Hydra. The generated Promela model will have appropriate print statements that will fire when states are entered or

[2]Note that this technique will only work for target languages with a print statement. Future work includes investigating a more general approach to instrumentation that adds boolean variables to the formal model to indicate when a state has been entered or when a transition has been taken.

transitions are taken during the exploration of the model.

Trace data captured from both random simulation and simulation of counterexamples obtained from model checking results enables animation of state diagram(s) originally drawn by the user (via the print statements just discussed), and generation of new sequence and collaboration diagrams. An example of raw trace data output from a Spin simulation with no runtime flags of a Promela model generated from an intermediate representation[3] augmented with print statements using the "States and Transitions" option is shown in Figure 6.12. The data has been manually elided and formatted for space considerations and readability as follows: (1) As line 1 indicates, data from the beginning of the trace has been elided so that the figure shows only the last five transitions and state entries (lines 2–31), plus the contents of all model queues at the end of the trace (lines 34–41) ; and (2) Normally, the information from a print statement pertaining to a transition firing (such as shown in lines 2–4) or a state being entered (such as shown in lines 6–7) would appear all on a single line, with data fields delimited by the character '@'. This data can be processed into either a human-oriented report, as shown in Figure 6.13, or visualization instructions for MINERVA as shown in Figure 6.14. The numbers delimited by '@' characters in the long strings in Figure 6.12 are the graphical element identity numbers from MINERVA's internal representation.

6.3.1 State Diagram Animation

A *state diagram*, originally drawn by the user, depicts the behavior of a class. Depending on the amount of detail present in the trace data (which, in turn, depends on the instrumentation option chosen at the time of generating the intermediate

[3]The intermediate representation used to generate the formal model that produced the trace data in Figures 6.12, 6.13, and 6.14 was created from a refined version of the diagrams in Figures 6.8 (page 84), 6.10 (page 86), and 6.11 (page 87). Figures 6.12, 6.13, and 6.14 are for illustration purposes only; however, the refinements to the diagrams presented in the next few sections will be described step-by-step in Section 6.3.4.

```
1  . . . [manually elided, formatted for readability]
2          TRANSITION@209130817257517@Producer@209130692280326
3            @Waiting_For_Demand@209130753228840
4            @Advertise@209130817257519@demand
5          in state Producer.Advertise
6          STATE@Producer@209130692280326
7            @Advertise@209130817257519
8          TRANSITION@20913430174105 8@Producer@209130692280326
9            @Advertise@209130817257519
10           @Waiting_For_Demand@2091307532288 40@
11         in state Producer.Waiting_For_Demand
12         STATE@Producer@209130692280326
13           @Waiting_For_Demand@209130753228840
14                 TRANSITION@209130748051490@Consumer@209130692280330
15                   @Waiting_For_Supply@209130747068447
16                   @Have_Money_Will_Spend@209130736058394@supply(edition_num)
17                 in state Consumer.Have_Money_Will_Spend
18                 STATE@Consumer@209130692280330
19                   @Have_Money_Will_Spend@209130736058394
20                 TRANSITION@209130748051488@Consumer@209130692280330
21                   @Have_Money_Will_Spend@209130736058394
22                   @Waiting_For_Supply@209130747068447@
23                 in state Consumer.Waiting_For_Supply
24                 STATE@Consumer@209130692280330
25                   @Waiting_For_Supply@209130747068447
26         TRANSITION@209130817257514@Producer@209130692280326
27           @Waiting_For_Demand@209130753228840
28           @Sorry_Sold_Out@2091308172575 16@
29         in state Producer.Sorry_Sold_Out
30         STATE@Producer@209130692280326
31           @Sorry_Sold_Out@209130817257516
32                 timeout
33 #processes: 3
34                 queue 2 (_SYSTEMCLASS__q):
35                 Producer_V.limited_ed = 5
36                 Producer_V.num_made = 5
37                 queue 3 (Producer_q): [demand]
38                 Consumer_V.edition_num = 5
39                 queue 4 (Consumer_q):
40                 queue 5 (Consumer_supply_p1):
41                 queue 1 (t): [free]
42 416:        proc  2 (Consumer) line 158 "producer_consumer_.pr" (state 37)
43 416:        proc  1 (Producer) line 110 "producer_consumer_.pr" (state 47)
44 416:        proc  0 (_SYSTEMCLASS_) line  64 "producer_consumer_.pr" (state 48)
45 3 processes created
```

Figure 6.12: Raw trace data output from Spin with no flags

```
1  Object "_SYSTEMCLASS_" transitions from state "Initial" to state
2         "Create_Producer" on event "modelstart"
3  Object "_SYSTEMCLASS_" enters state "Create_Producer"
4  Object "Producer" transitions from state "Initial" to state
5         "Waiting_For_Demand" on event "modelstart"
6  Object "_SYSTEMCLASS_" transitions from state "Create_Producer" to state
7         "Create_Consumer" on event "OK"
8  Object "Producer" enters state "Waiting_For_Demand"
9  Object "_SYSTEMCLASS_" enters state "Create_Consumer"
10 Object "Consumer" transitions from state "Initial" to state
11        "Have_Money_Will_Spend" on event "modelstart"
12 Object "_SYSTEMCLASS_" transitions from state "Create_Consumer" to state
13        "Done" on event "OK"
14 Object "_SYSTEMCLASS_" enters state "Done"
15 Object "Consumer" enters state "Have_Money_Will_Spend"
16 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
17        "Waiting_For_Supply" on event ""
18 Object "Consumer" enters state "Waiting_For_Supply"
19 Object "Producer" transitions from state "Waiting_For_Demand" to state
20        "Advertise" on event "demand"
21 Object "Producer" enters state "Advertise"
22 Object "Producer" transitions from state "Advertise" to state
23        "Waiting_For_Demand" on event ""
24 Object "Producer" enters state "Waiting_For_Demand"
25 Object "Consumer" transitions from state "Waiting_For_Supply" to state
26        "Have_Money_Will_Spend" on event "supply(edition_num)"
27 Object "Consumer" enters state "Have_Money_Will_Spend"
28 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
29        "Waiting_For_Supply" on event ""
30 Object "Consumer" enters state "Waiting_For_Supply"
31 Object "Producer" transitions from state "Waiting_For_Demand" to state
32        "Advertise" on event "demand"
33 Object "Producer" enters state "Advertise"
34 Object "Producer" transitions from state "Advertise" to state
35        "Waiting_For_Demand" on event ""
36 Object "Producer" enters state "Waiting_For_Demand"
37 Object "Consumer" transitions from state "Waiting_For_Supply" to state
38        "Have_Money_Will_Spend" on event "supply(edition_num)"
39 Object "Consumer" enters state "Have_Money_Will_Spend"
40 . . . [three more cycles, manually elided]
41 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
42        "Waiting_For_Supply" on event ""
43 Object "Consumer" enters state "Waiting_For_Supply"
44 Object "Producer" transitions from state "Waiting_For_Demand" to state
45        "Sorry_Sold_Out" on event ""
46 Object "Producer" enters state "Sorry_Sold_Out"
```

Figure 6.13: Trace data from Spin processed into human-oriented report

```
1  (list
2  . . . [12 previous instructions, manually elided]
3  (list "TRANSITION" "209130817257517" "Producer" "209130692280326"
4        "Waiting_For_Demand" "209130753228840"
5        "Advertise" "209130817257519" "demand")
6  (list "STATE" "Producer" "209130692280326"
7        "Advertise" "209130817257519")
8  (list "TRANSITION" "209134301741058" "Producer" "209130692280326"
9        "Advertise" "209130817257519"
10       "Waiting_For_Demand" "209130753228840" "")
11 (list "STATE" "Producer" "209130692280326"
12       "Waiting_For_Demand" "209130753228840")
13 . . . [38 more instructions, manually elided]
14 )
```

Figure 6.14: Trace data from Spin processed into visualization instructions for MINERVA corresponding to lines 19–24 of Figure 6.13

representation), the items highlighted in the state diagram may be states entered, transitions taken, or both. We find that highlighting the transitions taken is especially useful when there is more than one transition from the source state to the destination state (*e.g.*, several transitions using the same event but different guarding conditions). Figures 6.15–6.18 simulate the state diagram animation steps that correspond to lines 19–24 in Figure 6.13. Bold arcs in Figures 6.15 and 6.17 indicate the transition taken in each step, while bold rounded rectangles in Figures 6.16 and 6.18 indicate the state entered in each step.

6.3.2 Sequence Diagram Generation

Sequence diagrams complement state diagrams, portraying a single possible path through a collection of state diagrams. They are the isomorphic equivalent of collaboration diagrams, depicting a single sequence of message sends and receives (directed lines) over time (a vertical column per object, with time increasing from top to bottom). Message ordering and potential race conditions can be visualized with sequence

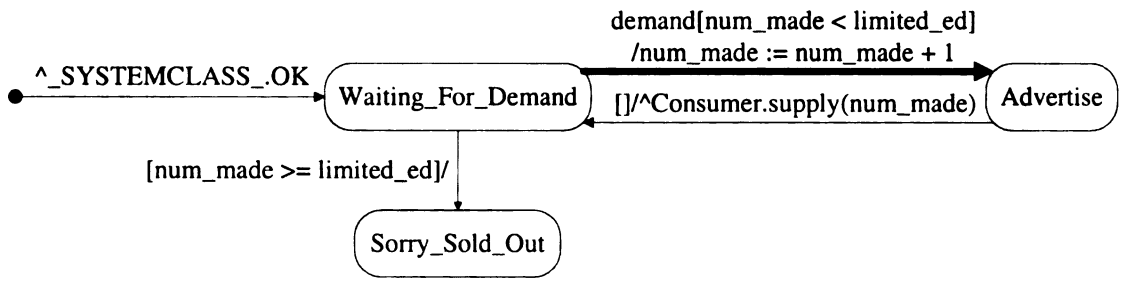


Figure 6.15: Highlighted transition corresponding to lines 19–20 of Figure 6.13. **Producer** transitions from state `Waiting_For_Demand` to state `Advertise` on event *demand*.

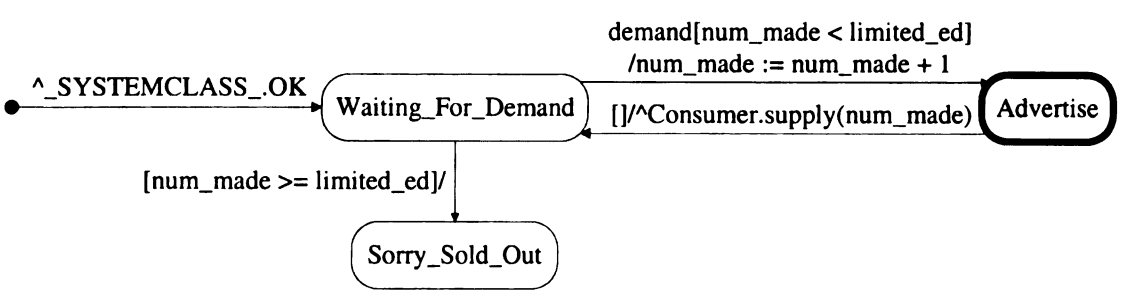


Figure 6.16: Highlighted state corresponding to line 21 of Figure 6.13. **Producer** enters state `Advertise`.

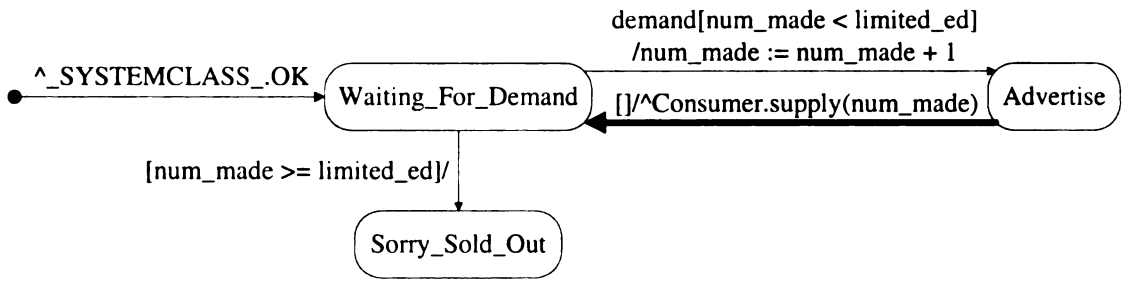


Figure 6.17: Highlighted transition corresponding to lines 22–23 of Figure 6.13 Producer transitions from state `Advertise` to state `Waiting_For_Demand` on the implicit event *done*.

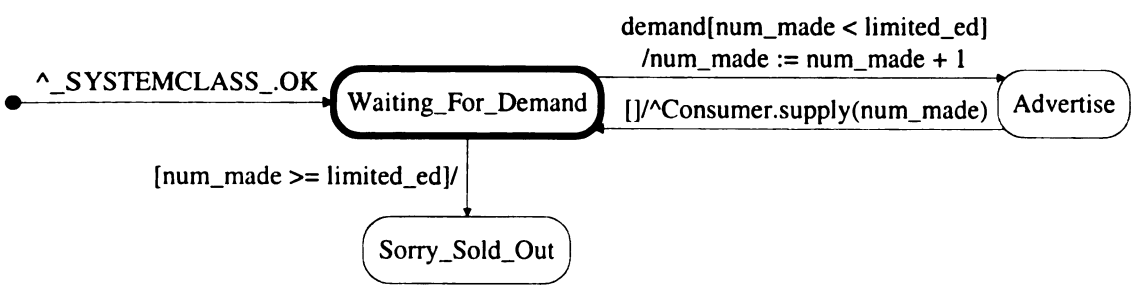


Figure 6.18: Highlighted state corresponding to line 24 of Figure 6.13 Producer enters state `Waiting_For_Demand`.

diagrams. Figure 6.19 shows the lifelines for three objects, _SYSTEMCLASS_, Producer, and Consumer, with messages depicted as arrows from the sender to the receiver labeled with the name of the message and the value of any parameters. This sequence of messages also corresponds to the series of events depicted in Figure 6.13 (page 91), although the trace data that generated this sequence diagram was produced by executing Spin with more verbose output options[4] than that used to generated the output in Figure 6.12 (page 90). For illustration purposes, a portion of the visualization commands generated by processing the verbose raw trace data is shown in Figure 6.20.

6.3.3 Collaboration Diagram Generation and Animation

A *collaboration diagram*[5] depicts a *snapshot* in time, a particular instance of communication between objects in the system (rectangles) via links (directed lines).[6] Animation of a collaboration diagram, then, depicts a series of these snapshots over time. When playing back trace data in a collaboration diagram, MINERVA highlights message pathways (links) as they are used. While state diagrams describe the order in which objects communicate via events, the actual communication pathway between the objects is not visualized. In collaboration diagrams, messages are preceded by sequence numbers denoting the order in which messages occur. Optional information displayed about a class instance at each step in an animation may include attribute values, the object's state, and for our visualization purposes, queue contents.[7] Figure 6.21 simulates a snapshot corresponding to the visualization instruction on line

[4]The command `spin -C -g -l -p -r -s -w specname.pr` includes information about channels used, global and local variables, message sends and receives, and forces print statements and a high level of verbosity in the output from Spin.

[5]Recall from Chapter 2 that collaboration and sequence diagrams are isomorphic.

[6]This notation convention is specific to our editor MINERVA. UML syntax suggests using labeled arrows to indicate the direction of message flow across a undirected link in a collaboration diagram, whereas we have attached the message label to a directed link to represent our asynchronous event flow.

[7]Recall (Chapter 2) that McUmber's formalization [31] uses queueing semantics.

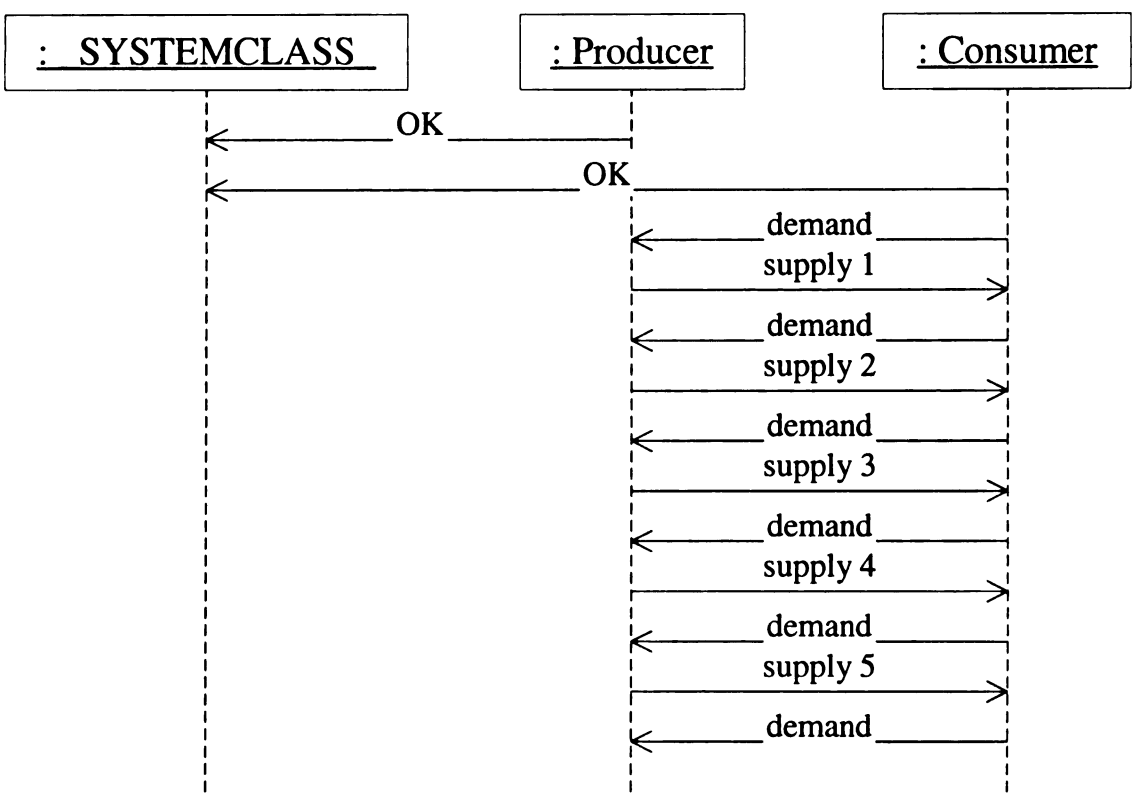


Figure 6.19: Example sequence diagram corresponding to events depicted in Figure 6.13 (page 91)

```
1  (list
2  (list "STARTOBJECTS")
3  (list "OBJECT" "_SYSTEMCLASS_" "0" "2")
4  (list "OBJECT" "Producer" "1" "3")
5  (list "OBJECT" "Consumer" "2" "4")
6  (list "ENDOBJECTS")
7  (list "STARTTHREADS")
8  (list "ENDTHREADS")
9  (list "STARTQUEUES")
10 (list "QUEUEDEF" "_SYSTEMCLASS_" "2" "0")
11 (list "QUEUEDEF" "Producer" "3" "1")
12 (list "QUEUEDEF" "Consumer" "4" "2")
13 (list "ENDQUEUES")
14 (list "STARTDATA")
15 (list "CHAN" "evq")
16 (list "CHAN" "evt")
17 (list "CHAN" "wait")
18 (list "CHAN" "_SYSTEMCLASS__q")
19 (list "CHAN" "Producer_q")
20 (list "CHAN" "Consumer_q")
21 (list "CHAN" "Consumer_supply_p1")
22 (list "CHAN" "t")
23 . . . [approximately 29 commands manually elided]
24 (list "STATE" "Consumer" "209130692280330"
25       "Have_Money_Will_Spend" "209130736058394")
26 (list "STATE" "Producer" "209130692280326"
27       "Sorry_Sold_Out" "209130817257516")
28 (list "SEND" "2" "Consumer" "demand" "3" "Producer_q")
29 (list "QUEUE" "3" "Producer_q" "1" "demand")
30 (list "STATE" "Consumer" "209130692280330"
31       "Waiting_For_Supply" "209130747068447")
32 (list "ENDDATA")
33 )
```

Figure 6.20: Visualization commands processed from raw trace data output from Spin with verbose flags

28 in Figure 6.20. As shown by the arrow on the link from Consumer to Producer, the Consumer is in the act of sending the message *demand* to the Producer, although the message has not yet reached the Producer's queue. The sequence number "13" indicates that this message is the thirteenth message in the sequence being animated.

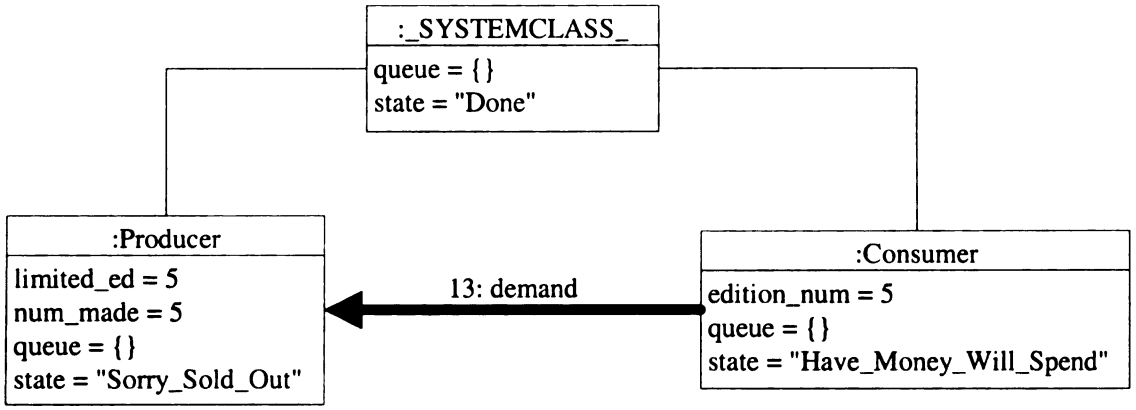


Figure 6.21: Example collaboration diagram corresponding to line 28 in Figure 6.20

6.3.4 Behavioral Analyses Applied to the *Producer-Consumer* Example, with Visualizations

We now return to our *Producer-Consumer* example, modeled by the diagrams from Figures 6.8, 6.10, and 6.11 after fixing structural problems but prior to final refinements. The diagrams are reproduced here in Figures 6.22, 6.23, and 6.24 respectively, for reference.

We generate the intermediate representation with the "States and Transitions" instrumentation option, translate it into a Promela model, and then run a simulation with Spin using verbose flags to determine whether the model behaves as we intended. The human-oriented report and sequence diagram generated from Spin's processed

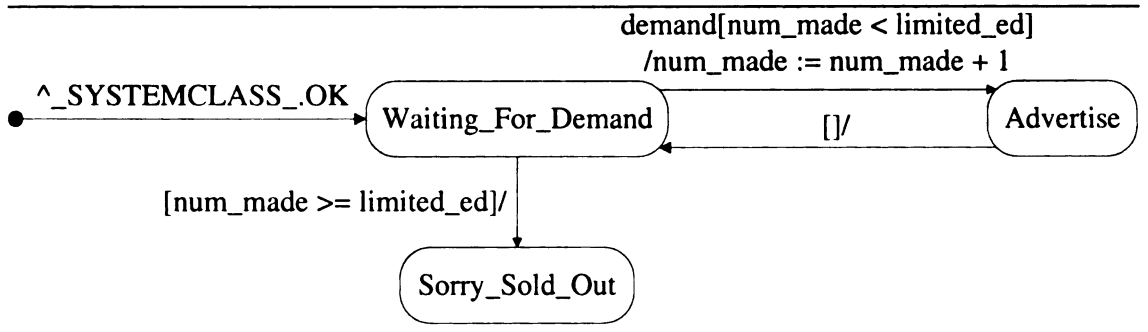


Figure 6.22: UML state diagram for Producer after fixing structural problems, repeated from Figure 6.8. (Used declared attribute `num_made` instead of undeclared `num` on transition from state `Waiting_For_Demand` to state `Advertise`.)

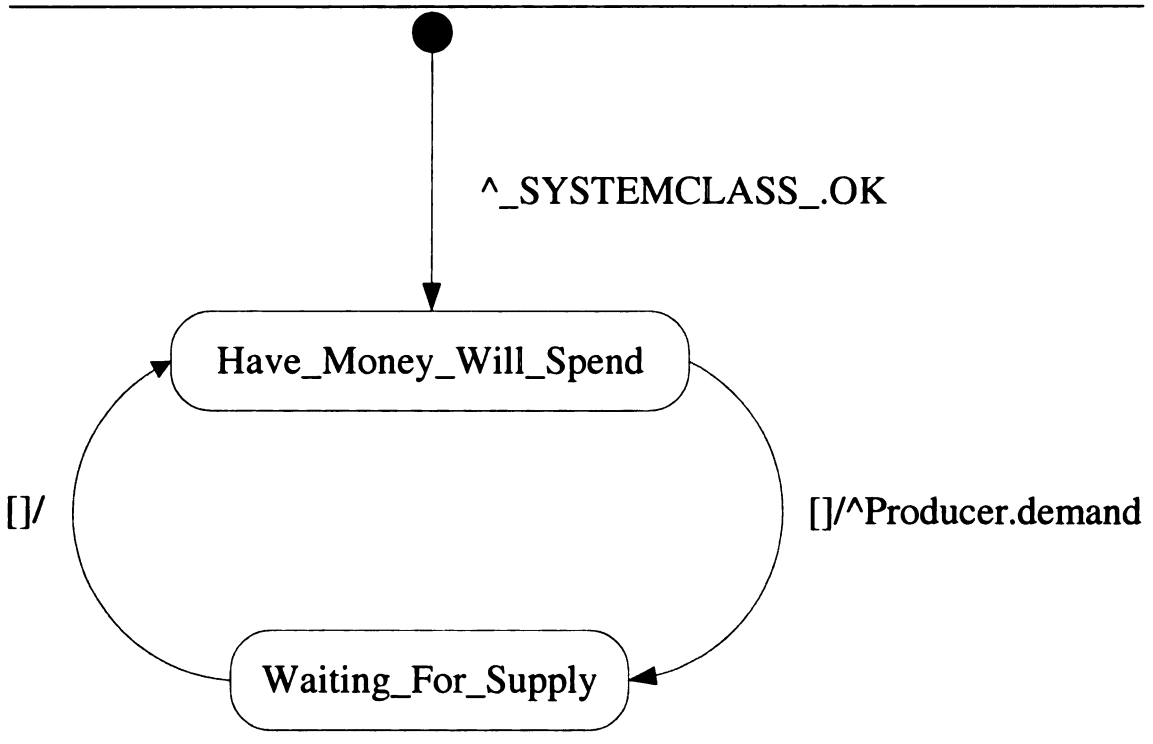


Figure 6.23: UML state diagram for Consumer after fixing structural problems, repeated from Figure 6.10. (Used declared signal *demand* instead of undeclared *request* on transition from state `Have_Money_Will_Spend` to state `Waiting_For_Supply`. Added initial transition to state `Have_Money_Will_Spend`.)

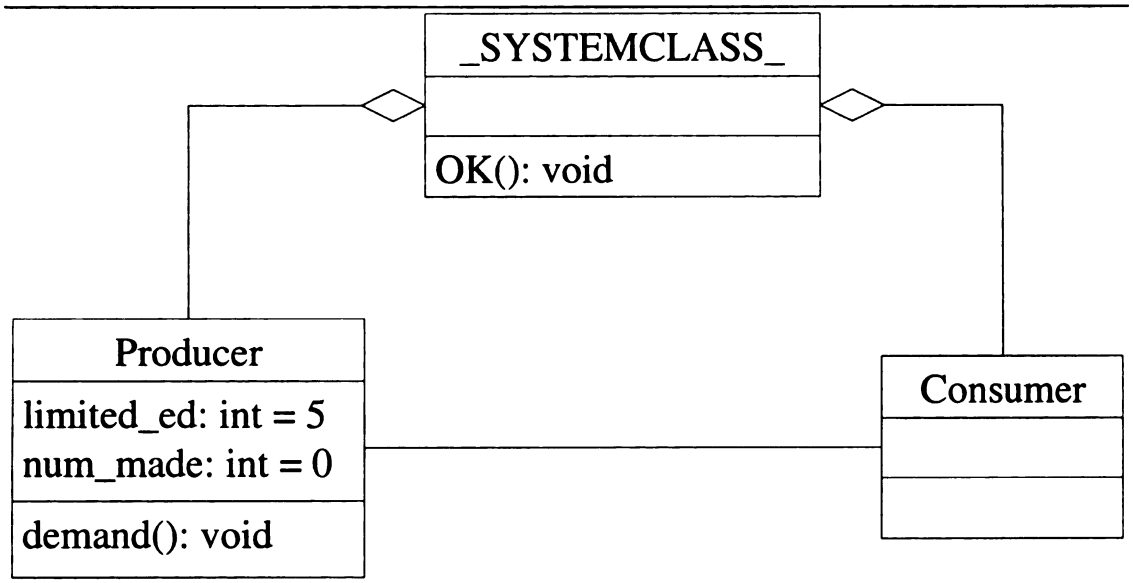


Figure 6.24: UML class diagram for *Producer-Consumer* example after fixing structural problems, repeated from Figure 6.11. (No changes were needed.)

output are shown in Figures 6.25 and 6.26, respectively. We can also generate visualization instructions for state diagram animation, not shown, from Spin's processed output, illustrating that the Consumer is able to send a second *demand* message before a first one has been handled by the Producer. The original intent was to have the Producer manufacture items on demand from the Consumer, but it is now apparent that in our naive model, based on the erroneous assumption that the Producer and the Consumer would take turns, the demands from the Consumer can out pace the Producer's ability to fulfill them. The simulation eventually terminates due to three factors: first, the Producer reaches its `Sorry_Sold_Out` state after making five items and thus is unable to fulfill any more demands; second, Promela queues must be of finite length;[8] and third, the default simulation behavior for Spin is to block when attempting to enqueue a message on an already-full queue rather than losing the message. By the end of the simulation, the Producer is in a state where it cannot respond

[8]The default queue length imposed by McUmber's Promela formalization [31] is five.

to any more demands, the Producer's queue is full of unmet *demand* messages, and the Consumer must block.

Spin's underlying execution model uses *interleaving*; that is, only one Promela process at a time may execute a step. Because Promela processes are not executed in *lockstep* (*i.e.*, not executed simultaneously), Spin has extremely weak *fairness* heuristics (*i.e.*, processes can starve), and our simple model has no special measures to ensure that the Producer and Consumer behave in a coordinated fashion, the Consumer is able to send multiple *demand* messages to the Producer without waiting for them to be handled. In order to resolve this problem, we introduce *handshaking*; that is, each *demand* message from the Consumer is acknowledged with a new *supply* message from the Producer. (Handshaking is a commonly used technique in the embedded systems domain to ensure coordinated inter-object communication [39].) The revised class diagram and Producer and Consumer state diagrams are shown in Figures 6.27 (page 104), 6.28 (page 104), and 6.29 (page 105), respectively. In the class diagram, class Consumer has been refined to include the new *supply* message, plus a new attribute called `edition_num`. The Producer state diagram has been refined to respond to a *demand* from the Consumer with a *supply* message that also carries the `num_made` value as a parameter to indicate which item in the limited edition series is being supplied. As shown in the Consumer state diagram, that value is stored in Consumer's `edition_num` variable upon receiving a *supply* message from the Producer.

After the diagrams are refined, we generate the intermediate representation again with the "States and Transitions" instrumentation option, translate it into a Promela model, and then run another simulation with Spin using verbose flags. Results are shown as a human-oriented report and a sequence diagram in Figures 6.30 (page 107) and 6.31 (page 108), respectively. These figures were also used earlier for illustration purposes in Figures 6.13 (page 91) and 6.19 (page 96). The refined version of the model seems to behave more in keeping with the original intent. The sequence dia-

```
1  . . . [elided 6 interleaved statements involving _SYSTEMCLASS_]
2  Object "Producer" transitions from state "Initial" to state
3         "Waiting_For_Demand" on event "modelstart"
4  Object "Producer" enters state "Waiting_For_Demand"
5  Object "Consumer" transitions from state "Initial" to state
6         "Have_Money_Will_Spend" on event "modelstart"
7  Object "Consumer" enters state "Have_Money_Will_Spend"
8  Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
9         "Waiting_For_Supply" on event ""
10 Object "Consumer" enters state "Waiting_For_Supply"
11 Object "Consumer" transitions from state "Waiting_For_Supply" to state
12        "Have_Money_Will_Spend" on event ""
13 Object "Consumer" enters state "Have_Money_Will_Spend"
14 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
15        "Waiting_For_Supply" on event ""
16 Object "Consumer" enters state "Waiting_For_Supply"
17 Object "Producer" transitions from state "Waiting_For_Demand" to state
18        "Advertise" on event "demand"
19 Object "Producer" enters state "Advertise"
20 Object "Consumer" transitions from state "Waiting_For_Supply" to state
21        "Have_Money_Will_Spend" on event ""
22 Object "Consumer" enters state "Have_Money_Will_Spend"
23 Object "Producer" transitions from state "Advertise" to state
24        "Waiting_For_Demand" on event ""
25 Object "Producer" enters state "Waiting_For_Demand"
26 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
27        "Waiting_For_Supply" on event ""
28 Object "Consumer" enters state "Waiting_For_Supply"
29 Object "Consumer" transitions from state "Waiting_For_Supply" to state
30        "Have_Money_Will_Spend" on event ""
31 Object "Consumer" enters state "Have_Money_Will_Spend"
32 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
33        "Waiting_For_Supply" on event ""
34 Object "Consumer" enters state "Waiting_For_Supply"
35 Object "Producer" transitions from state "Waiting_For_Demand" to state
36        "Advertise" on event "demand"
37 Object "Producer" enters state "Advertise"
38 . . . [elided 23 more steps]
```

Figure 6.25: Human-oriented report for *Producer-Consumer* example generated from the output of a Spin simulation with verbose flags

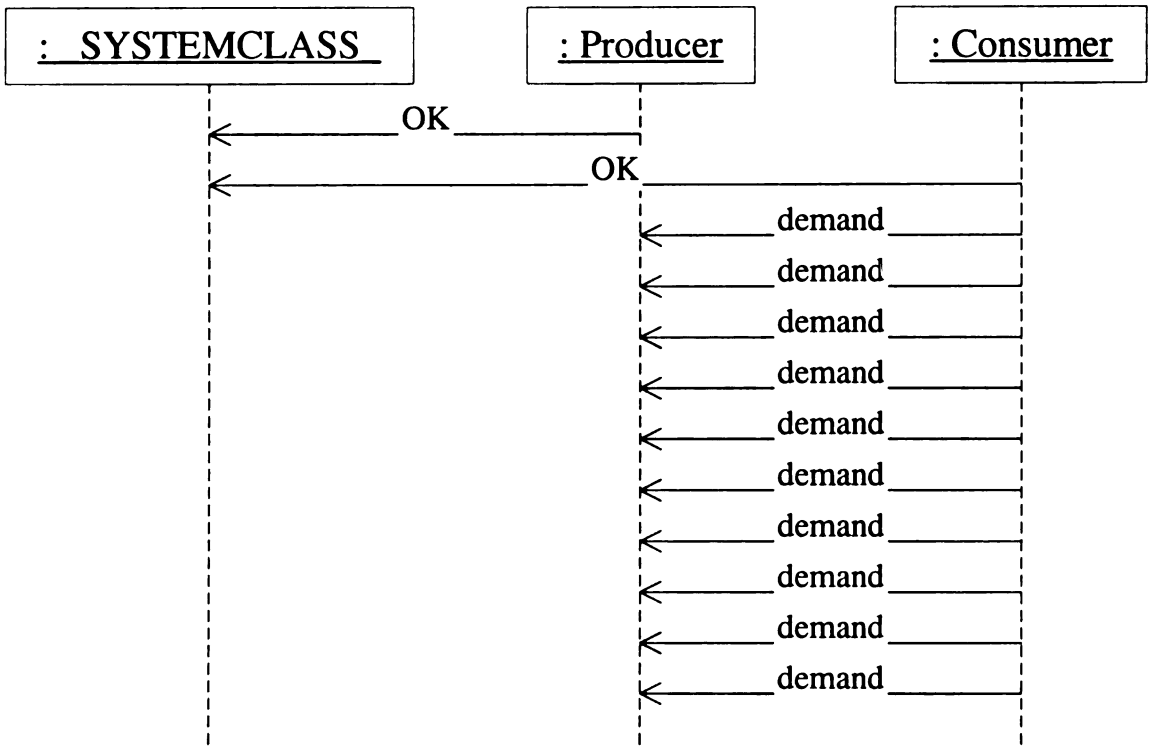


Figure 6.26: Sequence diagram for *Producer-Consumer* example generated from the output of a Spin simulation with verbose flags

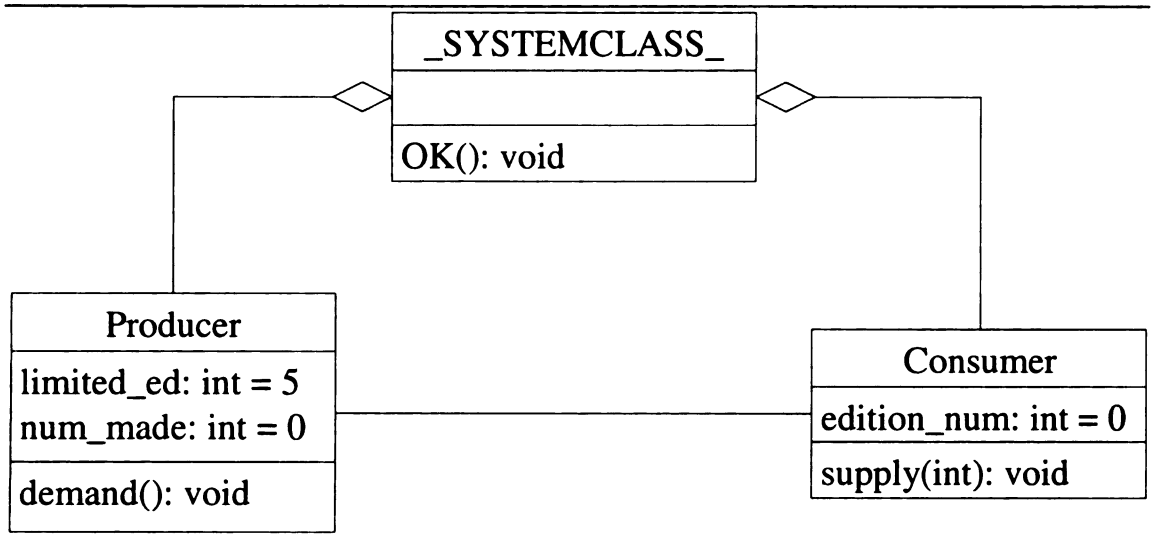


Figure 6.27: UML class diagram for *Producer-Consumer* example after fixing behavioral problems. (Added attribute `edition_num` and signal *supply* to class Consumer.)

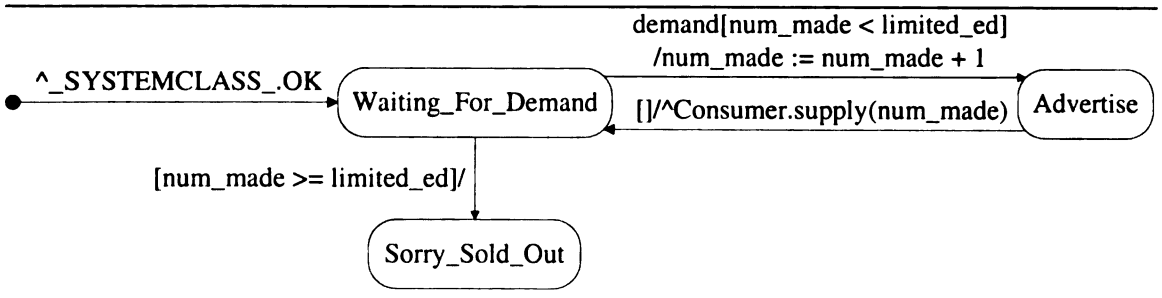


Figure 6.28: UML state diagram for Producer after fixing behavioral problems. (Added message *supply* sent to Consumer on transition from state `Advertise` to state `Waiting_For_Demand`.)

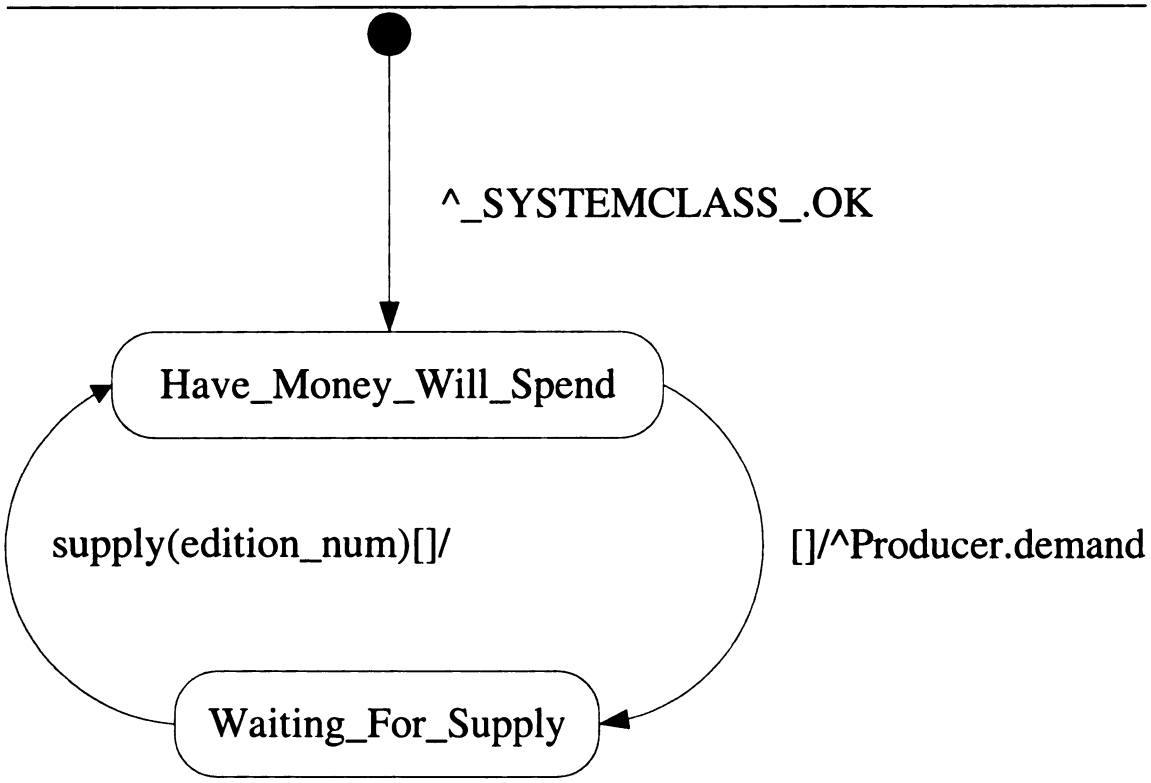


Figure 6.29: UML state diagram for Consumer after fixing behavioral problems. (Added event *supply* on transition from state `Waiting_For_Supply` to state `Have_Money_Will_Spend`.)

gram in Figure 6.31 shows that each *demand* message from the Consumer, save the last, is met with a *supply* message from the Producer. Lines 41–46 of the human-oriented report in Figure 6.30 show that the simulation ends with the Consumer in its `Waiting_For_Supply` state and the Producer in its `Sorry_Sold_Out` state. The simulation trace data can also be processed to create visualization instructions for state diagram animation, not shown, that would end with steps highlighting the Consumer's `Waiting_For_Supply` state and the Producer's `Sorry_Sold_Out` state. From the state diagrams in Figures 6.28 and 6.29, we know that the Producer will never respond to the final demand shown in the sequence diagram of Figure 6.31. The Consumer deadlocks in its `Waiting_For_Supply` state.

One approach to address the issue of the unmet demand is to alter the model as follows. Before accepting any demands from the Consumer, the Producer informs the Consumer of the number of items in the current collection as defined by `limited_ed`. The Consumer would thus have to add a new message with a parameter and an attribute in which to store the parameter's value. The Consumer then compares the latest edition it has received (`edition_num`) with the total number of items in the collection to determine when it has acquired the entire collection and can thus stop sending *demand* messages. To explore the ramifications of this possible solution, we would refine the diagrams, generate a Promela model from the intermediate representation once it has passed structural checks, and perform simulation or model checking with Spin. However, this example has served its purpose of illustrating types of structural and behavioral visualizations, and so we do not refine it further.

```
1  Object "_SYSTEMCLASS_" transitions from state "Initial" to state
2         "Create_Producer" on event "modelstart"
3  Object "_SYSTEMCLASS_" enters state "Create_Producer"
4  Object "Producer" transitions from state "Initial" to state
5         "Waiting_For_Demand" on event "modelstart"
6  Object "_SYSTEMCLASS_" transitions from state "Create_Producer" to state
7         "Create_Consumer" on event "OK"
8  Object "Producer" enters state "Waiting_For_Demand"
9  Object "_SYSTEMCLASS_" enters state "Create_Consumer"
10 Object "Consumer" transitions from state "Initial" to state
11        "Have_Money_Will_Spend" on event "modelstart"
12 Object "_SYSTEMCLASS_" transitions from state "Create_Consumer" to state
13        "Done" on event "OK"
14 Object "_SYSTEMCLASS_" enters state "Done"
15 Object "Consumer" enters state "Have_Money_Will_Spend"
16 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
17        "Waiting_For_Supply" on event ""
18 Object "Consumer" enters state "Waiting_For_Supply"
19 Object "Producer" transitions from state "Waiting_For_Demand" to state
20        "Advertise" on event "demand"
21 Object "Producer" enters state "Advertise"
22 Object "Producer" transitions from state "Advertise" to state
23        "Waiting_For_Demand" on event ""
24 Object "Producer" enters state "Waiting_For_Demand"
25 Object "Consumer" transitions from state "Waiting_For_Supply" to state
26        "Have_Money_Will_Spend" on event "supply(edition_num)"
27 Object "Consumer" enters state "Have_Money_Will_Spend"
28 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
29        "Waiting_For_Supply" on event ""
30 Object "Consumer" enters state "Waiting_For_Supply"
31 Object "Producer" transitions from state "Waiting_For_Demand" to state
32        "Advertise" on event "demand"
33 Object "Producer" enters state "Advertise"
34 Object "Producer" transitions from state "Advertise" to state
35        "Waiting_For_Demand" on event ""
36 Object "Producer" enters state "Waiting_For_Demand"
37 Object "Consumer" transitions from state "Waiting_For_Supply" to state
38        "Have_Money_Will_Spend" on event "supply(edition_num)"
39 Object "Consumer" enters state "Have_Money_Will_Spend"
40 . . . [three more cycles, manually elided]
41 Object "Consumer" transitions from state "Have_Money_Will_Spend" to state
42        "Waiting_For_Supply" on event ""
43 Object "Consumer" enters state "Waiting_For_Supply"
44 Object "Producer" transitions from state "Waiting_For_Demand" to state
45        "Sorry_Sold_Out" on event ""
46 Object "Producer" enters state "Sorry_Sold_Out"
```

Figure 6.30: Trace data from Spin processed into human-oriented report

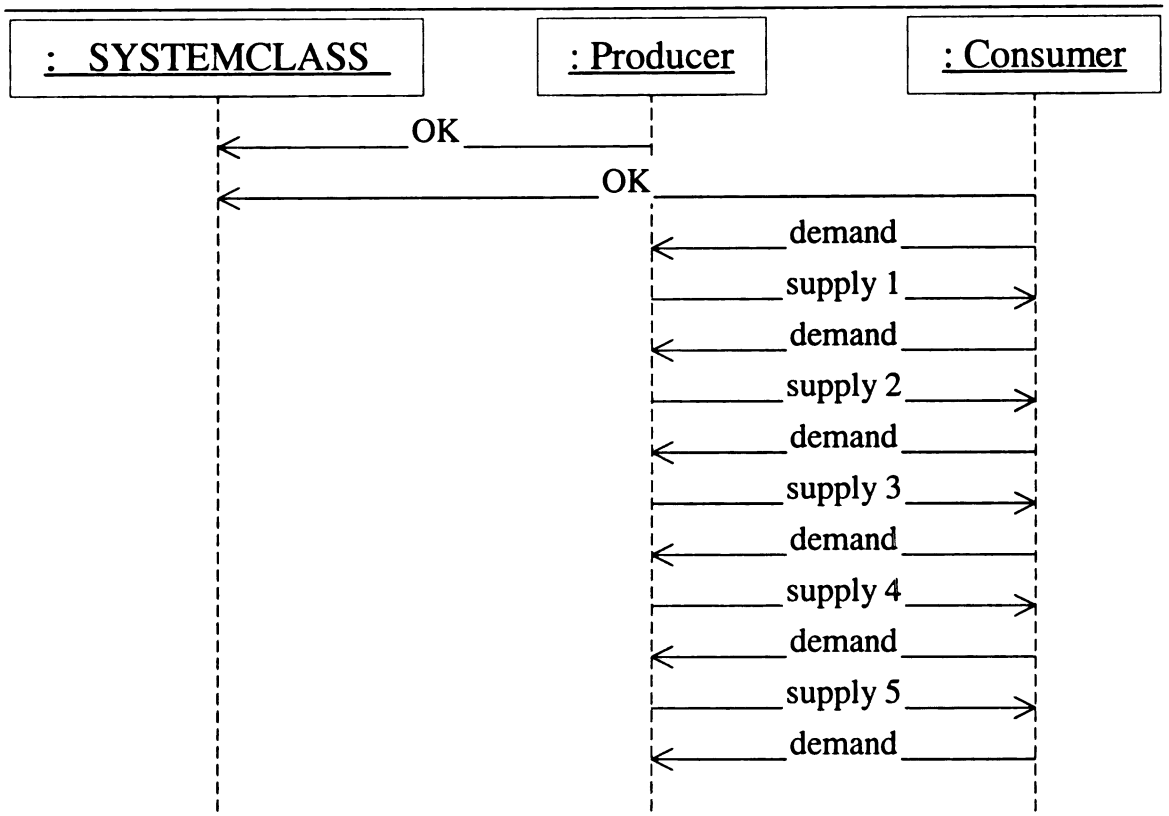


Figure 6.31: Sequence diagram corresponding to events depicted in Figure 6.30

Chapter 7

Validation: Industrial Case Study

This chapter overviews the Adaptive Cruise Control project [77], which we obtained from Siemens Automotive, and presents the results of a case study [40] we performed in order to validate our approach. Recently, this type of system was presented as a central part of an automotive collision avoidance system [78]. This chapter discusses how the system has been modeled with UML, analyzed via the (generated) formal model, and analysis results visualized in terms of both the original UML diagrams and automatically generated sequence diagrams. Results include detection of inconsistencies between diagrams using static checking, detection of an error on a guard using simulation and visualization, and deadlock detection in a critical brake scenario using model checking and visualization of the counterexample. These discoveries led to several refinements of the UML model.

7.1 Adaptive Cruise Control Project Overview

The Adaptive Cruise Control uses radar and an engine control module to control a car's speed when a vehicle, called the *lead vehicle*, is encountered in front of the car. When it detects a lead vehicle, Adaptive Cruise Control re-commands the engine control module to match the lead vehicle's speed. The *trail distance* behind the lead

vehicle is specified as the distance the lead vehicle travels in a given amount of time (usually two seconds). If the Adaptive Cruise Control-equipped car closes to within 90% of the specified trail distance (called the *safety zone*[1]) behind the lead vehicle, an audio warning must be sounded and the cruise disengaged. Otherwise, the system must match the speed of the lead 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 lead vehicle moves out of the radar's range (by speeding up or turning) or until the car's driver applies the brakes.

Figure 7.1 depicts the most common situations. Line 1 defines the various zones the control algorithm uses; Appendix D, page 248, provides more details. Line 2 depicts the radar acquiring the lead vehicle at about 400 feet, the range of the radar, at which time a periodic calculation is initiated to determine how long to maintain the current cruise speed. During this time, the car is in the *closing zone.* As soon as the relative speed and distance calculation determine coasting should be initiated, the system commands the engine control module to match the speed of the lead vehicle. The lead vehicle's speed is easily determined from two or more distance samples and the car's current speed. Because the car has inertia, the command to set the engine to a slower speed effectively results in the car decelerating until it assumes the correct trail position as shown on lines 3 and 4. Finally, line 4 shows the car having achieved the proper trail position of approximately two seconds behind the lead vehicle. If the radar loses the lead vehicle (*e.g.*, vehicle turned or the car driver changed lanes), then the initial cruise speed is resumed.

If the system determines that the closing speed is too fast, such that a collision is unavoidable without driver action, then it must produce visual and audio warnings, but not disengage the system. This behavior allows the driver to change lanes without

[1]The safety zone is approximately 1.8 seconds of lead vehicle travel, or approximately 185 feet at 70 miles per hour. While this distance between vehicles may appear small, human drivers often use much smaller trail distances.

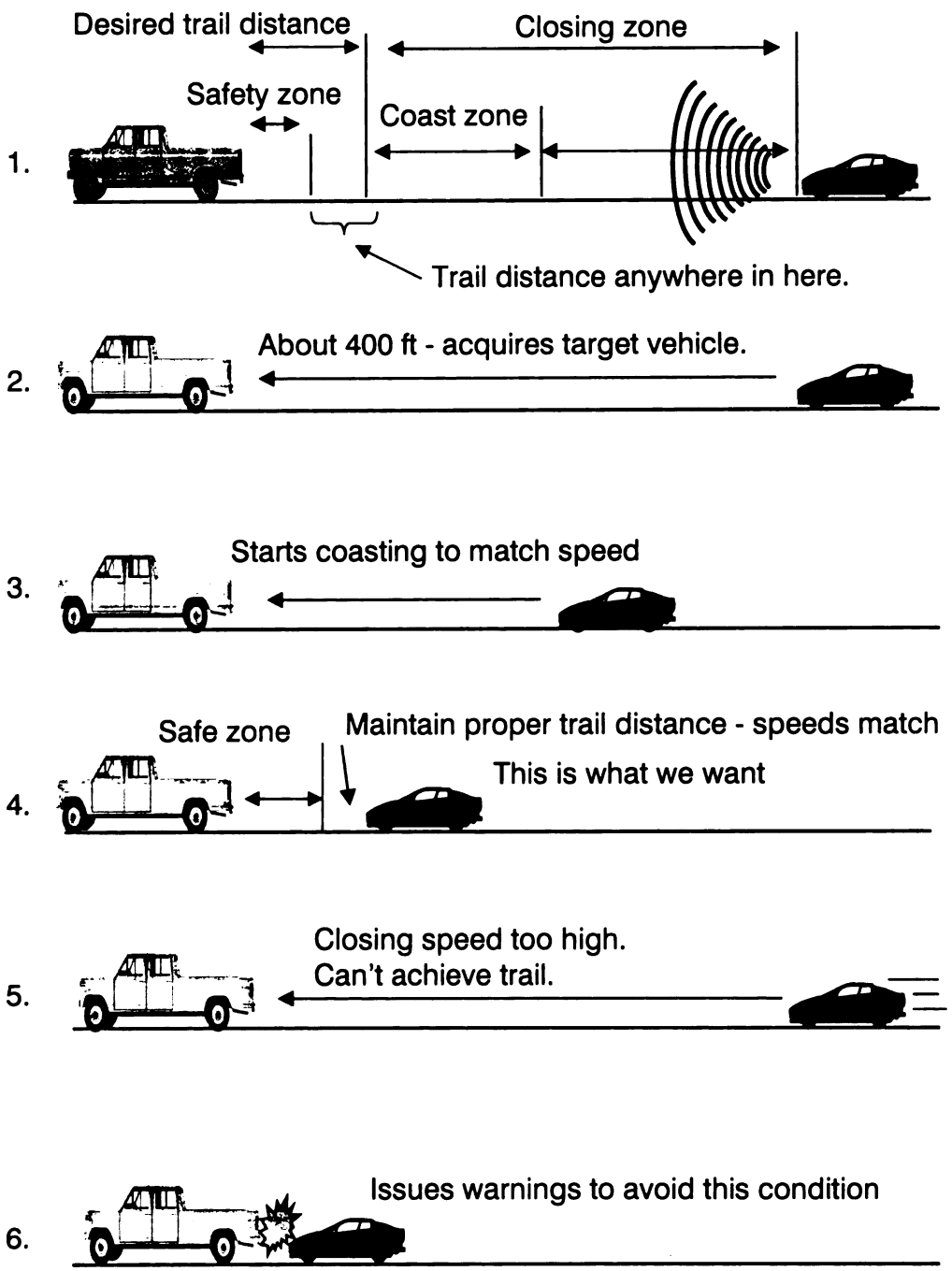


Figure 7.1: Common situations the Adaptive Cruise Control system must handle.

affect
The s
appli
ings t
page 2

7.2

After
that s
havio
opera
UML d
equip
distri
The sy
safety
celera
or turn
the bra
ners. S
check

The
ure 7.2
appear
a syste
repres

affecting the current cruise speed. Such a situation is shown on line 5 (Figure 7.1). The system will disengage only when the safety zone is entered, or when the driver applies the brakes. Otherwise, all unsafe conditions result in visual and audio warnings to the driver, line 6. (The preceding definitions are summarized in Appendix D, page 248.)

7.2 UML Modeling for Case Study

After discussing prose requirements with project engineers, we created an initial model that supports the main functionality of the system in the absence of exceptional behavior. The motivation for this approach is to establish and rigorously analyze normal operation of the system, and to add exception cases as later refinements. The initial UML diagrams created to model the system call for the Adaptive Cruise Control-equipped car to detect a slower-moving lead vehicle, calculate the appropriate trail distance, close on the lead vehicle, and decelerate to match the lead vehicle's speed. The system also warns if collision is imminent and disengages the system when the safety zone is violated. Exceptional behavior, such as the lead vehicle accelerating, decelerating, turning, or changing lanes, the driver changing lanes, applying the brakes, or turning off the cruise control, or the radar losing the target, is not examined in the initial version of the model. Furthermore, based on discussions with project engineers, several abstractions were made in order to make the model tractable for model checking. These abstractions are noted in the following paragraphs.

The initial class diagram for the Adaptive Cruise Control system is shown in Figure 7.2 (portions highlighted in bold will be discussed in Section 7.3). In our modeling approach, each class has a state subdiagram that describes its behavior. Additionally, a system is represented by a _SYSTEMCLASS_ class that is an aggregate of classes representing its main components, and a special class, Environment, that represents

the context or the environment for the system and can provide external stimuli to the system.

As shown in Figure 7.2, the three main classes that comprise the Adaptive Cruise Control system are Control, Car, and Radar. (Appendix D, page 248, contains a detailed explanation of these classes' attributes and signals.) Sensors, actuators, and the target vehicle are not represented explicitly but have been abstracted. The Car reports its current speed on request, and the Radar (for modeling purposes) calculates the distance to the target based on the Car's speed and the target vehicle's speed (represented in the model as attribute `vt` of Radar) rather than obtaining actual radar samples. Control accepts input from the driver (represented by the Environment) either to set the desired cruising speed by activating the Radar and controlling the throttle via adjustments to the Car's speed, or to disengage the system in the event that the driver applies the brakes. Once the Radar has been activated by the Control, it continually simulates scanning for a target and informs the Control of the distance to the target. For modeling purposes, the Radar sampling rate is assumed to be once per second.[2] The Adaptive Cruise Control-equipped Car continually adjusts its speed to match the desired speed as calculated by Control.

The initial state diagrams for the Car and Radar classes are shown in Figures 7.3 and 7.4 respectively (portions highlighted in bold will be discussed in Sections 7.3 and 7.4.1). Recall (Chapter 2) that the UML *dynamic model* or *state diagram* is based on Statechart [62] conventions and describes the dynamic behavior of objects. Transitions are labeled with an (optional) event followed optionally by a guard, an action list, and a message list.

Figure 7.3 describes the behavior of the Car class, which abstracts the speed management functionality of an Engine Control Module. There are four concurrent

[2]The radar sampling rate was not specified in the Adaptive Cruise Control requirements [77]. Experimentation with more frequent sampling rates exceeded available memory for model checking without clear benefit.

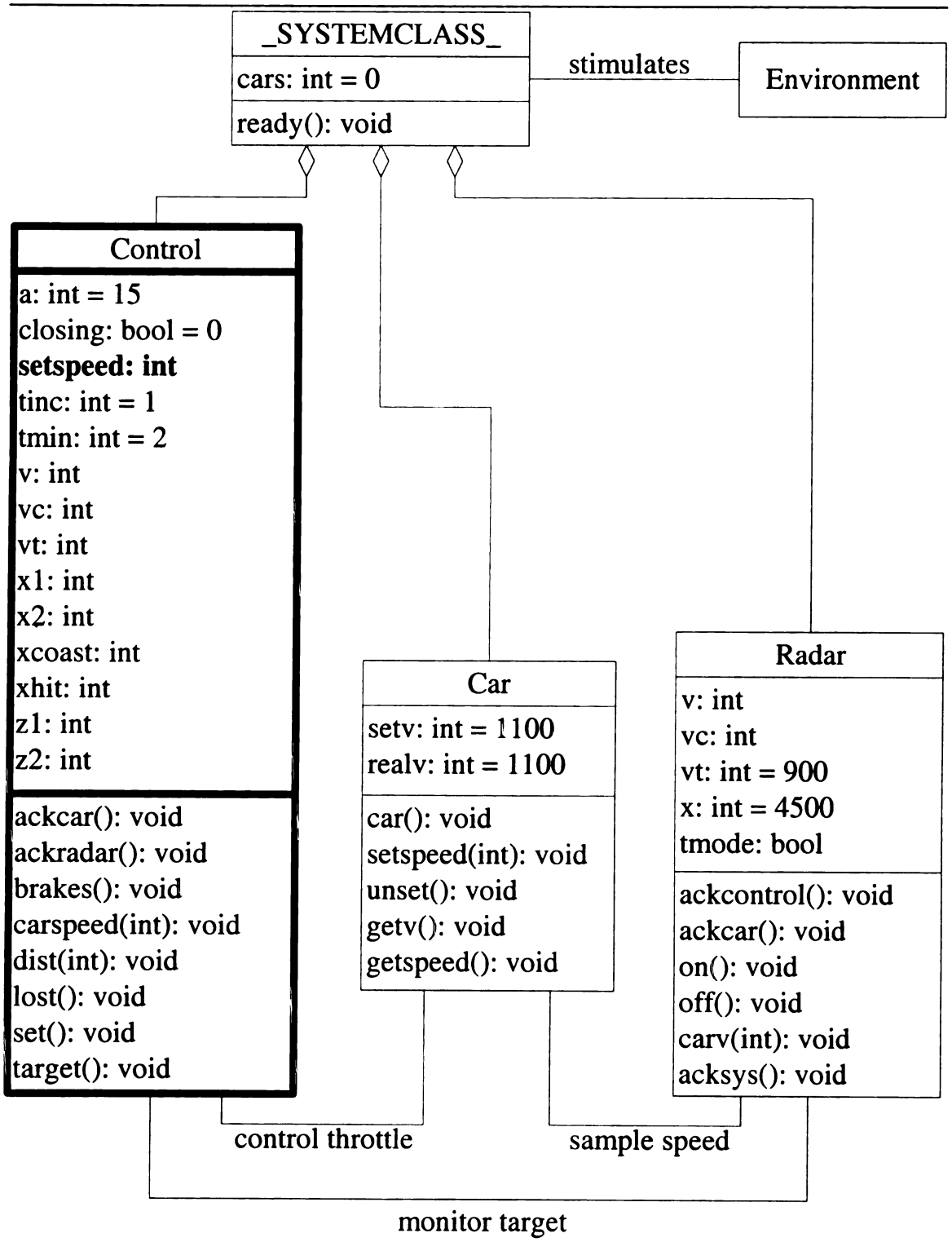


Figure 7.2: Class diagram for Adaptive Cruise Control

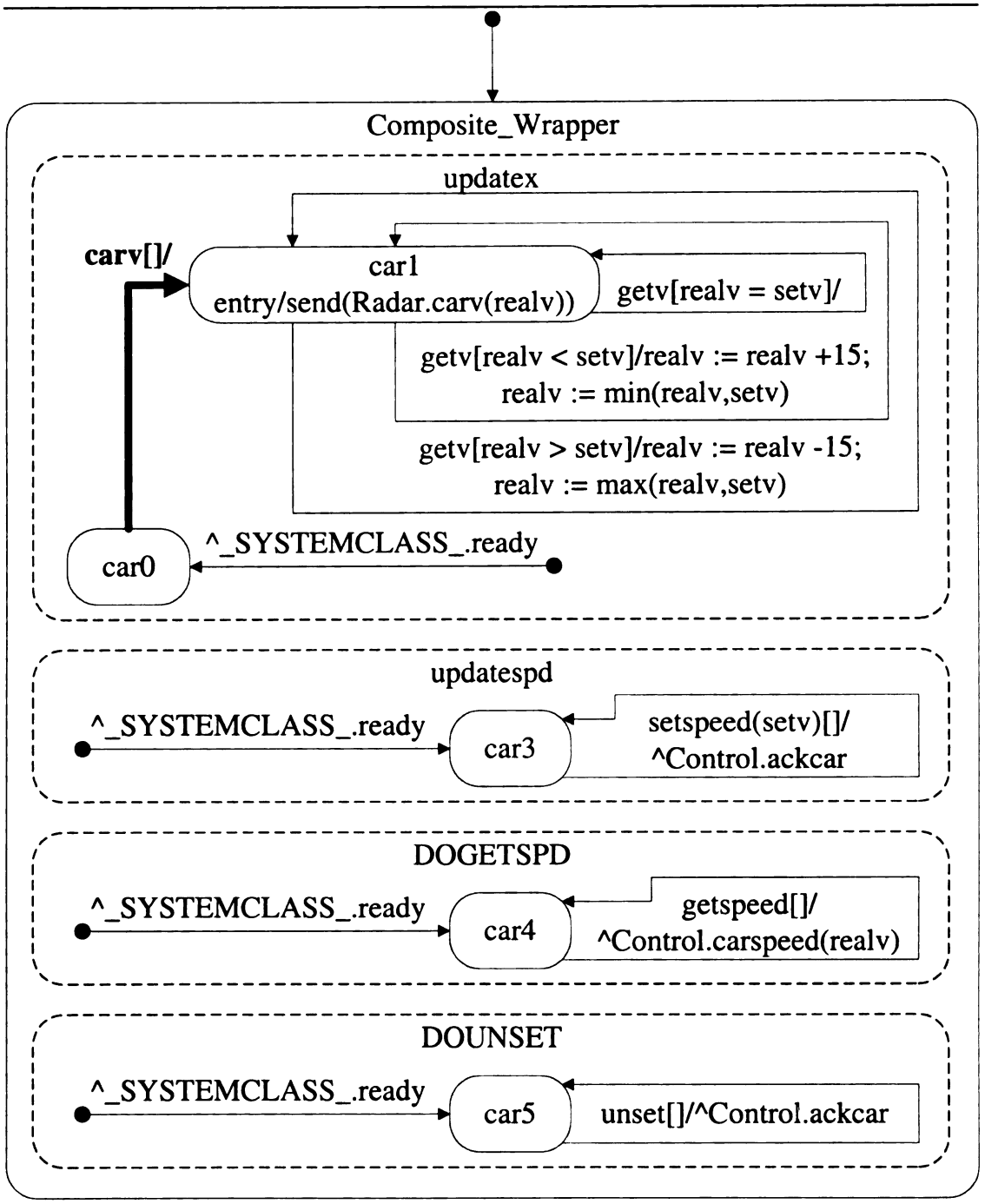


Figure 7.3: State diagram for Adaptive Cruise Control class Car

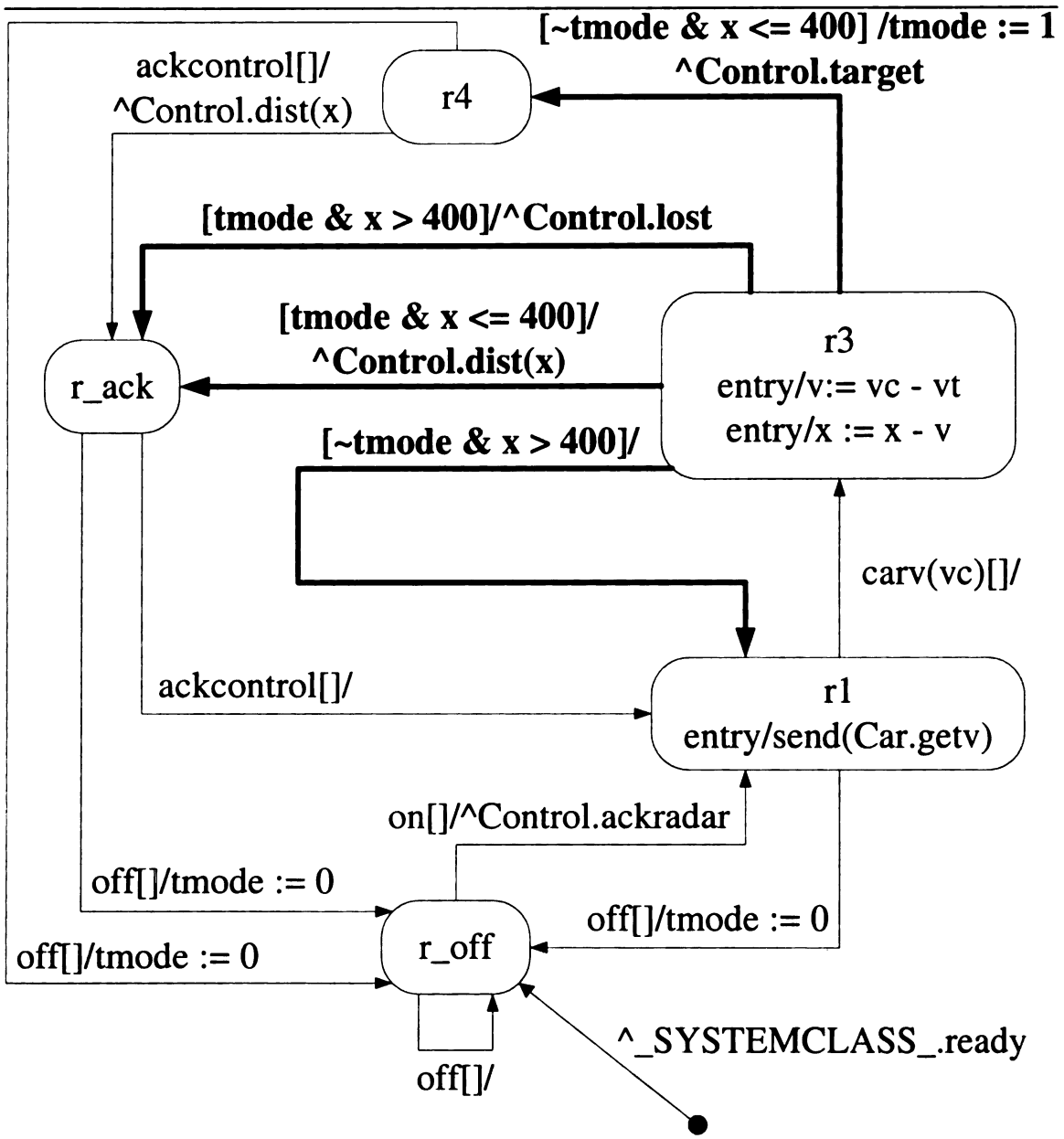


Figure 7.4: State diagram for Adaptive Cruise Control class Radar

partitions (delimited by dashed lines) to handle the main functionality of the Car. Partition `updatex` continually adjusts the car's speed (`realv`) to match the desired speed as calculated by Control (`setv`). The rate of change is specified as 1.5 feet per second (the distance scale for the model is tenths of a foot to give the Control algorithm enough distance samples to be effective[3]). Partitions `updatespd` and `DOGETSPD` receive the desired speed from Control and send the current speed to Control, respectively. Partition `DOUNSET` handles an *unset* message from Control, indicating that the system has been turned off.

Figure 7.4 (page 116) depicts the behavior of the Radar class. The Radar is off until activated by the Control. Rather than modeling actual radar samples, which would involve beam-intensity calculations and the possibility of losing the target, target acquisition is represented in the model by the distance `x` to the target becoming less than 400 feet (the range of the Radar), when the boolean flag `tmode` is set to *true*. The Radar continually performs distance calculations and, after target acquisition, informs the Control of the distance to the target.

We did not include multiple iterations of the Control state diagram in this chapter; however, the final refinement of the diagram is included in Figure E.2 in Appendix E, page 254. (The corresponding Promela code for the Control class and its state diagram is included in Appendix F, page 257.) For discussion purposes, the initial behavior of Control is captured by Figure E.2, minus the dashed and bold transitions which were added as later refinements. The Control accepts a *set* message from the environment, queries the Car for its current speed, and activates the Radar. After a target is acquired, Control receives distance samples and begins periodic calculations for the zones described in Figure 7.1, line 1 (page 111).

[3]Experimentation with finer distance scales exceeded available memory for model checking without clear benefit.

7.3

7.3 Structural Analysis

In this section, we briefly describe a few subtle structural errors detected by MINERVA and Hydra, using visualization techniques to highlight the source of the errors.

In the case study, the tool suite detected that an instance variable **setspeed** was included in the Control class but was not used, and that instance variable **setspd** was used in the state diagram for the Control class but was missing from the Control class (this error, if propagated to the Promela model, would cause a syntax error in Spin). Visualizations for these two problems within MINERVA highlighted the class Control and its instance variable **setspeed** as shown in bold in Figure 7.2 (page 114), and also highlighted transitions and states in the state diagram for Control that used the instance variable **setspd** (not shown). The actual error was due to the misspelling of instance variable **setspd** in the Control class that resulted in an inconsistency between the class diagram and the state diagram for the Control class.

Structural analysis also detected that the event *carv* was used on a transition in the concurrent partition **updatex** of the Car state diagram (Figure 7.3, page 115), but there was no class that sent the signal *carv* to class Car anywhere within the model. As shown in bold in Figure 7.3, MINERVA highlighted the transition using the signal *carv* in the **updatex** partition. Instead of the signal *carv*, the developer meant to use the signal *getv*, a signal from the Radar requesting the current speed of the Car. While this developer error, if not corrected prior to Promela generation, would likely be discovered later during simulation or model checking with Spin when deadlock occurred in state **car0** of the **updatex** partition, determining the exact cause of the problem and automatically tracing it back to the diagrams would be difficult due to its subtle nature.

7.4

7.4 Behavioral Analysis

We use Spin in either simulation or model-checking mode to perform behavioral analysis of UML diagrams via their automatically generated formal model. MINERVA's visualization techniques playback simulation and counterexample traces within UML diagrams, facilitating understanding of behavioral analysis results. Section 7.4.1 illustrates (via the Adaptive Cruise Control example) how simulation and visualization can be used to detect and highlight errors in UML models. In Sections 7.4.2 and 7.4.3 we model check several requirements-based properties against the Adaptive Cruise Control model to attempt to detect counterexamples in two situations: first, when the car successfully trails the lead vehicle; and second, when the car approaches the lead vehicle too quickly to avoid entering the safety zone. Finally, in Section 7.4.4, we use model checking and visualization to detect and highlight errors in a refined version of the model that includes the driver applying the brakes.

In our approach, we use requirements-based properties and model checking to detect counterexamples in order to validate UML models. That is, we are looking for counterexamples (indicating flaws in a UML model) rather that trying to verify correctness of a model. In this chapter, the phrase "verified successfully" means that Spin did not produce a counterexample in an exhaustive search of the generated state space, while "failed" means that the property did not hold, and Spin did produce a counterexample. The number of transitions explored, which Spin reports as a combination of the number of states stored, matched, and visited during a verification, is a measure of how much work Spin performed. We used Spin version 3.3.3 with the default memory allocation on a Solaris-based Sparc Ultra-60 workstation with 238 megabytes of free RAM and 305 megabytes of available swap space on disk. Unless otherwise noted, Spin's default memory allocation was sufficient for complete coverage of the state space for each property.

7.4.1 Simulation of Preliminary UML Diagrams

Simulation revealed an error in a constant used in a guard on several transitions in the Radar state diagram. During initial simulation, the model ran until the Radar issued a *target* message indicating that a vehicle had been detected at (presumably) about 400 feet (the situation depicted in Figure 7.1, page 111, line 2). However, Control then immediately raised an alarm that the car was too close to the target (*i.e.*, had violated the safety zone), and shut the system down. This behavior was unexpected, as the car should have closed with the lead vehicle and achieved proper trail distance.

The simulation trace data from Spin was processed by MINERVA to generate the sequence diagram in Figure 7.5 that reveals that the initial distance sent to class Control by the Radar after target acquisition was around 30 feet, which is much too small (the initial distance should have been close to 400 feet, the range of the Radar). Because class Radar generates the *target* event signifying target acquisition and sends distances to class Control, the state diagram for class Radar was a logical place to look for the problem.

The problem was found quickly by reviewing an animation of the Radar state diagram within MINERVA. This diagram is shown in Figure 7.4 (page 116). As mentioned previously, during the design of the Adaptive Cruise Control model we decided that the distance scale would be tenths of a foot, which gives the Control class enough distance samples to be effective without making the model intractable.[4] For example, 400 feet in the model is represented as 4000. However, in the Radar state diagram, the guards on state `r3` (shown in bold in Figure 7.4) to test whether a *target* message should be sent were inadvertently coded in the UML state diagram for Radar as 400 instead of 4000. This developer error had the effect of issuing a *target* message to class Control at approximately 30 to 40 feet, which does not allow

[4]Using a radar sampling rate of once per tenth of a second and a distance scale of hundredths of a foot causes about ten times more transitions per execution path explored during model checking, exceeding available memory without clear benefit.

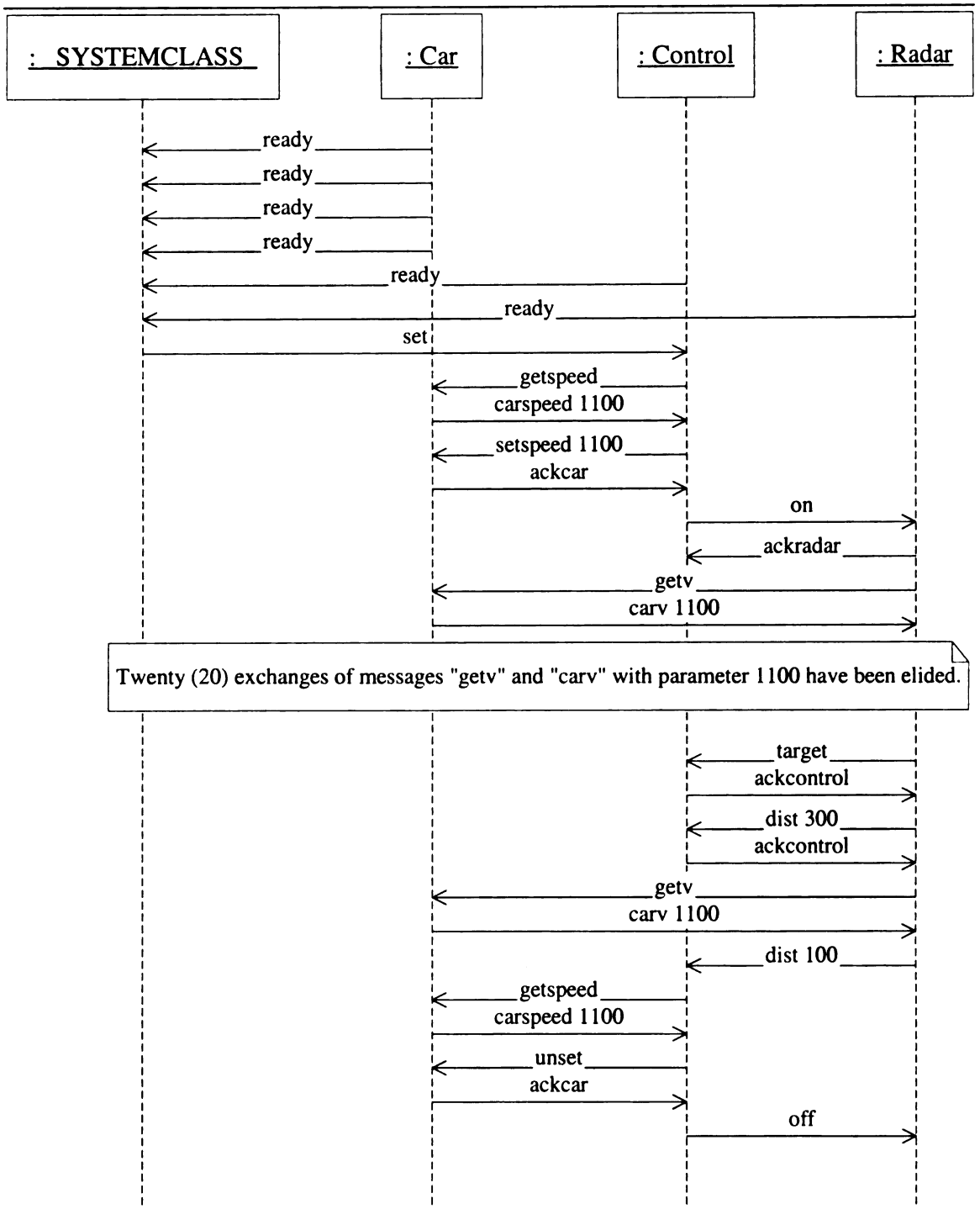


Figure 7.5: Sequence diagram from simulation.

sufficient time for the car to react.

When the guards in the Radar state diagram were changed to use 4000, the Promela model regenerated, and the simulation repeated, then the model ran as expected. A *target* message was sent around 400 feet, and the car closed with the lead vehicle to a distance of 186.5 feet. Afterwards, the car maintained proper trail distance for several minutes of simulated time until the simulation was terminated manually (without an external event, the car would trail the lead vehicle indefinitely). While this error was simple to fix once we detected it using MINERVA's sequence-diagram generation[5] and state-diagram animation of the Spin simulation trace data, relying solely on visual inspection of the UML diagrams might have meant that the error would be propagated through the design, coding, and testing stages before being detected.

7.4.2 Model Checking Initial UML Diagrams

Recall that this version of the model represents a typical scenario, where the Adaptive Cruise Control detects a slower-moving lead vehicle, calculates the appropriate trail distance, closes on the lead vehicle, and decelerates to match the lead vehicle's speed. The system also warns if collision is imminent and disengages the system when the safety zone is violated. In this section, we focus on the situation where the initial values for the speeds of both the car and the lead vehicle as well as the initial distance between them ensure that the car does not enter the safety zone. We examine several LTL variations of two requirements in an attempt to detect counterexamples.

Requirement: Normally, the car eventually matches the lead vehicle's speed. When the Adaptive Cruise Control is on and detects a lead vehicle, the requirements state that the car should eventually match the lead vehicle's speed. In terms of the model, this property could be expressed a number of different ways:

[5] MINERVA's sequence diagram depicts communication between UML objects whereas Spin generates message sequence charts (MSCs) in terms of Promela processes.

```
( []( sent(Control.target) ->                                    (1)
    <>(Car.realv == Radar.vt) ) )
```

says it is always (`[]`) the case that when class **Control** receives a *target* message (*i.e.*, a target has been acquired), eventually (`<>`) the speed of the **Car** (`realv`) will match the speed of the lead or target vehicle (`vt` as obtained by the **Radar**).

The above property may be modified slightly to hold only after **Control** receives a *set* message (*i.e.*, the cruising speed has been set) using Dwyer *et al.*'s *response* specification pattern[6] with an *after* scope (`[](q -> [](p -> <>s))`) [43], which says that s responds to p after q:

```
( []( sent(Control.set) ->                                       (2)
    []( sent(Control.target) ->
      <>(Car.realv == Radar.vt) ) ) )
```

This modification accurately reflects both the requirements and the model of the system, because the **Radar** will not issue *target* messages until after it has been activated by the **Control**.

Finally, we may choose to interpret the property as follows. The algorithm used for calculating trail distance and controlling the speed of the car sets a boolean flag, `closing`, to *true* after the car has become close enough to the lead vehicle that the car can begin decelerating. The property

```
( []( Control.closing ->                                         (3)
    <>(Car.realv == Radar.vt) ) )
```

says it is always the case that after the `closing` flag has been set to *true*, eventually the speed of the **Car** will match that of the lead vehicle.

These three claims were verified successfully with Spin, exploring approximately 1.14 million, 1.4 million, and 1.08 million transitions respectively.

Requirement: Normally, the car should not enter the safety zone. Under normal circumstances, the car should trail the lead vehicle at a safe distance as

[6]Chapter 2, Section 2.2.4, contains a brief overview of specification patterns [43].

calculated by the Control's algorithm. At no time should it enter the safety zone, as defined on line 1 of Figure 7.1. This requirement can be expressed as a system invariant

```
( [](Control.z1 <= Control.x1) )                                    (4)
```

meaning it is always the case that the calculated minimum safe distance from the lead vehicle (`z1`) is less than or equal to the last sampled distance to the lead vehicle (`x1`). The property is verified successfully in less than 270,000 transition explorations.

7.4.3 Model Checking After Changing Default Conditions

The system must warn if collision is imminent and disengage the system when the safety zone is violated. In this section, we focus on the situation where the car approaches the lead vehicle too quickly to avoid entering the safety zone. We modify the UML diagrams used in Section 7.4.2 to adjust some of the initial values (*e.g.*, decrease `vt`, the speed of the lead vehicle as obtained by Radar), thus creating such a situation. Then we automatically regenerate the corresponding formal model and re-examine Property 4, which fails as expected (*i.e.*, the car enters the safety zone), and examine LTL versions of two related requirements:

Requirement: Entering the safety zone disengages (turns off) the system. Once the car has entered the safety zone, it is too close to the lead vehicle to avoid a collision; so the cruise control must be disengaged. In terms of the UML diagrams, disengaging the system means that the Control class returns to its `idle` state and the Radar class returns to its `r_off` state:

```
( [](  (Control.x1 < Control.z1) ->                                 (5)
    <>( in(Control.idle) & in(Radar.r_off) ) ) )
```

This property is verified successfully in less than 220,000 transition explorations.

Requirement: If collision is imminent, yet the car is still outside the safety zone, then do not begin disengaging the system until the car enters

the safety zone. We say a collision is *imminent* if the Control algorithm determines that a collision will occur without action on the driver's part. However, the Control will not begin disengaging the system until the car enters the safety zone so that the driver may change lanes without slowing down.

We can express this requirement using Dwyer *et al.*'s *absence* pattern with a *between* scope (`[]((q & !r & <>r) -> (!p U r))`) [43], which says that p is *false* between q and r:

```
( []( ( (Control.x1 < Control.xhit) &                                    (6)
    ~(Control.x1 < Control.z1) &
      <>(Control.x1 < Control.z1) ) ->
        ( ~(in(Control.alloff) ) U
          (Control.x1 < Control.z1) ) ) )
```

where `xhit` is the calculated distance from the lead vehicle such that if the car were that distance behind the lead vehicle and began decelerating immediately, it would still hit the lead vehicle. This expression says that the Control will not enter its `alloff` state (*i.e.*, will not begin the process of disengaging the system) between the time that collision is imminent, yet the car is still outside the safety zone, and the time that the car enters the safety zone. The property is verified successfully in approximately 342,000 transition explorations.

7.4.4 Model Checking New Driving Scenarios

All of the preceding requirements-based properties were checked against the initial version of the Adaptive Cruise Control UML model (in two different configurations), which uses only one event external to the system, *set*, sent by the Environment to simulate the driver pressing a button on the steering column to activate the cruise control. The scenarios involving other external events, such as the driver applying the brakes, represent exceptional behavior and thus were not reflected in the original UML diagrams. We now refine the initial Adaptive Cruise Control UML diagrams,

specifically, the state diagram for Control that ultimately handles the *brakes* event, to accept a *brakes* message issued by the Environment, thus simulating the driver applying the brakes. These refinements are shown as dashed transitions in Figure E.2 in Appendix E, page 254. In order to focus our analysis efforts, we study the situation of when the brakes are engaged *after* the cruise control has been activated. After regenerating the formal model, we examine the following requirement:

Requirement: Brakes disengage the system. If the brakes are applied while Adaptive Cruise Control is controlling the speed of the car, then the requirements call for turning off the cruise control and radar. In terms of the UML diagrams, complete disengagement of the system is represented by the Control being in its `idle` state and Radar being in its `r_off` state.

```
( [] ( sent(Control.brakes) ->                                    (7)
    <> ( in(Control.idle) & in(Radar.r_off) ) ) )
```

Verification of the property failed initially, because the system model deadlocked before Control could reach its `idle` state. State diagram animation and sequence diagram generation within MINERVA using the trace data from the counterexample given by Spin illustrated that Control (Figure E.2) had stopped in state `caroff` after the following sequence of events (shown in the generated sequence diagram in Figure 7.6). Upon entry to state `calc`, Control issued a *getspeed* message to the Car to request its current speed, but before the Car had responded with a *carspeed* message, a *brakes* message arrived. The *brakes* message caused the Control class to enter its `caroff` state. A *carspeed* message, the response from the Car, was then at the top of the message queue for Control;[7] however, no transition from state `caroff` handled message *carspeed*. The Control class deadlocked in state `caroff`. After an additional transition was added to state `caroff` to handle the *carspeed* message (shown in bold in Figure E.2) and the Promela model was regenerated from the updated

[7]Recall that McUmber's formalization [31] uses queueing semantics. Spin outputs queue contents at the end of a verification run.

UML diagrams, the property was verified successfully, although the verifier had to be recompiled with the option -DMA=632 (suggested by Spin) when the default memory allocation proved to be inadequate. The verification explored approximately 3.9 million transitions.

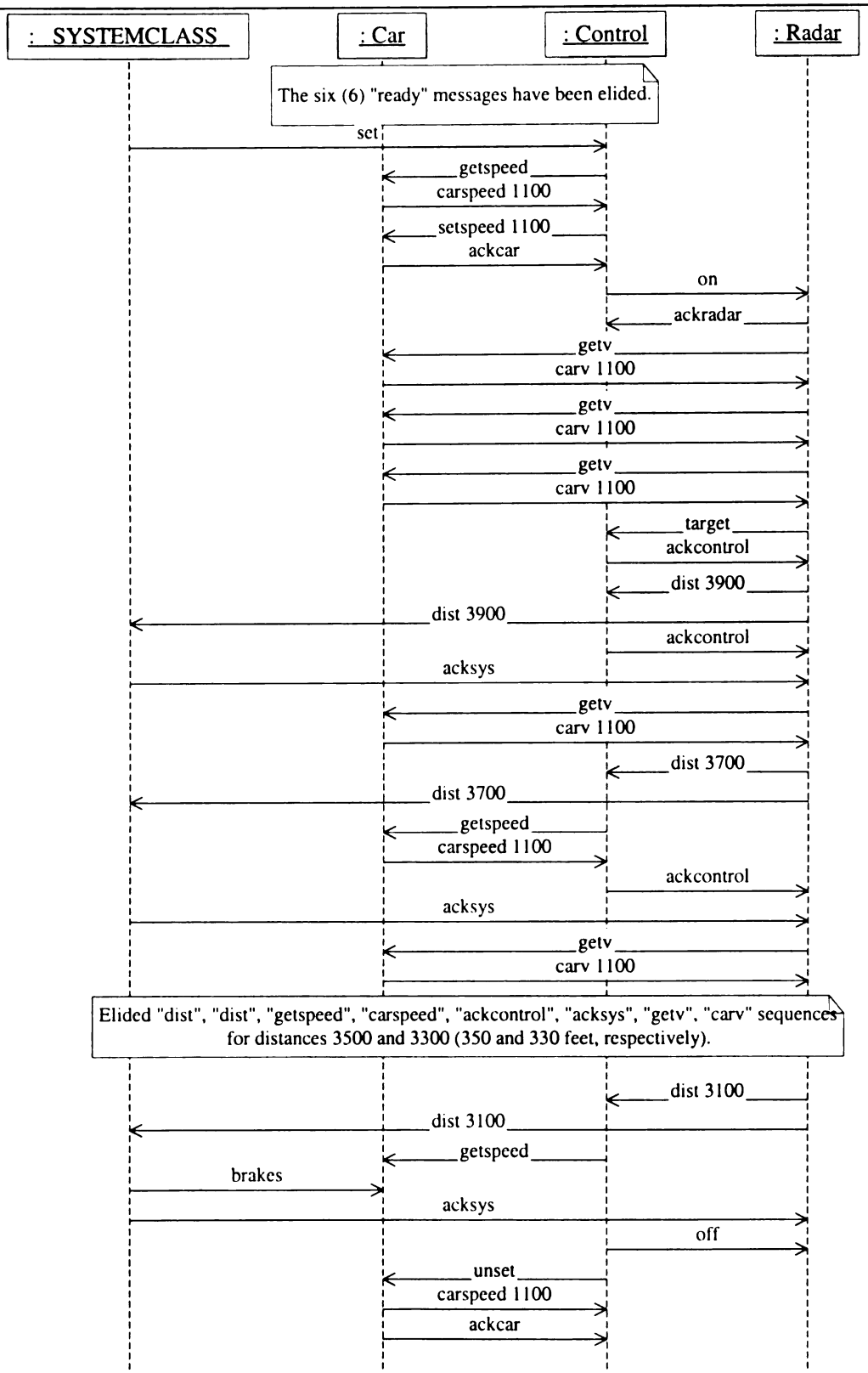


Figure 7.6: Sequence diagram representing counterexample.

Cha

Refl

In this ch
ners. S
meaning t
potential

8.1 C

In this s
approach
instrumen
the cost
model's s
model deri
size using
measure t
tion as ap
Fifth, we
potential

Chapter 8

Reflection: Cost and Applicability

In this chapter, we reflect on our approach to validating embedded systems requirements. Specifically, we examine two aspects of the approach: (1) the cost of instrumenting the formal model in order to enable UML-oriented visualization, and (2) the potential applicability of the approach to other application domains.

8.1 Cost of Instrumentation

In this section, we examine the cost of the visualization aspect of our overall approach to validating embedded systems requirements. First, we overview our instrumentation-based approach to UML-oriented visualization. Second, we define the cost of this instrumentation in terms of the change in the size of the formal model's state space. Third, we demonstrate the effect of instrumentation on formal model derivation, formal model internal representation, and formal model state space size using the *Producer-Consumer* model from Chapter 6 as an example. Fourth, we measure the cost of our instrumentation-based approach to UML-oriented visualization as applied to the Adaptive Cruise Control industrial case study from Chapter 7. Fifth, we present general conclusions regarding the cost of the approach and outline potential cost-containing enhancements to the approach.

8.1.1

As show
Chapter
a model
work [31
one acco
tegrated
framewo
rules [3
with our
the Prot
mapping
p. In or

Also sh
behavio
these be
the star

Part
visualiz
which a
UML-or
generat
various

8.1.1 Overview

As shown in Figure 8.1 (reprised for the reader's convenience from Figure 4.1 in Chapter 4), our overall approach to model development and analysis comprises both a model development and analysis framework [40] and an existing formalization framework [31, 33] that derives a formal (target language) model from a semi-formal (UML) one according to mapping rules. In this section, we focus on the portion of our integrated frameworks shown in Figure 8.2(a). In Figure 8.2(b), the formalization framework is instantiated with Hydra [31, 33] and McUmber's Promela mapping rules [31, 33], while the model development and analysis framework is instantiated with our graphical editor/visualization environment MINERVA [47, 50, 51, 52, 40] and the Promela analysis tool Spin [69]. Figure 8.2(b) illustrates that a set of Promela mapping rules F_{PROM} applied to a UML model u results in a derived Promela model p. In other words,

$$F_{PROM}(u) = p. \tag{8.1}$$

Also shown in Figure 8.2(b), our model development and analysis process applies the behavioral analyses of Spin, A_{SPIN}, to such a derived Promela model p. Results of these behavioral analyses applied to p, denoted as r_p, take the form of a trace through the state space of p. In other words,

$$A_{SPIN}(F_{PROM}(u)) = A_{SPIN}(p) = r_p. \tag{8.2}$$

Part of our model development and analysis process includes a mechanism for visualizing behavioral analysis results (*e.g.*, r_p) in terms of the UML model u from which a formal model (*i.e.*, p) was derived. For example, Chapter 6 describes several UML-oriented visualizations, including state diagram animation, sequence diagram generation, and collaboration diagram generation and animation. In order to enable various UML-oriented visualizations, information about a UML model itself (*e.g.*, in-

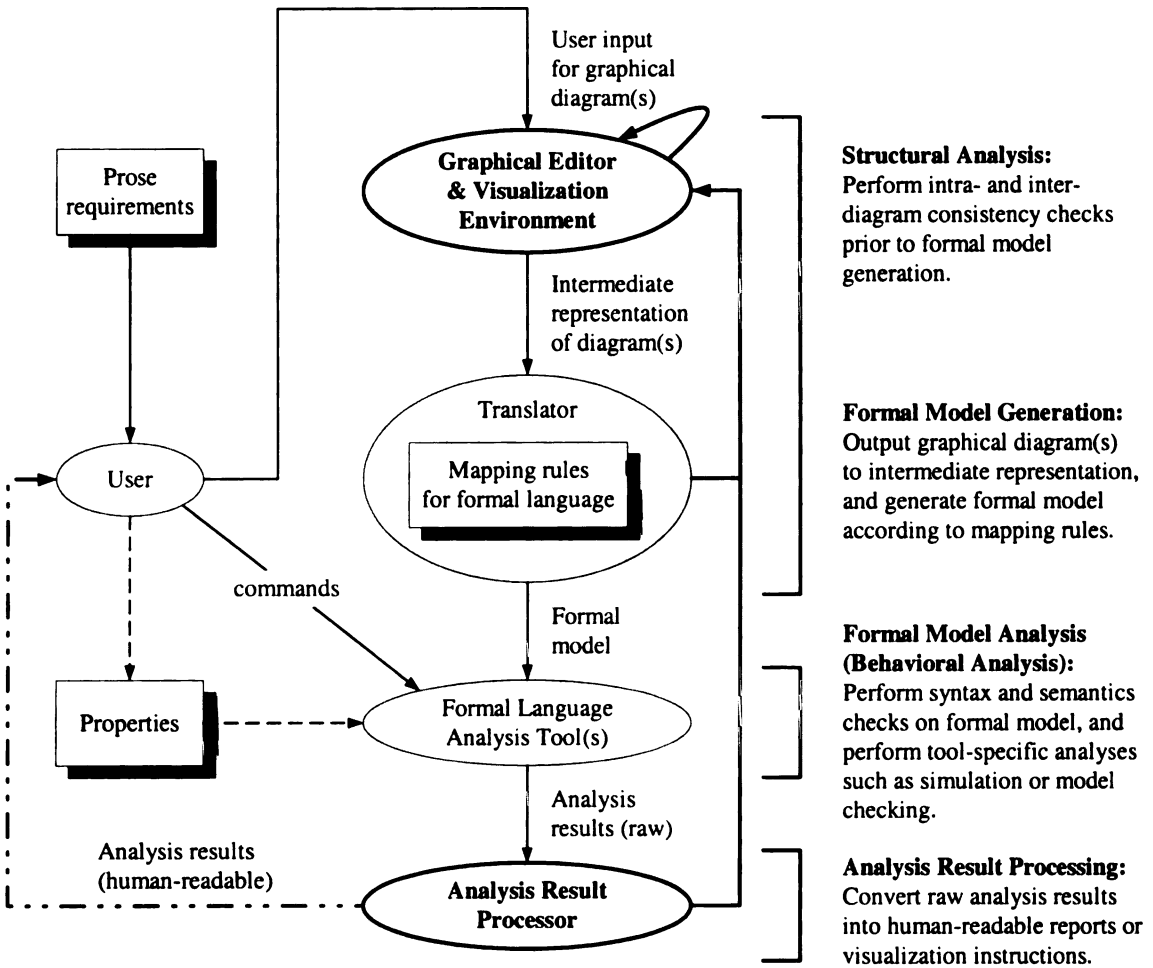


Figure 8.1: Model development and analysis framework encompassing formalization framework (reprised from Figure 4.1 in Chapter 4)

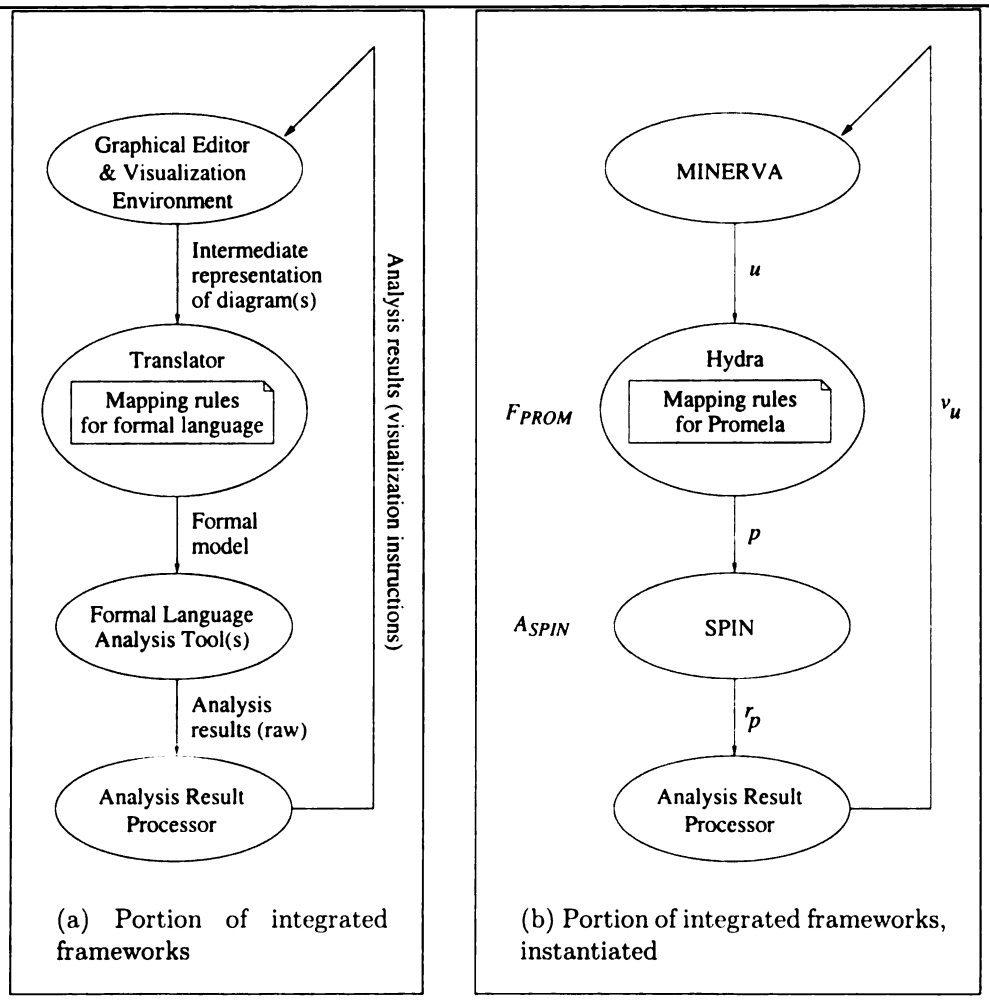


Figure 8.2: Portion of integrated frameworks from Figure 8.1 examined in this section

formation about when a UML state is entered or when a UML transition is taken) must be present in, or able to be inferred from, a formal model derived from it. Because formal languages in general have no concept of UML, UML-model-specific information must either be encoded in formalization rules from the beginning (potentially expensive) or added to a formal model during the translation process, ideally only when needed. For example, McUmber's Promela formalization rules [31, 33] take advantage of Promela's built-in queueing mechanisms to handle communication between Promela representations of UML state machines. Queue exchanges appear in and can be extracted from Spin's analysis results. However, a Promela model derived with McUmber's rules does not track which UML transition is currently enabled during execution. Therefore, such information does not appear in Spin's analysis results. Thus, McUmber's rules enable UML sequence diagram generation (message exchange), but not UML state diagram animation. To enable UML state diagram animation for a model using McUmber's rules, information about when a UML state is entered and/or when a UML transition is taken must be added to the formal model during the translation process.

Our approach to enable UML-oriented visualization has two parts. In the first part, we push information about a UML model not already encoded in formalization rules into the formal model derived from it as part of the translation process. This information then manifests itself in the results of analyses performed on the formal model. We depict this information as an asterisk (*) in Equations 8.3 and 8.4, which are refinements of Equations 8.1 and 8.2, respectively. Equation 8.3 states that a set of Promela mapping rules F_{PROM} applied to a UML model augmented with information about u, u^*, results in a derived Promela model augmented with information about u, p^*. Equation 8.4 states that results of Spin's behavioral analyses (A_{SPIN}) applied

to p^*. de

In the se
retrieves
converts
in Figur
mechanis
visualiza

V

We call
proach.
home [7

The
formation
will refer
formal m

- UM
- UM
- both

Note that
call this v

to p^*, denoted as r_{p^*}, take the form of a trace through the state space of p^*.

$$F_{PROM}(u^*) = p^* \tag{8.3}$$

$$A_{SPIN}(F_{PROM}(u^*)) = A_{SPIN}(p^*) = r_{p^*} \tag{8.4}$$

In the second part, a mechanism (*i.e.*, the **Analysis Result Processor** in Figure 8.2) retrieves such information about u, u^*, from the (raw) formal analysis results and converts it into visualization instructions in terms of the UML model (denoted as v_u in Figure 8.2(b)). Because different formal analysis tools have differing outputs, this mechanism is specific to the formal analysis tool used. Equation 8.5 illustrates the visualization mechanism specific to Spin, V_{SPIN}, applied to Spin's analysis results r_{p^*}.

$$V_{SPIN}(A_{SPIN}(F_{PROM}(u^*))) = V_{SPIN}(A_{SPIN}(p^*)) = V_{SPIN}(r_{p^*}) = v_u \tag{8.5}$$

We call this approach to enabling UML-oriented visualization the *breadcrumb approach*, likening it to Hansel dropping breadcrumbs in the forest to mark the way home [79].

The breadcrumb approach currently offers three options for pushing diagram information from a UML model into the formal model derived from it (in the sequel we will refer to these options as *breadcrumb options*). We may choose to augment the formal model with information about

- UML states only,
- UML transitions only, or
- both UML states and UML transitions.

Note that the default is to instrument no additional UML diagram elements. We call this version of the breadcrumb approach *naive* because it affects an *entire set* of

UM
just
trai
requ
I
with
these
flows
Hydra
in the
Similar
printf
tion has
analysis

8.1.2

One of t
State spa
seemingly
have a la
of the nu
number o
the state
model, w
State spa
Promela
and then

UML elements (*e.g.*, all UML states of all state diagrams in a model) rather than just a *select subset* of UML elements (*e.g.*, a group of user-identified "interesting" transitions of all state diagrams in a model, only those UML states referenced in a requirements-based property to be checked against the model).

In the case where we instantiate our model development and analysis framework with the formal analysis tool Spin and our Spin-specific visualization mechanism, these breadcrumb options transparently instruct MINERVA to add special *print actions* to UML states and/or UML transitions prior to their translation to Promela. Hydra translates such an action added to a given UML state into a `printf` statement in the corresponding Promela model indicating that the UML state has been entered. Similarly, Hydra translates such an action added to a given UML transition into a `printf` statement in the corresponding Promela model indicating that the transition has been taken. The Spin-specific visualization mechanism then searches Spin's analysis results for the output of these `printf` statements (*i.e.*, the breadcrumbs).

8.1.2 Defining Cost

One of the most important issues in model checking is the size of a formal model's state space [70] (measured as the number of unique states in the state space). A seemingly small change to a formal model (*e.g.*, adding a `printf` statement) may have a large effect on the size of its state space. Therefore, we must measure the *cost* of the naive breadcrumb approach. As part of its analysis output, Spin reports the number of unique states in a derived Promela model's state space (*i.e.*, the size of the state space). For each of the three breadcrumb options applied to a given UML model, we measure the change in the size of the corresponding derived formal model's state space. To obtain baseline measurements for a given UML model, we derive a Promela model from the UML model without using any of the breadcrumb options and then use Spin to perform a state space exploration of this Promela model. We

use as a

For a P:

the char

where s

the brea

1. U

2. U

3. be

stea

crumb c

and a ne

8.1.3

To dem

formal n

Produce

that U

ures 8.3.

state the

states of

Effect o

Consume

use as a baseline the number of unique states in this Promela model's state space. For a Promela model derived with any one of the breadcrumb options, we measure the change in the size of the state space as

$$\frac{(size_{breadcrumb_i} - size_{baseline})}{size_{baseline}} \tag{8.6}$$

where $size_{breadcrumb_i}$ is the size of the state space for a formal model generated with the breadcrumb options

1. UML states only,
2. UML transitions only, or
3. both UML states and UML transitions;

$size_{baseline}$ is the size of the state space for a formal model generated with no breadcrumb options; a positive result indicates an increase in the size of the state space; and a negative result indicates a decrease in the size of the state space.

8.1.3 *Producer-Consumer* Example

To demonstrate the effect of each breadcrumb option on formal model derivation, formal model internal representation, and formal model state space size, we use the *Producer-Consumer* example from Chapter 6. For convenience to the reader, the (final) UML diagrams for the *Producer-Consumer* example are reproduced here (Figures 8.3, 8.4, 8.5, and 8.6). Recall that the role of the _SYSTEMCLASS_ is to instantiate the other objects of the system, as indicated by the "new" entry actions in the states of the _SYSTEMCLASS_ state diagram in Figure 8.4.

Effect on formal model derivation. From the UML model of *Producer-Consumer*, we derive the corresponding baseline Promela model (generated with no

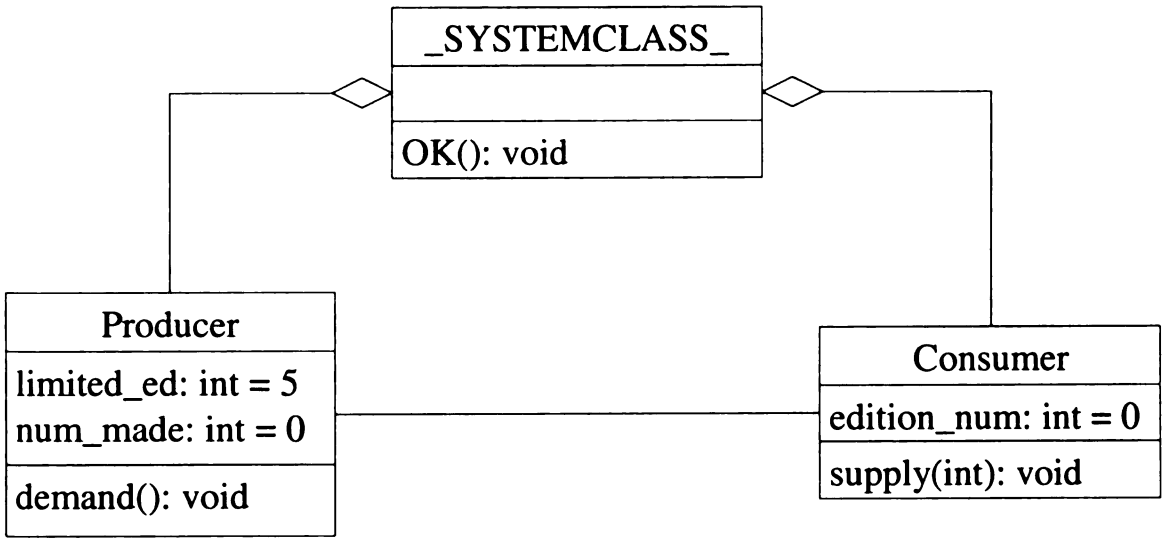


Figure 8.3: Class diagram for *Producer-Consumer*

breadcrumb options), as well as Promela models generated with each of the three breadcrumb options: UML states only, UML transitions only, and both UML states and UML transitions. The Promela code for each of these four models is listed in Appendix G (page 266) in Sections G.1, G.2, G.3, and G.4, respectively. Breadcrumb options that transparently add print actions to a UML transition cause `printf` statements to appear in the generated Promela code for that transition (*e.g.*, Section G.3, page 278, line 93; Section G.4, page 285, line 107), while breadcrumb options that transparently add print entry actions to a UML state cause `printf` statements to appear in the generated Promela code for that state (*e.g.*, Section G.2, page 272, line 32; Section G.4, page 285, line 99).

Effect on formal model's internal representation. Recall (Chapter 2) that McUmber's Promela formalization [31, 33] maps each UML state diagram into a set of `proctypes` (flat UML state diagrams, such as those in the *Producer-Consumer* ex-

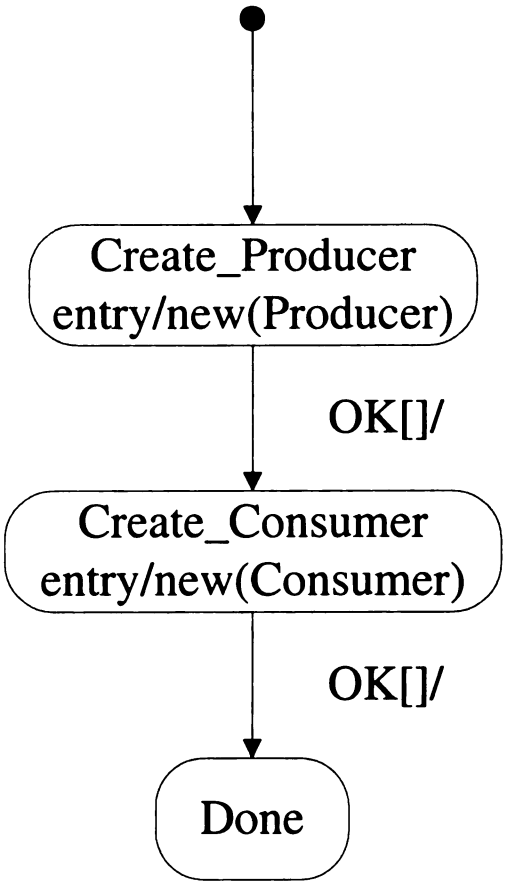


Figure 8.4: State diagram for *Producer-Consumer* class _SYSTEMCLASS_

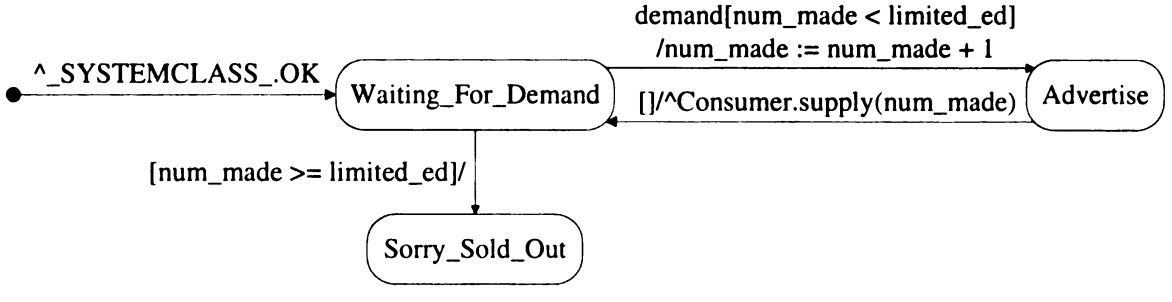


Figure 8.5: State diagram for *Producer-Consumer* class Producer

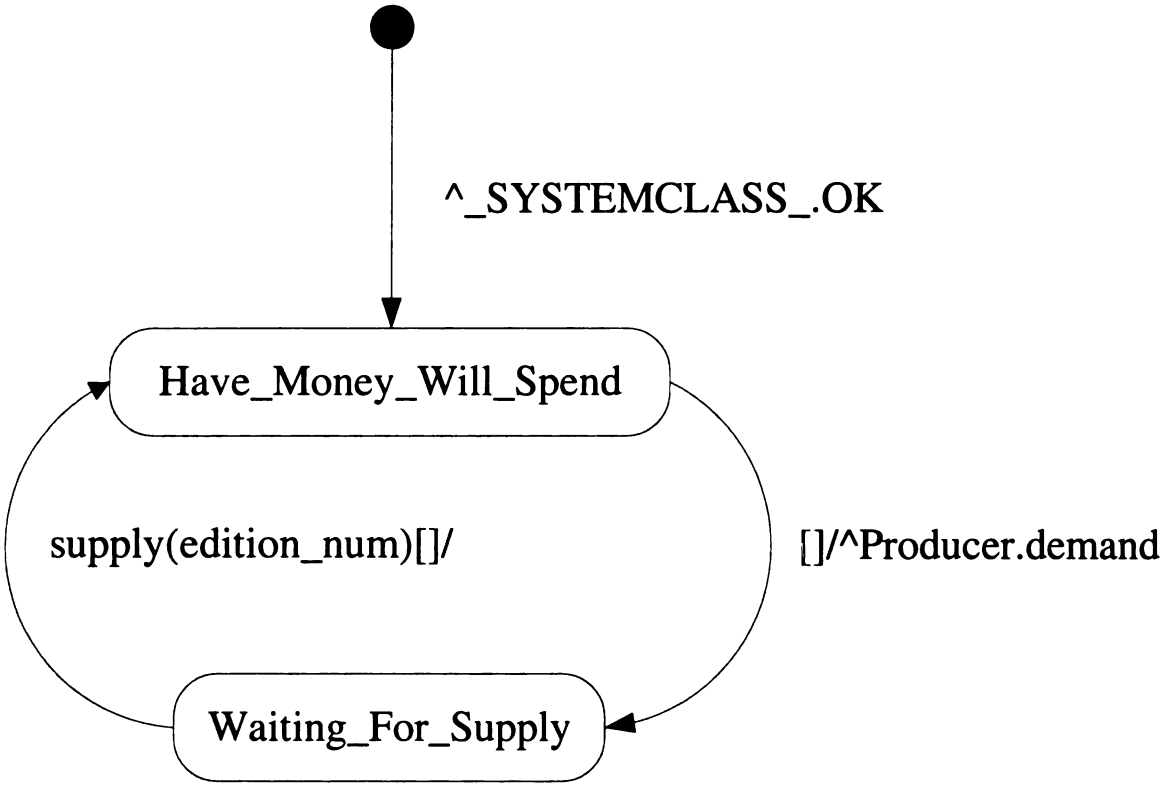


Figure 8.6: State diagram for *Producer-Consumer* class Consumer

ample, are mapped into a single `proctype` each; *i.e.*, a singleton set). Spin internally converts each `proctype` into a finite-state machine (FSM) with uniquely numbered states and offers an option to display it. For example, the FSMs for the baseline Promela model derived from the UML state diagrams of the _SYSTEMCLASS_, Producer and Consumer are shown in Figures 8.7, 8.8, and 8.9, respectively. Each arc in Figures 8.7, 8.8 and 8.9 corresponds to a Promela statement in the `proctype` derived from the UML state diagrams of the _SYSTEMCLASS_, Producer, and Consumer, respectively. Therefore, when breadcrumb options introduce `printf` statements into a derived Promela model, each `printf` statement in a `proctype` becomes a new arc and state in the FSM for that `proctype`. For example, the FSMs for the Promela model derived from the UML state diagrams of the _SYSTEMCLASS_, Producer and Consumer using the breadcrumb option "Both UML States and UML Transitions" are shown in Figures 8.10, 8.11, and 8.12, respectively. Note that each one of these FSMs (with instrumentation) has more unique states and arcs than its corresponding baseline FSM (without instrumentation).

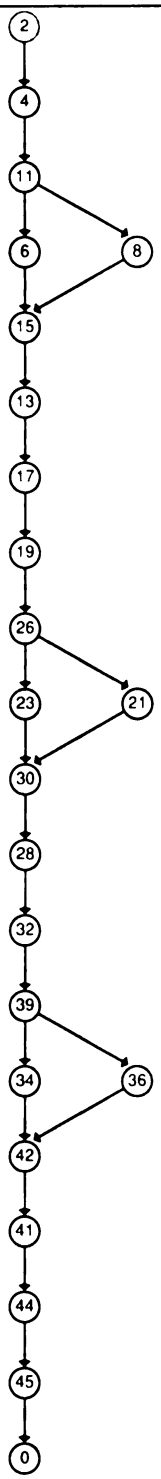


Figure 8.7: Finite state machine for _SYSTEMCLASS_ proctype, baseline

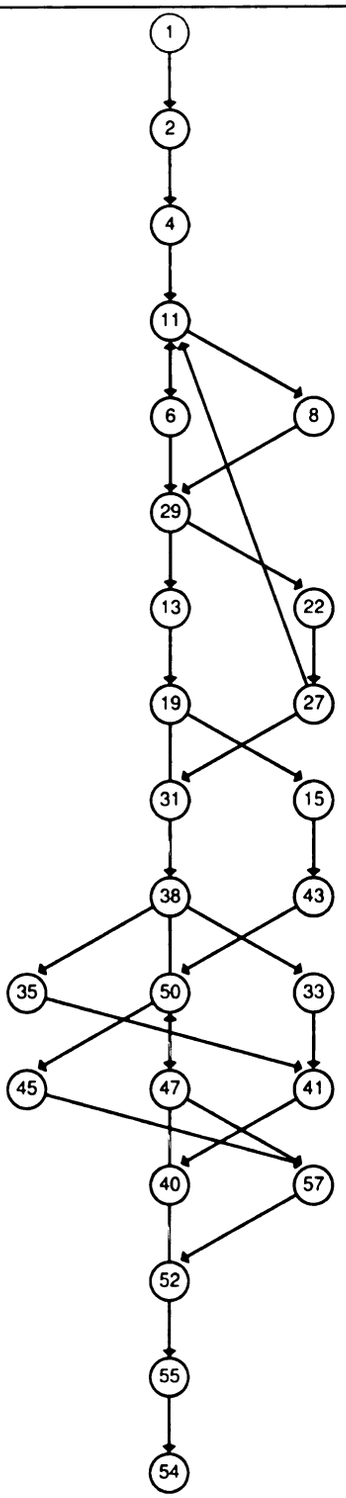


Figure 8.8: Finite state machine for **Producer** `proctype`, baseline

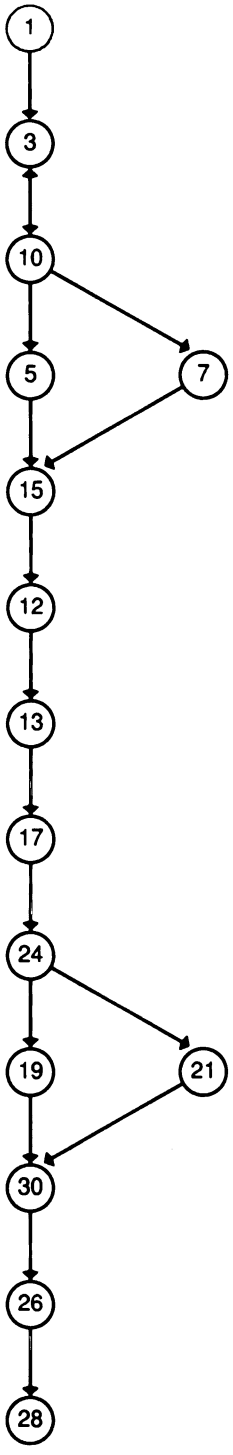


Figure 8.9: Finite state machine for **Consumer** `proctype`, baseline

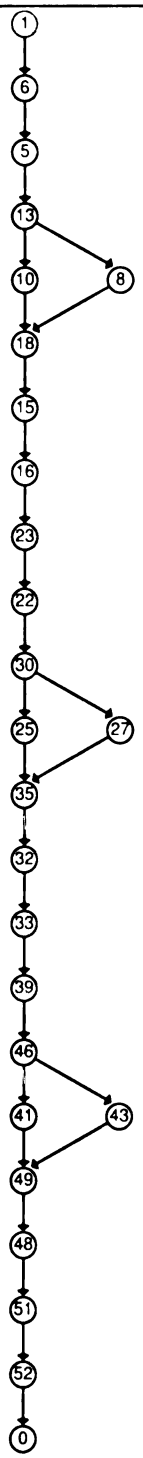


Figure 8.10: Finite state machine for _SYSTEMCLASS_ `proctype`, option **Both**

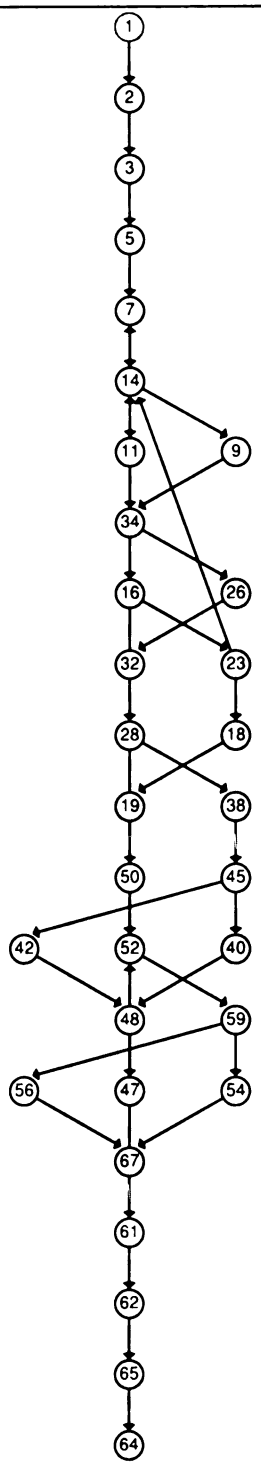


Figure 8.11: Finite state machine for **Producer** `proctype`, option **Both**

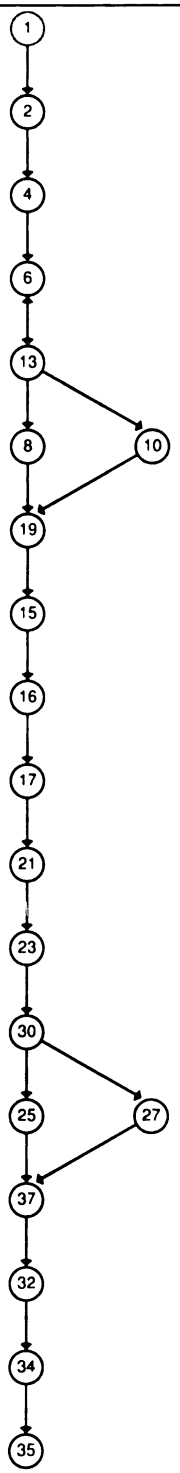


Figure 8.12: Finite state machine for **Consumer proctype**, option **Both**

Effect on formal model's state space size. For each derived Promela model for *Producer-Consumer* (*i.e.*, one model derived without any instrumentation and one model apiece derived for each of the three breadcrumb options), we then generate and explore its state space with the following commands:

1. `spin -a` *modelname*`.pr`

   Executes Spin on the Promela model *modelname*`.pr` (in this case, the automatically generated Promela model for *Producer-Consumer* with a given breadcrumb option, or lack thereof) with option `-a`. Option `-a` generates a *verifier*, a C program file called `pan.c` that when compiled and executed performs an exhaustive *verification* (*i.e.*, exploration) of the model's state space according to various run-time options.

2. `gcc -o` *modelname* `pan.c`

   Executes the `gcc` (GNU Compiler Collection) [80] C and C++ compiler on the file `pan.c` and outputs the executable to a file named *modelname*.

3. *modelname*

   Executes the file *modelname*, performing an exhaustive verification of the derived Promela model's state space (*i.e.*, in this case, checking user-specified *assertions*, or boolean conditions about the state of the model, and checking for *invalid endstates*, or states in which the overall model's execution ends before each `proctype` has reached its final statement or with non-empty channels). In the *Producer-Consumer* model, there are no user-specified assertions. The analysis results (contained in Appendix H, page 294) indicate an invalid endstate because at the end of execution, there is still one *demand* message left in the queue for the Producer. Recall (Chapter 6) that we did not refine the *Producer-Consumer* model further to handle this final message explicitly. For the purposes of this discussion, we do not consider this invalid endstate an error.

The verification generates the reachable state space of the model, a product of the interleaved FSMs of the `proctypes` of the derived Promela model. At the end of the verification, Spin reports the number of unique Promela states stored, *i.e.*, the size of the state space. The Spin analysis results for exploring the state space of the baseline Promela model and Promela models for the three breadcrumb options for the *Producer-Consumer* example are shown in Appendix H in Figures H.1, H.2, H.3, and H.4, respectively. The size of the state space for each model is shown in Figure 8.13. The amount of change in the size of the state space for the *Producer-Consumer* example, as calculated by Equation 8.6 (page 136), for each breadcrumb option is shown in Figure 8.14 (values have been rounded to two decimal places).

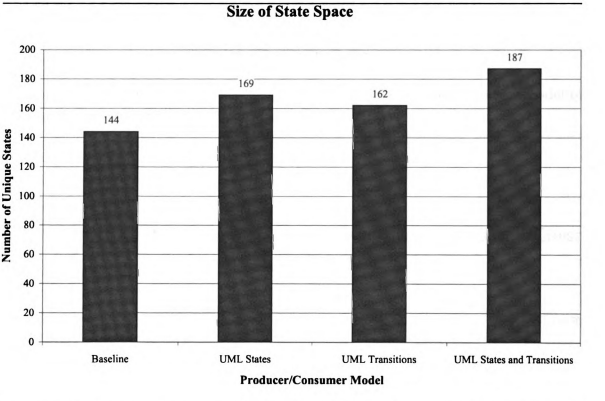


Figure 8.13: Number of unique (Promela) states in formal model of *Producer-Consumer* example for baseline and each breadcrumb option

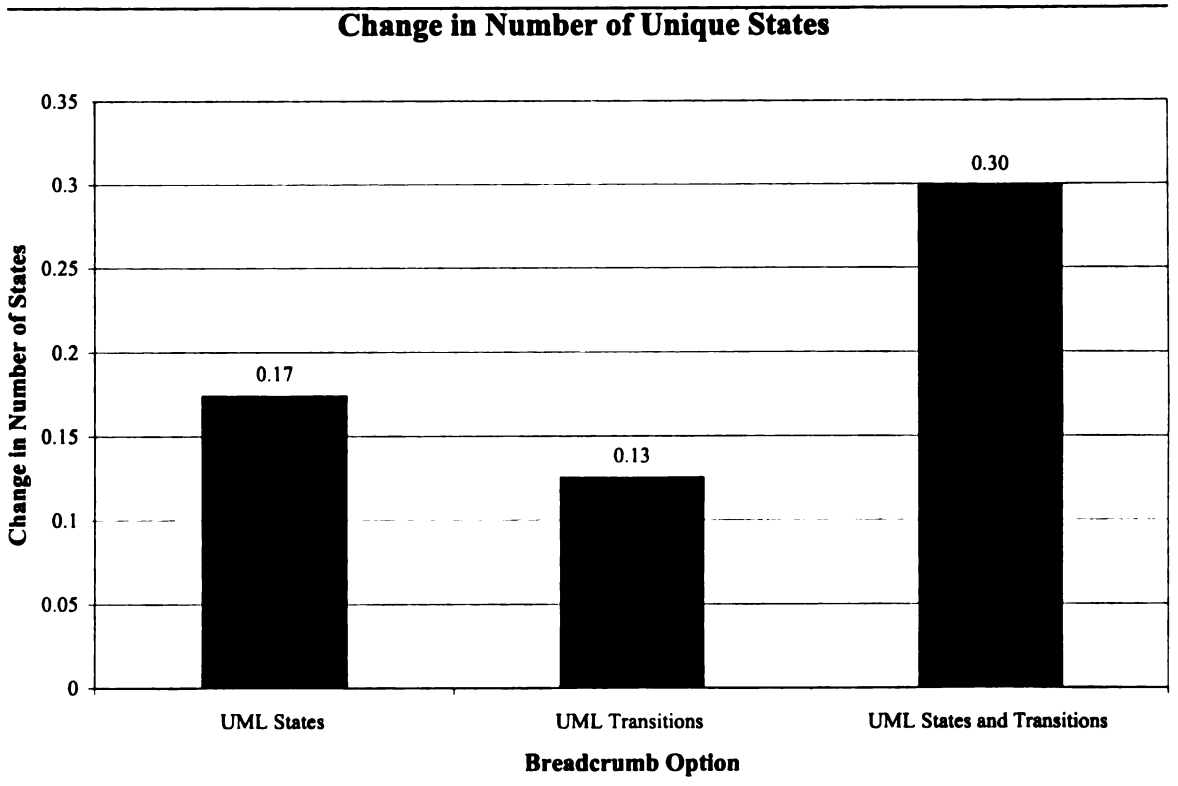


Figure 8.14: Cost: Change in number of unique (Promela) states in formal model of *Producer-Consumer* example for each breadcrumb option

8.1.4 Cost of Instrumentation as Applied to Adaptive Cruise Control

In this section, we consider the cost of our approach as applied to two Adaptive Cruise Control UML models from [40]. The models are identical in every way except one. In the first model, the Adaptive Cruise Control-equipped car acquires the lead vehicle as a target, achieves the proper trailing distance, and trails the lead vehicle indefinitely. In the second model, the (constant) speed of the lead vehicle (represented as an attribute in the UML model) is set too low for the Adaptive Cruise Control-equipped car to safely achieve the proper trailing distance. Thus, the cruise-control powers off almost immediately. These models represent a *normal* or *sunny-day* and an *exceptional* or

rainy-day scenario, respectively.

After generating the Promela model for each scenario, we apply Spin to the Promela models as described in the previous section to obtain baseline measurements for the size of the state space. Figure 8.15 shows the size of the state space for the model of each scenario. Note that although the UML models and subsequently the corresponding derived Promela models for each scenario are identical in every way except the value of the lead vehicle's speed, the state spaces of the formal models for each scenario differ significantly in size. This difference is explained by the fact that Spin takes into account only reachable states during analysis. The sunny-day scenario exercises a large portion of the **Adaptive Cruise Control** UML (and thus its corresponding formal) model, including several iterations through a calculation algorithm to determine the proper trailing distance and car speed. On the other hand, the rainy-day scenario exercises a much smaller portion of the **Adaptive Cruise Control** UML (and thus its corresponding formal) model, determining on the first iteration through the calculation algorithm that achieving a safe trailing distance is not possible and shutting down the cruise control.

For comparison, we also generate three more Promela models for each scenario, one for each of the three breadcrumb options, and perform analysis with Spin as described in the previous section. For both scenarios, the change in the size of the state space for each breadcrumb option is shown in Figure 8.16 (values have been rounded to two decimal places). The amount of change across all breadcrumb options is slightly larger for the rainy-day scenario. This effect is due to the smaller baseline state space size of the rainy-day model. The baseline state space size is the denominator in Equation 8.6 (page 136), so a similar absolute increase (decrease) in the state space size between two models will yield a greater amount of change in the model with the smaller baseline state space size.

We next examine Properties 1 – 4 from Chapter 7 (page 123) against the sunny-

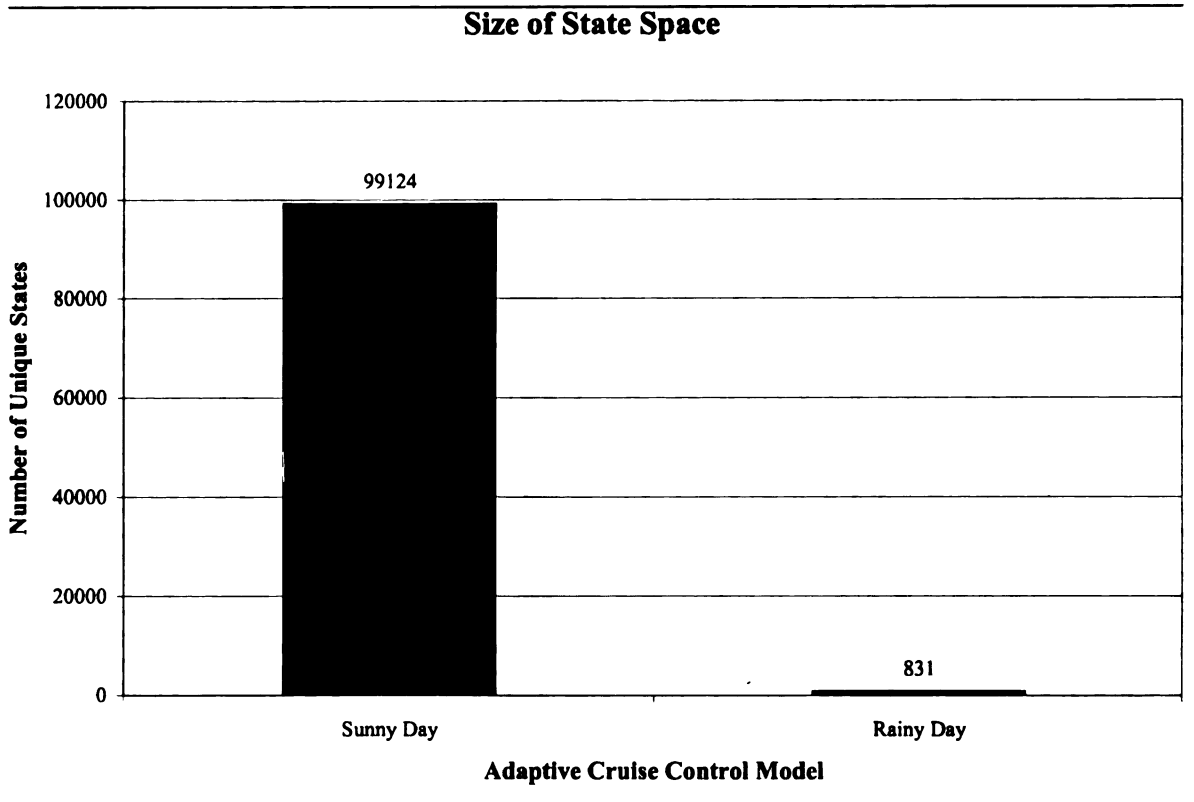


Figure 8.15: Number of unique (Promela) states in formal model of Adaptive Cruise Control for sunny-day and rainy-day scenarios

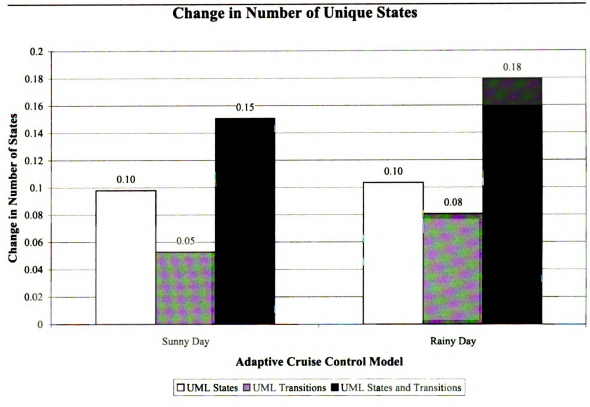


Figure 8.16: Cost: Change in number of unique (Promela) states in formal model of Adaptive Cruise Control for sunny-day and rainy-day scenarios

day scenario. The first three properties represent different ways in LTL to express the Adaptive Cruise Control requirement that the car eventually matches the lead vehicle's speed, while the fourth property expresses the requirement that the car should not enter the safety zone. The state space explored in each case is the product of the formal model executed synchronously with the FSM for the given property. The resulting state space size for each exploration is shown in Figure 8.17, while the change in state space size is shown in Figure 8.18 (values have been rounded to two decimal places). Although the state space size of the models for these properties vary widely, the change in state space size across all the breadcrumb options for these models is very similar.

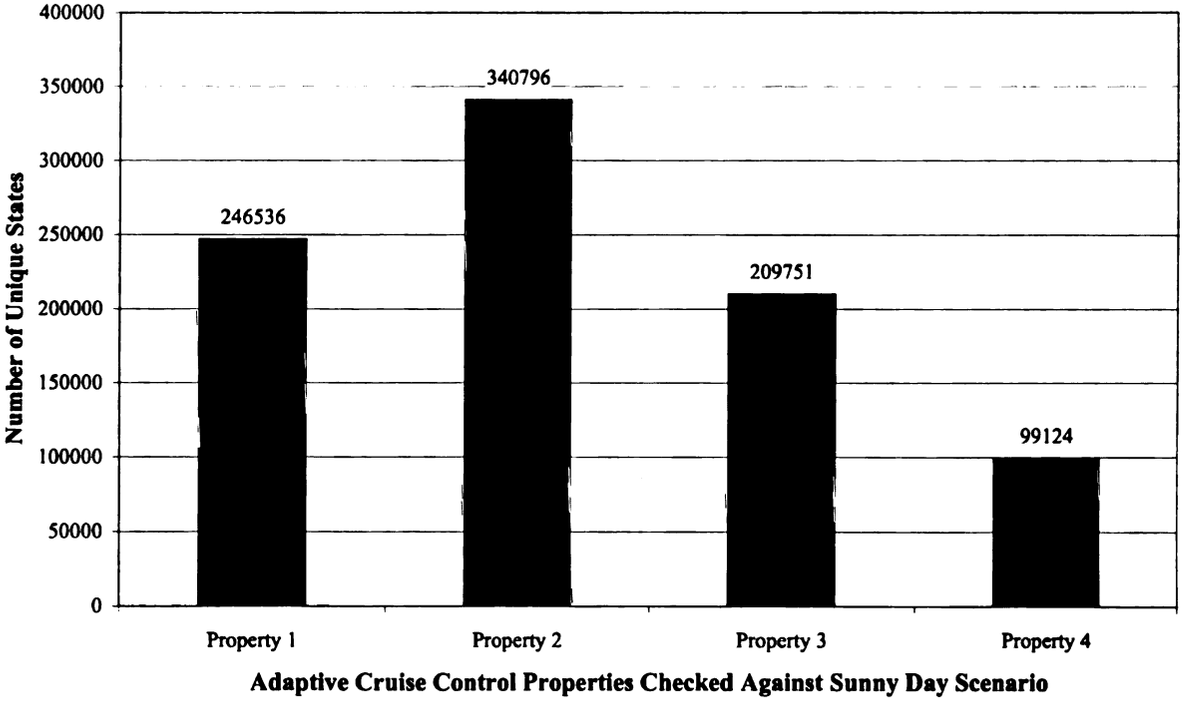


Figure 8.17: Number of unique (Promela) states in formal model of Adaptive Cruise Control for Properties 1 – 4 checked against sunny-day scenario

Finally, we examine Properties 4 – 6 from Chapter 7 (page 124) against the rainy-

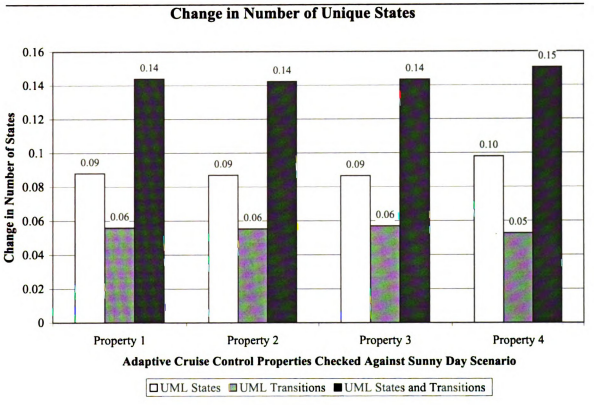


Figure 8.18: Cost: Change in number of unique (Promela) states in formal model of Adaptive Cruise Control for Properties 1–4 checked against sunny-day scenario

day scenario. Property 4 expresses the requirement that the car should not enter the safety zone, which produces a counterexample when checked against the rainy-day model. Property 5 expresses the requirement that entering the safety zone disengages the system, and Property 6 expresses the requirement that when collision is imminent, disengagement does not occur until the car enters the safety zone. The resulting state space size for each exploration is shown in Figure 8.19, while the change in state space size is shown in Figure 8.20 (values have been rounded to two decimal places). The state space size for the model checked against Property 4 is significantly smaller than the other two because Spin detected a counterexample almost immediately and terminated the analysis. Again, although the state space size of the models for these properties vary widely, the change in state space size across all the breadcrumb options for these models is very similar.

8.1.5 Conclusions

Figure 8.21 shows, for all the models measured, the cost for each of the three breadcrumb options as a function of state space size. The breadcrumb option **Both UML States and UML Transitions** pushes the most information into the formal model, instrumenting both UML states and UML transitions with print actions. As demonstrated in Section 8.1.3, these print actions manifest as printf statements in the Promela model, and as additional arcs in the FSMs for each proctype. Therefore it is not surprising that this option incurs the most cost in terms of the amount of change in the size of the state space for all the models measured (shown in Figure 8.21 as the dotted line). Beyond 100,000 states, the cost for this option drops below 15 percent. The breadcrumb option **UML Transitions** (solid line) appears the least costly for the models measured, dropping below six percent beyond 100,000 states. As current model checking endeavors may need to handle 10^{20} states and beyond, six percent is a more practical bound.

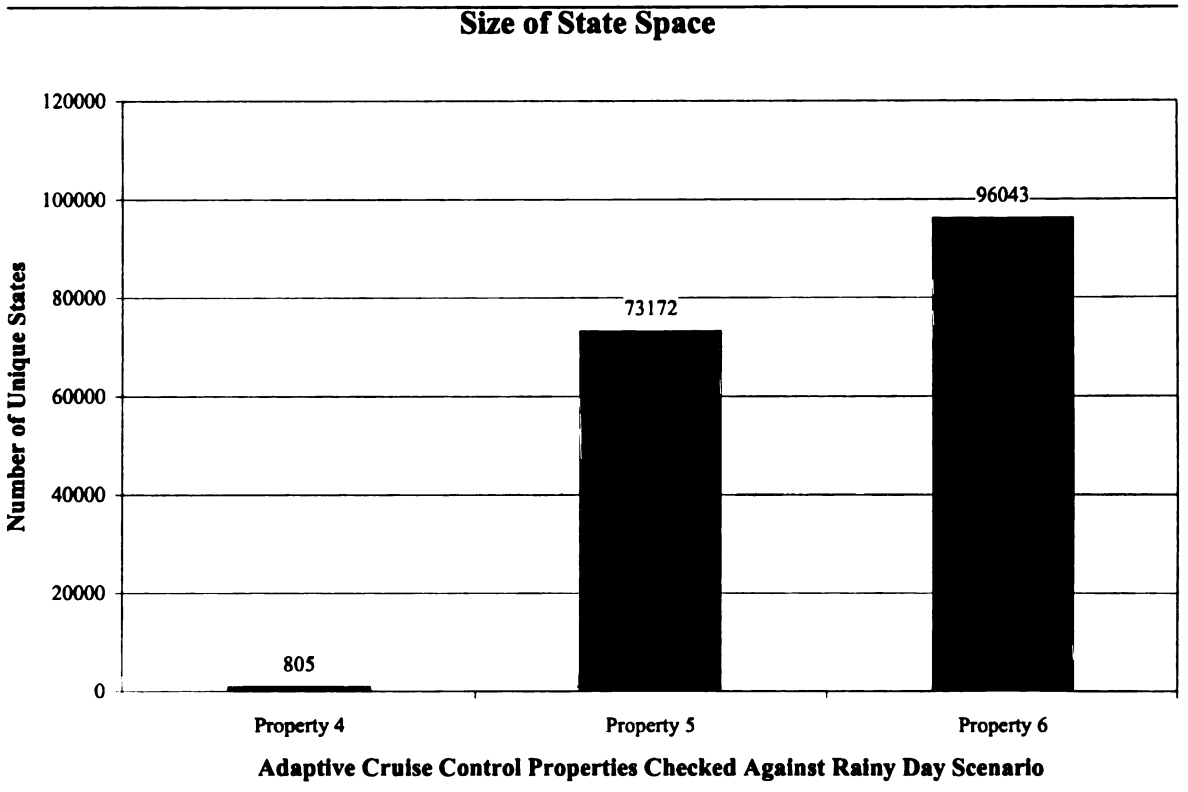


Figure 8.19: Number of unique (Promela) states in formal model of Adaptive Cruise Control for Properties 4 – 6 checked against rainy-day scenario

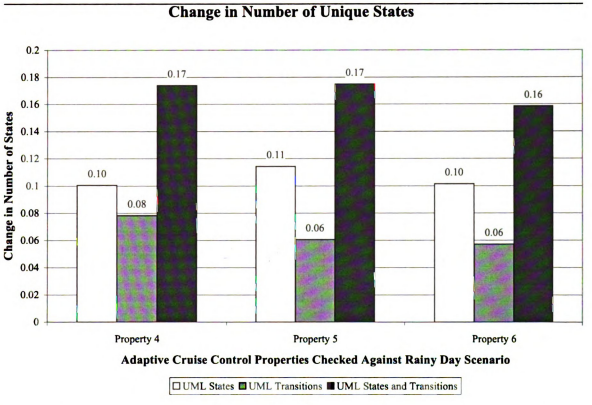


Figure 8.20: Cost: Change in number of unique (Promela) states in formal model of Adaptive Cruise Control for Properties 4 – 6 checked against rainy-day scenario

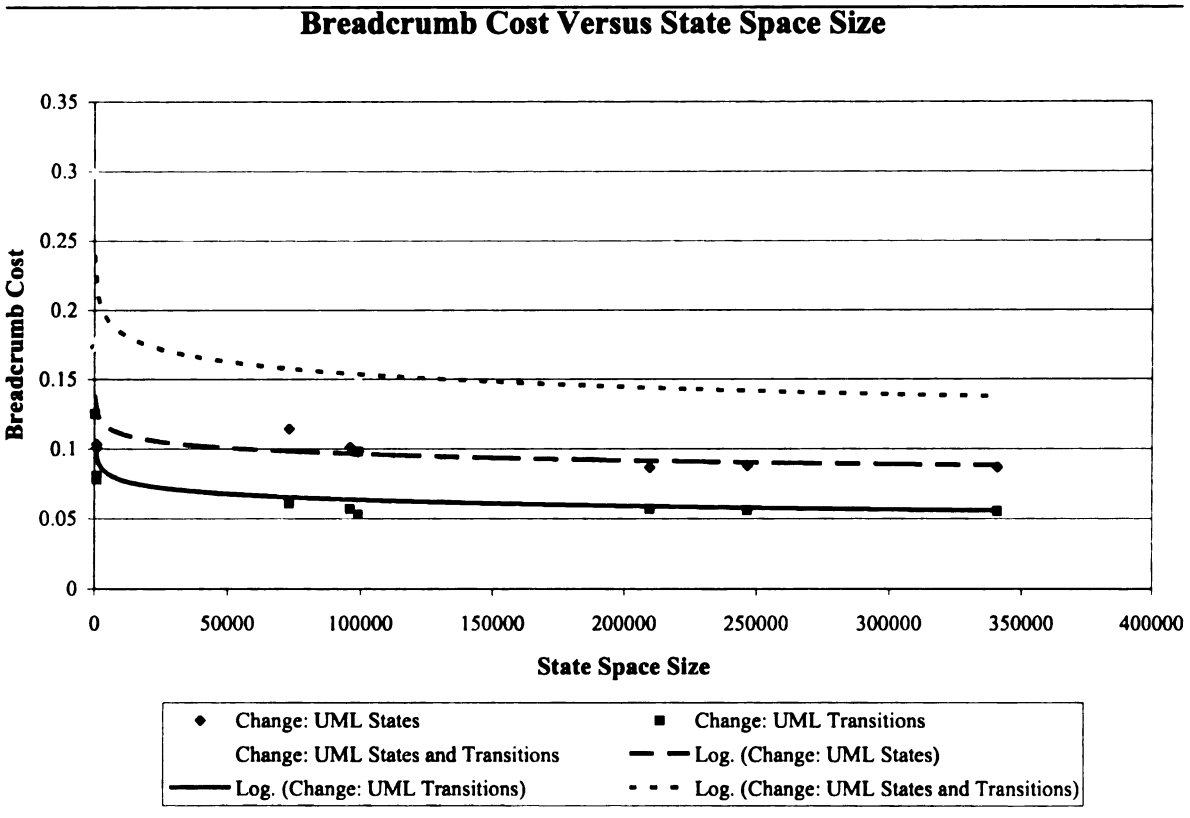


Figure 8.21: Cost versus state space size per breadcrumb option

In addition to focusing on the **UML Transitions** breadcrumb option that appears to incur the least cost of all the options, future work may consider various other strategies to limit the cost of the breadcrumb approach while still providing relevant feedback to the user. Three such strategies are outlined below:

User-selected elements. A *selective* approach to instrumentation of UML elements would, in general, incur less cost than the naive approach. To implement this strategy, we would need to add to MINERVA the capability for the user to select which UML elements to instrument and/or to ignore, similar to setting breakpoints in a debugging tool.

Elements in requirements-based properties. Automatically instrumenting only those UML states and/or UML transitions involved in a requirements-based property would, in general, incur less cost than the naive approach. Intuitively, the elements involved in a requirements-based property are of interest and should be instrumented. To implement this strategy, we would need to add to MINERVA the capability to automatically instrument the UML states and/or UML transitions involved in a requirements-based property.

Combination strategy. While either of the above two strategies would, in general, incur less cost than the naive approach, the first offers no guidance to users as to which elements(s) to instrument, while the second instruments only a small subset of UML elements of possible interest to the user. A combination strategy would automatically instrument elements involved in a requirements-based property thus giving a starting point for visualization of formal analysis results. However, this strategy would also enable the user to select additional UML states and/or UML transitions for instrumentation in order to extend the focus of visualizations.

8.2 Applicability of the Approach to Other Domains

This research applies UML formalization to enable developers to use automated tools to validate embedded-systems requirements. Specifically, we describe a model development and analysis framework that integrates an existing formalization framework [31, 33], and a model development and analysis process for a specific instantiation of the integrated frameworks (Chapter 4). This process systematically applies the steps of model creation, formal model generation, formal specification translation, model analysis, and visual interpretation of analysis results. In this section, we discuss the applicability of the presented approach to other application domains.

Our instantiation of the model development/analysis and formalization frameworks with tools MINERVA [40, 47, 50, 51, 52], Hydra [31, 48], and Spin [69] incorporates several assumptions relevant to the embedded systems domain that may preclude the use of the approach for other application domains. These assumptions include the use of state diagrams to model requirements, static model configurations, and object-level concurrency with single-threaded objects. Below, we discuss the ramifications of these assumptions:

Use of state diagrams. This approach uses state diagrams to model requirements. Therefore, other application domains that *might* be able to use this approach are those that typically rely on state diagrams to model requirements (*e.g.*, controllers, reactive systems, some user interfaces). Application domains that are not traditionally modeled with state diagrams, such as databases, would not use this approach.

Static model configurations. Embedded systems are control systems with static configurations; that is, objects are not dynamically created and destroyed during execution. McUmber's Promela formalization rules [31, 33], realized in Hydra [31, 48],

make this assumption. Therefore, application domains that typically have dynamic configurations where objects are created and destroyed on-the-fly, such as user interfaces, would not use this approach.

Object-level concurrency and single-threading. Our approach to modeling embedded systems requirements assumes that each object has a single thread of control, and that all objects in the system execute concurrently. Again, McUmber's Promela formalization rules [31, 33], realized in Hydra [31, 48], make this assumption. Therefore, this approach would not be applied to multi-threaded applications, or applications with many passive objects such as user interfaces.

Chapter 9

Extension: A Pattern-Driven Approach to Fault Handling

High-assurance systems must deliver their services in a manner that satisfies certain critical properties, which may include *fault-tolerance* properties [81]. A fault-tolerant system provides a certain guaranteed level of service despite the presence of *faults* (*e.g.*, hardware failures, environmental anomalies, dangerous or illegal system conditions). This chapter discusses how we can extend our modeling and analysis framework with *patterns* for handling fault-tolerance requirements.

Faults manifest themselves as *errors* (*e.g.*, a flag indicating a hardware error is set, a value is out of acceptable range), that in turn lead to system *failure*, *i.e.*, the system deviating from desired or acceptable behavior. The relationship between faults, errors, and failure is shown in Expression (9.1).

$$\text{fault} \rightarrow \text{error} \rightarrow \text{failure} \tag{9.1}$$

According to Lee and Anderson [82], there are four phases of fault tolerance that, taken together, prevent faults from leading to system failure:

1. **Error detection.** While a fault itself cannot be detected, the presence of a fault generates detectable error conditions.

2. **Damage confinement and assessment.** Because it is more likely than not that a delay has occurred between the occurrence of a fault and the detection of the offending error condition(s), overall damage to the system must be assessed before recovery is attempted.

3. **Error recovery.** Recovery techniques bring the system back into a well-defined and error-free state.

4. **Fault treatment and continued system service.** Detecting an error condition does not necessarily unambiguously identify the fault that caused it (*i.e.*, several different faults may cause the same error condition to arise). Ideally, the particular fault that caused the error condition to arise is identified and treated so that it does not recur. This phase differs from the previous three phases in that it addresses faults themselves rather than their resulting error conditions.

In this chapter, we focus on Lee and Anderson's phase 1, error detection, and phase 3, error recovery or correction. First, we provide an overview of two fault-tolerance concepts, namely *error detection* and *error correction*, and introduce three *Detector-Corrector Patterns* for modeling these concepts in UML. Second, we summarize Konrad and Cheng's *object analysis patterns*[1] [1] for (non-distributed) embedded systems; *i.e.*, templates in the spirit of *design patterns* [83] that include example UML class, state and sequence diagrams describing structural and behavioral patterns common to embedded systems. Object analysis patterns, and their included diagram templates, guide developers in modeling embedded systems requirements with UML. A given object analysis pattern also includes *constraints* [53], specification-pattern-based [43] temporal logic templates for properties of interest with respect to

[1] *Object analysis patterns* [1] were formerly termed *requirements patterns* [2, 3, 4].

the object analysis pattern providing their context. These temporal logic templates guide developers in specifying and instantiating requirements-based properties appropriate to check against a UML model created using the given object analysis pattern. Object analysis patterns and their constraints can be used to drive our model development and analysis process presented in Chapter 4. Third, because this chapter focuses on patterns for fault tolerance, we overview a specific requirements pattern[2], the *Fault Handler* requirements pattern [2, 3, 4], that describes how to incorporate a centralized **FaultHandler** into an embedded system. While this requirements pattern provides a state diagram template for the **ComputingComponent** of an embedded system, it does not give any modeling guidelines for the behavior of the **FaultHandler**. Therefore, using our previously introduced Detector-Corrector Patterns, we next extend the original *Fault Handler* requirements pattern with both a revised UML class diagram template and a new UML state diagram template for the **FaultHandler**. We then extend the underlying formalization framework with semantics for centralized fault handling. Finally, we apply this pattern-driven approach to the **Adaptive Cruise Control** example from Chapter 7.

9.1 Detectors and Correctors

To facilitate the modeling and analysis of fault-tolerance requirements, we propose to explore how the concepts of *detectors* and *correctors* from the fault-tolerance community [57] can be used to develop structural patterns for modeling fault-tolerance in non-distributed embedded systems. A detector is a system component that "detects" whether some state predicate (*i.e.*, a boolean expression over system variables) is satisfied by the system state [57]. Examples of detectors in embedded systems

[2]At the time of this writing, object analysis patterns for fault handling were still under development. This chapter presents preliminary work in revising the original *Fault Handler* requirements pattern. The overall pattern-driven approach to model development and analysis, first presented in [41], remains the same whether the patterns are termed requirements patterns or object analysis patterns.

include comparators, watchdog programs, and exception conditions. A corrector is a system component that "corrects" the system state (*i.e.*, adjusts the values of system variables) in order to satisfy some state predicate whenever it is not satisfied [57]. Examples of correctors in embedded systems include reset procedures and exception handlers.

In this work, we model specific types of detectors and correctors as classes in the UML class diagram as shown in Figure 9.1. The `detector` attribute of a PredicateDetector is modeled as a boolean expression over system variables, while the `corrector` attribute of an ActionCorrector is modeled as a sequence of actions adjusting the values of system variables. Modeling detectors and correctors as classes rather than as data types allows us to model the relationships between them with associations. A detector can be modeled without a corrector, as denoted by the arity of the "corrected by" association in Figure 9.1. Such a detector can be used to prevent a system from entering an undesirable state. Otherwise, a detector can be used together with a corrector that restores the system to a desirable state, as denoted by the arity of the "responds to" association in the figure. The type `boolExprString` indicates that the value of the PredicateDetector attribute `detector` is a string that should be parsed as a boolean expression, where the terms are constants, or references to attributes from the aggregating class or from other classes in a UML model. Attributes may be compared with constants or with other attributes. The type `actionSeqString` indicates that the value of the ActionCorrector attribute `corrector` is a string that should be parsed as a sequence of UML actions that adjust the values of system attributes (*i.e.*, attribute assignments or message sends). Because we model detectors and correctors in UML class diagrams as aggregated classes, PredicateDetector and ActionCorrector attributes (`detector` and `corrector`, respectively) may then be referenced in the UML state diagram owned by the aggregating class.[3]

[3]Recall (Section 2.2.5, page 25) that according to McUmber *et al.*'s diagram integration conventions, a state diagram expressing the behavior of a class is said to be *owned by* the class.

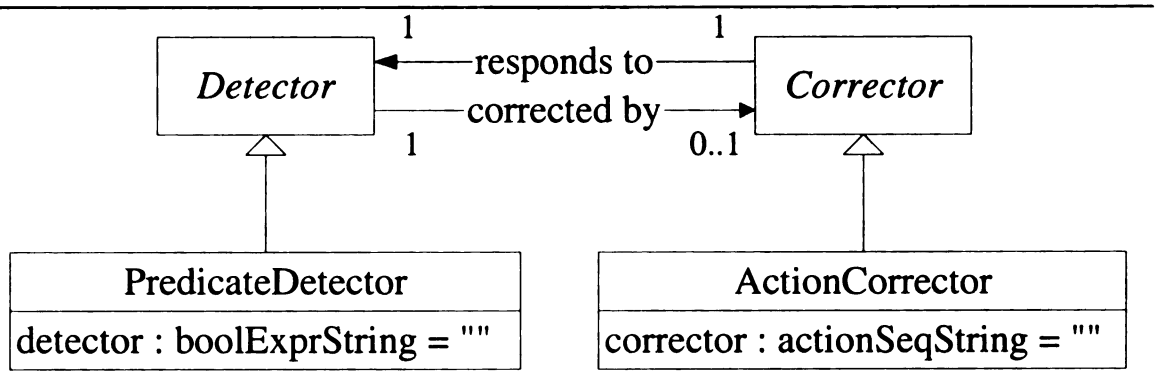
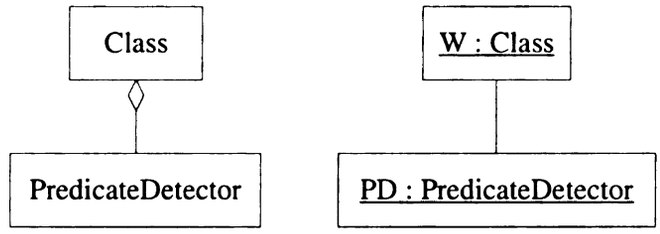


Figure 9.1: Modeling detectors and correctors in the UML class diagram

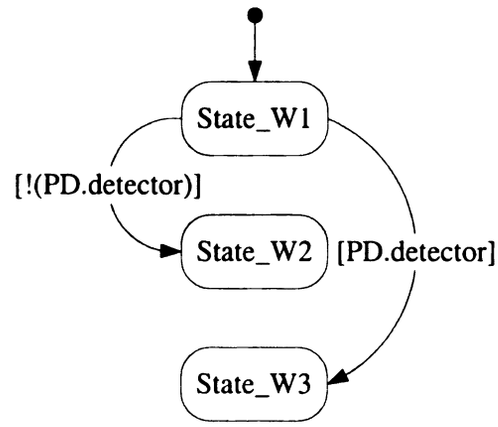
Thus far, we have determined three patterns for modeling detectors and correctors in UML. We refer to these patterns collectively as *Detector-Corrector Patterns.* Figures 9.2, 9.3, and 9.4 illustrate patterns 1, 2, and 3, respectively. In the first pattern (Figure 9.2), a detector is modeled without a corresponding corrector. There are no associations between the detector and any corresponding corrector. For example, the PredicateDetector in the class and object diagrams in Figures 9.2(a) and 9.2(b) is not associated with any ActionCorrector. Detectors modeled without correctors are used to prevent entry to a particular state when the error condition detected by the detector is *true.* For example, in one of our case studies, the Diesel Filter System (DFS) [41, 84] may not enter its cleaning cycle if the current system pressure is below 8,000 or above 10,000 *Pascals* (*Pa*). We model this usage in the UML state diagram with a negated detector as a guard on transitions targeting a particular state in order to permit entry to the state when the error condition detected by the detector is *false* (*e.g.*, the DFS may enter its cleaning cycle if the current system pressure is between 8,000 and 10,000 *Pa*). The detector is used as a guard on other transitions in order to transition to another state when the error condition detected by the detector is *true* (*e.g.*, when the pressure is above 10,000 *Pa*, the DFS shuts down). In the example state diagram in Figure 9.2(c), corresponding to the object diagram in Figure 9.2(b),

it is undesirable to enter `State_W2` when the condition detected by PD is *true*. Therefore, the condition `!(PD.detector)`, the negated detector, guards the transition from `State_W1` to `State_W2` (*i.e.*, if the condition detected by PD is *false*, then entering state `State_W2` is permissible). Additionally, the condition `PD.detector` guards another transition from `State_W1` to a desirable state when the condition detected by PD is *true* (*i.e.*, a state other than `State_W2`).



(a) Class diagram

(b) Object diagram



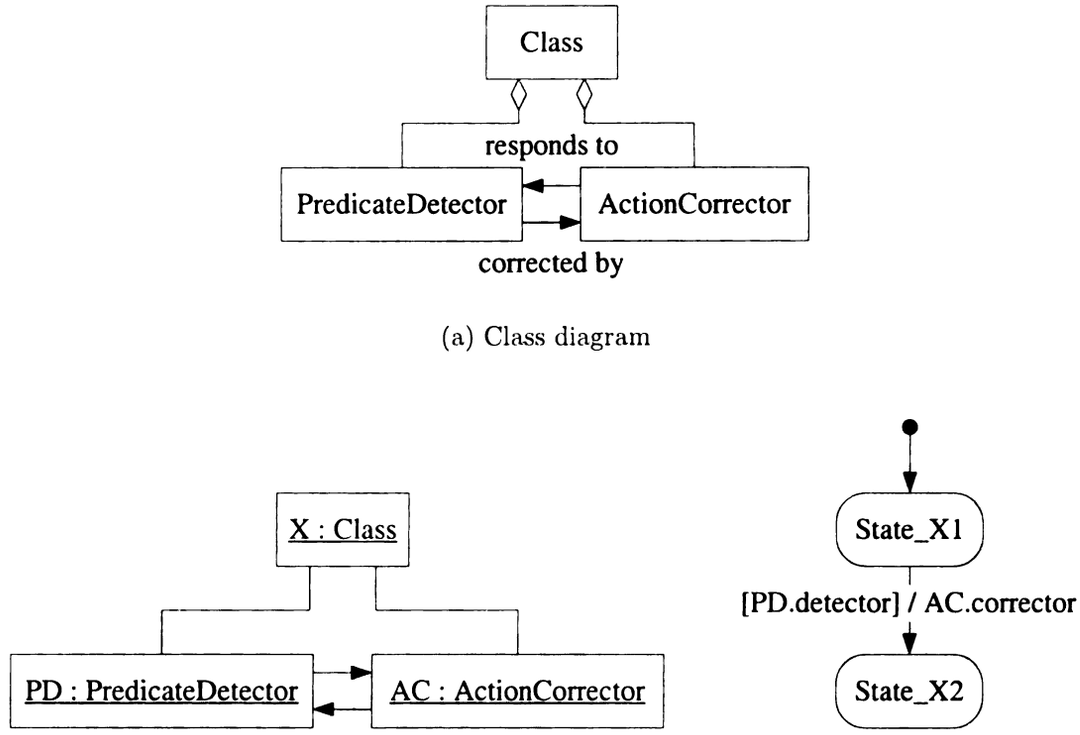
(c) State diagram

Figure 9.2: Pattern 1 for modeling detectors and correctors

In the second pattern (Figure 9.3), both a detector and its corresponding corrector are aggregated by the same class. There are associations between the detector and its corresponding corrector. For example, the PredicateDetector and ActionCorrector in the class and object diagrams in Figures 9.3(a) and 9.3(b) have associations between

them. These associations relate a particular detector with a particular corrector, since different error conditions most likely require different corrective actions. For example, in the Integrated Starter/Generator (ISG) [85] project, if the ISG is in running power generation mode, then low battery voltage requires that drive assist be disabled, whereas excessive battery charge requires that the charging system be turned off. This relationship between a detector and its associated corrector carries over into the state diagram, as illustrated in Figure 9.3(c). For illustrative purposes, the transition from `State_X1` to `State_X2` in the state diagram (that corresponds to the object diagram in Figure 9.3(b)) is annotated with the detector `PD.detector`, as a guard, and its corresponding corrector `AC.corrector`, as an action sequence. The transition is enabled when the condition detected by PD is *true*. Firing the transition performs the corrective action described by AC.

In the third pattern (Figure 9.4), corresponding detectors and correctors are aggregated by two different classes. For example, in the Anti-Lock Braking System (ABS) [53, 86] project, a centralized FaultHandler activates a redundant brake sensor when a Watchdog detects that the primary brake sensor has failed. There are associations between the detector and its corresponding corrector as illustrated in Figures 9.4(a) and 9.4(b). These diagrams also show the PredicateDetector PD aggregated by object Y, while object Z aggregates its corresponding ActionCorrector AC. In this pattern, the detecting object (*e.g.*, Y) must somehow inform the correcting object (*e.g.*, Z) that a specific error condition has been detected in order to take the appropriate corrective action. In the state diagrams for Y and Z in Figure 9.4(c) that correspond to the object diagram in Figure 9.4(b), this information is imparted via a parameterized signal *message* that Y sends to Z when the condition detected by PD (via a guard on a transition in the state diagram of Y) is *true*. The class of object Z must have declared the parameterized signal *message*, and the parameter `error`, in the class diagram in order for Z to use *message* as an event on a transition in its



(a) Class diagram

(b) Object diagram

(c) State diagram

Figure 9.3: Pattern 2 for modeling detectors and correctors

state diagram. If Z aggregates more than one ActionCorrector, then there must be more than one transition handling the signal *message*, with each guard testing the parameter value against known error codes to determine the appropriate transition to fire, and thus the corrective action sequence to perform.

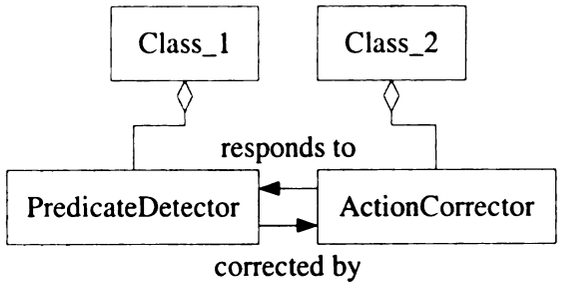
9.2 Object Analysis Patterns and Constraints

Konrad and Cheng have identified several patterns to describe requirements for the main elements of an embedded system [1, 2, 3, 4]. Figure 9.5 gives a list of the object analysis patterns[4] that have been identified to date with a brief description of each. The complete set of object analysis patterns and their full descriptions are given elsewhere [1]. In order to address the needs of requirements engineering, they developed a template to describe these object analysis patterns [1, 2, 3, 4] by modifying the original design pattern template [83]. Such modifications include extending the original design pattern template with a **Constraints** field [53] that contains specification-pattern-based [43] representations of properties of interest.

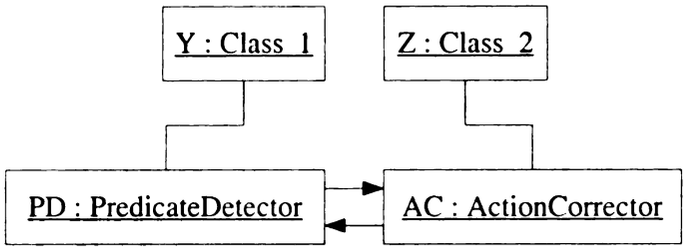
Object analysis patterns can provide both guidance to novices of embedded systems development for determining the key elements of many embedded systems, and examples of how to model these elements with a commonly accepted diagramming notation, UML. Additionally, object analysis pattern *constraints* provide a template for instantiating properties specific to a modeled system in terms of the UML diagrams describing the system. Thus far, the constraints have included representations of two of Dwyer *et al.*'s [43] most commonly used general specification pattern categories, *universality/absence* (to capture invariant properties) and *response* (to capture cause/effect relationships in system behavior).

Preliminary feedback from industrial collaborators indicates that object analysis patterns can be an effective mechanism for describing requirements of embedded

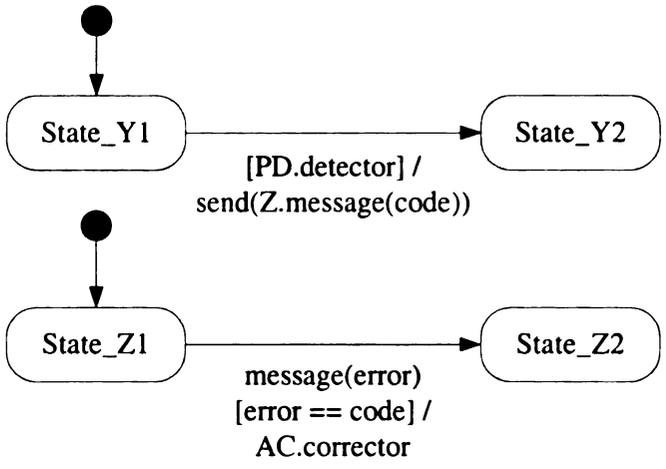
[4] *Object analysis patterns* were formerly termed *requirements patterns*.



(a) Class diagram



(b) Object diagram



(c) State diagrams

Figure 9.4: Pattern 3 for modeling detectors and correctors

- ***Actuator-Sensor*:** The *Actuator-Sensor* pattern specifies basic types of sensors and actuators in an embedded system. In addition, the interaction with the environment via sensors and actuators is one of the main responsibilities of an embedded system. Therefore, the pattern also describes how relationships between actuators and sensors and other components in the system can be captured.
- ***Communication Link*:** Due to the growth in demand for distributed real-time embedded systems, communication capabilities are becoming more important. The *Communication Link* pattern describes how to capture high-level information about communication capabilities offered by an embedded system, such as sending periodic "heartbeat" messages to other systems.
- ***Computing Component*:** Embedded systems have to offer various operational modes as they often function in an environment where a shutdown of the system would lead to a significant loss. In this pattern, various operational modes of an embedded system are specified, such as fail-safe modes that a system enters in response to occurring faults.
- ***Controller Decompose*:** This pattern describes how to decompose an embedded system into different components according to their responsibilities. It is the foundation upon which all other patterns are based. It introduces a high-level view on an embedded system and refers to other object analysis patterns for refinement.
- ***Fault Handler*:** Fault handing is crucial for embedded systems. In this pattern, a fault handler for an embedded system is specified. The fault handler collects fault messages from various sources and initiates corresponding recovery actions.
- ***User Interface*:** User interaction is an important aspect of an embedded system. The system interacts with the user via so-called controls and indicators. Differing from sensors and actuators, interaction via controls and indicators is usually not as tightly constrained (*e.g.*, timing constraints are less strict). The *User Interface* pattern describes how to specify an object model for a user interface that is extensible and reusable.
- ***Watchdog*:** Embedded systems commonly have tight timing and operational constraints. Monitoring these constraints and ensuring that they are not violated is the responsibility of a so-called watchdog. The watchdog monitors a device or system conditions and initiates corrective action(s) if a violation is found.

Figure 9.5: Current list of object analysis patterns for embedded systems [1]

systems. Furthermore, by adding specification-pattern-based [43] representations of properties to the **Constraints** field of the object analysis pattern template [53], developers have some guidance as to what kinds of properties can be checked for a given system when a particular pattern is applied. The specification patterns [43] used in conjunction with object analysis patterns [1, 2, 3, 4] enable even novice developers to easily formulate claims to check the system for specific constraints. Then using McUmber's formalization work [31, 33] and our model development and analysis process (Chapter 4), developers have a mechanism to rigorously check the requirements using simulation and model checking techniques [53, 41].

9.2.1 Object-Analysis-Pattern-Driven Modeling and Analysis

Figure 9.6 illustrates how object analysis patterns [1, 2, 3, 4] and constraints [53] can be used to drive the iterative model development and analysis process described in Chapter 4. The user begins by selecting appropriate object analysis patterns based on the requirements of the system. Using the structural and behavioral diagrams in the object analysis patterns as a guide, the user constructs UML class and state diagrams in MINERVA's graphical editors (Figure 9.6, part A). Hydra performs consistency checks (Figure 9.6, part B), and MINERVA visualizes structural consistency-checking results (dash-dotted arc in Figure 9.6, part F). (See Chapters 5 and 6 for details of these capabilities.) Hydra then generates formal models from textual representations of UML diagrams (Figure 9.6, part C); these formal models can be used to validate the UML diagrams through simulation using Spin (Figure 9.6, part D). Furthermore, constraints from the object analysis patterns can guide novices in constructing formal properties to check against their UML models. The user may instantiate (as LTL claims) properties from the **Constraints** field of those object analysis patterns used to guide the modeling of the system (Figure 9.6, part E). These LTL claims, defined

in terms of attributes, signals, and states of the UML model, can then be checked against the UML diagrams (Figure 9.6, part D) through model checking using Spin. Finally, MINERVA visualizes behavior simulation and counterexample traces (solid arc, Figure 9.6, part F) via state diagram animation, generation/animation of collaboration diagrams, and generation of sequence diagrams (Chapter 6), thus facilitating the debugging and refinement of the original UML diagrams. A complete example applying this process to an industrial case study can be found in [41] (where object analysis patterns were termed *requirements patterns*).

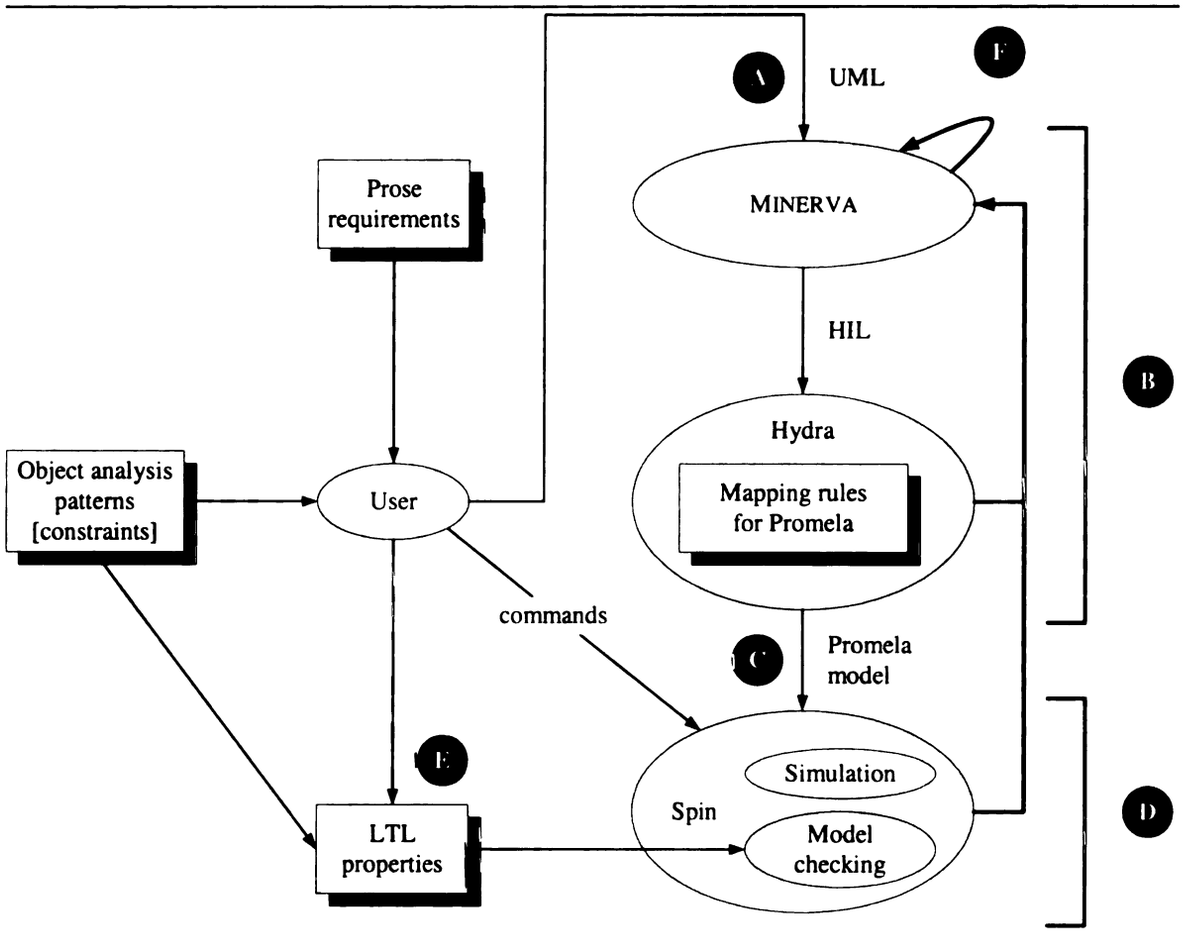


Figure 9.6: Using object analysis patterns and constraints to drive the model development and analysis process

9.3 *Fault Handler* Requirements Pattern

Because this chapter focuses on patterns for handling fault-tolerance requirements, we next overview a particular requirements pattern, the *Fault Handler* requirements pattern [2, 3], that describes how to incorporate centralized fault handling into an embedded system (the complete *Fault Handler* requirements pattern is included in Appendix I). The pattern contains the UML class diagram template shown in Figure 9.7 that suggests a structure or architecture for an embedded system with a centralized FaultHandler. A Watchdog, Examiner, or Monitor monitors a given Device and reports errors to the FaultHandler (the Device itself may also report errors to the FaultHandler). Based on the particular error message received, the FaultHandler performs an appropriate recovery action, such as sending a message to the ComputingComponent (to initiate transitioning to a designated "safe" state, described in the next paragraph), the UserInterface (to give warning feedback to the user), or a FailSafeDevice (to activate a redundant Device, such as a backup sensor, in case of hardware failure).

The original *Fault Handler* requirements pattern as developed by Konrad and Cheng [2, 3, 4] also contains a UML state diagram template for the ComputingComponent of an embedded system, as shown in Figure 9.8. The diagram template includes many different UML states, including either acceptable recovery states or "safe" states for the system (not all recovery states will be required by every system). Generally, recovery states (the states other than `PowerOff` and `Hold` shown as rounded rectangles in Figure 9.8) are intermediate states that eventually lead to either a *system shutdown* (*i.e.*, the ComputingComponent transitions to state `PowerOff`) or a *system reset* (*i.e.*, the ComputingComponent transitions to state `Initialize`). These intermediate states are described in more detail in the *Fault Handler* requirements pattern, included in Appendix I. The state `PowerOff` is considered "safe" because the system is not executing and therefore cannot perform harmful actions. For our pur-

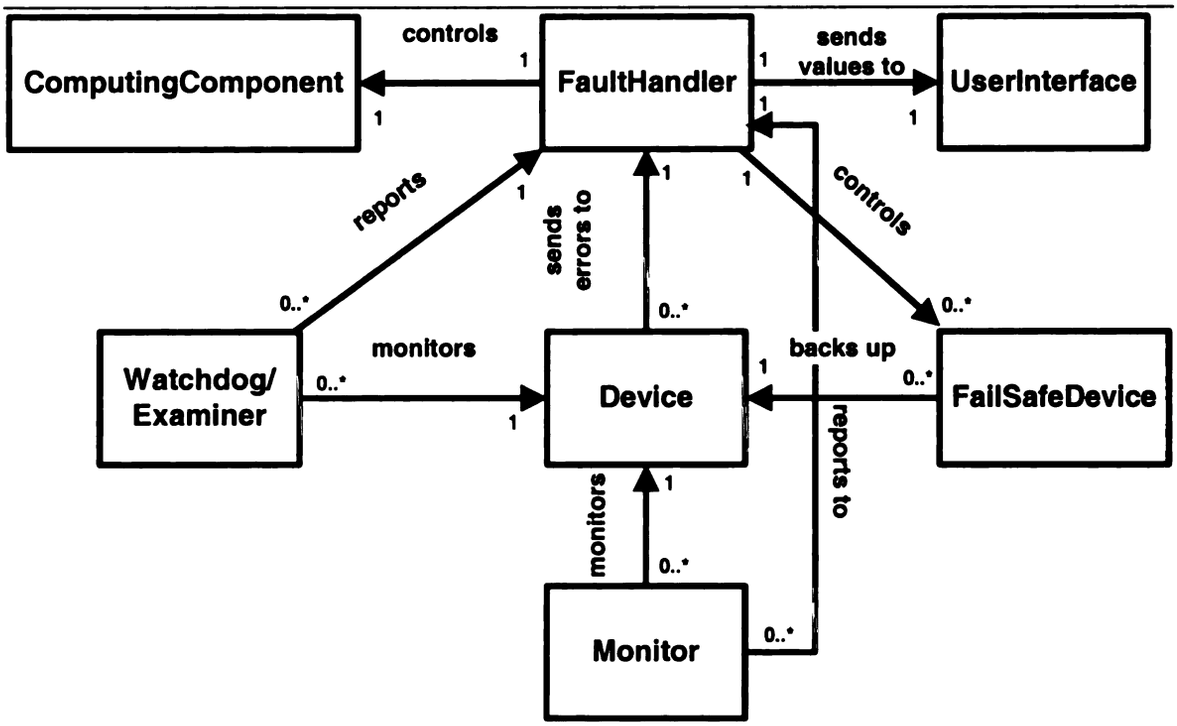


Figure 9.7: UML class diagram template from the original *Fault Handler* requirements pattern incorporating a FaultHandler into an embedded system

poses, the state `Initialize` is considered "safe" because (1) the system is believed to start in an acceptable or legal configuration (so that a system reset can successfully re-start the system in an acceptable state), and (2) during initialization, embedded systems generally perform built-in tests (BITs) or power-on self-tests (POSTs) and immediately shut the system down if a problem is encountered.

9.3.1 UML Modeling and Semantics for the FaultHandler

While the original *Fault Handler* requirements pattern [2, 3, 4] includes a state diagram template for the Controller, it does not include a state diagram template for the FaultHandler. However, several of our case studies involving fault-handling (*e.g.*, an Anti-Lock Braking System [53], a Diesel Filter System [41]) have modeled the behavior of the FaultHandler in a way similar to that shown in Figure 9.9, that has now been codified as part of Detector-Corrector Pattern 3 (each detector is aggregated by a Watchdog, Examiner, Monitor, or Device as shown in Figure 9.10, a revised UML class diagram template also utilizing Detector-Corrector Pattern 3). The FaultHandler begins in state `Monitoring`. The reception of error messages is modeled using the signal *StoreError(Error)*, where the FaultHandler attribute `Error` stores a numeric code indicating the specific error received (each modeled error for a given system uses its own unique code, often indicated in the requirements document for the system). When the FaultHandler receives an error message (due to a Watchdog, Examiner, Monitor, or Device detecting an error), it self-transitions to state `Monitoring` based on the error received. To ensure coverage for all modeled errors, there must be as many self-transitions for `Monitoring` as there are possible modeled errors. In Figure 9.9, these transitions are indicated by n transitions annotated with `[Error == E_i] / ActionCorrector_i`, $1 \leq i \leq n$, where n is the number of possible modeled errors. The particular error code received (stored in the attribute `Error`) determines, via the guards, which one of these n self-transitions for state `Monitoring` is taken, thereby

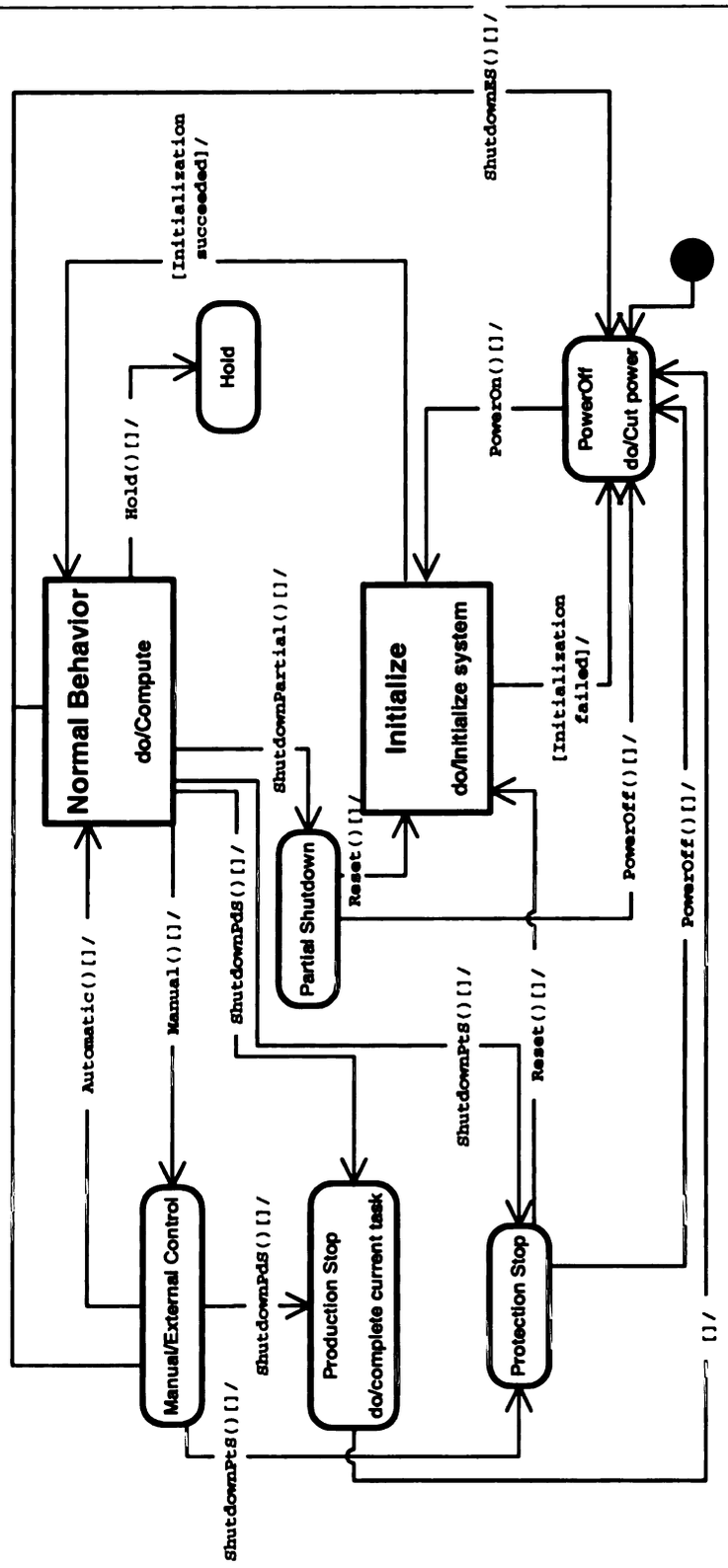


Figure 9.8: UML state diagram template for the ComputingComponent from the original *Fault Handler* requirements pattern, including "safe" states [2, 3, 4]

determining which corresponding recovery action is performed.

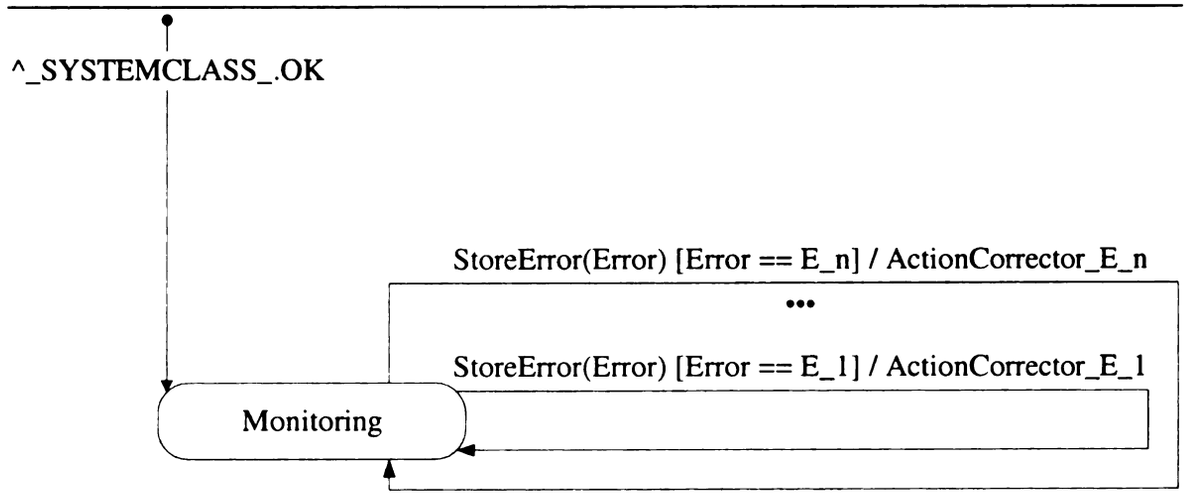


Figure 9.9: Sample UML state diagram template for a centralized **FaultHandler** (not included in the original *Fault Handler* requirements pattern) utilizing Detector-Corrector Pattern 3

The sample UML diagram templates for the **FaultHandler** shown in Figures 9.9 and 9.10 have the following characteristics. First, the **FaultHandler** receives error messages indicating particular problems from non-**FaultHandler** sources (*e.g.*, a **Watchdog**, **Examiner**, **Monitor**, or **Device**). Therefore, the **FaultHandler** itself is not involved in the actual detection of errors, but only the response to errors. Second, a **Watchdog**, **Examiner**, or **Monitor** typically monitors a single **Device** for problems. Thus, the errors detected and reported to the **FaultHandler** are localized to a particular **Device** or part of the system, rather than indicative of global error conditions (*i.e.*, error conditions involving more than one component). Third, the original requirements patterns, including the *Fault Handler* requirements pattern, were developed with the idea of leveraging McUmber's UML-to-Promela formalization [31, 33] in mind. The **FaultHandler** state diagram would therefore be translated to a Promela process and execute in the same manner as all other state diagrams in the UML model (Figure 9.11). Spin interleaves all processes and has only weak fairness, so the Promela

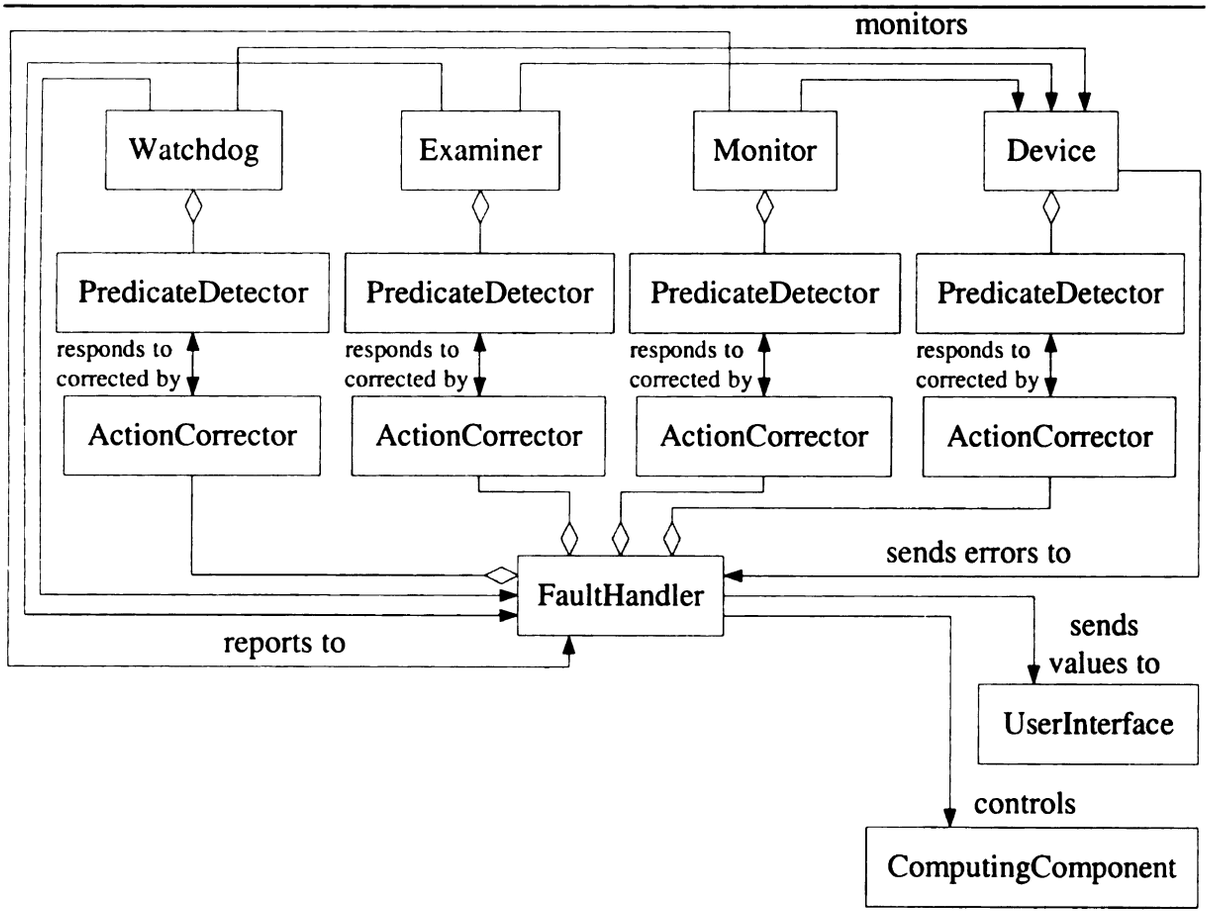


Figure 9.10: Refined UML class diagram template for FaultHandler utilizing Detector-Corrector Pattern 3

process corresponding to the behavior (*i.e.*, UML state diagram) of the FaultHandler might not execute immediately after an error is detected and reported.

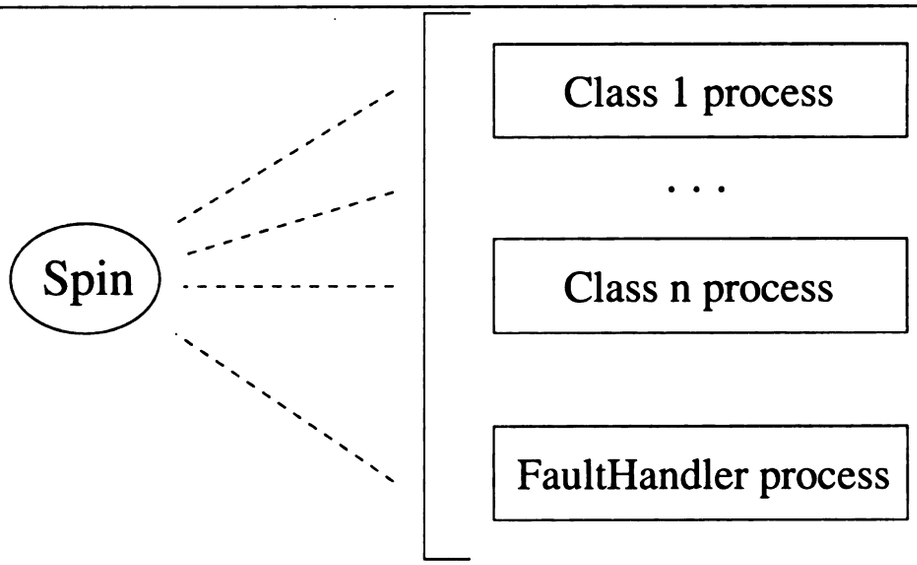


Figure 9.11: Spin non-deterministically interleaves all processes (dashed lines), each representing the behavior of a UML class.

9.4 Refined UML Modeling, Semantics, and Formalization for the FaultHandler

We refine modeling and semantics for the FaultHandler in two ways. First, we now allow the FaultHandler itself to perform *global* error detection, rather than only responding to error messages received from external components due to the *local* error detection of those components. Global error detection enables detection of system conditions where, taken individually, local conditions do not indicate a problem, but when examined together (globally), they do indicate a problem. We enable the FaultHandler to perform global error detection by allowing it access to all system variables. Because we allow the FaultHandler access to all system variables (*i.e.*, all attributes in a UML model), global error conditions (*e.g.*, the accelerator pedal is depressed to

a level indicating high demand, but the battery charge, while adequate for normal performance, is below the lowest acceptable level to provide power assist for a high-demand situation) can now be modeled by detectors aggregated by the FaultHandler. Second, because fault handling is a critical task, we may opt to increase its scheduling priority. The original semantics assumed by the *Fault Handler* requirements pattern executes all objects, including the FaultHandler, with the same priority, so that there may be a significant delay after an error has been detected before the FaultHandler performs a corresponding corrective action. Scheduling the FaultHandler to execute more often shortens the delay before corrective action is performed; however, doing so may incur a significant penalty in terms of increasing the size of the formal model's state space.

Global error detection. The refined UML class diagram template for the FaultHandler utilizing Detector-Corrector Patterns 2 and 3 is shown in Figure 9.12. The FaultHandler itself now aggregates one or more *pairs* of PredicateDetector and ActionCorrector objects (*i.e.*, utilizes Detector-Corrector Pattern 2). Thus, the FaultHandler itself can now perform error detection. The refined UML state diagram template for the FaultHandler utilizing Detector-Corrector Patterns 2 and 3 is shown in Figure 9.13. Similar to Figure 9.9 (page 179), the FaultHandler begins in state `Monitoring`. Using Detector-Corrector Pattern 3, we model the reception of error messages using the signal *StoreError(Error)*, where the FaultHandler attribute `Error` stores a numeric code indicating the specific error received. When the FaultHandler receives an error message (due to a Watchdog, Examiner, Monitor, or Device detecting an error with a *local* detector, *i.e.*, a detector examining only those attributes belonging to that component), it self-transitions to state `Monitoring` based on the error code received. In contrast to Figure 9.9, Figure 9.13 now also includes *global* detectors (detectors aggregated by the FaultHandler itself that may examine attributes from more than

one non-FaultHandler component) with corresponding correctors (Detector-Corrector Pattern 2). To enable the FaultHandler to perform *global* error detection, we revise our modeling and formalization frameworks to allow any Detector aggregated by the FaultHandler access to all system variables. In an actual embedded system, access to all system variables can be achieved through polling or shared memory. The existing UML-to-Promela formalization [31, 33] already declares all variables globally, so we assume shared memory in our model.

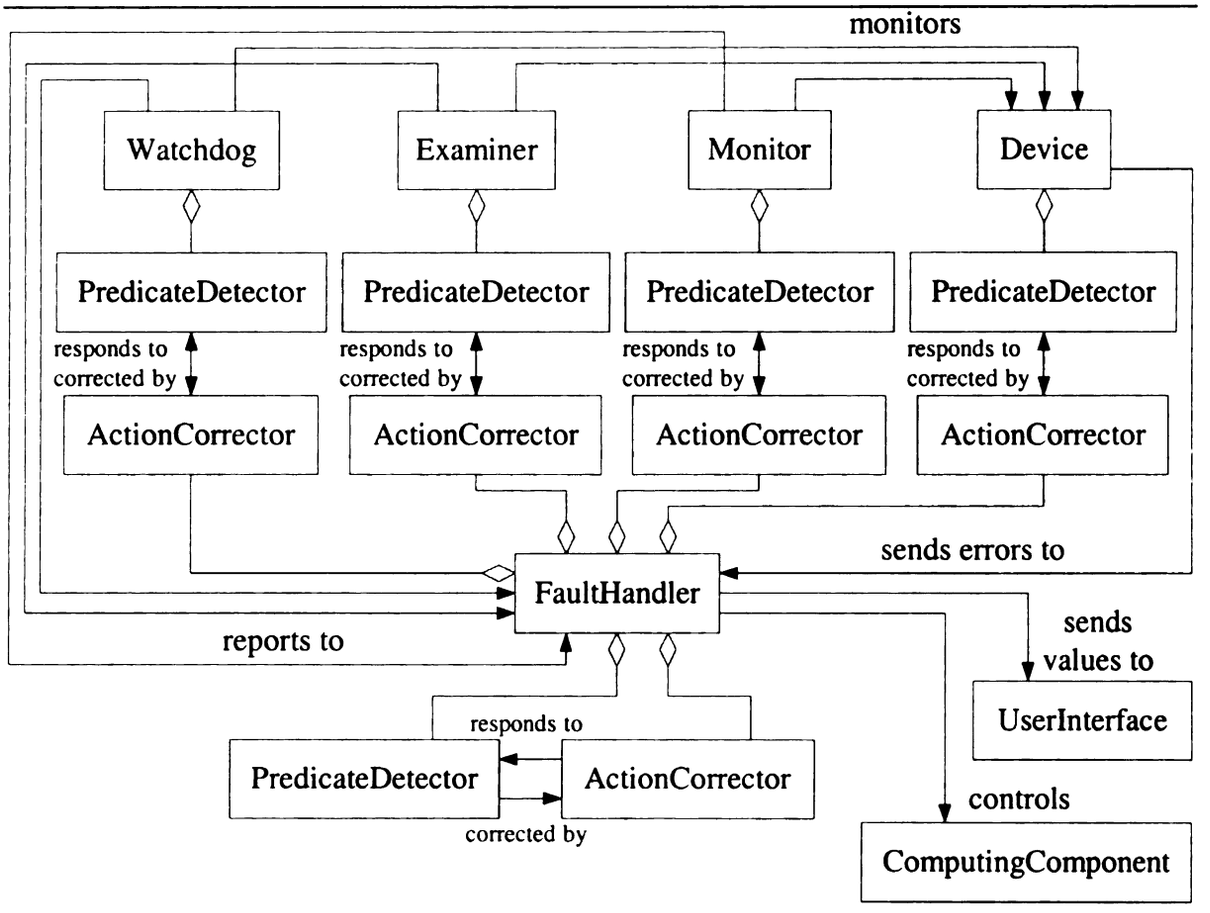


Figure 9.12: Refined UML class diagram template for FaultHandler utilizing Detector-Corrector Patterns 2 and 3

Increased scheduling priority. As a further extension to modeling fault handling, we may adjust the underlying formalization semantics to increase the Fault-

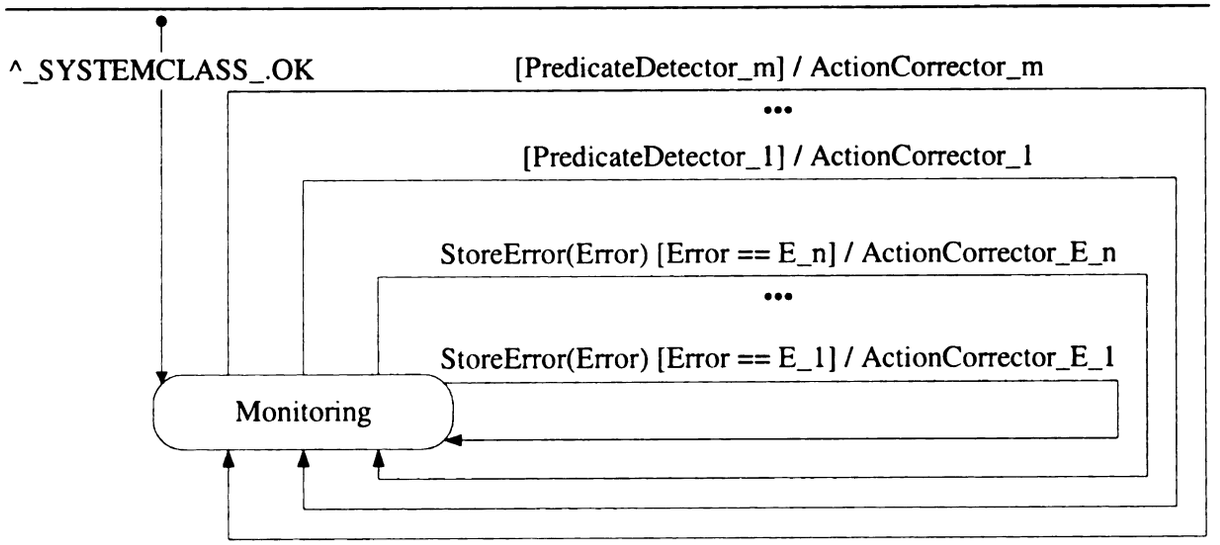


Figure 9.13: Refined UML state diagram template for FaultHandler utilizing Detector-Corrector Patterns 2 and 3

Handler's scheduling priority. This approach may increase the size of the formal model's state space, so additional abstractions may be needed. Spin has no built-in way to manage scheduling of processes, so we explicitly force the FaultHandler's behavior to be interleaved with the behavior of the rest of the system via a boolean flag (Figure 9.14). We implement an increase in the scheduling priority of the FaultHandler in Promela via a global boolean flag called `global_UMLStep` in the following manner. When `global_UMLStep` is true (*i.e.*, equal to one), the rest of the system excluding the FaultHandler may take a step (take a UML transition). We model this constraint by adding, as a conjunct, the boolean condition `(global_UMLStep == 1)` to all guards on non-initial transitions in all state diagrams in the UML model except for the _SYSTEMCLASS_ (a modeling construct responsible for instantiating objects in the system), Environment (a modeling construct responsible for providing external stimulus to the system), and FaultHandler. Similarly, we add the entry action `global_UMLStep := 0` to all states in all state diagrams in the UML model except for the _SYSTEMCLASS_, Environment, and FaultHandler. For convenience to the devel-

oper, these additions to the guards and entry actions can be performed automatically (and transparently) by either MINERVA (during generation of the intermediate textual representation), as shown in Figures 9.15 and 9.16, or by Hydra (during translation to Promela code, not shown).

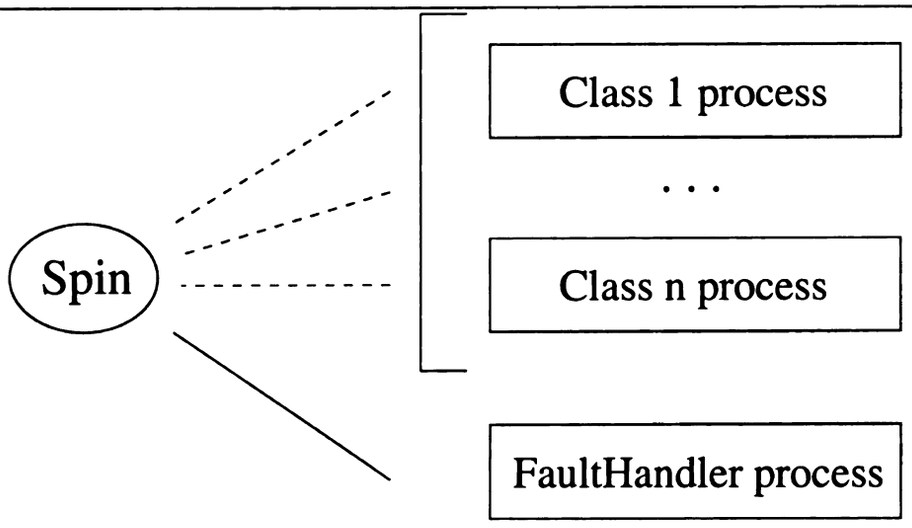


Figure 9.14: Forcing Spin to interleave the behavior of the FaultHandler with the behavior of the rest of the system. Spin alternates between allowing the FaultHandler to take a step (solid line) and allowing one of many Promela processes, each representing the behavior of a UML class, to take a step (dashed lines).

When `global_UMLStep` is false (*i.e.*, equal to zero), the FaultHandler may take a step (*i.e.*, a UML transition). We model this constraint by adding, as a conjunct, the boolean condition `(global_UMLStep == 0)` to all guards on non-initial transitions in the FaultHandler state diagram. We also add the action `global_UMLStep := 1` to all non-initial transitions in the FaultHandler state diagram. For convenience to the developer, these additions to the guards and actions can be performed automatically (and transparently) by either MINERVA (during generation of the intermediate textual representation), as shown in Figures 9.17 and 9.18, or by Hydra (during translation to Promela code, not shown).

FaultHandler **Scheduling Priority Rule 1:**

Description: Adds the conjunct `(global_UMLStep == 1)` to guards on all non-initial transitions in all state diagrams in a UML model except for the state diagrams of the _SYSTEMCLASS_, Environment, and FaultHandler classes.

Purpose: Ensures that non-FaultHandler, non-modeling-construct objects only execute when the FaultHandler is not executing.

Note: This addition can be performed transparently by MINERVA during generation of the intermediate textual representation as follows:

(1) **let** u be a UML model
(2) **foreach** class $c \in u$
(3) **if** $c \notin$ {_SYSTEMCLASS_, Environment, FaultHandler}
(4) **let** sd be the state diagram of c
(5) **foreach** non-initial transition $t \in sd$
(6) **let** g be the guard for t
(7) **if** g is empty **then** $g \leftarrow$ `global_UMLStep == 1`
(8) **else** $g \leftarrow$ `(` g `)` $\wedge$ `(global_UMLStep == 1))`

Figure 9.15: Rule for increasing the scheduling priority of the FaultHandler: non-FaultHandler components may only take a step when `(global_UMLStep == 1)`

FaultHandler **Scheduling Priority Rule 2:**

Description: Adds entry action `global_UMLStep := 0` to all states in all state diagrams in a UML model except for the state diagrams of the _SYSTEMCLASS_, Environment, and FaultHandler classes.

Purpose: Ensures that after every step performed by non-FaultHandler, non-modeling-construct objects, the FaultHandler is enabled to execute (and non-FaultHandler, non-modeling-construct objects are disabled).

Note: This addition can be performed transparently by MINERVA during generation of the intermediate textual representation as follows:

```
(1) let u be a UML model
(2) foreach class c ∈ u
(3)    if c ∉ {_SYSTEMCLASS_, Environment, FaultHandler}
(4)       let sd be the state diagram of c
(5)       foreach state s ∈ sd
(6)          let eas be the entry action sequence for s
(7)          if eas is empty then eas ← global_UMLStep := 0;
(8)                          else eas ← eas ; global_UMLStep := 0
```

Figure 9.16: Rule for increasing the scheduling priority of the FaultHandler: after every UML step, enable the FaultHandler to take a step

FaultHandler Scheduling **Priority Rule 3:**

Description: Adds conjunct `(global_UMLStep == 0)` to guards on all non-initial *transitions* in the **FaultHandler** state diagram.

Purpose: Ensures that the **FaultHandler** only executes when non-**FaultHandler**, non-*modeling*-construct objects are not executing.

Note: *This* addition can be performed transparently by MINERVA during generation *of the intermediate* textual representation as follows:

```
(1) let u be a UML model
(2) foreach class c ∈ u
(3)   if c ∈ {FaultHandler}
(4)     let sd be the state diagram of c
(5)     foreach non-initial transition t ∈ sd
(6)       let g be the guard for t
(7)       if g is empty then g ← global_UMLStep == 0
(8)                      else g ← (( g ) ∧ (global_UMLStep == 0))
```

Figure 9.17: Rule for increasing the scheduling priority of the **FaultHandler**: the *FaultHandler may only* take a step when `(global_UMLStep == 0)`

FaultHandler **Scheduling Priority Rule 4:**

Description: Adds action `global_UMLStep := 1` to all non-initial transitions in the FaultHandler state diagram.

Purpose: Ensures that after every step performed by the **FaultHandler**, non-FaultHandler, non-modeling-construct objects are enabled to execute (and the **FaultHandler** is disabled).

Note: This addition can be performed transparently by MINERVA during generation of the intermediate textual representation as follows:

```
(1) let u be a UML model
(2) foreach class c ∈ u
(3)    if c ∈ {FaultHandler}
(4)       let sd be the state diagram of c
(5)       foreach non-initial transition t ∈ sd
(6)          let as be the action sequence for t
(7)          if as is empty then as ← global_UMLStep := 1
(8)                          else as ← as ; global_UMLStep := 1
```

Figure 9.18: Rule for increasing the scheduling priority of the **FaultHandler**: after *every* ***FaultHandler*** step, enable the rest of the model to take a step

9.5 Adaptive Cruise Control Example

In this section, we apply our approach for modeling and analysis of fault handling to the Adaptive Cruise Control model from Chapter 7. Recall from Chapter 7, Property 6 (page 125), that if collision is imminent, yet the Adaptive Cruise Control-equipped car is still outside the safety zone, then the system will not begin disengaging until the Adaptive Cruise Control-equipped car has violated the safety zone. (Definitions of zones used by the Adaptive Cruise Control algorithm can be found in Appendix D.) This stipulation allows the driver time to change lanes without slowing down the car. We now examine the following related requirement:

***Requirement:* If collision is imminent, yet the Adaptive Cruise Control-equipped car is still outside the safety zone, then an alarm sounds to warn the driver that action is required on the driver's part (*e.g.*, applying brakes, changing lanes) to avoid a collision.**

In the original UML model for Adaptive Cruise Control developed in Chapter 7, the concept of an alarm sounding was modeled by the Control periodically entering a state called `sendwarn`, as shown in the elided Control state diagram in Figure 9.19. The disadvantage of modeling any occurrence (such as an alarm sounding) as periodic *entry* to a particular non-final state s rather than as a particular boolean flag f becoming (and remaining) true is that any temporal logic claim involving $\Diamond(\text{in}(s))$ (*eventually* the model is in state s) becomes violated as soon as the model leaves state s. In the case of the Adaptive Cruise Control model from Chapter 7, the claim

```
( []( ( (Control.x1 < Control.xhit) & (Control.z1 <= Control.x1) ) ->          (8)
    <>( ( in(Control.sendwarn) & (Control.z1 <= Control.x1) ) ) ) )
```

becomes violated as soon as the Control leaves the (non-final) state `sendwarn`.

In order to address this shortcoming, we refine the way the concept of an alarm sounding is modeled in two ways. First, we model *the alarm being triggered* as a boolean attribute `alarm` in the Control class that is initially *false* (*i.e.*, zero) and

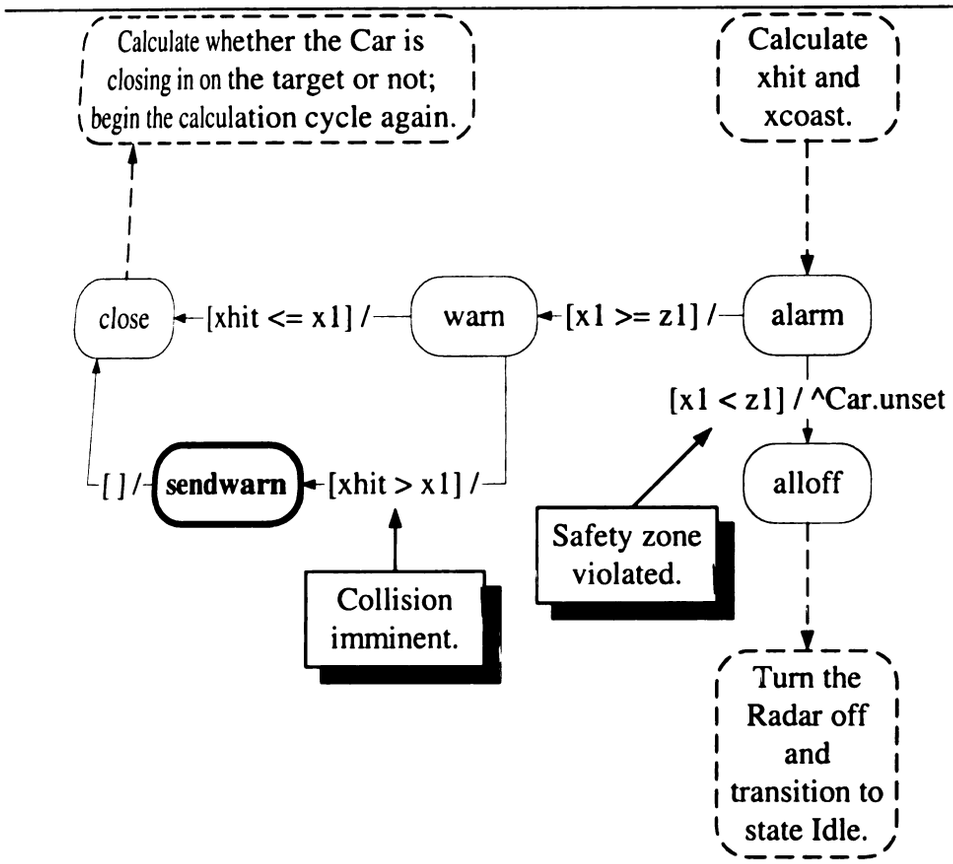


Figure 9.19: Elided Control state diagram. Control periodically enters state **sendwarn** *(shown as a bold rounded rectangle). Portions of the state diagram that have been abstracted for illustrative purposes are shown with dashed states and transitions (dashed rounded rectangles and directed arcs, respectively). Notes indicating critical conditions in the model* are shown as shadowed callout boxes.

becomes *true* the first time the Control state **sendwarn** is entered (*i.e.*, we add an entry action to state **sendwarn** that sets the value of **alarm** to *true* (*i.e.*, one)). However, the alarm being triggered does not necessarily mean that the alarm emits a sound. The Adaptive Cruise Control requirements stipulate that a microphone check to make sure that if the alarm is triggered, then a sound is actually emitted to warn the driver. Therefore, we additionally model *the alarm emitting a sound* as a boolean attribute **sound** in the Control class. The value of this attribute is initially *true*. These additions to the Control class and state diagrams are shown in Figures 9.20 and 9.21, respectively. In Figure 9.20, previously declared attributes have been elided, indicated with ellipses. In Figure 9.21, the state **sendwarn** is indicated in bold, while annotation boxes describe portions of the state diagram that have been elided. The Control enters the state **sendwarn** if the safety zone has not been violated, yet collision is imminent.

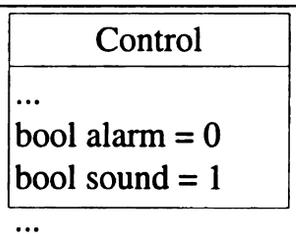


Figure 9.20: Refinements to Control class: Added boolean attributes **alarm** and **sound**.

9.5.1 Fault Handling in Adaptive Cruise Control

We inject the following *fault* into the model: the alarm's speaker fails during execution (*i.e.*, after the cruise-control is in operation, as speaker failure prior to execution would be detected during initialization, not modeled here). As shown in Figure 9.22, we make a non-deterministic self-transition from the Control state **warn** to itself that sets the value of **sound** to false, indicating that the alarm fails to emit a sound (*i.e.*, the alarm's speaker has failed during execution). To handle this fault, we

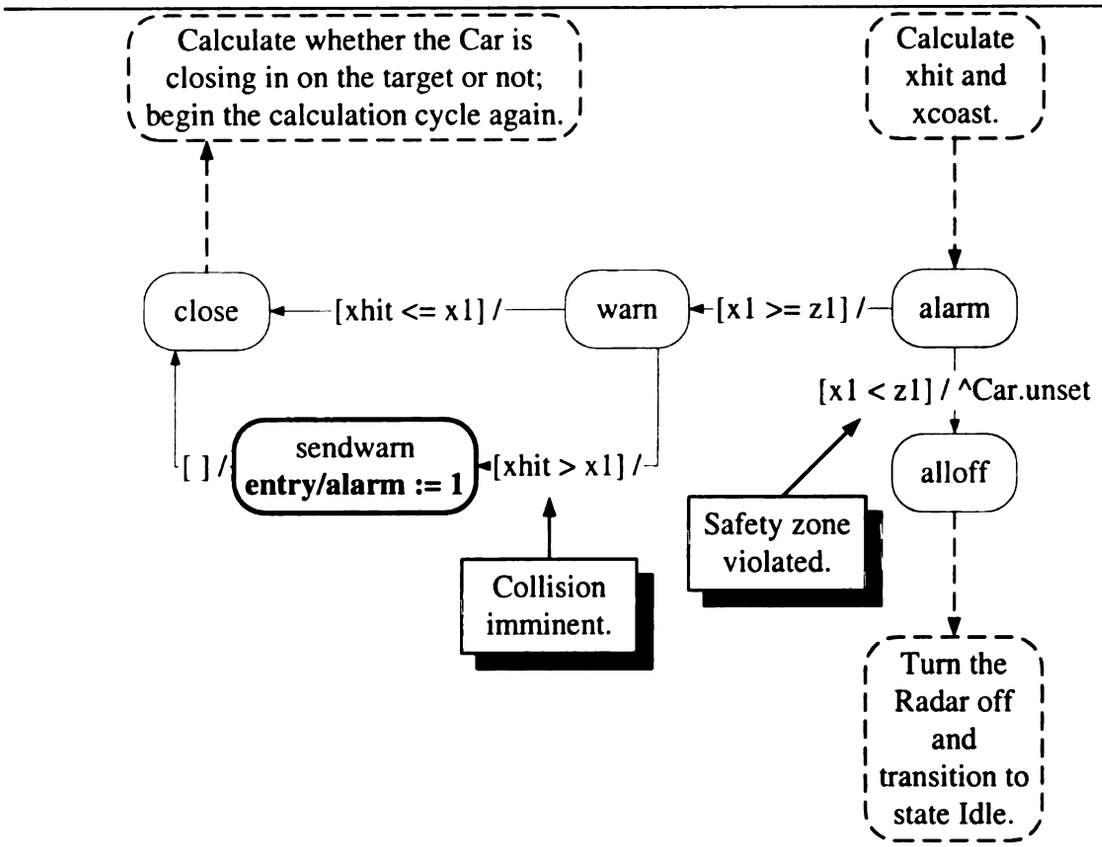


Figure 9.21: Refinements to Control state diagram: Added entry action "`alarm := 1`" to state `sendwarn` (both the action and the state are bold). Portions of the state diagram that have been abstracted for illustrative purposes are shown with dashed states and transitions (dashed rounded rectangles and directed arcs, respectively). Notes indicating critical conditions in the model are shown as shadowed callout boxes.

then introduce a FaultHandler into the model according to the *Fault Handler* object analysis pattern in the following manner. We *detect* the fault by testing the following condition: `(Control_V.alarm & !Control_V.sound)`. If the alarm is triggered, but the alarm does not emit a sound, then the driver is in potential danger and not aware of it. The best course of action is to disengage the cruise control, thus slowing the car by disengaging the throttle (the Adaptive Cruise Control does not have control of the brake). We therefore *correct* the problem with a recovery action that consists of sending a *shutdown* message to the Control. This condition and recovery action are modeled using Detector-Corrector Pattern 2 in the FaultHandler class and state diagrams as shown in Figures 9.23 and 9.24, respectively. We also refine the Control state diagram according to the *Fault Handler* object analysis pattern by encapsulating the previously modeled behavior in a composite state called `Normal_Behavior` that has an outgoing transition to handle the new signal *shutdown* from the FaultHandler, with additional actions to turn off the remaining components in the system. We add the signal *shutdown* to the Control class diagram. These refinements are shown in Figures 9.25 and 9.26, respectively.

From the refined UML diagrams, we generate the formal Promela model (without the optional increased scheduling priority). We then check the property

```
( []( ( Control.alarm & !Control.sound ) -> <>( send(Control.shutdown) ) ) )    (9)
```

meaning that it is always the case that if the alarm is triggered and does not produce a sound, then eventually the system shuts down. The results are shown in Figure 9.27.

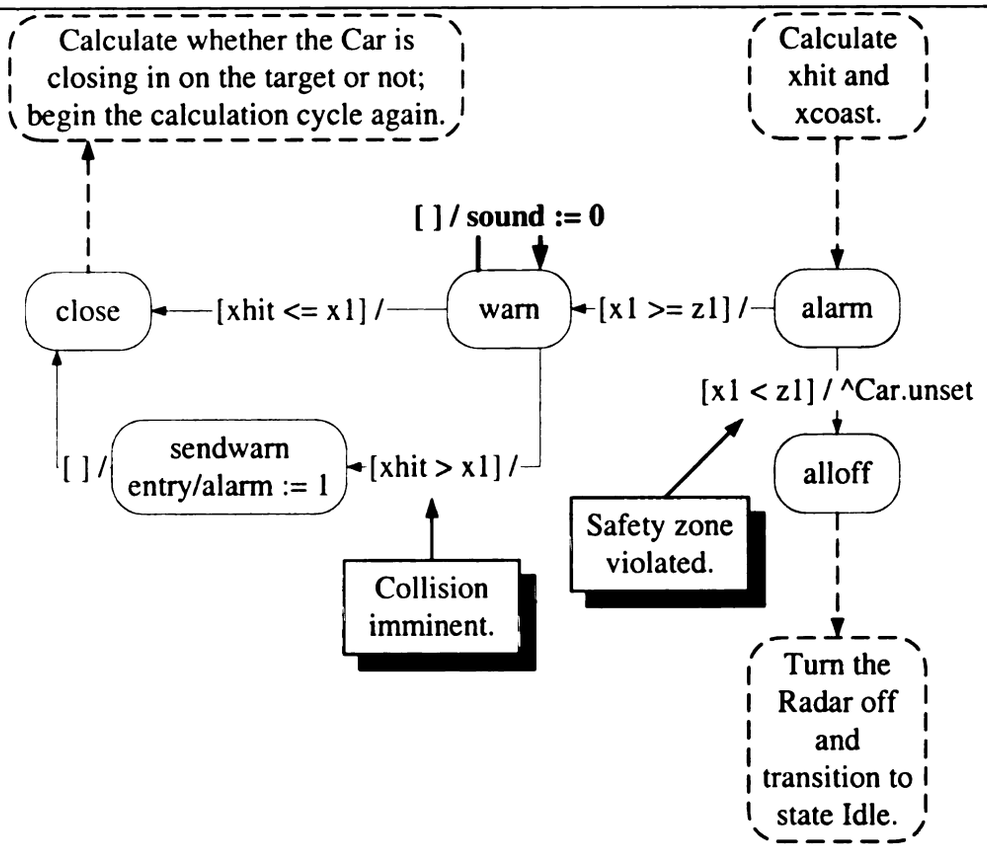


Figure 9.22: Injecting a fault into the Adaptive Cruise Control model: Added non-deterministic self-transition with action "sound := 0" to state warn in the Control state diagram to indicate that the alarm's speaker fails (both the action and the transition are bold). Portions of the state diagram that have been abstracted for illustrative purposes are shown with dashed states and transitions (dashed rounded rectangles and directed arcs, respectively). Notes indicating critical conditions in the model are shown as shadowed callout boxes.

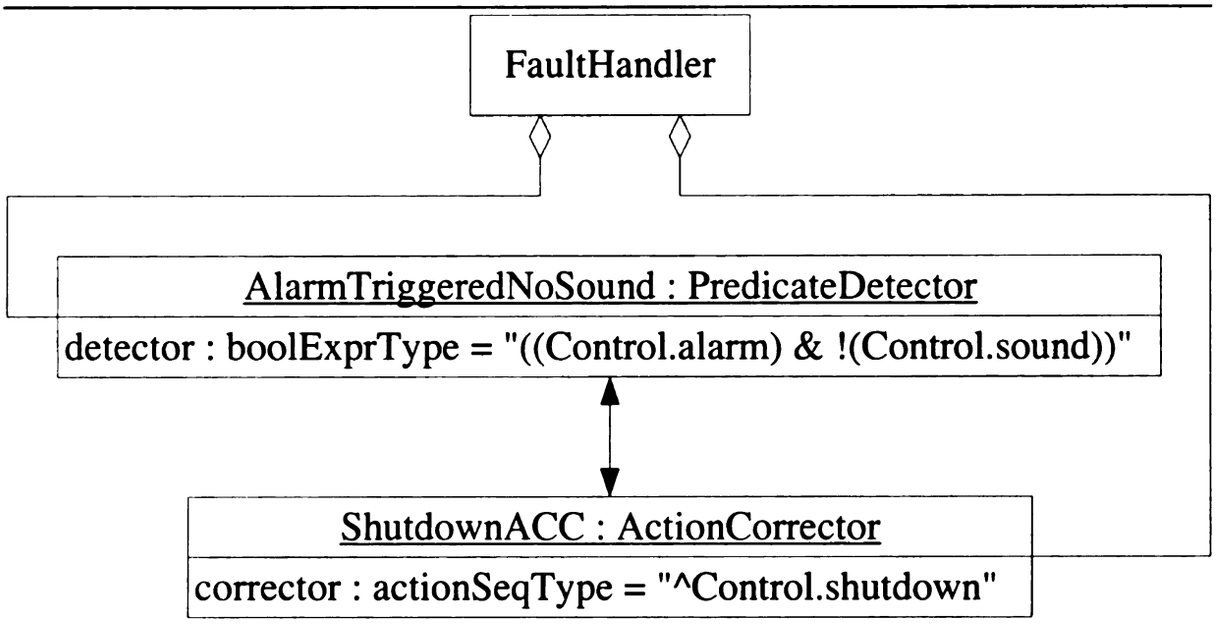


Figure 9.23: Adaptive Cruise Control FaultHandler class utilizing Detector-Corrector Pattern 2

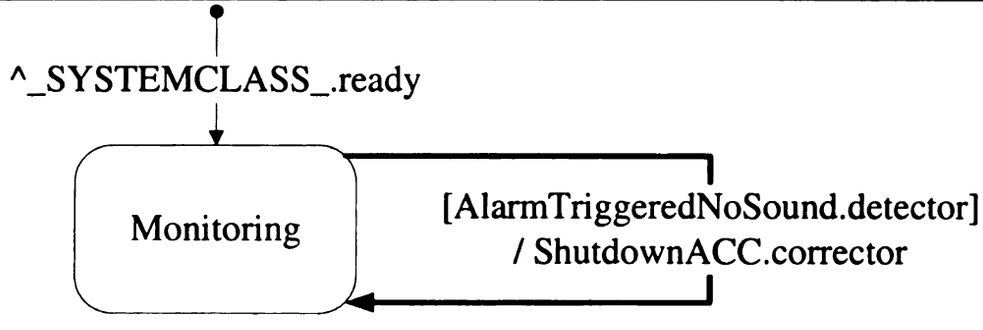


Figure 9.24: Adaptive Cruise Control FaultHandler state diagram utilizing Detector-Corrector Pattern 2

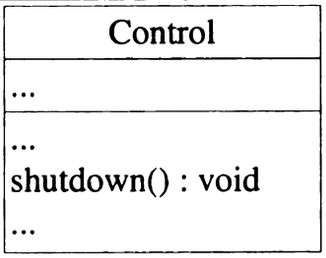


Figure 9.25: Refinements to Control class to handle new signal *shutdown* from FaultHandler

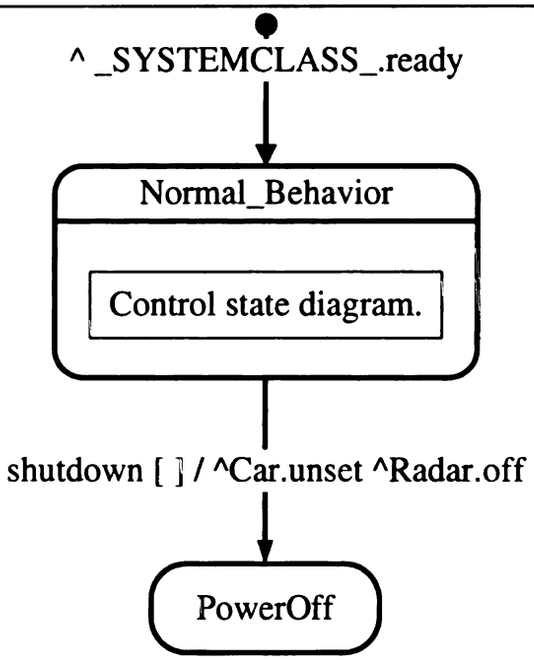


Figure 9.26: Refinements to Control state diagram to handle new signal *shutdown* from FaultHandler

```
(Spin Version 3.3.3 -- 21 July 1999)
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     - (not selected)
        invalid endstates       - (disabled by never-claim)

State-vector 688 byte, depth reached 755, errors: 0
     317 states, stored
      19 states, matched
     336 transitions (= stored+matched)
     209 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.596   memory usage (Mbyte)
```

Figure 9.27: Spin results for analyzing property 9

Chapter 10

Literature Review

Developing specifications in any formal language, including Promela, manually has the potential to be an error-prone task. In an effort to leverage the benefits and address the shortcomings of both formal and semi-formal approaches to requirements analysis and design, many projects [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35] have proposed techniques for associating formal semantics with semi-formal modeling notations, including UML [26, 28, 29, 30, 31, 32, 33, 35]. The work described in this dissertation leverages a particular approach to UML formalization [31, 33] but is not, itself, concerned with formalization approaches or techniques for automated generation of formal specifications. We refer the interested reader to [31] for a detailed discussion of these topics.

While formalization of UML enables rigorous analysis of formal models derived from UML diagrams, formalization itself is not sufficient to broaden the community of (embedded systems) developers who can use formal methods to rigorously analyze requirements. To enable developers to model and analyze requirements in UML without having to know details of formal models requires a framework and process that takes advantage of a UML formalization yet insulates developers from the formal models produced by such a formalization. This research presents an approach to validating

embedded systems requirements modeled as both UML diagrams and temporal logic properties using formal verification (*e.g.*, model checking) techniques. We describe a model development and analysis framework, complementary to a previously developed formalization framework [31, 33], that insulates the developer from formal models and outputs of tools. As shown in Figure 10.1, this framework includes a graphical editor and visualization environment, and an analysis result processor (both shown as bold ovals). As shown in Figure 10.2, we instantiate the framework with tools (*e.g.*, MINERVA, Hydra, and Spin) and describe an overall model development and analysis process that incorporates the following enabling techniques: model construction (Part A), automated generation of formal models (Part B), guidance for instantiation of formal properties (Part C), model analysis (Parts D and E), and visual interpretation of analysis results (Part F, bold arcs to the MINERVA process oval).

This chapter overviews work related to these enabling techniques (with the exception of automated generation of formal models; as noted previously, see [31] for details regarding this topic), as well as environments for embedded systems development, including formalized approaches and the leading commercial UML Computer-Aided Software Engineering (CASE) tools.

10.1 Guidance for Model Construction

Our approach enables formal analysis of UML models of embedded systems requirements. However, such models must first be constructed, and, as discussed in Chapter 3, shown to be well-formed, before they can be formally analyzed. Sections 10.1.1, 10.1.2, and 10.1.3 overview, respectively, how *graphical editing environments*, *stereotypes*, and *patterns* can provide user guidance for constructing well-formed UML-based requirements models for a given application domain, and how these techniques compare with our approach.

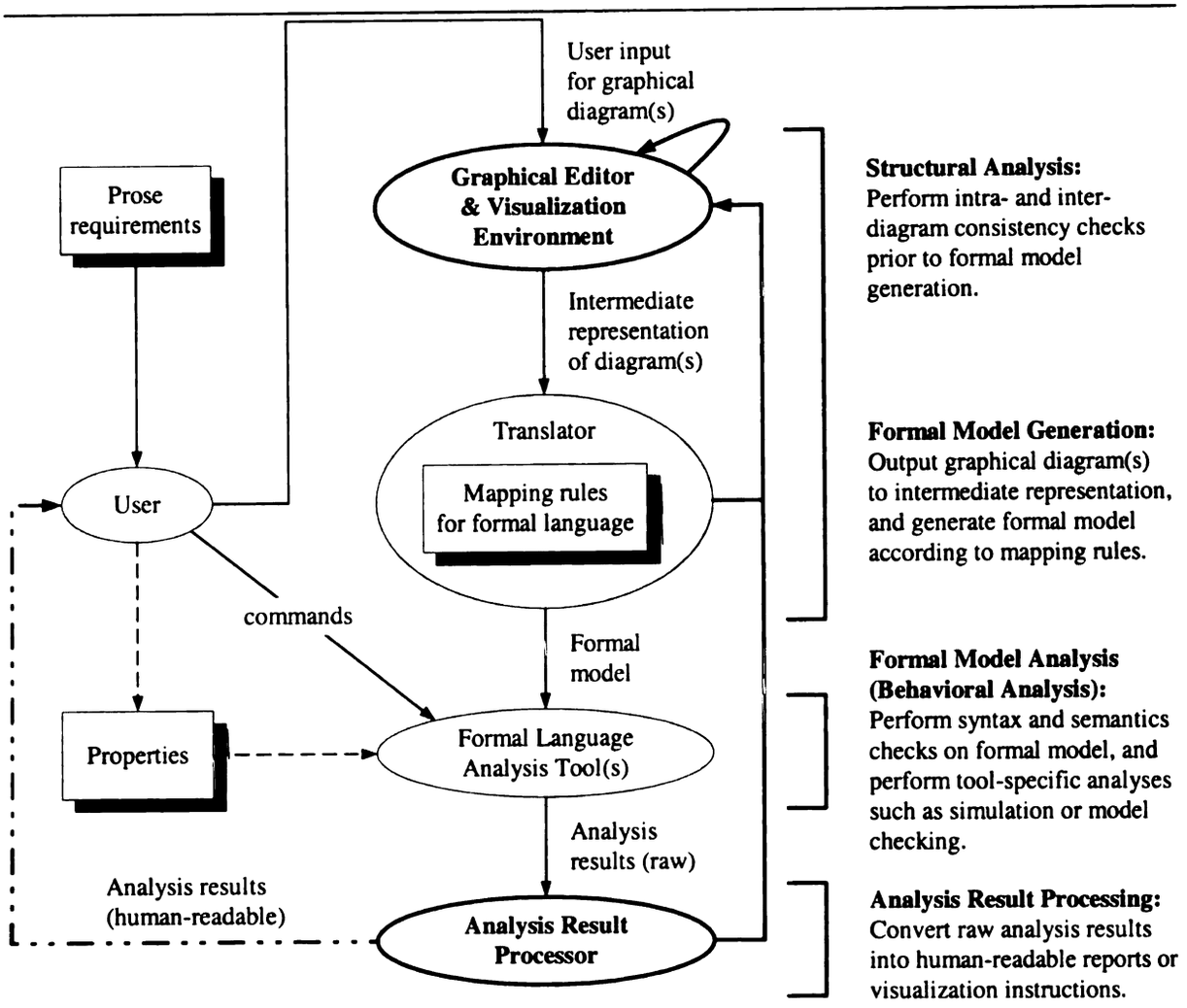


Figure 10.1: Model development and analysis framework

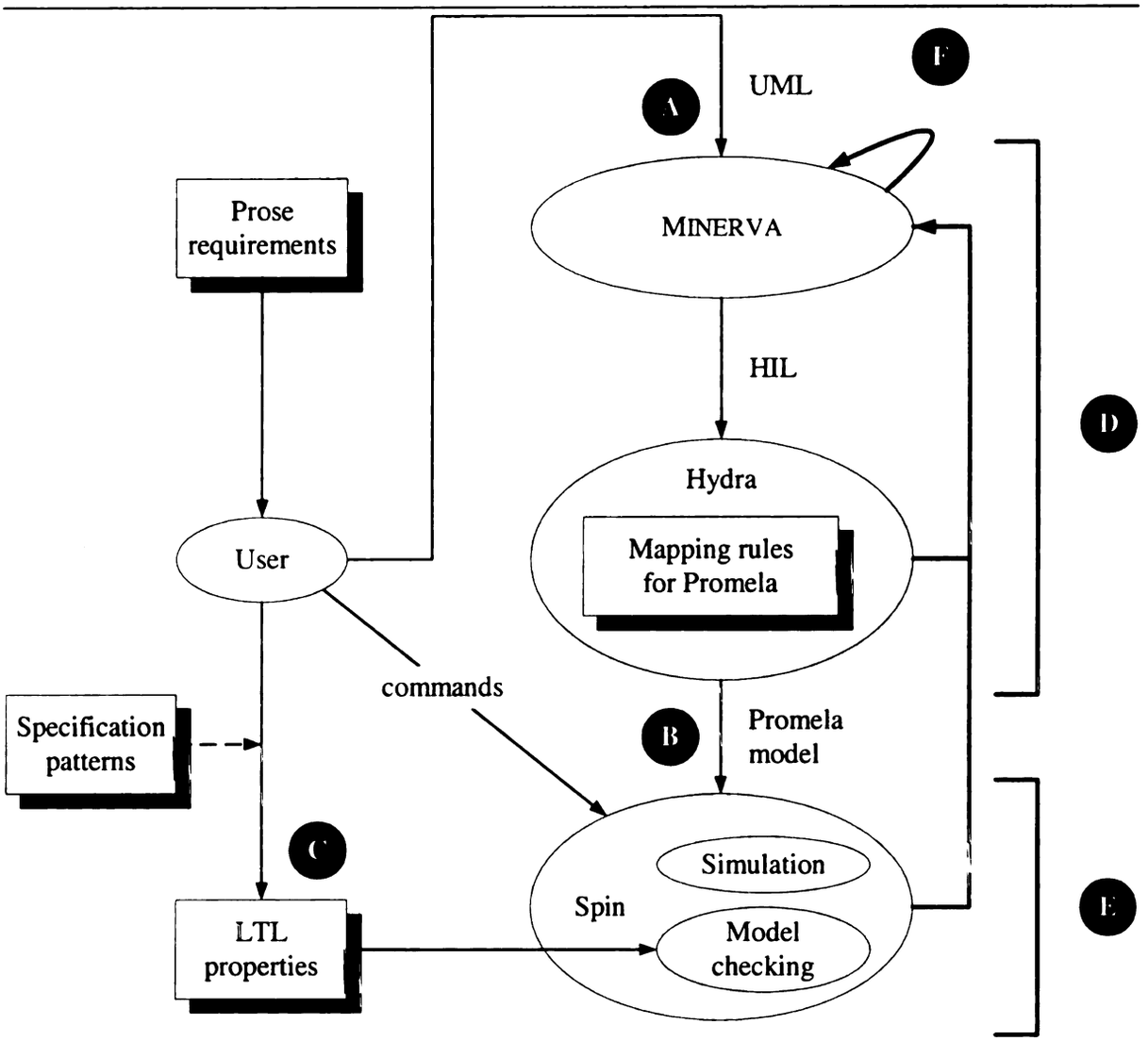


Figure 10.2: Framework from Figure 10.1 instantiated with tools

10.1.1 Graphical Editing Environments

As discussed in Chapter 5, there are essentially three techniques that a graphical editing environment may use, in some combination, in order to ensure diagram well-formedness: *prevention/containment*, *static consistency checking*, and *dynamic consistency checking*. In this section we examine the first and third techniques; we consider static consistency checking as a structural analysis technique later in this chapter in Section 10.3.1.

Prevention/Containment. In Chapter 5, Section 5.1.1, we discussed how a graphical editing environment, when augmented with knowledge of an object-oriented notation's graphical syntax, can prevent certain types of graphical syntax violations from being drawn, or guide a user towards creating consistent diagrams. Our graphical editor and visualization environment, MINERVA, is built atop Honeywell's DoME (Domain Model Editing) utility [87, 88]. DoME supports custom-built graphical editors, where the syntax of a diagram is described in terms of *metamodels*.[1] The resulting graphical editors prevent developers (or automated diagram generation utilities such as our sequence- and collaboration-diagram visualizations) from constructing syntactically incorrect diagrams with respect to the constraints imposed by the underlying metamodels.

As discussed in Chapter 5, generic drawing applications (*e.g.*, `xfig` [76]) that have no knowledge of an object-oriented notation's graphical syntax cannot impose any constraints on the graphical elements drawn. Such an application places the entire burden for well-formedness checking on a separate utility for static consistency checking. While UML does not require that a graphical editor supporting its notation use a metamodel-based approach for its own underlying implementation, it makes sense to leverage the UML's metamodel. Therefore, we do not consider generic drawing

[1] Recall (Chapter 2) that a *metamodel* is a class diagram that describes the constructs of a modeling language and the relationships between the constructs.

applications further. It is possible, however, to provide a drawing application with text-based rules rather than a graphical metamodel depiction in order to constrain what can be drawn. For example, Method Workbench and MetaEdit+ [89] comprise a suite of tools that provides a framework for defining domain-specific modeling languages and realizing customized domain-specific CASE tools. While the domain modeling language supports features similar to MINERVA's underlying DoME metamodels (domain concepts and properties, visual symbols to represent them, and rules about how they are connected), the tools use text-based forms to define the concepts, properties, and rules rather than a graphical class diagram metamodel depiction.

Dynamic Consistency Checking. Whether or not a graphical editing environment has been augmented with knowledge of an object-oriented graphical notation's syntax, the environment must still perform some consistency checking (*i.e.*, metamodels or rules cannot prevent or contain every error). This task can be performed either *statically* (as a separate step after the user reaches a stopping point in the drawing of a diagram), or *dynamically* (*i.e.*, the environment constantly monitors the status of the diagram-in-progress, alerting the user to possible errors). As described in Chapter 5, Sections 5.1.2 and 5.1.3, MINERVA and Hydra use *static consistency checking* to detect diagram anomalies not inherently prevented by MINERVA's metamodel-based editors; in Section 10.3.1 we examine static consistency checking in more detail. In contrast, here we consider how *dynamic consistency checking* can also be incorporated into a graphical editing environment in order to guide a developer in producing well-formed UML models. ArgoUML [90], for example, is a graphical editing environment for UML diagrams that provides interactive guidance during model construction via analysis agents, called *design critics*, based on "best practices" for software design. Design critics continually monitor a UML model-in-progress, offering advice and warnings in the form of dynamically updated to-do lists and highlighted diagram

elements. While this approach is based on cognitive psychology principles that enable the user to continually reflect upon and improve a design-in-progress, there may be an implicit cost in terms of negative user attitude incurred by the "constant critique" approach. Additionally, the critics are implemented as separate threads running in parallel, so there is a much higher computational cost than for static consistency checking.

Comparison with Our Approach. We find that using MINERVA's underlying metamodels and structural guidance within its graphical user interface to prevent and/or contain graphical and textual errors as much as possible, and then using static consistency checking to detect definition/usage and other discrepancies, including graphical syntax anomalies such as a missing start state, effectively distributes the responsibilities for consistency checking between the graphical editing environment and other utilities without unduly constraining the user during model development.

10.1.2 Stereotypes

UML was developed to be a general-purpose software modeling language. However, despite its attempt to be all things to all people, developers may still need modeling elements more specific to a given application domain than those already present in the UML. *Stereotypes* [16, 17, 18] in UML address this problem, allowing developers to extend the UML with new modeling elements specific to a given application domain. This approach has been used generally for modeling product lines [91] and system families [92], as well as more specifically for modeling middleware [93], fault-tolerant real-time systems [94] and embedded systems [95, 96], to give some examples.

Comparison with Our Approach. McUmber's formalization framework [31, 33], leveraged by our model development and analysis framework, uses only the modeling elements already present in the UML without extending it, in order to make the

approach available to any general-purpose UML modeling tool, and potentially applicable to other application domains besides embedded systems. However, in the past several years, stereotypes (an extension mechanism), and more recently, *profiles*[2] [97] (in part, a constraining mechanism) have become more prevalent for domain-specific modeling (*e.g.*, real-time modeling [98]; data modeling [99]; web modeling [100]; business modeling [101]; schedulability, performance, and time modeling [102]; CORBA modeling [103]). Stereotypes allow developers to model their systems using modeling elements specific to their application domain, *i.e.*, elements that model familiar objects and concepts. Profiles combine this approach with the technique of limiting the developers' modeling "palette" to only the necessary elements; for example, the approach discussed in this dissertation uses only the UML class and state diagrams for modeling embedded systems. An UML-based approach that extends UML with stereotypes that represent familiar objects and concepts within a specific application domain, yet limits the available modeling elements to only the ones necessary for modeling that domain, can guide developers unfamiliar with UML in both constructing models that have domain-specific meaning, and avoiding constructing extra diagrams or diagram elements that have not been found generally useful for modeling that domain.

If we chose to incorporate stereotypes into our approach, possible candidates would be the special modeling constructs _SYSTEMCLASS_ and Environment. These constructs are really specialized classes used for modeling purposes, so stereotypes representing them could be reused for other application domains. Other, embedded-system-domain-specific, candidates for stereotypes would be sensors and actuators. However, because these concepts are domain-specific, adding such stereotypes would make the approach more explicitly tied to the embedded-systems domain, and therefore less generally applicable.

[2] A *profile* is a subset of UML tailored to a specific application domain.

10.1.3 Patterns

Patterns are a third way to provide user guidance for constructing UML-based models for a given development phase and/or for a given application domain. Patterns supercede well-formedness constraints and stereotypes by offering diagram templates that express relationships between diagram elements specific to a given problem or application domain (*e.g.*, the *Visitor* design pattern [83] enables the definition of new operations on an object without changing the object; the *Watchdog* real-time safety and reliability design pattern [104] enables time-based detection of deadlock in real-time systems). Patterns largely entered the software engineering scene with the advent of Gamma *et al.*'s book *Design Patterns* [83] that provided a catalog of creational, structural, and behavioral patterns to guide construction of object-oriented models during the design phase of development. Patterns for other development phases (*e.g.*, analysis patterns [105], architectural patterns [106], *etc.*) and specific application domains (*e.g.*, database access patterns [107], fault-tolerant telecommunication system patterns [108], design patterns tailored to distributed real-time embedded systems (DREs) [109, 110, 111, 112], design patterns for avionics control systems [113], real-time design patterns [104, 114], security patterns [115], *etc.*) soon followed. Here, we overview design patterns and analysis patterns for the embedded systems domain, as well as fault-tolerance patterns.

Design Patterns and Analysis Patterns. Currently much of the embedded systems industry uses *ad hoc* development approaches [37] that emphasize design and coding over analysis [116]. The large number of design patterns [83], especially those tailored to real-time systems [104, 114] and distributed real-time embedded systems (DREs) [109, 110, 111], is further evidence of this focus. Despite its importance, the analysis phase is often neglected in current embedded systems development *practice*, often causing conceptual errors to be propagated to design and coding [36].

To address this problem, Konrad *et al.* propose *requirements patterns* [2, 3, 4] (now termed *object analysis patterns* [1]) for use in the analysis phase of embedded systems development to guide the construction of a conceptual model of a system. Analysis patterns are not new *per se* (*e.g.*, [36, 105, 117, 118]); for example, Fowler [105] identified several patterns that might be used during the analysis phase to represent conceptual models of business processes, such as abstractions from accounting, trading, and organizational relationships. While Fowler's analysis patterns may be relevant to only one application domain, or may span several domains, Konrad *et al.*'s object analysis patterns focus explicitly on software development for the embedded systems domain. Fowler also uses an informal description style for his patterns, while Konrad *et al.* use a template similar to the one used by Gamma *et al.* for their design patterns [83], with some modifications. (To distinguish the contents and objectives of their work from Fowler's, and to leverage Douglass's ROPES [37] development process, Konrad *et al.* use the term *object analysis patterns* instead of *analysis patterns.*)

Object analysis patterns not only guide developers in constructing UML-based conceptual models of requirements, but, unlike the other pattern approaches previously mentioned, also provide property templates so that developers are able to validate their conceptual models, prior to design, by leveraging McUmber's previously developed UML formalization framework [31, 33].

Fault-Tolerance Patterns. *Fault-tolerance*, the ability of a system to continue to execute correctly in the presence of a finite number of hardware and software faults, is a non-functional property potentially desired by many high-assurance applications, including fault-tolerant (individual or distributed) embedded systems. For example, several of Konrad *et al.*'s previously identified requirements patterns for individual embedded systems [2, 3, 4], namely the *Actuator-Sensor*, *Watchdog*, *Fault Handler*,

Communication, and *Actuation-Monitor* patterns, offer informal guidelines for modeling fault-tolerance under different contexts. Some of these, such as the *Actuator-Sensor*, *Communication*, and *Actuation-Monitor* patterns, suggest introducing fault-tolerance at the architectural level via the common technique of *redundancy* (*e.g.*, if a primary sensor fails, then a backup sensor takes over).

Indeed, design patterns for fault-tolerance in non-distributed systems exist that make use of hardware and/or software redundancy. For example, Daniels *et al.* [119] present a general design pattern for software fault-tolerance utilizing redundant voting strategies to minimize the potential for common-cause software faults. Ferreira and Rubira [120] present a system of design patterns to implement fault-tolerance that utilize replication, diversity, and exception-handling redundancy strategies for hardware, software, and environmental faults, respectively.

Comparison with Our Approach. As discussed in Chapter 9, requirements patterns [2, 3, 4] (now termed object analysis patterns [1]) can be used to drive our model development and analysis process introduced in Chapter 4. In comparison to our work, none of the other aforementioned pattern approaches offers a combination of patterns to guide construction of conceptual models, templates for the formal specification of critical properties, and a formalization framework tailored to the software analysis process of embedded systems development.

With regards to fault-tolerance patterns, while redundancy can reduce (but not eliminate) the risk of system failure due to random faults in hardware or common-cause software faults [37], it does not address detecting or handling faults due to inappropriate, missing, ill-timed, or out-of-sequence actions, or dangerous and/or illegal system conditions. The fault-handling patterns described in Chapter 9, based on the concepts of detectors and correctors [57], enables developers to model fault handling requirements and analyze them using specification-pattern-based constraints

related to the occurrence and ordering of actions such as message reception.

10.2 Guidance for Instantiating Formal Properties

Using our approach, developers can check requirements-based properties against the formal model derived from UML diagrams. Developers instantiate properties in terms of UML elements, such as attributes, states, and messages. Our approach recommends the use of *specification patterns* [43], temporal logic templates for specifying commonly occurring properties, to guide the formation and instantiation of requirements-based properties in LTL to check against the derived formal Promela model using Spin.

Requirements patterns [2, 3, 4], now termed object analysis patterns [1], take this idea one step further. They contain specification-pattern-based *constraints* [53] that can be checked against UML models that have been constructed using the diagram templates in the object analysis patterns. Thus, the object analysis patterns provide context for the constraints, so that developers not only know *how* but also *when* to instantiate and check a given property.

Comparison with Our Approach. Chapter 2, Section 2.2.4, briefly overviews specification patterns, and Chapter 4 discusses their role in our approach. As discussed in Chapter 9, object analysis patterns [1] can be used to drive our model development and analysis process introduced in Chapter 4. Both specification patterns and object analysis patterns provide guidance to developers for instantiating formal properties; however, without a framework and process to leverage this capability, their utility is limited. Our model development and analysis framework and process leverage these patterns, providing developers with a mechanism to construct diagrams against which to check instantiated properties in order to validate requirements.

10.3 Model Analysis

As discussed in Chapter 5, we divide analysis into two phases, structural analysis and behavioral analysis. Structural analyses (*i.e.*, consistency checks) are performed on UML models (*i.e.*, diagrams), while behavioral analyses (*i.e.*, simulation and model checking with Spin) are performed on formal (*i.e.*, Promela) models. We overview work related to these tasks in Sections 10.3.1 and 10.3.2, respectively.

10.3.1 Diagram Consistency Checking

The Object Management Group (OMG) developed well-formedness rules for UML that CASE tools for UML (*e.g.*, IBM's Rational Rose) should support. We have already discussed in Chapter 5, Section 5.1.1, and this chapter, Section 10.1.1, how some such rules can be enforced *a priori* by a metamodel-based or otherwise rule-enhanced graphical editing environment; however, consistency checking is still required for those rules that a graphical editing environment cannot enforce *a priori*. Existing CASE tools for UML support the well-formedness rules in a variety of ways (*e.g.*, some combination of metamodel- or rule-based constraints, and static or dynamic consistency checking); here we overview one general-purpose approach to *static consistency checking*, xlinkit [21], that could be utilized by any UML CASE tool that supports XML[3] [121] and the XMI[4] format. (ArgoUML's design critics [90], discussed in Section 10.1.1, are an example of *dynamic consistency checking*.) The xlinkit [21] tool offers a framework for rule-based detection of inconsistencies between distributed XML [121] documents. In particular, static consistency checking based on the OMG well-formedness rules for UML can be performed on UML diagrams in XMI format. The tool automatically generates hyperlinks, where an *inconsistent link* relates elements that violate consistency rules.

[3]Extensible Markup Language
[4]XML Metadata Interchange

Comparison with Our Approach. The application of the xlinkit framework to static consistency checking of UML diagrams is complementary to our approach and quite powerful. Its ability to check distributed documents is especially pertinent to industry, where projects may contain thousands of diagrams in documents spread across several networks or even continents. However, xlinkit is targeted entirely towards consistency management rather than dynamic property verification.

10.3.2 Behavioral Analysis of UML Diagrams with Spin

The model checker Spin [69] was developed in the 1980's at Bell Labs specifically for the verification of distributed software systems. While Spin originated in the telecommunications industry, it has gained increasing use in other industrial domains involving distributed [122] and embedded systems, such as flight systems [123] and railway systems [124]. Its non-deterministic input language, Promela, is loosely based on Dijkstra's guarded command language [125] and Hoare's CSP (Communicating Sequential Processes) [126].

Our approach uses the Spin model checker to validate (embedded systems) requirements expressed as LTL properties and UML diagrams. Therefore, we limit our discussion here to two UML-based graphical front-ends to Spin: vUML [127] and the Visual Interface for Promela (VIP) tool [128]. The vUML tool translates UML state diagrams to Promela and displays counterexamples as UML sequence diagrams. VIP supports a visual extension to Promela, called v-Promela, that extends the Promela language with graphical notations and object-oriented concepts. VIP translates UML-style collaboration diagrams (termed *structure diagrams* in v-Promela) and state diagrams to Promela, using many Promela-specific notations in the UML diagrams.

Comparison with Our Approach. McUmber's formalization framework [31, 33], leveraged by our approach, was designed to be as target-language independent as possible at the graphical modeling level. It includes a generic syntax for commonly occurring concepts such as boolean expressions, variable assignments, and message sends that are transparently translated to formal language syntax (*e.g.*, Promela) during formal specification generation; developers do not need to know formal language syntax. Additionally, the formalization framework may be extended with formalization rules for other target languages (*e.g.*, VHDL [30]), so it, and thus our approach, is not specifically tied to Promela and Spin. Finally, Hydra generates Promela from both the class and state diagrams, rather than state diagrams only, while MINERVA animates the original UML state diagrams, as well as generating UML sequence and collaboration diagrams, based on the behavioral analysis results from simulation and model checking using Spin.

10.4 Visual Interpretation of Analysis Results

We divide analysis into structural and behavioral phases, with corresponding visualization techniques, so we briefly overview approaches to both types of visualizations here. Regarding structural visualization techniques, static consistency checking tools generally highlight questionable diagrams and/or diagram elements (*e.g.*, xlinkit [21] highlights questionable relationships between such items), while dynamic consistency checking tools such as ArgoUML [90] highlight questionable diagrams and/or diagram elements and offer textual messages to the user (*e.g.*, additional warnings, error messages, or advice). Regarding behavioral visualization techniques, commercial UML CASE tools for embedded systems development (*e.g.*, [19, 20, 129, 130]) overwhelmingly visualize simulation traces with state diagram animation and sequence diagram generation and/or animation.

Comparison with Our Approach. Chapter 6 discusses our visualization techniques for displaying results of both structural and behavioral analyses to the user. MINERVA's structural visualizations include textual messages and highlighting questionable diagram elements. MINERVA's behavioral visualizations include state diagram animation, sequence diagram generation, and collaboration diagram generation and animation. In addition to displaying message ordering, we include the capability to display attribute values and the current state of each object during collaboration diagram animation, thus augmenting the amount and type of information usually displayed with either a state diagram (current state) or a sequence diagram (message ordering). Additionally, our approach incorporates both structural and behavioral analyses rather than only one or the other, so we likewise incorporate both structural and behavioral visualizations into one environment.

10.5 Development Environments for Embedded Systems

This research presents an approach to validating embedded systems requirements modeled as both UML diagrams and linear time temporal logic (LTL) properties using formal verification (*e.g.*, model checking) techniques. We developed a model development and analysis framework (Chapter 4) that leverages, integrates, and encapsulates a previously developed formalization framework for object-oriented notations [31, 33], including feedback to diagrams from formal analysis tools in order to insulate users from the outputs of such tools. We also developed an iterative and incremental model development and analysis process (Chapter 4) that comprises steps for (1) model construction, (2) structural and behavioral analyses, and (3) refinements based on feedback from both types of analyses. While previous sections explored work related to the enabling techniques for these steps, this section com-

pares our approach with leading environments for embedded systems development, including formal and commercial approaches. Specifically, we overview the Omega project [131], the Software Cost Reduction (SCR) toolset SCR* [132], the NIMBUS environment [133] that leverages RSML [134] specifications, and the four major commercial UML CASE tool offerings for embedded systems development according to current market research [135]: ARTiSAN's Real-time Studio [129], IBM Rational's Rose RealTime [19], I-Logix's Rhapsody [20], and Telelogic's TAU Generation2 [130]. The main difference between our approach and the commercial environments is that they rely on validation and testing rather than verification techniques; therefore, we do not include a separate comparison section for each but highlight salient differences within the overview of each environment.

10.5.1 Omega Project

The EU-IST Omega project [131] (Correct Development of Real-Time Embedded Systems) selects a subset of UML, including class and state diagrams, applicable to the embedded systems domain and extends this subset by leveraging the Schedulability, Performance and Time profile [102] to create an *Omega kernel model*, or intermediate representation in XMI format [121], used as the basis for a common input to different formal analysis tools. The intermediate representation has a fixed formal semantics, and connects commercial UML CASE tools such as Rhapsody [20] and TAU [130] with formal tools such as model checkers or interactive theorem provers. Requirements may be expressed via Live Sequence Charts (LSCs) [136], a subset of OCL [137], or special state machines stereotyped as observers. Omega supports interactive simulation of LSCs, while error traces from formal analysis tools are provided to the user as scenarios.

Comparison with Our Approach. Our approach leverages McUmber's formalization framework [31, 33], that maps a unified metamodel of a subset of UML into the metamodel of the syntax of a formal target language. The source UML metamodel remains fixed, while the metamodels for different target languages will vary as the syntax of each target language varies. The Omega project translates UML models into an intermediate representation, the Omega kernel model, that has a fixed semantics. This model is used as the basis for further translation to the input format of formal analysis tools. The Omega project formalizes a few additional UML concepts not addressed in McUmbers's formalization embodied by Hydra: Hydra supports asynchronous signal-based communications only, whereas Omega also supports synchronous operation calls; in Hydra, all classes have associated state diagrams, whereas Omega makes a distinction between reactive and non-reactive classes (only reactive classes have associated state diagrams); in Hydra, each class has its own thread of control and event queue, whereas in Omega, only active classes do so. Additionally, the Omega project includes support for timing, whereas McUmber's original formalization framework did not; current work has focused on extending McUmber's formalization with timing [54]. While Omega provides error traces to the user in the form of *scenarios* (it is unclear whether this terminology refers to LSCs, sequence diagrams, or use case diagrams—the most likely case is sequence diagrams), more elaborate feedback mechanisms are outside of the scope of the project. As we have learned from case studies (*e.g.*, [1, 41]), sequence diagrams display the sequence of events leading up to an error, but do not always provide enough information to determine *why* the error occurred. Therefore, in addition to sequence diagram generation, our approach enables state diagram animation and collaboration diagram generation and animation in order to provide the user with additional information.

10.5.2 SCR*

SCR* [132] is an integrated tool suite supporting the Software Cost Reduction (SCR) method. SCR uses a tabular notation to capture behavioral information about relations between monitored and controlled variables. The tool suite SCR* offers a simulator and verification capabilities through the use of model checkers and theorem provers. Recently, Gargantini and Riccobene developed an approach to model-driven animation of SCR specifications [138]. This approach derives *animation goals* from either predicates over states or temporal logic formulas. Predicates or formulas are specified by the user based on SCR tables, and are used as input to model checkers integrated into SCR* (*e.g.*, Spin [69] or SMV [75]). A detected counterexample then becomes an *animation scenario* animated via graphical user interface (GUI) widgets that may mimic actual control panels, or otherwise display information about monitored and controlled variables.

Comparison with Our Approach. Although its tabular notation is much different from the UML diagrammatic notation used in our approach and currently being adopted by industry, SCR has been successfully applied to large-scale critical systems (*e.g.*, avionics systems [139], weapons control systems [140], *etc.*). When combined with Gargantini and Riccobene's animation approach [138], formal analysis results can be displayed in formats more accessible to users, such as control panels that mimic the actual deployed system. In this way, SCR* meets our goal of insulating developers from formal analysis tools, although the analysis results are not displayed in terms of the original model (in SCR*, tables; in our approach, UML diagrams). Because we also utilize counterexamples to drive our animations, we could extend our visualization techniques to include animating GUI widgets like Gargantini and Riccobene in order to present information to users in terms of the context of the deployed system.

10.5.3 NIMBUS

The NIMBUS environment [133] is a framework for specification-based prototyping of embedded systems. The environment enables simulation of RSML (Requirements State Machine Language) [134] specifications of embedded systems components. RSML, based on Harel's Statecharts [62], was originally developed as a specification language for embedded systems requirements. RSML specifications can be analyzed for completeness, consistency, and adherence to safety and liveness properties. The NIMBUS environment provides for iterative refinement of an RSML specification to the point of executing it directly within the context of actual hardware, or "hardware-in-the-loop" simulation.

Comparison with Our Approach. The NIMBUS environment [133] directly incorporates RSML specifications, whereas our approach leverages McUmber's general formalization framework [31, 33] that enables flexibility for targeting different formal languages and analysis tools. It is not clear from the description of the NIMBUS environment whether guidance is provided to developers for creating RSML specifications or formulating properties to check against them; however, like SCR [132], the specifications utilize a tabular notation. A software emulation of the environment drives simulation, and information about controlled variables is collected in text files for later examination with spreadsheets rather than being presented in terms of RSML.

10.5.4 ARTiSAN's Real-time Studio

ARTiSAN's Real-time Studio [129], a suite of tools for software modeling and component based development, enables developers to model system architecture and functional requirements. It supports both stereotypes and profiles, including real-time extensions to UML. Executable code is automatically generated from UML state diagrams, including test harnesses. Animation of sequence and state diagrams en-

able developers to validate system behavior via simulation. There are no underlying formal semantics; this tool suite relies on validation and testing rather than formal verification.

10.5.5 IBM Rational's Rose RealTime

Rational Rose [19], a general-purpose UML modeling tool, is part of a family of integrated products that support the analysis, design, implementation, and testing phases of software development. It offers a "modelcheck" utility that detects diagram construction errors such as unconnected associations or transitions, which we avoid inherently with MINERVA's metamodel-based graphical editors. On the other hand, the Rational Rose RealTime suite, aimed at embedded systems software development, has been optimized to model concurrent, event-driven, reactive, and state-based systems. Validation is performed by generating executable code from UML models, while the visual debugger shows message traces and state changes on the UML models at runtime. Nevertheless, both families of products, oriented towards code generation, rely on validation and testing rather than formal verification.

10.5.6 I-Logix's Rhapsody

I-logix's Rhapsody [20] family of products offers a UML-driven approach to analysis, design, testing and implementation of embedded systems: Use cases and sequence diagrams capture requirements; object diagrams depict system architecture; statecharts and activity diagrams model component behavior; and component diagrams describe run-time artifacts. Design-level debugging features include highlighting states in statecharts, displaying message traces in sequence diagrams, and monitoring attribute value changes. While the underlying semantics of the executable models developed with these products is based on Harel's Statecharts [62], McUmber's formalization framework [31, 33] that underlies our approach enables support for different seman-

tics and integration with various formal verification tools.

10.5.7 Telelogic's TAU Generation2

Telelogic's TAU Generation2 tool suite [130] consists of four products: TAU/Architect, TAU/Developer, TAU/Tester, and TAU/Logiscope. These tools are used for systems architecture and design, model-driven software development, systems and integration testing, and software quality assurance and metrics, respectively. TAU/Architect and TAU/Developer are the most closely related to our work. TAU/Architect enables developers to create UML-based structural (*e.g.*, component diagrams) and behavioral models (*e.g.*, state diagrams) and use simulation to validate them. It allows user-defined symbols for domain-specific modeling (*i.e.*, stereotypes). TAU/Developer compiles executable models from the diagrams and includes a model debugger that includes visualizing execution traces with on-the-fly creation of sequence diagrams and animation of the model's state diagrams. Again, this tool suite relies on validation and testing rather than formal verification.

Chapter 11

Conclusions and Future Investigations

While requirements errors can be costly for software systems in general [9], they can be especially costly for high-assurance or safety-critical embedded systems where failure can have dire consequences. Therefore, methods for modeling and rigorously analyzing embedded systems requirements are needed. However, the *ad hoc* development approaches currently used in embedded systems lack systematic methods for both modeling and analyzing requirements [37, 38].

Although the embedded systems community has expressed interest in exploring how object-oriented modeling, specifically the UML, can be used for embedded systems development [37, 38, 39], UML lacks a formal semantics, thus precluding rigorous analysis of requirements expressed as UML models. While formalization of UML enables rigorous analysis of formal models derived from UML diagrams, formalization itself is not sufficient to broaden the community of embedded systems developers who can use formal methods to rigorously analyze requirements. To enable developers to model and analyze requirements in UML without having to know details of formal models requires a framework and process that takes advantage of a UML formalization

yet insulates developers from the formal models produced by such a formalization.

This research presents an approach to validating embedded systems requirements modeled as both UML diagrams and temporal logic properties using formal verification (*e.g.*, model checking) techniques. We describe a model development and analysis framework, complementary to a previously developed formalization framework [31, 33], that insulates the developer from formal models and outputs of tools, and an overall model development and analysis process.

We have validated this work by applying the approach to several case studies from industrial collaborators:

- an **Adaptive Cruise Control** system [40, 77] (also described in Chapter 7) that uses radar to avoid collisions,
- an **Anti-Lock Braking System** [53, 86] with redundant brake sensors,
- a self-cleaning **Diesel Filter System** [41, 42, 84] that removes soot from diesel truck exhaust, and most recently
- an **Electronically Controlled Steering** system [54, 55, 141] that provides variable-assistance power steering.

Each case study uses the integrated model development/analysis and formalization frameworks from Chapter 4 instantiated with tools (*e.g.*, MINERVA [40, 47, 49, 50, 51, 52, 56], Hydra [31, 33, 48], and Spin [69]) as described in the process diagram in Figure 4.3, page 55, and demonstrated in the remainder of the dissertation. Additionally, the latter three studies leverage object analysis patterns [4] as described in Chapter 9, Section 9.2, while the **Electronically Controlled Steering** system uses an extension of McUmber's formalization framework with timing information [54, 55]. These case studies demonstrate that our framework and process enables embedded systems developers to model and analyze their requirements in UML with formal-verification

(*e.g.*, model checking) techniques while being insulated from formal models and outputs of tools.

11.1 Summary of Contributions

In summary, this research makes several contributions [1, 40, 41, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56]:

- **Bifurcated approach to analysis [40, 47, 51].** Preliminary investigations [44, 45, 46] (Chapter 3) indicated that formal analysis tools are not well-suited to detecting structural problems within the formal models generated from diagrams; that is, diagram well-formedness should be checked prior to formal model generation. Therefore, we describe and demonstrate a bifurcated approach to analysis (Chapter 5) that incorporates both structural (diagram level) and behavioral (formal model level) analyses.

  Impact. While formalization of object-oriented notations combined with automated generation of formal specifications enables rigorous analysis of diagrams (via the formal specifications), such formalizations make implicit assumptions about the well-formedness of diagrams. Ill-formed diagrams may result in degenerate formal models, but in ways that cannot be detected by the formal analysis tools (Chapter 3 presented several examples). Therefore, an approach to formalizing object-oriented notations intended for use by typical software developers must include diagram consistency-checking (structural analysis) prior to formal model generation and behavioral analysis.

- **Visualization of analysis results to guide diagram refinements [40, 41, 47, 50].** We developed visualization techniques for results of both structural analyses of diagrams and behavioral analyses of automatically generated formal

models. Visualizations (Chapter 6) include highlighting portions of diagrams, animating diagrams, and generating new diagrams.

Impact. As discussed in Chapter 6, analysis results from formal language analysis tools are often cryptic, and are expressed in terms of the formal model rather than the original diagrams. Therefore, an approach to rigorous analysis of object-oriented diagrams incorporating formal language analysis tools intended for typical software developers must include a mechanism for interpreting formal analysis results in familiar terms; we have chosen to interpret and visualize results in terms of the original UML diagrams or complementary ones in order to facilitate model refinement and understanding.

- **Framework and process for development and analysis of (formalized) UML models [40, 41, 49, 56].** We developed a model development and analysis framework (Chapter 4) that leverages, integrates, and encapsulates a previously developed formalization framework for object-oriented notations [31, 33], including feedback to diagrams from formal analysis tools in order to insulate users from the outputs of such tools. We also developed an iterative and incremental model development and analysis process (Chapter 4) that comprises steps for (1) model construction, (2) structural analyses, (3) behavioral analyses, and (4) refinements based on feedback from both types of analyses.

Impact. Automated tools are necessary to broaden the community of users who can take advantage of the benefits of formal methods. The model development and analysis framework (embodied by MINERVA) described in Chapter 4 encapsulates an existing formalization framework (embodied by Hydra), thus providing a graphical diagram editing and visualization environment as both a front-end for the formalization framework and a back-end for formal analysis

tools. While analysis and visualization techniques, and a framework to support them, are important, they have greater impact and utility when incorporated into a systematic process that guides typical software developers in modeling and analyzing their requirements. Chapter 4 describes our process, and the industrial case study presented in Chapter 7 illustrates its application to a real-world embedded system. Typical software developers thus have a systematic process and tools with which to model their requirements and view analysis results in UML.

- **Structural and behavioral patterns for modeling fault handling.** High-assurance systems must often remain operational even in the presence of faults. We developed structural and behavioral patterns for modeling the fault-tolerance concepts of *detectors* and *correctors* [57] in UML (Chapter 9) to provide guidance for modeling and analyzing fault handling requirements.

Impact. In Chapter 9, we extended our framework and process to incorporate requirements patterns [2, 3, 4] (now termed object analysis patterns [1]) that use templates for (UML) diagrams and (temporal logic) system properties to guide developers in creating UML models and instantiating properties to check against them. We used our patterns for detectors and correctors to refine Konrad *et al.*'s original Fault Handler requirements pattern [2, 3, 4], and applied the refined requirements pattern to the Adaptive Cruise Control example from Chapter 7. Patterns for specific application domains, such as embedded systems, provide guidance for typical software developers who may not be familiar with UML, formal methods, or a given application domain, in constructing UML models of their requirements and in instantiating temporal logic properties to check against their models.

Collective impact. An overarching objective of this research was to facilitate technology transfer of rigorous software engineering techniques to typical software developers, especially in the embedded systems domain. As the complexity and critical nature of embedded systems increase, developers can no longer rely on *ad hoc* methods. Effective analysis techniques are needed to minimize the number of errors introduced during the early stages of development, and to assist in detecting the causes of such errors. Our model development and analysis framework and process, integrated with McUmber's formalization framework [31, 33], enable rigorous analysis of UML-based requirements using consistency-checking, simulation, and model checking techniques, all combined with visualizations of analysis results. Therefore, the collective contribution of this work enables developers from the broad UML user community to make use of formal-verification tools to support model validation while being insulated from the formal models used by these tools.

11.2 Future Investigations

Several investigations complementary to the research presented in this dissertation may be pursued in future work. These investigations include creating formalization (*i.e.*, mapping) rules for other target languages, codifying mapping rules in a more precise way, extending both the model development and analysis framework and the formalization framework to support timing, and developing patterns to guide modeling and analysis of distributed real-time embedded systems. These ideas are further elaborated below.

- **Creating formalization (mapping) rules for other target languages.**

  McUmber developed a general framework for formalizing object-oriented modeling notations [31, 33]. The model development and analysis framework presented in this dissertation leverages and integrates this formalization framework.

To demonstrate the flexibility of the formalization framework, McUmber created mapping rules from UML to both VHDL [30, 31], a simulation language used in embedded systems development, and Promela [31, 33], the input language for the model checker Spin [69]. The instantiation of the model development/analysis and formalization frameworks described in this dissertation uses his UML-to-Promela mapping rules [31, 33] and the Spin [69] model checker. Using McUmber's metamodel-based approach to formalization, it would be possible to create mapping rules to other target languages, such as SMV [75], to enable the use of other formal analysis tools.

Model checkers for SMV (*e.g.*, Cadence SMV [142], NuSMV [143, 144]) support *Computational Tree Logic* (CTL), a *branching-time* temporal logic where temporal operators quantify over the paths possible from a given state, as well as LTL, a linear-time temporal logic where temporal operators describe events along a single path of computation. Because CTL and LTL are not expressively equivalent [5], a mapping to SMV would provide the potential for checking properties previously not expressible with the Spin instantiation of the integrated frameworks, such as $AG(EFp)$ [70], or the mutual-exclusion property described in Figure 11.1. Specification patterns [43] and some object analysis pattern constraints [53] also support CTL [145], so developers would still have some guidance when instantiating CTL properties. Currently, there are few UML formalizations to SMV (*e.g.*, [146, 147, 148]).

- **Codifying mapping rules more precisely.**

State-of-the-art formalization approaches (*e.g.*, [26, 27, 31, 33, 146]) overwhelmingly use what we term *correspondence-style* rules [35] to define mappings from a source to a target language for every source language feature formalized. That is, each rule presents, in an inherently informal style, a prose description of the

Example: Consider an instance of the mutual exclusion problem, where each process P_i has a *non-critical section* NCS_i, a *trying section* TRY_i, and a *critical section* CS_i. It should be possible for a particular process P_i either to remain in its non-critical section forever while other processes P_j perform their tasks in a mutually exclusive fashion, or to eventually enter its trying section. The key property is that once P_i is in NCS_i, it either remains there forever or eventually enters TRY_i, which can be expressed in CTL as

$$EG(inNCS_i) \wedge EF(inTRY_i) \wedge A(G(inNCS_i) \vee F(inTRY_i)) \qquad (11.1)$$

where path quantifiers A ("for all paths") or E ("for some paths") may prefix assertions composed of linear-time temporal operators G ("always"), F ("sometimes"), X ("next"), and U ("until"). Thus, Expression (11.1) states (in CTL) that for some paths, process P_i is always in its non-critical section NCS_i; and for some paths, process P_i is sometimes in its trying section TRY_i; and for all paths, process P_i is either always in its non-critical section NCS_i or sometimes in its trying section TRY_i. Because LTL cannot explicitly express the existence of alternate computation paths, but only describe events along a single path, the closest candidate expression using the linear-time temporal operators described above for Expression (11.1) is

$$A(G(inNCS_i) \vee F(inTRY_i)), \qquad (11.2)$$

which unfortunately allows for a degenerate model where all paths satisfy $F(inTRY_i)$, the possibility that process P_i eventually enters its trying section, and no paths satisfy $G(inNCS_i)$, the possibility that process P_i remains in its non-critical section forever. Using the LTL notation of Chapter 2, Expression (11.2) can be written as

$$\Box(inNCS_i) \vee \Diamond(inTRY_i). \qquad (11.3)$$

Figure 11.1: Example CTL property not fully expressible in LTL [5]

context for instantiating the rule, and examples of the target-language code snippets to be generated. For example, Wang's OMT-to-LOTOS formalization [27] and McUmber's UML-to-Promela formalization [31, 33], described in this dissertation, both use this approach. However, recent investigations by Cheng *et al.* [35, 149] suggest that a combination of a natural deduction system (NDS) and metamodel-based approach to formalization has an advantage over correspondence-style rule-based ones in that NDS rules themselves can precisely capture a mapping from source- to target-language metamodels. Potentially ambiguous, inconsistent, and/or incomplete correspondence-style rules, on the other hand, must first be interpreted by a human (an inherently error-prone step by itself) in order to be encoded into an executable tool. Cheng *et al.* [35] found that an NDS and metamodel-based approach forces developers to be explicit about assumptions when creating a set of formalization rules, and also revealed ambiguities and missing cases in an existing set of correspondence-style rules mapping a subset of UML to SMV [146]. Precisely codifying a mapping from UML to a given target language in a format amenable to automated reasoning (*i.e.*, NDS rules) will enable unambiguous generation of a formal model from UML diagrams.

- **Extending both frameworks to support timing.**

  Konrad *et al.*'s preliminary work with validating timing-based embedded systems requirements [54, 55] has shown that checking untimed properties is not sufficient for systems that rely critically on timing. For example, a property for a cruise control system that states that "it is always the case that if the driver taps the brake pedal, the cruise control system eventually disengages" does not include any timing information about the actual time delay that it takes from the activation of the brake until the disengagement of the system. Therefore, it is also necessary to check the property that "it is always the case that if

the driver taps the brake pedal, the cruise control system disengages within a specific time period." Because (real-time) embedded systems frequently have strict timing constraints, methods for modeling and analyzing time-based requirements have value for embedded systems developers.

Konrad *et al.* [54, 55] introduce a *timer* type in UML class diagrams and use a notation similar to timed automata [150] to add the ability to manipulate and evaluate timers in UML state diagrams. Correspondingly, they extend McUmber's previous UML-to-Promela formalization [31, 33] to incorporate semantics similar to timed automata, using the digital-clock model [150]. The result is that requirements-based properties involving time (expressed in a subset of metric temporal logic (MTL) [151]) can be checked against the formal Promela model automatically generated from UML diagrams.

Modified versions of Spin [152, 153] for analyzing timing have been developed, as well as other tools for the verification of real-time systems, such as Kronos [154] and HyTech [155]. However, these tools lack both a mapping from UML into their respective target languages, and a UML-based graphical editing and visualization environment. Therefore, these tools do not offer any support for the graphical UML-based modeling of a system nor for visualizing property violation traces in terms of UML diagrams.

MINERVA's graphical editors could be extended to support timing syntax, and Hydra's automated generation of formal models could be extended to support timing semantics, at first using Konrad *et al.*'s extension to McUmber's UML-to-Promela formalization [54, 55] described above, and then potentially using mappings to other target languages and tools that have built-in support for timing. Additionally, timing-based visualizations might be helpful for indicating *when* a particular property is violated.

- **Developing patterns for distributed real-time embedded systems.**

  Given the potentially critical nature of embedded systems (*e.g.*, X-by-wire, medical devices, *etc.*) in which faulty behavior of a system could lead to significant loss, methods for modeling and developing embedded systems and rigorously analyzing behavior before starting the design phase and committing to code are increasingly important. However, currently much of the embedded systems industry uses *ad hoc* development approaches [37] that emphasize design and coding over analysis [116]. The large number of *design patterns* [83], especially design patterns tailored to real-time systems (*e.g.*, [104, 109, 110, 111, 114]), is further evidence of this focus. Despite its importance, the analysis phase is often neglected in current embedded systems development *practice*, often causing conceptual errors to be propagated to design and coding [36]. To address this problem for (non-distributed) embedded systems, Konrad *et al.* propose *object analysis patterns* [1, 2, 3, 4] to be used in the analysis phase of development that not only guide developers in constructing UML-based conceptual models of their systems, but also provide property templates so that developers are able to validate these models, prior to design, using McUmber's formalization framework [31, 33]. In Chapter 9, we discussed and demonstrated how these patterns can be used to drive our model development and analysis process from Chapter 4 by guiding both model construction and property instantiation.

  The demand for distributed real-time embedded systems (DREs) has increased considerably in recent years and is expected to continue to grow. DREs occur in many application domains, including automotive, aerospace, manufacturing, and telecommunication. The complexity of DREs has increased in order to add new services and features in an effort to keep these applications competitive in a global market. These embedded devices often operate in environments where a failure could lead to significant losses, such as human life or financial losses. The

increase in the number and complexity of DREs strongly motivates the need for more rigorous, repeatable, and cost-effective development techniques, of which patterns can play an important role. However, Konrad *et al.*'s object analysis patterns [1, 2, 3, 4] focus on an individual embedded system, rather than a collection of embedded systems that work cooperatively together as they would in a DRE. These patterns could be expanded significantly to address various concerns of the DRE domain, such as decentralized fault-handling, real-time hard timing constraints, concurrency, synchronization, and safety.

APPENDICES

Appendix A

Wang's Design Process

In order to formalize OMT, Wang developed a systematic design process for constructing and refining the object-oriented models [60]. The design process contains iterations of model development. For each step of model development during a given iteration, corresponding formal models are derived or refined (see Figure A.1). The process explicitly addresses the consistency between the formal models of two adjacent levels of abstraction thus enabling stepwise refinement and consistency checking [60].

In the design process, Steps 1-3 focus on creating system-level versions of object, dynamic, and functional models, respectively. Steps 4-7 are refinements and decompositions of the models from Steps 1-3. An object functional model (OFM) in Step 5 is a variation of data flow diagrams, and it depicts visible services offered by the object. A service refinement functional model (SRFM) in Step 6b is also a variation of data flow diagrams, depicting a system/object service in terms of the services provided by the aggregate objects of the system/object. Step 8 composes the dynamic models for all aggregate objects to depict the overall system behavior. The formal models of the diagrams enable automated analysis to check that the diagram and model refinements are consistent with earlier versions of diagrams and models.

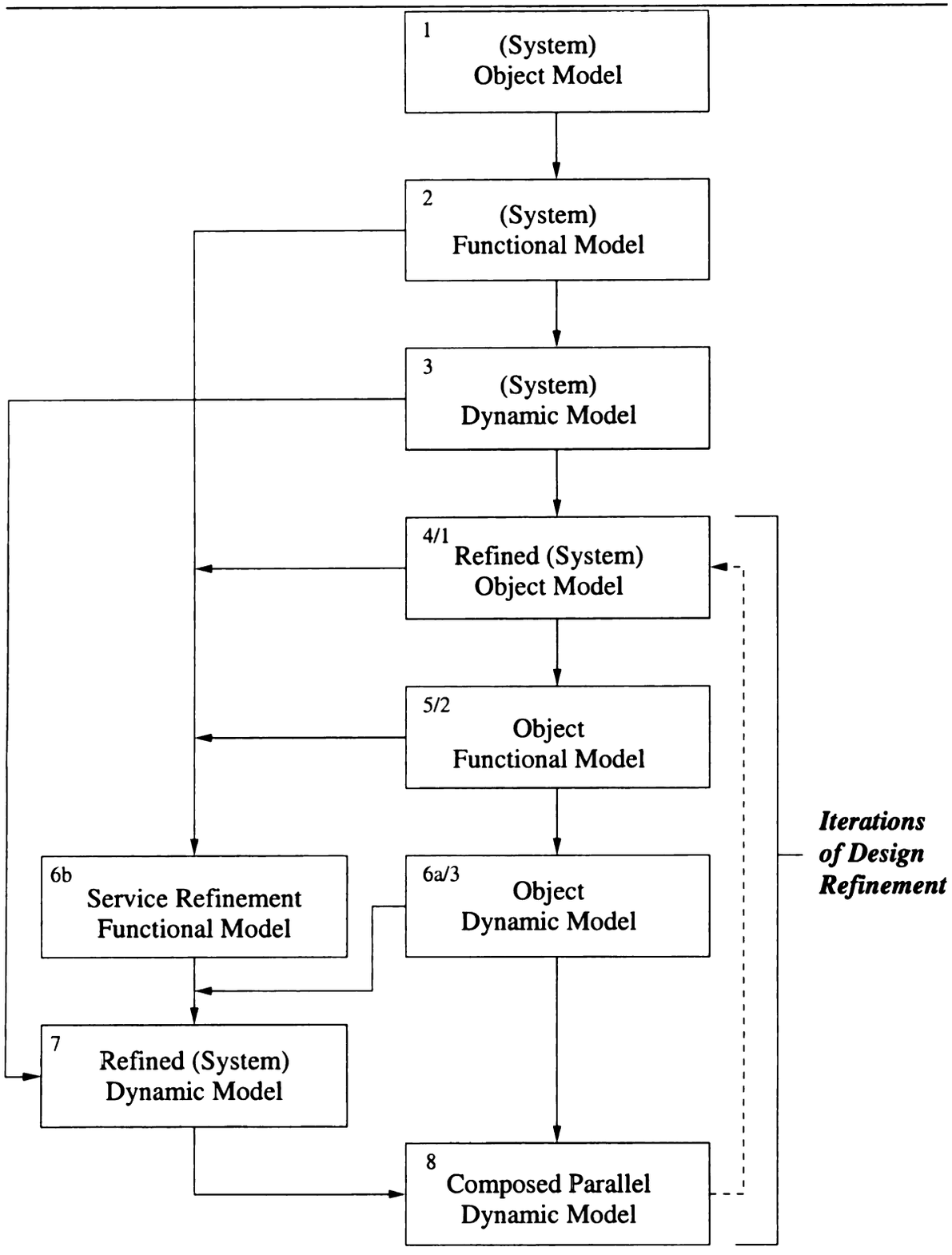


Figure A.1: Wang's iterative design process

Appendix B

TRMCS Refinement

The *Teleservices and Remote Medical Care System* (TRMCS) design proposed in [46] is a distributed system that realizes its functionality via a collection of communicating software components. The distributed operation is supported by a *client-server* architecture. The main components of interest examined in the refinement cooperate to realize the services expressed in the high-level model. This Appendix overviews object, service, and dynamic model refinement for the TRMCS to complement the discussion in Chapter 3 illustrating different LOTOS analyses.

Object refinement. The system-level object model for the TRMCS is refined (according to Step 4 of Wang's design process, Appendix A, Figure A.1, page 235) to include new objects as shown in bold in Figure B.1, where the diamond indicates aggregation and a filled circle indicates the "zero or more" relationship (see Chapter 2, Section 2.1.1, page 11, for more details on OMT syntax). As shown in Figure B.1, a Data Repository handles requests for patient records issued by Clients, while a Name Monitor is responsible for maintaining a list of active Data Repositories and handling Client requests for copies of this list. A Channel represents the connection between a Client object and any of the Data Repositories. LOTOS models (not shown) are derived for all of the aggregate objects in the same manner in which a LOTOS model

was derived for the high-level system (Chapter 3).

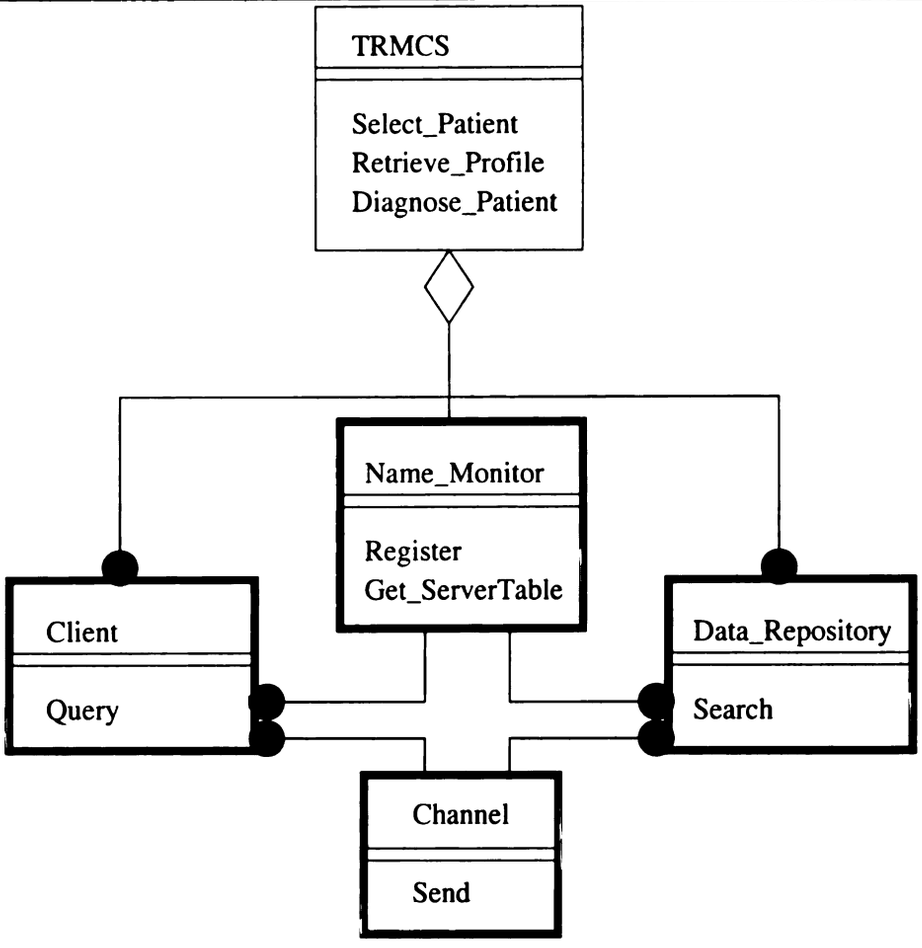


Figure B.1: Refined TRMCS object model (attributes have been elided)

Service refinement. The refinement of the high-level TRMCS into aggregate objects Client, Data Repository, Name Monitor, and Channel allows us to refine the high-level *Retrieve Profile* service in the TRMCS object into a composition of services offered by the aggregates. TRMCS uses the Client's *Query* service to implement the high-level *Retrieve Profile* service. Thus the client-server architecture suggests a refinement of the dynamic model of the TRMCS system (Figure B.2) to depict the *Retrieve Profile* service being handled by the Client's *Query* service. The WaitQuery

state is introduced to handle the redirection of the **Query Request** and the conversion of the returned **Query Result** as described below. The modified parts of the model (indicated by bold states or transition text outlined by dotted rectangles) specify that:

1. When a request for the high-level *Retrieve Profile* service occurs, the input argument **Patient ID** is decomposed into **Data Repository Name** and **Query Request**.

2. The **Query Request** is redirected to a **Client** object for a *Query* service.

3. The **TRMCS** system enters the WaitQuery state to await a **Query Result** from the **Client**.

4. When a **Query Result** is received from the **Client**, the **TRMCS** converts the **Query Result** to a **Patient Record** and delivers it to the user.

Dynamic model refinement. As indicated in Step 8 of Wang's design process (Appendix A, Figure A.1, page 235), the dynamic models of *instantiations* of all the aggregate objects and the **TRMCS** object are composed concurrently, and the refined object-oriented models are again translated to LOTOS to create the refined model (not shown). For our case study [46], we composed the instantiated behaviors of one **Client** (Figure B.3), one **Name Monitor** (Figure B.4), and one **Channel** (Figure B.5) with the instantiated behaviors of two **Data Repository** instances (Figures B.6 and B. 7). The behaviors of the **Client**, **Name Monitor**, **Channel**, and each **Data Repository** were instantiated as described in Figure B.8.

We included two instances of **Data Repository** in order to validate correct concurrent behavior for the distributed system. The two instances of **Data Repository**, **LANSING** and **DETROIT**, are fully interleaved.

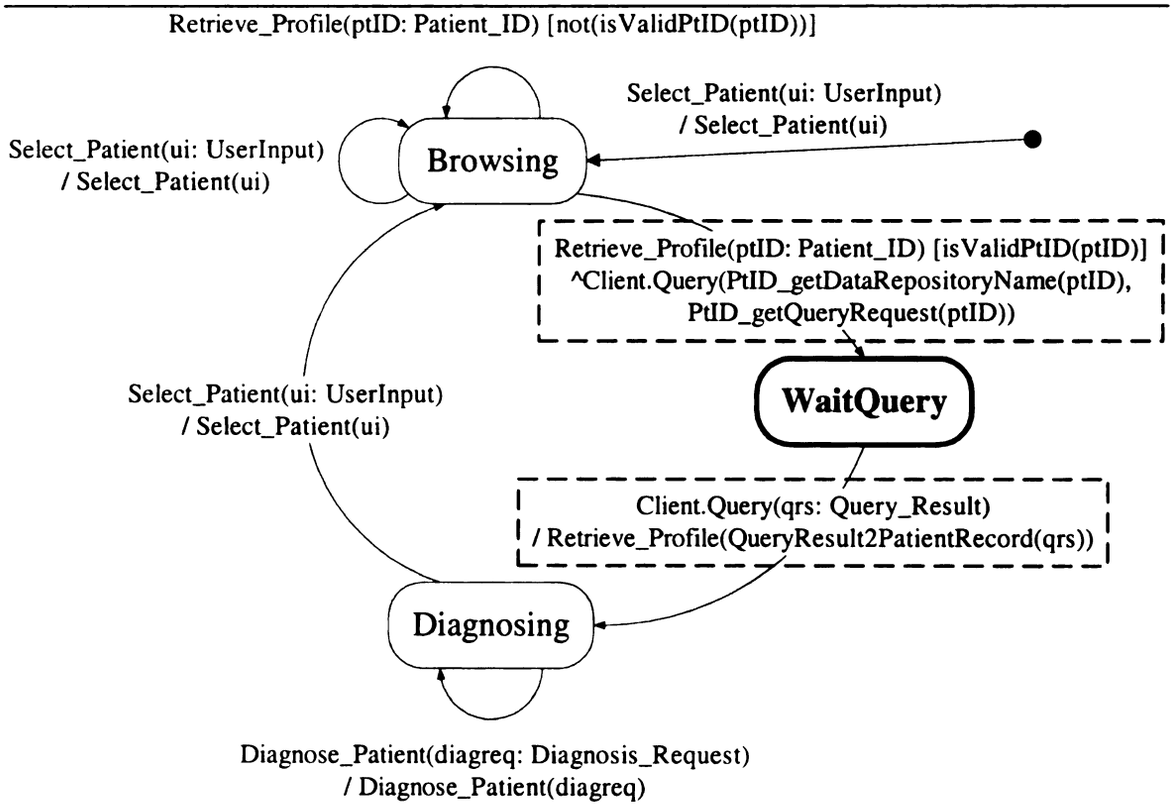


Figure B.2: Refined TRMCS high-level dynamic model (state diagram)

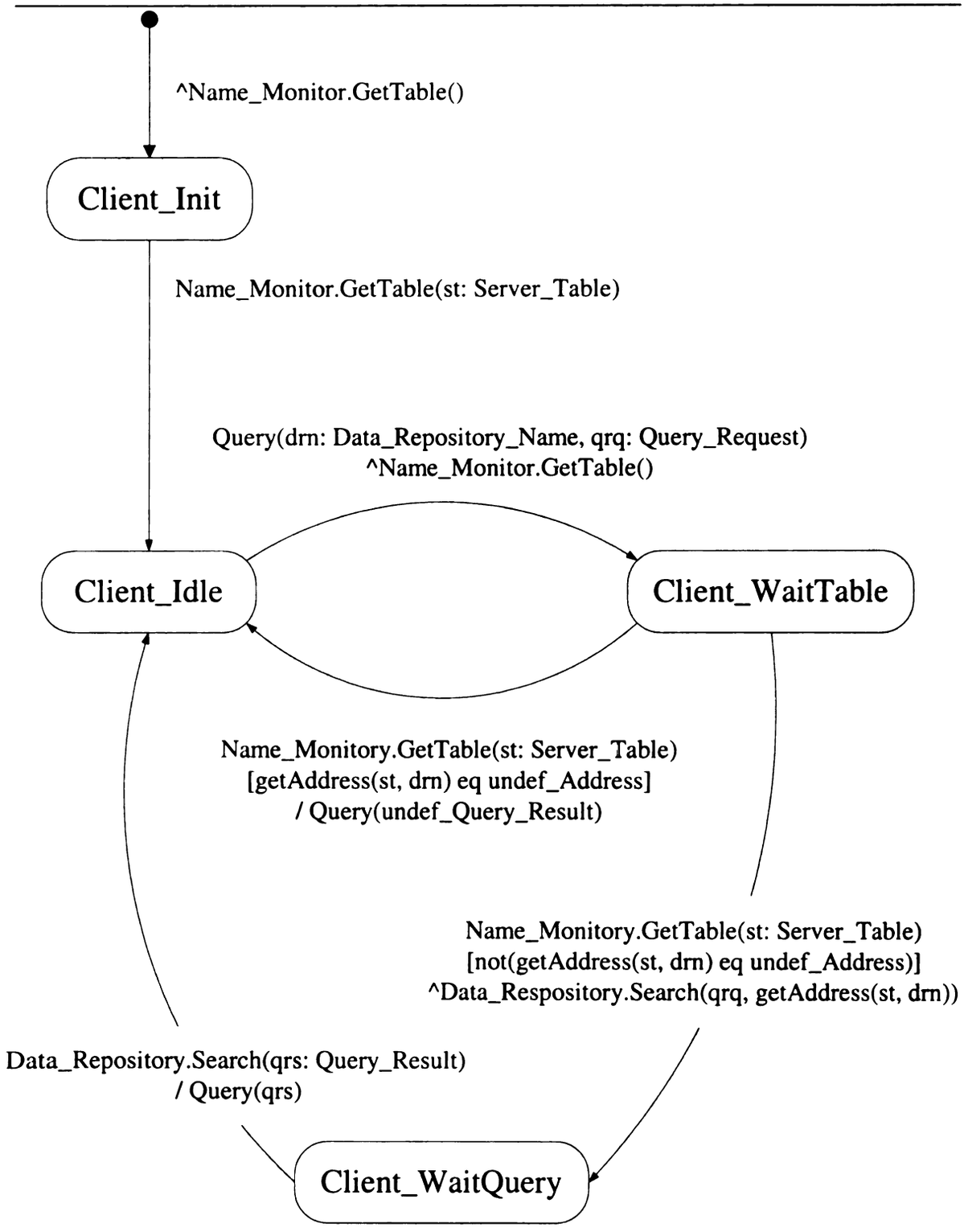


Figure B.3: Dynamic model (state diagram) of the TRMCS Client

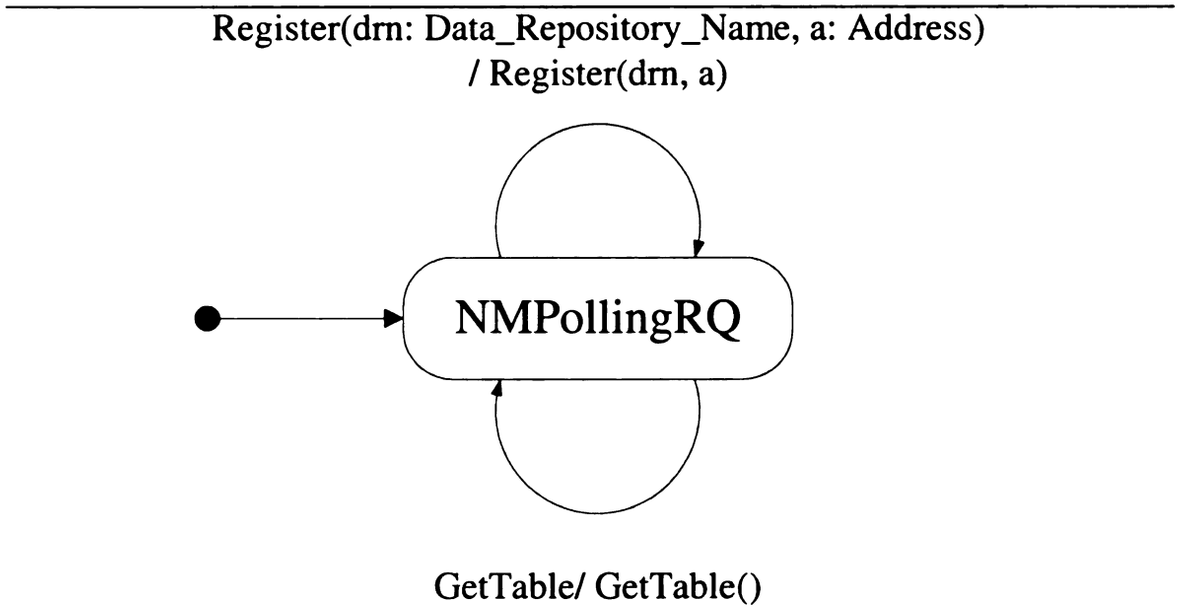


Figure B.4: Dynamic model (state diagram) of the TRMCS Name Monitor

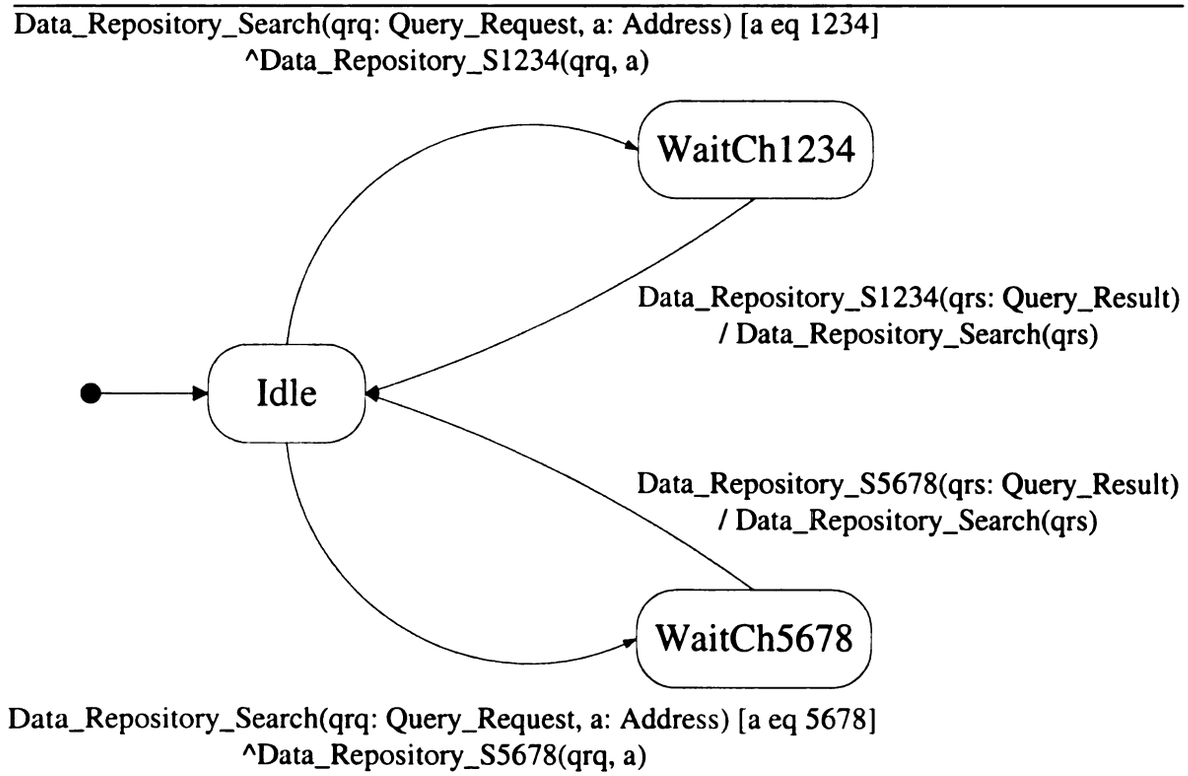


Figure B.5: Dynamic model (state diagram) of the TRMCS Channel

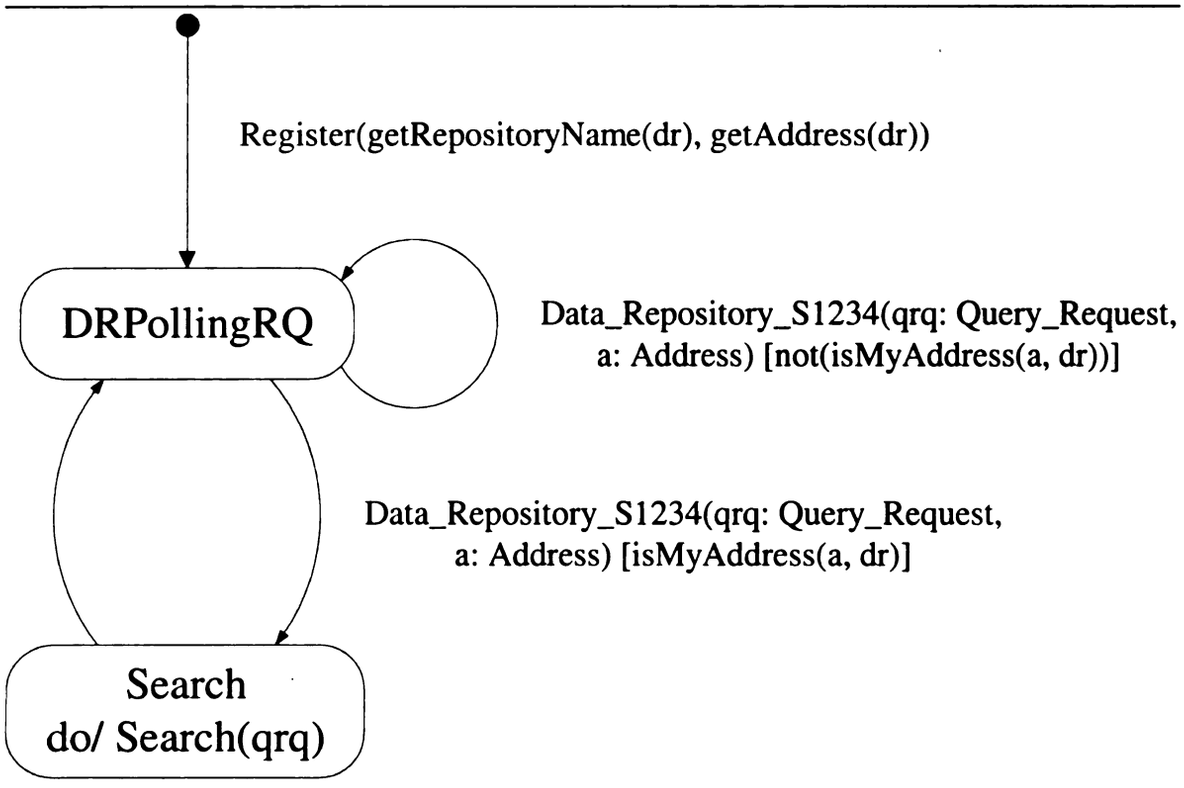


Figure B.6: Dynamic model (state diagram) of first TRMCS Data Repository

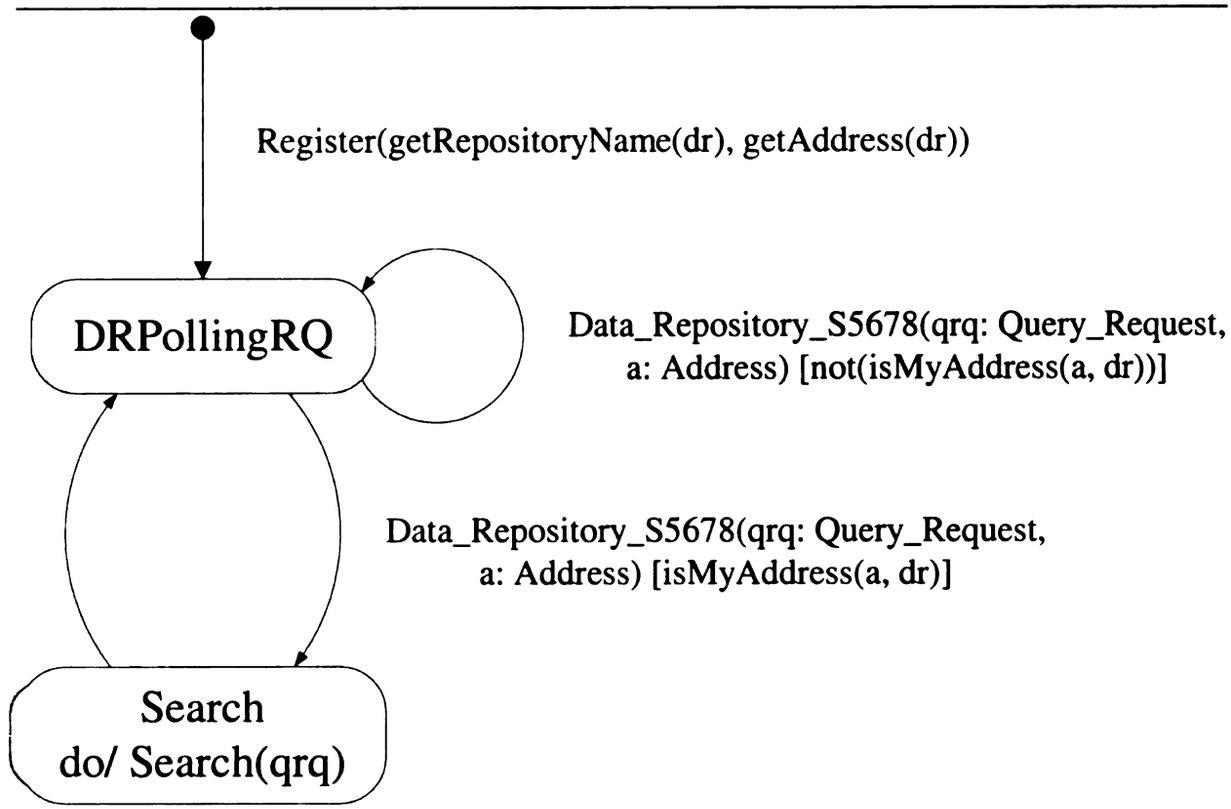


Figure B.7: Dynamic model (state diagram) of second TRMCS Data Repository

Client	`make_Client(empty)`
Name Monitor	`make_Name_Monitor(empty)`
Channel	`make_Channel(empty)`
First Data Repository	`make_Data_Repository(lansing, lansing_db, 1234)`
Second Data Repository	`make_Data_Repository(detroit, detroit_db, 5678)`

Figure B.8: TRMCS LOTOS instantiations

Appendix C

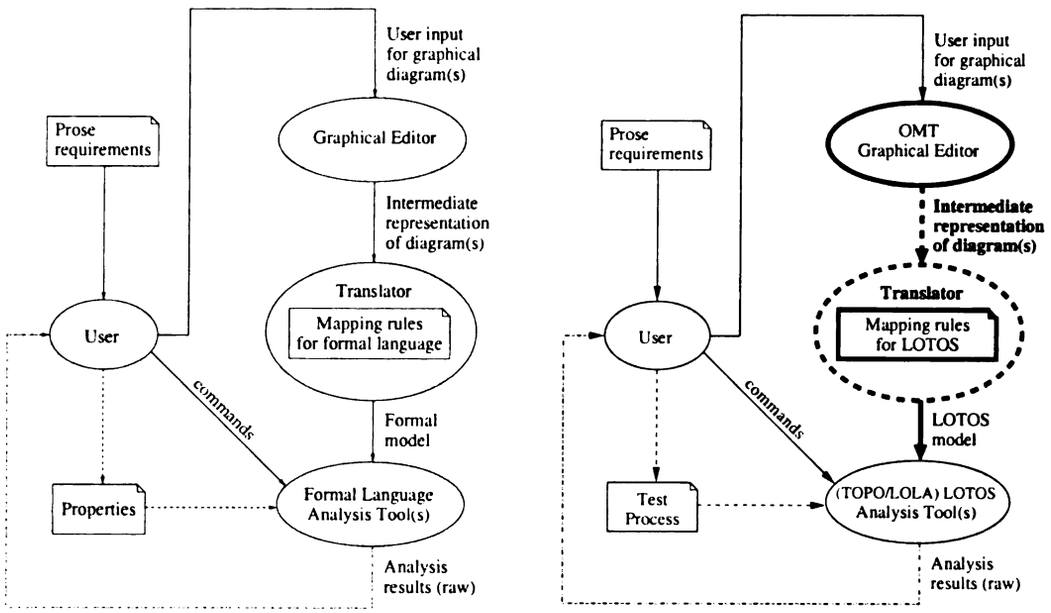
Formal Model Generator Architectures

This Appendix compares both the architecture realized in Wang's approach (Section C.1) and the architecture realized in McUmber's approach (Section C.2) to the general formal model generator architecture first introduced in Chapter 4.

C.1 Wang's Approach

Figure C.1 illustrates the architecture realized in Wang's approach as compared to the general formal model generator architecture (Figure C.1(a)) first introduced in Chapter 4. Wang created a set of **mapping rules** from OMT to LOTOS, shown as a bold solid rectangle in Figure C.1(b). Wang then augmented a simple Motif-based **graphical editor** for OMT class diagrams, VISUALSPECS [156], to add support for OMT state and data flow diagrams (shown as a bold solid oval in Figure C.1(b)). This augmented editor was, in future work, to have output an intermediate (textual) representation of OMT diagrams. Wang intended to write a parser for this intermediate representation in order to generate LOTOS models based on his mapping rules from OMT to LOTOS. These unfinished portions are shown as the bold dashed

intermediate representation data flow arrow and bold dashed **Translator** process oval in Figure C.1(b), respectively. The bold solid **LOTOS model** data flow arrow in Figure C.1(b) represents LOTOS models generated by manual application of Wang's mapping rules. Finally, with this architecture it may be possible to analyze some types of properties against the generated formal model as represented by the dot-dashed data flows and solid **properties** rectangle in Figure C.1(a). As an example of a type of property that can be analyzed with LOTOS analysis tools, such as the TOPO/LOLA LOTOS tool shown in Figure C.1(b), a developer can compose a **test process** (solid rectangle between dashed data flows) with a LOTOS model to check whether the behavior described by the test process is present in the model. An example of test composition is described in Chapter 3, page 36.



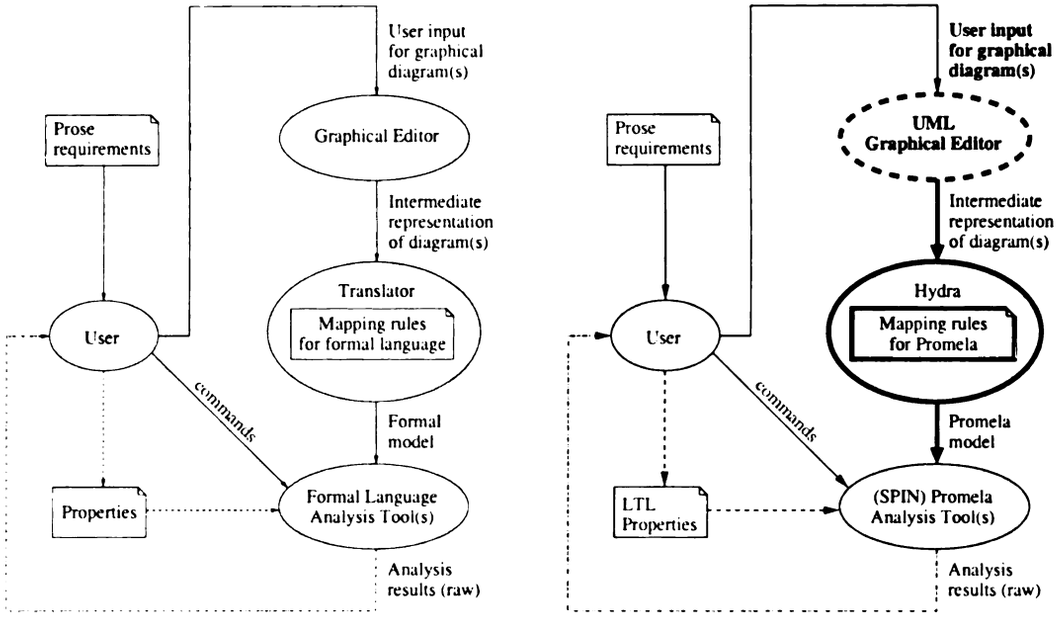
(a) General architecture

(b) Wang's approach

Figure C.1: Architecture for formal model generator realized in Wang's approach

C.2 McUmber's Approach

Figure C.2 illustrates the architecture realized in McUmber's approach as compared to the general formal model generator architecture (Figure C.2(a)) first introduced in Chapter 4. As represented by the bold solid **intermediate representation** data flow in Figure C.2(b), McUmber created a textual representation language of UML class and state diagrams, called *Hydra Intermediate Language* (HIL). As his **Translator**, he created a parser for HIL, Hydra [31], to generate both VHDL and Promela models (each target language had its own set of mapping rules). Figure C.2(b) shows Hydra (a bold solid oval) instantiated with his **mapping rules** for Promela (a bold rectangle). The instantiation shown generates Promela models, as represented by the bold solid **Promela model** data flow arrow. He did not create a graphical editor for UML that would output HIL; thus, the **UML Graphical Editor** is shown as a bold dashed process oval in Figure C.2(b). Finally, with this architecture it may be possible to analyze some types of properties against the generated formal model as represented by the dot-dashed data flows and solid **properties** rectangle in Figure C.2(a). As an example of a type of property that can be analyzed with Promela analysis tools, such as the Spin tool shown in Figure C.2(b), a developer can check **LTL properties** (solid rectangle between dashed data flows) against a Promela model. An example of checking LTL properties is described in Chapter 7.



(a) General architecture

(b) McUmbers's approach

Figure C.2: Architecture for formal model generator realized in McUmber's approach

Appendix D

Adaptive Cruise Control Attributes and Signals

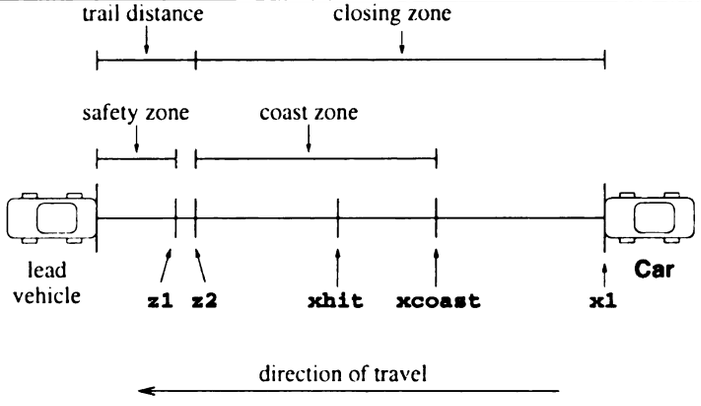


Figure D.1: Variables and zones used by Adaptive Cruise Control Control algorithm.

Scales

- *Time* is measured in seconds.
- *Distance* is measured in tenths of a foot.
- *Speed* is measured in tenths of a foot per second.

Signals

- All signals are asynchronous.
- Signals that carry parameters include each parameter type in the signal signature.

Glossary

- Car: Adaptive Cruise Control-equipped vehicle.
- *Lead vehicle* (also called the *target vehicle*): Vehicle encountered in front of the Car.
- *Trail distance*: Distance the lead vehicle travels in a given amount of time (usually two seconds).
- *Safety zone*: 90% of the specified trail distance.
- *Closing zone*: Zone in which the Radar has acquired the lead vehicle as a target, but the Car has not yet achieved proper trail distance behind the lead vehicle.
- *Coasting*: Continued (forward) movement without throttle.
- *Closing speed*: Speed at which the Car approaches the lead vehicle. In a typical scenario, the Car approaches a slower moving lead vehicle at the set cruising speed, but will not begin decelerating until the Control algorithm's relative speed and distance calculations determine that coasting should be initiated.
- *Coast zone*: Zone in which the Car is coasting.
- *Adjusted speed*: Speed calculated by the Control algorithm. The Control algorithm simulates adjusting the Car's throttle by calculating a new (increased or decreased) speed for the Car.

Car Attributes	
Attribute	**Description**
`setv`	Adjusted speed of the Car as set by the Control algorithm. For modeling purposes, this speed is initially set to 1100 tenths of a foot per second.
`realv`	Current speed of the Car. For modeling purposes, this speed is initially set to 1100 tenths of a foot per second.

Figure D.2: Car attributes

Car Signals	
Signal	**Description**
carv	Developer error: Should have been *getv*. Detected and corrected in Section 7.3.
setspeed(int)	Signal (from the Control) whose parameter is the adjusted speed of the Car as set by the Control algorithm.
unset	Signal from the Control indicating the beginning of the cruise control disengagement process.
getv	Signal from the Radar requesting the current speed of the Car.
getspeed	Signal from the Control requesting the current speed of the Car.

Figure D.3: Car signals

Radar Attributes	
Attribute	**Description**
v	Closing speed of the Car. Used by the Radar simulation algorithm to calculate the next sampled distance to the target vehicle. Calculated as $v = vc - vt$.
vc	Most recent Car speed as obtained from the Car.
vt	(Constant, 900) Speed of the target vehicle.
x	Current distance to the target vehicle. Calculated as $x = x - v$ every time the Radar simulates sampling the distance to the target vehicle (assumed to be once per second). For modeling purposes, this distance is initially set to 4500 tenths of a foot, greater than the range of the Radar (4000 tenths of a foot).
tmode	Boolean flag used by the Radar simulation algorithm. Set to *true* when Radar acquires a target.

Figure D.4: Radar attributes

Radar Signals	
Signal	**Description**
ackcontrol	Acknowledgment signal received from the Control in response to *target*, *dist*, and *lost* signals.
on	Signal received from the Control indicating that the Radar should be turned on.
off	Signal received from the Control indicating that the Radar should be turned off.
carv(int)	Signal (received from the Car) whose parameter is the current speed of the Car.
ackcar	Unused signal.
acksys	Unused signal.

Figure D.5: Radar signals

Control Attributes	
Attribute	**Description**
a	(Constant, 15) Acceleration/deceleration adjustment to the speed of the Car.
closing	Boolean flag used in the Control algorithm. Set to *true* when the Control algorithm's relative speed and distance calculations determine that coasting should be initiated.
setspeed	Developer error: Should have been **setspd**. Detected and corrected in Section 7.3.
setspd	Cruising speed set by the driver.
tinc	(Constant, 1) Increment of time (in seconds) between Radar distance samples.
tmin	(Constant, 2) The number of seconds of target vehicle travel used to calculate the desired trail distance.
v	Calculated closing speed of the Car as it approaches the target vehicle. Calculated as $v = (x1 - x2)/tinc$.
vc	Current speed of the Car. Obtained from the Car.
vt	Calculated current speed of the target vehicle. Calculated as $vt = vc - v$.
x1	Previous sampled distance to the target vehicle as obtained from the Radar. After v, vt, $z2$, and $z1$ are calculated in a single cycle of the Control algorithm, $x1$ is set equal to $x2$ in Control state getxc.
x2	Current sampled distance to the target vehicle as obtained from the Radar.
xcoast	Calculated distance from the target vehicle at which to start coasting in order to achieve the desired trail distance. Calculated as $xcoast = xhit + z2 + tinc * v$.
xhit	Calculated distance from the target vehicle at which to start coasting in order for the Car to exactly match the speed of the target vehicle at zero trail distance. Calculated as $xhit = (v * v)/(2 * a)$.
z1	Calculated closest safe distance from the target vehicle. Boundary of the safety zone. Calculated as 90% of the desired trail distance, or $z1 = z2 - (z2/10)$.
z2	Calculated desired trail distance from the target vehicle. Calculated as $z2 = vt * tmin$.

Figure D.6: Control attributes

Control Signals	
Signal	**Description**
ackcar	Acknowledgment signal received from the Car in response to *setspeed* and *unset* signals.
ackradar	Acknowledgment signal received from the Radar in response to an *on* signal.
brakes	External signal that simulates the driver has applied the brakes.
carspeed(int)	Signal (received from the Car) whose parameter is the current speed of the Car.
dist(int)	Signal (received from the Radar) whose parameter is the current distance from the target vehicle.
lost	Signal received from the Radar indicating that a previously acquired target has been lost.
set	External signal that simulates the driver has set the desired cruising speed.
target	Signal received from the Radar indicating that a target has been acquired.

Figure D.7: Control signals

Appendix E

Adaptive Cruise Control Control State Diagram

The modified Control state diagram as described in Section 7.4.4 includes transitions that handle the *brakes* message (shown as dashed arcs). The missing transition from state `caroff` to handle the *carspeed* message is shown in bold. Due to printing limitations, the transitions have been annotated with unique numbers that correspond to the legend in Figure E.1.

```
1  ^_SYSTEMCLASS_.ready
2  set [] / ^Car.getspeed
3  carspeed(setspd) [] / ^Car.setspeed(setspd)
4  ackcar [] / ^Radar.on
5  ackradar [] /
6  target [] / ^Radar.ackcontrol
7  lost [] /
8  ackcar [] / ^Radar.ackcontrol
9  dist(x1) [] / xcoast :=0
10 lost [] /
11 dist(x2) [] /
12 carspeed(vc) [] /
13 [~closing] / xhit:=(v*v)/(2*a); xcoast:=xhit+z2+tinc*v
14 [closing] /
15 [x1 >= z1] /
16 [xhit <= x1] /
17 [xhit > x1] /
18 [] /
19 [closing] /
20 [~closing] /
21 [x1 > xcoast] /
22 [x1 <= xcoast] / closing:=1 ^Car.setspeed(vt)
23 ackcar [] /
24 brakes [] /
25 brakes [] /
26 brakes [] /
27 brakes [] /
28 brakes [] /
29 brakes [] /
30 carspeed(vc) [] /
31 lost [] /
32 dist(x1) [] /
33 target [] /
34 ackcar [] /
35 [x1 < z1] / ^Car.unset
36 brakes [] /
37 ackcar [] / ^Radar.off
```

Figure E.1: Legend for transitions in Figure E.2

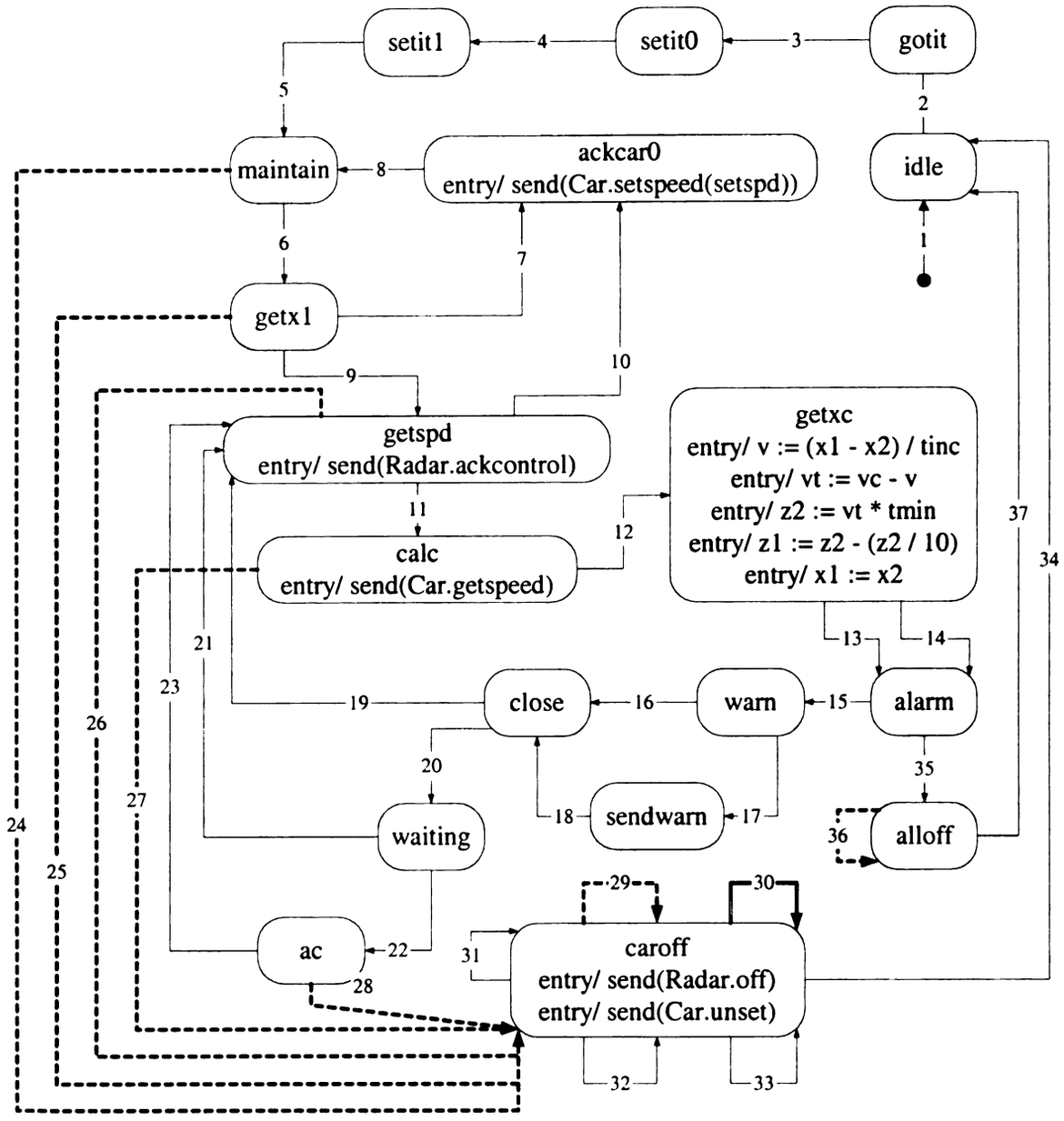


Figure E.2: State diagram for the Adaptive Cruise Control class Control

Appendix F

Hydra-Generated Promela Code for Adaptive Cruise Control

This appendix contains the Promela model for class **Control** and its state diagram. (This code has been excerpted from the Promela model automatically generated for the **Adaptive Cruise Control** UML diagrams in Chapter 7.) Extra linebreaks have been added in order to ensure that the formal model is readable; no attempt has been made to add additional comments to annotate the formal model. The inclusion of the formal model is intended to illustrate the correspondence between the UML diagrams and the Promela code.

```
1 typedef Control_T {
2           int a;
3           int tmin;
4           int vc;
5           int x1;
6           int vt;
7           int tinc;
8           int x2;
9           bool closing;
```

```
10          int v;
11          int z1;
12          int xhit;
13          int z2;
14          int setspd;
15          int xcoast;
16          }
17 Control_T Control_V;
18 chan Control_q=[5] of {mtype};
19 chan Control_carspeed_p1=[5] of {int};
20 chan Control_dist_p1=[5] of {int};
21
22 proctype Control()
23 {
24 mtype m;
25 int dummy;
26      Control_V.a = 15;
27      Control_V.tmin = 2;
28      Control_V.tinc = 1;
29 /*  Init state      */
30 /* Initial actions / messages */
31      _SYSTEMCLASS__q!ready;
32      goto idle;
33 /* State idle   */
34 idle:     printf("in state Control.idle\n");
35      atomic {if :: !t?[free] -> t!free :: else skip fi;}
36      if
37      :: Control_q?set -> t?free;  Car_q!getspeed; goto gotit
38      fi;
```

```
39 /* State gotit   */
40 gotit:    printf("in state Control.gotit\n");
41      atomic {if :: !t?[free] -> t!free :: else skip fi;}
42      if
43      :: atomic{Control_q?carspeed ->
44         Control_carspeed_p1?Control_V.setspd} -> t?free;
45         atomic{Car_setspeed_p1!Control_V.setspd; Car_q!setspeed};
46         goto setit0
47      fi;
48 /* State caroff   */
49 caroff:   printf("in state Control.caroff\n");
50 /* entry actions */
51      atomic{
52         Radar_q!off;Car_q!unset;
53      }
54      atomic {if :: !t?[free] -> t!free :: else skip fi;}
55      if
56      :: atomic{Control_q?carspeed ->
57         Control_carspeed_p1?Control_V.vc} -> t?free;  goto caroff
58      :: Control_q?ackcar -> t?free;  goto idle
59      :: atomic{Control_q?dist -> Control_dist_p1?Control_V.x1} ->
60         t?free;  goto caroff
61      :: Control_q?brakes -> t?free;  goto caroff
62      :: Control_q?target -> t?free;  goto caroff
63      :: Control_q?lost -> t?free;  goto caroff
64      fi;
65 /* State getxc   */
66 getxc:    printf("in state Control.getxc\n");
67 /* entry actions */
```

```
68      atomic{
69         Control_V.v=(Control_V.x1-Control_V.x2)/Control_V.tinc;
70         Control_V.vt=Control_V.vc-Control_V.v;
71         Control_V.z2=Control_V.vt*Control_V.tmin;
72         Control_V.z1=Control_V.z2-Control_V.z2/10;
73         Control_V.x1=Control_V.x2;
74      }
75 getxc_G:
76      atomic {if :: !t?[free] -> t!free :: else skip fi;}
77      if
78      :: 1 -> t?free; if
79         ::  Control_V.closing ->  goto alarm
80         ::  !Control_V.closing ->
81            Control_V.xhit=(Control_V.v*Control_V.v)/(2*Control_V.a);
82            Control_V.xcoast=Control_V.xhit+Control_V.z2+
83              Control_V.tinc*Control_V.v;
84            goto alarm
85         :: else -> goto getxc_G
86         fi
87      fi;
88 /* State alarm   */
89 alarm:    printf("in state Control.alarm\n");
90 alarm_G:
91      atomic {if :: !t?[free] -> t!free :: else skip fi;}
92      if
93      :: 1 -> t?free; if
94         :: Control_V.x1<Control_V.z1 ->  Car_q!unset; goto alloff
95         ::  Control_V.x1>=Control_V.z1 ->  goto warn
96         :: else -> goto alarm_G
```

```
97          fi
98      fi;
99  /* State alloff   */
100 alloff:   printf("in state Control.alloff\n");
101     atomic {if :: !t?[free] -> t!free :: else skip fi;}
102     if
103     :: Control_q?ackcar -> t?free;  Radar_q!off; goto idle
104     :: Control_q?brakes -> t?free;  goto alloff
105     fi;
106 /* State warn   */
107 warn:     printf("in state Control.warn\n");
108 warn_G:
109     atomic {if :: !t?[free] -> t!free :: else skip fi;}
110     if
111     :: 1 -> t?free; if
112        ::  Control_V.xhit>Control_V.x1 ->  goto sendwarn
113        ::  Control_V.xhit<=Control_V.x1 ->  goto close
114        :: else -> goto warn_G
115        fi
116     fi;
117 /* State close   */
118 close:    printf("in state Control.close\n");
119 close_G:
120     atomic {if :: !t?[free] -> t!free :: else skip fi;}
121     if
122     :: 1 -> t?free; if
123        ::  !Control_V.closing ->  goto waiting
124        ::  Control_V.closing ->  goto getspd
125        :: else -> goto close_G
```

```
126         fi
127      fi;
128 /* State sendwarn   */
129 sendwarn: printf("in state Control.sendwarn\n");
130      atomic {if :: !t?[free] -> t!free :: else skip fi;}
131      if
132      :: 1 -> t?free;  goto close
133      fi;
134 /* State waiting   */
135 waiting:  printf("in state Control.waiting\n");
136 waiting_G:
137      atomic {if :: !t?[free] -> t!free :: else skip fi;}
138      if
139      :: 1 -> t?free; if
140         ::  Control_V.x1<=Control_V.xcoast ->
141            Control_V.closing=1;
142            atomic{Car_setspeed_p1!Control_V.vt; Car_q!setspeed};
143            goto ac
144         ::  Control_V.x1>Control_V.xcoast ->  goto getspd
145         :: else -> goto waiting_G
146         fi
147      fi;
148 /* State ac   */
149 ac:       printf("in state Control.ac\n");
150      atomic {if :: !t?[free] -> t!free :: else skip fi;}
151      if
152      :: Control_q?ackcar -> t?free;  goto getspd
153      :: Control_q?brakes -> t?free;  goto caroff
154      fi;
```

```
155 /* State getspd   */
156 getspd:   printf("in state Control.getspd\n");
157 /* entry actions */
158       atomic{
159          Radar_q!ackcontrol;
160       }
161       atomic {if :: !t?[free] -> t!free :: else skip fi;}
162       if
163       :: atomic{Control_q?dist -> Control_dist_p1?Control_V.x2} ->
164          t?free;   goto calc
165       :: Control_q?brakes -> t?free;  goto caroff
166       :: Control_q?lost -> t?free;  goto ackcar0
167       fi;
168 /* State calc   */
169 calc:     printf("in state Control.calc\n");
170 /* entry actions */
171       atomic{
172          Car_q!getspeed;
173       }
174       atomic {if :: !t?[free] -> t!free :: else skip fi;}
175       if
176       :: atomic{Control_q?carspeed ->
177          Control_carspeed_p1?Control_V.vc} -> t?free;  goto getxc
178       :: Control_q?brakes -> t?free;  goto caroff
179       fi;
180 /* State setit0   */
181 setit0:   printf("in state Control.setit0\n");
182       atomic {if :: !t?[free] -> t!free :: else skip fi;}
183       if
```

```
184      :: Control_q?ackcar -> t?free;  Radar_q!on; goto setit1
185      fi;
186 /* State setit1   */
187 setit1:   printf("in state Control.setit1\n");
188      atomic {if :: !t?[free] -> t!free :: else skip fi;}
189      if
190      :: Control_q?ackradar -> t?free;  goto maintain
191      fi;
192 /* State ackcar0   */
193 ackcar0:  printf("in state Control.ackcar0\n");
194 /* entry actions */
195      atomic{
196         atomic{Car_setspeed_p1!Control_V.setspd; Car_q!setspeed};
197      }
198      atomic {if :: !t?[free] -> t!free :: else skip fi;}
199      if
200      :: Control_q?ackcar -> t?free;  Radar_q!ackcontrol; goto
201         maintain
202      fi;
203 /* State maintain   */
204 maintain: printf("in state Control.maintain\n");
205      atomic {if :: !t?[free] -> t!free :: else skip fi;}
206      if
207      :: Control_q?brakes -> t?free;  goto caroff
208      :: Control_q?target -> t?free;  Radar_q!ackcontrol; goto getx1
209      fi;
210 /* State getx1   */
211 getx1:    printf("in state Control.getx1\n");
212      atomic {if :: !t?[free] -> t!free :: else skip fi;}
```

```
213       if
214       :: atomic{Control_q?dist -> Control_dist_p1?Control_V.x1} ->
215          t?free;
216          Control_V.xcoast=0;Control_V.closing=0;  goto getspd
217       :: Control_q?brakes -> t?free;  goto caroff
218       :: Control_q?lost -> t?free;  goto ackcar0
219       fi;
220 exit:     skip
221 }
```

A

H

fo

Th

Co

iza

mo

op

pr

ter

hov

G

Appendix G

Hydra-Generated Promela Code for *Producer-Consumer*

This Appendix contains four versions of the formal (Promela) model for the *Producer-Consumer* example. The first version is the baseline, generated according to formalization rules and without pushing additional diagram information into the formal model. The next three versions were generated according to the three breadcrumb options, states, transitions, and both, respectively. In the latter three versions, each `printf` for `STATE` and `TRANSITION` information was manually wrapped at the character '`@`' for readability in this Appendix (*e.g.*, lines 37–38 and 44–47 in Section G.4); however, when generated, a `printf` statement appears all on one line of code.

G.1 Baseline

```
1 #define min(x,y) (x<y->x:y)
2 #define max(x,y) (x>y->x:y)
3 chan evq=[10] of {mtype,int};
4 chan evt=[10] of {mtype,int};
5 chan wait=[10] of {int,mtype};
```

```
6 mtype={demand, OK, supply};
7 chan _SYSTEMCLASS__q=[5] of {mtype};
8 typedef Producer_T {
9           int limited_ed;
10          int num_made;
11          }
12 Producer_T Producer_V;
13 chan Producer_q=[5] of {mtype};
14 typedef Consumer_T {
15          int edition_num;
16          }
17 Consumer_T Consumer_V;
18 chan Consumer_q=[5] of {mtype};
19 chan Consumer_supply_p1=[5] of {int};
20 chan t=[1] of {mtype};
21 mtype={free};
22 active proctype _SYSTEMCLASS_()
23 {
24 mtype m;
25 int dummy;
26 /*  Init state      */
27          goto Create_Producer;
28 /* State Create_Producer   */
29 Create_Producer: printf("in state _SYSTEMCLASS_.Create_Producer\n");
30 /* entry actions */
31          atomic{
32             run Producer();
33          }
34          atomic {if :: !t?[free] -> t!free :: else skip fi;}
```

```
35          if
36          :: _SYSTEMCLASS__q?OK -> t?free;  goto Create_Consumer
37          fi;
38 /* State Create_Consumer   */
39 Create_Consumer: printf("in state _SYSTEMCLASS_.Create_Consumer\n");
40 /* entry actions */
41          atomic{
42             run Consumer();
43          }
44          atomic {if :: !t?[free] -> t!free :: else skip fi;}
45          if
46          :: _SYSTEMCLASS__q?OK -> t?free;  goto Done
47          fi;
48 /* State Done   */
49 Done:     printf("in state _SYSTEMCLASS_.Done\n");
50          atomic {if :: !t?[free] -> t!free :: else skip fi;}
51          if
52          :: skip -> false
53          fi;
54 exit:     skip
55 }
56
57
58 proctype Producer()
59 {
60 mtype m;
61 int dummy;
62          Producer_V.limited_ed = 5;
63 /*  Init state      */
```

```
64 /* Initial actions / messages */
65           _SYSTEMCLASS__q!OK;
66           goto Waiting_For_Demand;
67 /* State Waiting_For_Demand   */
68 Waiting_For_Demand: printf("in state Producer.Waiting_For_Demand\n");
69 Waiting_For_Demand_G:
70           atomic {if :: !t?[free] -> t!free :: else skip fi;}
71           if
72           :: Producer_q?demand -> t?free; if
73              ::  Producer_V.num_made<Producer_V.limited_ed ->
74                 Producer_V.num_made=Producer_V.num_made+1; goto
75                 Advertise
76              :: else -> goto Waiting_For_Demand_G
77              fi
78           :: 1 -> t?free; if
79              ::  Producer_V.num_made>=Producer_V.limited_ed ->  goto
80                 Sorry_Sold_Out
81              :: else -> goto Waiting_For_Demand_G
82              fi
83           fi;
84 /* State Sorry_Sold_Out   */
85 Sorry_Sold_Out: printf("in state Producer.Sorry_Sold_Out\n");
86           atomic {if :: !t?[free] -> t!free :: else skip fi;}
87           if
88           :: skip -> false
89           fi;
90 /* State Advertise   */
91 Advertise:printf("in state Producer.Advertise\n");
92           atomic {if :: !t?[free] -> t!free :: else skip fi;}
```

```
93            if
94            :: 1 -> t?free;
95               atomic{Consumer_supply_p1!Producer_V.num_made;
96               Consumer_q!supply}; goto Waiting_For_Demand
97            fi;
98 exit:     skip
99 }
100
101
102 proctype Consumer()
103 {
104 mtype m;
105 int dummy;
106 /*  Init state      */
107 /* Initial actions / messages */
108           _SYSTEMCLASS__q!OK;
109           goto Have_Money_Will_Spend;
110 /* State Have_Money_Will_Spend   */
111 Have_Money_Will_Spend:
112           printf("in state Consumer.Have_Money_Will_Spend\n");
113           atomic {if :: !t?[free] -> t!free :: else skip fi;}
114           if
115           :: 1 -> t?free;  Producer_q!demand; goto Waiting_For_Supply
116           fi;
117 /* State Waiting_For_Supply   */
118 Waiting_For_Supply: printf("in state Consumer.Waiting_For_Supply\n");
119           atomic {if :: !t?[free] -> t!free :: else skip fi;}
120           if
121           :: atomic{Consumer_q?supply ->
```

```
122                Consumer_supply_p1?Consumer_V.edition_num} -> t?free;
123                goto Have_Money_Will_Spend
124             fi;
125 exit:       skip
126 }
127
128
129
130 /* This is the universal event dispatcher routine */
131 proctype event(mtype msg)
132 {
133         mtype type;
134         int pid;
135
136         atomic {
137         do
138         :: evq??[eval(msg),pid] ->
139            evq??eval(msg),pid;
140            evt!msg,pid;
141            do
142            :: if
143               :: evq??[type,eval(pid)] -> evq??type,eval(pid)
144               :: else break;
145               fi
146            od
147         :: else -> break
148         od}
149 exit:         skip
150 }
```

G.2 UML States

```
1 #define min(x,y) (x<y->x:y)
2 #define max(x,y) (x>y->x:y)
3 chan evq=[10] of {mtype,int};
4 chan evt=[10] of {mtype,int};
5 chan wait=[10] of {int,mtype};
6 mtype={demand, OK, supply};
7 chan _SYSTEMCLASS__q=[5] of {mtype};
8 typedef Producer_T {
9           int limited_ed;
10          int num_made;
11          }
12 Producer_T Producer_V;
13 chan Producer_q=[5] of {mtype};
14 typedef Consumer_T {
15          int edition_num;
16          }
17 Consumer_T Consumer_V;
18 chan Consumer_q=[5] of {mtype};
19 chan Consumer_supply_p1=[5] of {int};
20 chan t=[1] of {mtype};
21 mtype={free};
22 active proctype _SYSTEMCLASS_()
23 {
24 mtype m;
25 int dummy;
26 /*  Init state      */
27          goto Create_Producer;
```

```
28 /* State Create_Producer   */
29 Create_Producer: printf("in state _SYSTEMCLASS_.Create_Producer\n");
30 /* entry actions */
31           atomic{
32              printf("STATE@_SYSTEMCLASS_@209130692280324
33                           @Create_Producer@209130704732175\n");
34              run Producer();
35           }
36           atomic {if :: !t?[free] -> t!free :: else skip fi;}
37           if
38           :: _SYSTEMCLASS__q?OK -> t?free;  goto Create_Consumer
39           fi;
40 /* State Create_Consumer   */
41 Create_Consumer: printf("in state _SYSTEMCLASS_.Create_Consumer\n");
42 /* entry actions */
43           atomic{
44              printf("STATE@_SYSTEMCLASS_@209130692280324
45                           @Create_Consumer@209130704732176\n");
46              run Consumer();
47           }
48           atomic {if :: !t?[free] -> t!free :: else skip fi;}
49           if
50           :: _SYSTEMCLASS__q?OK -> t?free;  goto Done
51           fi;
52 /* State Done   */
53 Done:     printf("in state _SYSTEMCLASS_.Done\n");
54 /* entry actions */
55           atomic{
56              printf("STATE@_SYSTEMCLASS_@209130692280324
```

```
57                          @Done@209130706632727\n");
58          }
59          atomic {if :: !t?[free] -> t!free :: else skip fi;}
60          if
61          :: skip -> false
62          fi;
63 exit:    skip
64 }
65
66
67 proctype Producer()
68 {
69 mtype m;
70 int dummy;
71          Producer_V.limited_ed = 5;
72 /*  Init state      */
73 /* Initial actions / messages */
74          _SYSTEMCLASS__q!OK;
75          goto Waiting_For_Demand;
76 /* State Waiting_For_Demand   */
77 Waiting_For_Demand: printf("in state Producer.Waiting_For_Demand\n");
78 /* entry actions */
79          atomic{
80             printf("STATE@Producer@209130692280326
81                          @Waiting_For_Demand@209130753228840\n");
82          }
83 Waiting_For_Demand_G:
84          atomic {if :: !t?[free] -> t!free :: else skip fi;}
85          if
```

```
86          :: Producer_q?demand -> t?free; if
87             ::  Producer_V.num_made<Producer_V.limited_ed ->
88                Producer_V.num_made=Producer_V.num_made+1; goto
89                Advertise
90             :: else -> goto Waiting_For_Demand_G
91             fi
92          :: 1 -> t?free; if
93             ::  Producer_V.num_made>=Producer_V.limited_ed ->  goto
94                Sorry_Sold_Out
95             :: else -> goto Waiting_For_Demand_G
96             fi
97          fi;
98 /* State Sorry_Sold_Out   */
99 Sorry_Sold_Out: printf("in state Producer.Sorry_Sold_Out\n");
100 /* entry actions */
101          atomic{
102             printf("STATE@Producer@209130692280326
103                          @Sorry_Sold_Out@209130817257516\n");
104          }
105          atomic {if :: !t?[free] -> t!free :: else skip fi;}
106          if
107          :: skip -> false
108          fi;
109 /* State Advertise   */
110 Advertise:printf("in state Producer.Advertise\n");
111 /* entry actions */
112          atomic{
113             printf("STATE@Producer@209130692280326
114                          @Advertise@209130817257519\n");
```

```
115             }
116             atomic {if :: !t?[free] -> t!free :: else skip fi;}
117             if
118             :: 1 -> t?free;
119                atomic{Consumer_supply_p1!Producer_V.num_made;
120                Consumer_q!supply}; goto Waiting_For_Demand
121             fi;
122 exit:       skip
123 }
124
125
126 proctype Consumer()
127 {
128 mtype m;
129 int dummy;
130 /*  Init state      */
131 /* Initial actions / messages */
132             _SYSTEMCLASS__q!OK;
133             goto Have_Money_Will_Spend;
134 /* State Have_Money_Will_Spend   */
135 Have_Money_Will_Spend:
136             printf("in state Consumer.Have_Money_Will_Spend\n");
137 /* entry actions */
138             atomic{
139                printf("STATE@Consumer@209130692280330
140                                @Have_Money_Will_Spend@209130736058394\n");
141             }
142             atomic {if :: !t?[free] -> t!free :: else skip fi;}
143             if
```

```
144           :: 1 -> t?free;  Producer_q!demand; goto Waiting_For_Supply
145           fi;
146 /* State Waiting_For_Supply   */
147 Waiting_For_Supply: printf("in state Consumer.Waiting_For_Supply\n");
148 /* entry actions */
149           atomic{
150              printf("STATE@Consumer@209130692280330
151                           @Waiting_For_Supply@209130747068447\n");
152           }
153           atomic {if :: !t?[free] -> t!free :: else skip fi;}
154           if
155           :: atomic{Consumer_q?supply ->
156              Consumer_supply_p1?Consumer_V.edition_num} -> t?free;
157              goto Have_Money_Will_Spend
158           fi;
159 exit:     skip
160 }
161
162
163
164 /* This is the universal event dispatcher routine */
165 proctype event(mtype msg)
166 {
167         mtype type;
168         int pid;
169
170         atomic {
171         do
172         :: evq??[eval(msg),pid] ->
```

```
173             evq??eval(msg),pid;
174             evt!msg,pid;
175             do
176             :: if
177                :: evq??[type,eval(pid)] -> evq??type,eval(pid)
178                :: else break;
179                fi
180             od
181          :: else -> break
182          od}
183 exit:         skip
184 }
```

G.3 UML Transitions

```
1 #define min(x,y) (x<y->x:y)
2 #define max(x,y) (x>y->x:y)
3 chan evq=[10] of {mtype,int};
4 chan evt=[10] of {mtype,int};
5 chan wait=[10] of {int,mtype};
6 mtype={demand, OK, supply};
7 chan _SYSTEMCLASS__q=[5] of {mtype};
8 typedef Producer_T {
9            int limited_ed;
10           int num_made;
11           }
12 Producer_T Producer_V;
13 chan Producer_q=[5] of {mtype};
14 typedef Consumer_T {
```

```
15          int edition_num;
16          }
17 Consumer_T Consumer_V;
18 chan Consumer_q=[5] of {mtype};
19 chan Consumer_supply_p1=[5] of {int};
20 chan t=[1] of {mtype};
21 mtype={free};
22 active proctype _SYSTEMCLASS_()
23 {
24 mtype m;
25 int dummy;
26 /*  Init state      */
27 /* Initial actions / messages */
28          printf("TRANSITION@2091307066327210_SYSTEMCLASS_
29                              @2091306922803240Initial
30                              @2091307047321740Create_Producer
31                              @2091307047321750modelstart\n");
32          goto Create_Producer;
33 /* State Create_Producer   */
34 Create_Producer: printf("in state _SYSTEMCLASS_.Create_Producer\n");
35 /* entry actions */
36          atomic{
37             run Producer();
38          }
39          atomic {if :: !t?[free] -> t!free :: else skip fi;}
40          if
41          :: _SYSTEMCLASS__q?OK -> t?free;
42             printf("TRANSITION@2091307066327230_SYSTEMCLASS_
43                                  @2091306922803240Create_Producer
```

```
44                                  @2091307047321750Create_Consumer
45                                  @2091307047321760OK\n");
46              goto Create_Consumer
47          fi;
48 /* State Create_Consumer   */
49 Create_Consumer: printf("in state _SYSTEMCLASS_.Create_Consumer\n");
50 /* entry actions */
51          atomic{
52              run Consumer();
53          }
54          atomic {if :: !t?[free] -> t!free :: else skip fi;}
55          if
56          :: _SYSTEMCLASS__q?OK -> t?free;
57              printf("TRANSITION@2091307066327250_SYSTEMCLASS_
58                                  @2091306922803240Create_Consumer
59                                  @2091307047321760Done
60                                  @2091307066327270OK\n");
61              goto Done
62          fi;
63 /* State Done   */
64 Done:       printf("in state _SYSTEMCLASS_.Done\n");
65          atomic {if :: !t?[free] -> t!free :: else skip fi;}
66          if
67          :: skip -> false
68          fi;
69 exit:       skip
70 }
71
72
```

```
73 proctype Producer()
74 {
75 mtype m;
76 int dummy;
77           Producer_V.limited_ed = 5;
78 /*  Init state      */
79 /* Initial actions / messages */
80           printf("TRANSITION@2091307532288837@Producer
81                                 @2091306922803260@Initial
82                                 @2091307532288839@Waiting_For_Demand
83                                 @2091307532288840@modelstart\n");
84           _SYSTEMCLASS__q!OK;
85           goto Waiting_For_Demand;
86 /* State Waiting_For_Demand   */
87 Waiting_For_Demand: printf("in state Producer.Waiting_For_Demand\n");
88 Waiting_For_Demand_G:
89           atomic {if :: !t?[free] -> t!free :: else skip fi;}
90           if
91           :: Producer_q?demand -> t?free; if
92              ::  Producer_V.num_made<Producer_V.limited_ed ->
93                 printf("TRANSITION@2091308172575170Producer
94                                       @2091306922803260@Waiting_For_Demand
95                                       @2091307532288400@Advertise
96                                       @2091308172575190@demand\n");
97                 Producer_V.num_made=Producer_V.num_made+1;
98                 goto Advertise
99              :: else -> goto Waiting_For_Demand_G
100             fi
101          :: 1 -> t?free; if
```

```
102             ::  Producer_V.num_made>=Producer_V.limited_ed ->
103                printf("TRANSITION@2091308172575l4@Producer
104                                  @2091306922803 26@Waiting_For_Demand
105                                  @2091307532288 40@Sorry_Sold_Out
106                                  @2091308172575 16@\n");
107                goto Sorry_Sold_Out
108             :: else -> goto Waiting_For_Demand_G
109             fi
110          fi;
111 /* State Sorry_Sold_Out   */
112 Sorry_Sold_Out: printf("in state Producer.Sorry_Sold_Out\n");
113          atomic {if :: !t?[free] -> t!free :: else skip fi;}
114          if
115          :: skip -> false
116          fi;
117 /* State Advertise   */
118 Advertise:printf("in state Producer.Advertise\n");
119          atomic {if :: !t?[free] -> t!free :: else skip fi;}
120          if
121          :: 1 -> t?free;
122             printf("TRANSITION@2091343017410 58@Producer
123                                  @2091306922803 26@Advertise
124                                  @2091308172575 19@Waiting_For_Demand
125                                  @2091307532288 40@\n");
126             atomic{Consumer_supply_p1!Producer_V.num_made;
127             Consumer_q!supply}; goto Waiting_For_Demand
128          fi;
129 exit:    skip
130 }
```

```
131
132
133 proctype Consumer()
134 {
135 mtype m;
136 int dummy;
137 /*  Init state      */
138 /* Initial actions / messages */
139           printf("TRANSITION@2091307403182360Consumer
140                                 @2091306922803300Initial
141                                 @2091307360583930Have_Money_Will_Spend
142                                 @2091307360583940modelstart\n");
143           _SYSTEMCLASS__q!OK;
144           goto Have_Money_Will_Spend;
145 /* State Have_Money_Will_Spend   */
146 Have_Money_Will_Spend:
147           printf("in state Consumer.Have_Money_Will_Spend\n");
148           atomic {if :: !t?[free] -> t!free :: else skip fi;}
149           if
150           :: 1 -> t?free;
151              printf("TRANSITION@2091307480514880Consumer
152                                    @2091306922803300Have_Money_Will_Spend
153                                    @2091307360583940Waiting_For_Supply
154                                    @2091307470684470\n");
155              Producer_q!demand; goto Waiting_For_Supply
156           fi;
157 /* State Waiting_For_Supply   */
158 Waiting_For_Supply: printf("in state Consumer.Waiting_For_Supply\n");
159           atomic {if :: !t?[free] -> t!free :: else skip fi;}
```

```
160             if
161             :: atomic{Consumer_q?supply ->
162                Consumer_supply_p1?Consumer_V.edition_num} -> t?free;
163                printf("TRANSITION@2091307480514900Consumer
164                                    @2091306922803300Waiting_For_Supply
165                                    @2091307470684470Have_Money_Will_Spend
166                                    @2091307360583940supply(edition_num)\n");
167                goto Have_Money_Will_Spend
168             fi;
169 exit:       skip
170 }
171
172
173
174 /* This is the universal event dispatcher routine */
175 proctype event(mtype msg)
176 {
177          mtype type;
178          int pid;
179
180          atomic {
181          do
182          :: evq??[eval(msg),pid] ->
183             evq??eval(msg),pid;
184             evt!msg,pid;
185             do
186             :: if
187                :: evq??[type,eval(pid)] -> evq??type,eval(pid)
188                :: else break;
```

```
189                fi
190             od
191          :: else -> break
192          od}
193 exit:         skip
194 }
```

G.4 Both UML States and UML Transitions

```
1 #define min(x,y) (x<y->x:y)
2 #define max(x,y) (x>y->x:y)
3 chan evq=[10] of {mtype,int};
4 chan evt=[10] of {mtype,int};
5 chan wait=[10] of {int,mtype};
6 mtype={demand, OK, supply};
7 chan _SYSTEMCLASS__q=[5] of {mtype};
8 typedef Producer_T {
9            int limited_ed;
10           int num_made;
11           }
12 Producer_T Producer_V;
13 chan Producer_q=[5] of {mtype};
14 typedef Consumer_T {
15           int edition_num;
16           }
17 Consumer_T Consumer_V;
18 chan Consumer_q=[5] of {mtype};
19 chan Consumer_supply_p1=[5] of {int};
20 chan t=[1] of {mtype};
```

```
21 mtype={free};
22 active proctype _SYSTEMCLASS_()
23 {
24 mtype m;
25 int dummy;
26 /*  Init state      */
27 /* Initial actions / messages */
28          printf("TRANSITION@2091307066327721@_SYSTEMCLASS_
29                                  @2091306922803240@Initial
30                                  @2091307047321740@Create_Producer
31                                  @2091307047321750@modelstart\n");
32          goto Create_Producer;
33 /* State Create_Producer   */
34 Create_Producer: printf("in state _SYSTEMCLASS_.Create_Producer\n");
35 /* entry actions */
36          atomic{
37             printf("STATE@_SYSTEMCLASS_@209130692280324
38                                 @Create_Producer@209130704732175\n");run
39                Producer();
40          }
41          atomic {if :: !t?[free] -> t!free :: else skip fi;}
42          if
43          :: _SYSTEMCLASS__q?OK -> t?free;
44             printf("TRANSITION@2091307066327230@_SYSTEMCLASS_
45                                  @2091306922803240@Create_Producer
46                                  @2091307047321750@Create_Consumer
47                                  @2091307047321760@OK\n");
48             goto Create_Consumer
49          fi;
```

```
50 /* State Create_Consumer   */
51 Create_Consumer: printf("in state _SYSTEMCLASS_.Create_Consumer\n");
52 /* entry actions */
53           atomic{
54              printf("STATE@_SYSTEMCLASS_@209130692280324
55                           @Create_Consumer@209130704732176\n");run
56                 Consumer();
57           }
58           atomic {if :: !t?[free] -> t!free :: else skip fi;}
59           if
60           :: _SYSTEMCLASS__q?OK -> t?free;
61              printf("TRANSITION@209130706632725@_SYSTEMCLASS_
62                                 @209130692280324@Create_Consumer
63                                 @209130704732176@Done
64                                 @209130706632727@OK\n");
65              goto Done
66           fi;
67 /* State Done   */
68 Done:     printf("in state _SYSTEMCLASS_.Done\n");
69 /* entry actions */
70           atomic{
71              printf("STATE@_SYSTEMCLASS_@209130692280324
72                           @Done@209130706632727\n");
73           }
74           atomic {if :: !t?[free] -> t!free :: else skip fi;}
75           if
76           :: skip -> false
77           fi;
78 exit:     skip
```

```
79 }
80
81
82 proctype Producer()
83 {
84 mtype m;
85 int dummy;
86           Producer_V.limited_ed = 5;
87 /*  Init state      */
88 /* Initial actions / messages */
89           printf("TRANSITION@2091307532288837@Producer
90                                 @2091306922803260@Initial
91                                 @2091307532288839@Waiting_For_Demand
92                                 @2091307532288840@modelstart\n");
93           _SYSTEMCLASS__q!OK;
94           goto Waiting_For_Demand;
95 /* State Waiting_For_Demand   */
96 Waiting_For_Demand: printf("in state Producer.Waiting_For_Demand\n");
97 /* entry actions */
98           atomic{
99              printf("STATE@Producer@209130692280326
100                              @Waiting_For_Demand@2091307532288840\n");
101          }
102 Waiting_For_Demand_G:
103          atomic {if :: !t?[free] -> t!free :: else skip fi;}
104          if
105          :: Producer_q?demand -> t?free; if
106             ::  Producer_V.num_made<Producer_V.limited_ed ->
107                printf("TRANSITION@2091308172575170Producer
```

```
108                                   @2091306922803260@Waiting_For_Demand
109                                   @2091307532288400@Advertise
110                                   @2091308172575190@demand\n");
111                 Producer_V.num_made=Producer_V.num_made+1;
112                 goto Advertise
113              :: else -> goto Waiting_For_Demand_G
114              fi
115           :: 1 -> t?free; if
116              ::  Producer_V.num_made>=Producer_V.limited_ed ->
117                 printf("TRANSITION@2091308172575140@Producer
118                                   @2091306922803260@Waiting_For_Demand
119                                   @2091307532288400@Sorry_Sold_Out
120                                   @2091308172575160@\n");
121                 goto Sorry_Sold_Out
122              :: else -> goto Waiting_For_Demand_G
123              fi
124           fi;
125 /* State Sorry_Sold_Out   */
126 Sorry_Sold_Out: printf("in state Producer.Sorry_Sold_Out\n");
127 /* entry actions */
128           atomic{
129              printf("STATE@Producer@209130692280326
130                              @Sorry_Sold_Out@2091308172575160\n");
131           }
132           atomic {if :: !t?[free] -> t!free :: else skip fi;}
133           if
134           :: skip -> false
135           fi;
136 /* State Advertise   */
```

```
137 Advertise:printf("in state Producer.Advertise\n");
138 /* entry actions */
139           atomic{
140              printf("STATE@Producer@209130692280326
141                           @Advertise@209130817257519\n");
142           }
143           atomic {if :: !t?[free] -> t!free :: else skip fi;}
144           if
145           :: 1 -> t?free;
146              printf("TRANSITION@2091343017410 58@Producer
147                                @209130692280326@Advertise
148                                @209130817257519@Waiting_For_Demand
149                                @2091307532288 40@\n");
150              atomic{Consumer_supply_p1!Producer_V.num_made;
151              Consumer_q!supply}; goto Waiting_For_Demand
152           fi;
153 exit:     skip
154 }
155
156
157 proctype Consumer()
158 {
159 mtype m;
160 int dummy;
161 /*  Init state      */
162 /* Initial actions / messages */
163           printf("TRANSITION@209130740318236@Consumer
164                           @209130692280330@Initial
165                           @209130736058393@Have_Money_Will_Spend
```

```
166                              @209130736058394@modelstart\n");
167           _SYSTEMCLASS__q!OK;
168           goto Have_Money_Will_Spend;
169 /* State Have_Money_Will_Spend   */
170 Have_Money_Will_Spend:
171           printf("in state Consumer.Have_Money_Will_Spend\n");
172 /* entry actions */
173           atomic{
174              printf("STATE@Consumer@209130692280330
175                           @Have_Money_Will_Spend@209130736058394\n");
176           }
177           atomic {if :: !t?[free] -> t!free :: else skip fi;}
178           if
179           :: 1 -> t?free;
180              printf("TRANSITION@209130748051488@Consumer
181                                 @209130692280330@Have_Money_Will_Spend
182                                 @209130736058394@Waiting_For_Supply
183                                 @209130747068447@\n");
184              Producer_q!demand; goto Waiting_For_Supply
185           fi;
186 /* State Waiting_For_Supply   */
187 Waiting_For_Supply: printf("in state Consumer.Waiting_For_Supply\n");
188 /* entry actions */
189           atomic{
190              printf("STATE@Consumer@209130692280330
191                           @Waiting_For_Supply@209130747068447\n");
192           }
193           atomic {if :: !t?[free] -> t!free :: else skip fi;}
194           if
```

```
195             :: atomic{Consumer_q?supply ->
196                Consumer_supply_p1?Consumer_V.edition_num} -> t?free;
197                printf("TRANSITION@2091307480514900Consumer
198                                    @2091306922803300Waiting_For_Supply
199                                    @2091307470684470Have_Money_Will_Spend
200                                    @2091307360583940supply(edition_num)\n");
201                goto Have_Money_Will_Spend
202             fi;
203 exit:       skip
204 }
205
206
207
208 /* This is the universal event dispatcher routine */
209 proctype event(mtype msg)
210 {
211         mtype type;
212         int pid;
213
214         atomic {
215         do
216         :: evq??[eval(msg),pid] ->
217            evq??eval(msg),pid;
218            evt!msg,pid;
219            do
220            :: if
221               :: evq??[type,eval(pid)] -> evq??type,eval(pid)
222               :: else break;
223               fi
```

```
224             od
225         :: else -> break
226         od}
227 exit:      skip
228 }
```

Appendix H

Producer-Consumer Spin Analysis Results

This Appendix contains the Spin analysis results described in Chapter 8 for the *Producer-Consumer* model of Chapter 6. Figure H.1 gives the analysis results for the baseline Promela model, while Figures H.2, H.3, and H.4 give the analysis results for the breadcrumb options **UML States**, **UML Transitions**, and **Both UML States and UML Transitions**, respectively. Recall (Chapter 6, Section 6.3.4) that the "error" in each case is that a final *demand* message remains in the Producer queue. In Chapter 6 we did not refine the example further to handle this message.

```
pan: invalid endstate (at depth 183)
pan: wrote pc_exp_none.pr.trail
(Spin Version 3.3.3 -- 21 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 356 byte, depth reached 183, errors: 1
     144 states, stored
       3 states, matched
     147 transitions (= stored+matched)
      42 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

Figure H.1: Spin analysis results for *Producer-Consumer*, baseline

```
pan: invalid endstate (at depth 210)
pan: wrote pc_exp_states.pr.trail
(Spin Version 3.3.3 -- 21 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 356 byte, depth reached 210, errors: 1
     169 states, stored
       3 states, matched
     172 transitions (= stored+matched)
      44 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

Figure H.2: Spin analysis results for *Producer-Consumer*, breadcrumb option **UML States**

```
pan: invalid endstate (at depth 201)
pan: wrote pc_exp_trans.pr.trail
(Spin Version 3.3.3 -- 21 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 356 byte, depth reached 201, errors: 1
     162 states, stored
       3 states, matched
     165 transitions (= stored+matched)
      42 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

Figure H.3: Spin analysis results for *Producer-Consumer*, breadcrumb option **UML Transitions**

```
pan: invalid endstate (at depth 228)
pan: wrote pc_exp_both.pr.trail
(Spin Version 3.3.3 -- 21 July 1999)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 356 byte, depth reached 228, errors: 1
     187 states, stored
       3 states, matched
     190 transitions (= stored+matched)
      44 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

Figure H.4: Spin analysis results for *Producer-Consumer*, breadcrumb option **Both UML States and UML Transitions**

Appendix I

Original *Fault Handler* Requirements Pattern

This Appendix contains Konrad *et al.*'s original *Fault Handler* requirements pattern [3, 4] prior to refinements discussed in Chapter 9.

I.1 *Fault Handler*: Behavioral Pattern

Intent:

Specify a centralized fault handler for an embedded system.

Motivation:

Fault handling is essential for embedded systems. Embedded systems frequently need to determine what responses are necessary to recover from errors. Consider a flight control system in an airplane, where the system should never shut down completely in response to an error. The system has to decide if it should perform a partial shutdown and offer basic functionality, or if the error is no threat to system safety and logging is sufficient. This fault handler must offer the possibility for other devices to read the error log. But it should also have access to a user interface to

signal that errors have occurred. An important function of the fault handler is to send the system into different safety states depending on the severity of the error. These safety states have to be implemented in the computing component, such as the operation for performing an emergency stop. If an error is reported to the fault handler justifying this action, then the fault handler will activate this state.

Therefore, the fault handler acts as a centralized coordinator for safety monitoring and, hence, control of system recovery.

The following inputs are usually captured [37]:

- Timeout messages by watchdogs, examiners, or monitors.
- Assertions of software errors.
- Built-in-tests (BITs) that run on a periodic or continuous basis.

The centralized safety control facilitates the verification and validation of the safety measures and eases the reuse of the fault handle r in different systems.

Figure I.1 gives the use-case diagram for the *Fault Handler* Pattern, with the major goals being to detect and to handle faults.

Use-Case: System running.

Actors: None

Description: This use-case represents the system when it is functioning.

Includes: Handle faults, Interact with user

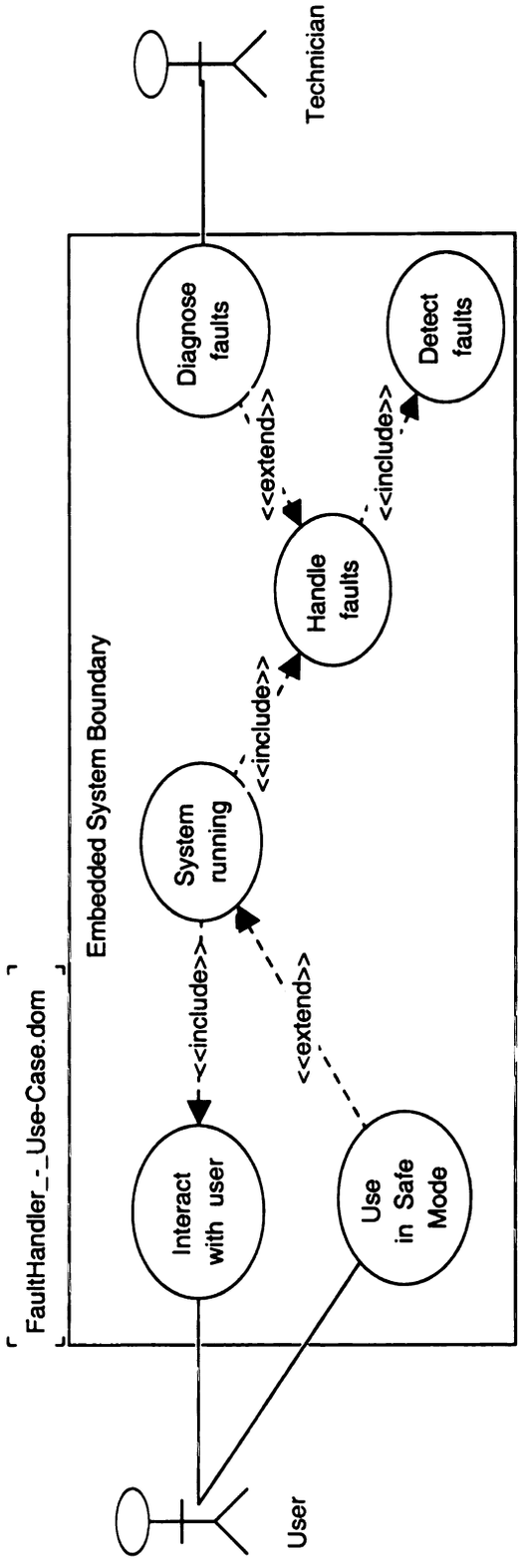


Figure I.1: UML use-case diagram for the *Fault Handler* Pattern

Use-Case: Use in safe mode.

Actors: User

Description: Special case of the use-case *System running*. System offers basic functionality due to errors that have occurred. The exact level of functionality is system-dependent.

Includes: -

Use-Case: Interact with user.

Actors: User

Description: Read user settings and activate indicators.

Includes: -

Use-Case: Handle faults.

Actors: None

Description: Initiate corrective actions if needed.

Includes: Detect faults.

Use-Case: Diagnose faults.

Actors: Technician

Description: Special case of the use-case *Handle faults.* The system offers extended diagnostic functions instead of handling faults to identify the source of the fault(s).

Includes: -

Use-Case: Detect faults.

Actors: None

Description: The system offers fault detection functionality.

Includes: -

Applicability:

The *Fault Handler* Pattern is applicable

- in embedded systems where fault handling is to be centralized.

Structure:

The UML class diagram of the *Fault Handler* Pattern can be seen in Figure I.2. The FaultHandler sends messages to the UserInterface to activate warning levels and sends the ComputingComponent into different safety states. For every safety state defined in the requirements, an operation in the ComputingComponent is needed. The safety states are listed in the *Behavior* field.

The FaultHandler also receives error messages from Watchdogs, Examiners, and Monitors. The Device class represents possible devices in the system that also send error messages to the FaultHandler.

Depending on the safety measures and policies defined, the FaultHandler decides what action to take, for example, such as activating a FailSafeDevice.

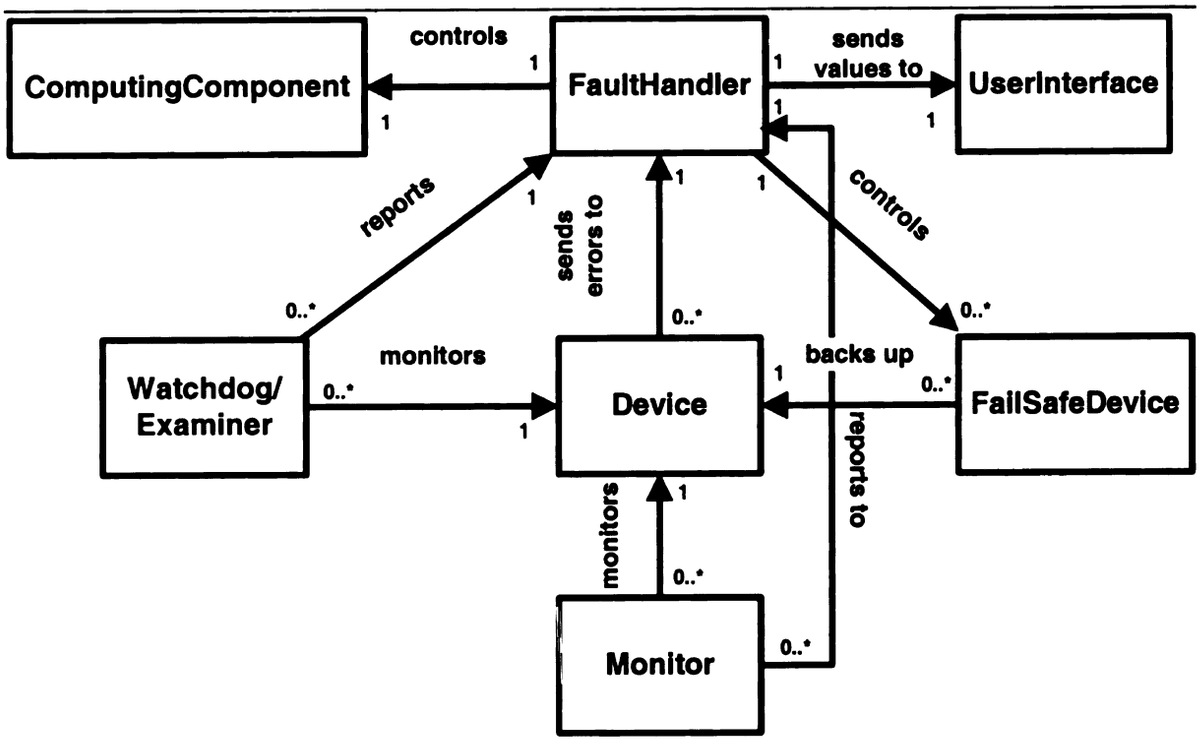


Figure I.2: Structural Diagram for the *Fault Handler* Pattern

Behavior:

Figure I.3 shows the state diagram of the ComputingComponent of the *Fault Handler* Pattern. The state diagram shows which states are possible and what messages activate them. Not all of the states are needed in every system. For example, ABS systems generally do not have partial shutdown states because the system constraints require that an inactive system should not affect the basic functionality of the brakes. Therefore, an emergency stop where the ABS system cuts power immediately is sufficient. These states are defined for the class ComputingComponent; when an error occurs, the FaultHandler decides which state is appropriate and sends the respective message to the ComputingComponent to activate the corresponding state if needed.

The FaultHandler also activates the UserInterface to notify the system user of the current system state. The definitions for the possible system states are as follows [37]:

- **Normal Behavior:** This state captures the system when no errors have occurred and it is functioning normally.

- **Manual/External:** In this state, the system is controlled by an external entity, such as a diagnostic device.

- **Production Stop:** This state is useful, for example, when a human enters a hazardous area. The system should be able to complete its current task and secure the environment, but it should shutdown as soon as possible.

- **Protection Stop:** Ceases operation immediately, but does not turn off power. This state is appropriate, for example, when a machine needs to be stopped, but a device should continue to operate to avoid hazardous situations. For example, a cooling device should remain working even in case of a system malfunction.

- **Partial Shutdown:** The system only offers basic functionality; for example, medical devices may remain in a monitoring state.

- **Hold:** No functionality is provided in this state, but safety actions are taken; for example, a rocket self-destructs in the case of abnormal functions. There is no outgoing transition from this state; a system can only be reactivated by a complete restart.

- **Initialize:** In this state, the system initializes itself.

- **Power Off:** The system might be connected to a power supply in this state, but is not yet activated. For example, a television set can operate in a standby mode.

Furthermore, an emergency stop can be performed by the system. This stop state is not modeled as a separate state because this action takes the system to the `PowerOff` state immediately.

Participants:

- FaultHandler: Fault handler of the system. Contains safety measures and policies.
- ComputingComponent: Central computing component of the system.
- UserInterface: Class offering functionality to notify the user about errors.
- Device: Component representative for a number of possible devices in the system.
- Watchdog/Examiner: Watchdog or examiner in the system.
- Monitor: Possible monitor monitoring the Device.
- FailSafeDevice: Possible backup component for the Device.

Collaborations:

- The FaultHandler receives error messages and stores those messages in an error log. Furthermore, the FaultHandler decides, depending on the safety measures and policies, if a fail-safe state in the ComputingComponent should be entered, or whether the user interface or recovery device should be activated.
- Watchdog, Examiner, and/or Monitor monitor the device and report violations to the FaultHandler.
- FailSafeDevice is activated to recover from faults.

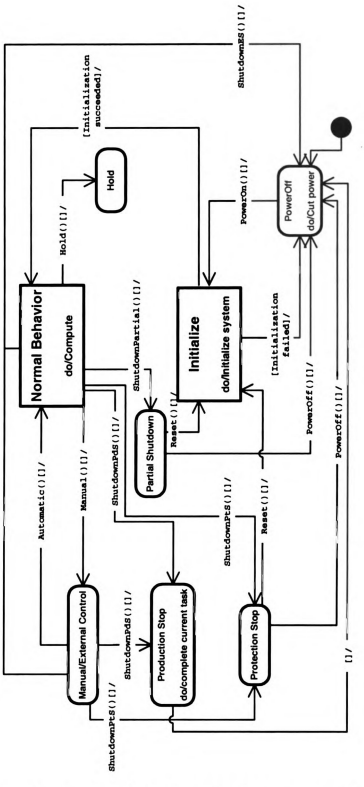


Figure I.3: UML state diagram of the ComputingComponent in the *Fault Handler* Pattern

- The UserInterface gets activated by the FaultHandler.

Consequences:

1. Required safety states have to be implemented in the ComputingComponent.
2. Only one fault handler should exist in the system and should handle all error messages to avoid inconsistent handling of faults [37].
3. The fault handler is one of the critical elements for system safety. Therefore, during the development process of this component, techniques should be used that result in a high assurance of the software component, such as formal methods and thorough testing.
4. Hardware and software redundancies exist in the system, thus meaning higher system costs.
5. Overall safety of the system can significantly be improved by the centralized fault handling component.

Constraints:

- **Absence Pattern:**

  If system initialization fails, then the system should remain in a powered-off state. Therefore, the system should never be in a state where the initialization failed and the system power is on.

  □(!(``Initialization failed'' && ``System power on''))

- **Response Pattern:**

  When an error message is sent to the fault handler, it should process the error and, depending on the error classification, perform the predefined recovery

action as a result of the error. This action can range from "Do nothing" to "Perform emergency shutdown of the system".

□(``Error reported to fault handler'' →

◊(``Start defined recovery action''))

- **Response Pattern:**

  When an error message is sent, it should be stored in an error log for system diagnosis purposes.

  □(``Error reported to fault handler'' →

  ◊(``Store error in error log''))

- **Response Pattern:**

  If an error message is sent to the fault handler, then it should activate the appropriate user interface warning level if required.

  □(``Error reported to fault handler'' →

  ◊(``Activate appropriate user interface warning level''))

- **Response Pattern:**

  If some device, such as a diagnostic device, requests the current error list, then the error list should be sent to the device.

  □(``Error list requested from fault handler'' →

  ◊(``Return list of errors in error log''))

Design Patterns:

- ***Singleton* Design Pattern [83]:**

  Assure that only one fault handler exists in the system.

- ***Strategy* Design Pattern [83]:**

  Encapsulate algorithms for the safety states and make them interchangeable.

Also Known As:

To be determined.

Known Uses:

To be determined.

Related Requirements Patterns:

- ***Controller Decompose* Requirements Pattern:**

  This requirements pattern describes how the fault handler relates to other components in a system.

- ***User Interface* Requirements Pattern:**

  This pattern can be used for the user interface to signal a user the current system state.

Bibliography

Bibliography

[1] Sascha Konrad, Betty H.C. Cheng, and Laura A. Campbell. Object-analysis patterns for embedded systems. *Transactions on Software Engineering*, 2004. Submitted for publication.

[2] Sascha Konrad and Betty H.C. Cheng. Requirements patterns for embedded systems. Technical Report MSU-CSE-02-4, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2002.

[3] Sascha Konrad. Identification, classification, and application of requirements patterns. Technical Report MSU-CSE-02-6, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2002.

[4] Sascha Konrad and Betty H.C. Cheng. Requirements patterns for embedded systems. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002.

[5] 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, Jan 1986.

[6] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2), 1994.

[7] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 4^{th} edition, 1997.

[8] Sue Conger. *The New Software Engineering.* The Wadsworth Series in Management Information Systems. Wadsworth Publishing Company, 1994.

[9] Robyn R. Lutz. Targeting safety-related errors during software requirements analysis. In *Proceedings of the First ACM Symposium on the Foundations of Software Engineering*, Los Angeles, CA, December 1993.

[10] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[11] C. Michael Holloway and Ricky W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, April 1996.

[12] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood, New Jersey, 1990.

[13] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood, New Jersey, 1990.

[14] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[15] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1994.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[17] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[18] Object Management Group (OMG). OMG Unified Modeling Language (UML) 1.3 Specification. Technical report, OMG, March 2000. `www.omg.org/cgi-bin/doc?formal/00-03-01`.

[19] Rational. Rational Rose. `www.rational.com`.

[20] I-logix. Rhapsody. `www.ilogix.com`.

[21] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Static consistency checking for distributed specifications. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE01)*, San Diego, November 2001.

[22] Fiona Hayes and Derek Coleman. Coherent models for object-oriented analysis. In *Proceedings of OOPSLA '91*, pages 171–183, 1991.

[23] Lesley T. Semmens, Robert B. France, and Thomas W.G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, 1992.

[24] Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics of object models. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[25] Ana Maria Dinis Moreira and Robert G. Clark. Adding rigour to object-oriented analysis. *Software Engineering Journal*, 11(5):270–280, 1996.

[26] Malcolm Shroff and Robert B. France. Towards a Formalization of UML Class Structures in Z. In *Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC97)*, pages 646–651. IEEE Computer Society, Los Alamitos, CA, August 1997.

[27] Yile Enoch Wang. *Integrating Informal and Formal Approaches to Object-Oriented Analysis and Design.* PhD thesis, Michigan State University, East Lansing, Michigan, March 1998.

[28] Diego Latella, István Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), IFIP TC6/WG6.1*, Florence, Italy, February 1999. Kluwer.

[29] Prasanta Bose. Automated translation of UML models of architecture for verification and simulation using SPIN. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE99)*, Cocoa Beach, FL, October 1999.

[30] William E. McUmber and Betty H.C. Cheng. UML-based analysis of embedded systems using a mapping to VHDL. In *Proceedings of the 4th IEEE International Symposium on High Assurance Systems Engineering (HASE99). Special Theme: Integration Issues in High Assurance Embedded Systems*, Washington, DC, November 1999.

[31] William E. McUmber. *A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development.* PhD thesis, Michigan State University, East Lansing, Michigan, August 2000.

[32] Rik Eshuis and Roel Wieringa. Requirements-level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS) IV*, pages 121–140. Kluwer Academic Publishers, October 2000.

[33] William E. McUmber and Betty H.C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, May 2001.

[34] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[35] Betty H.C. Cheng, Laura A. Campbell, Min Deng, and R.E.K. Stirewalt. Enabling validation of UML formalizations. Technical Report MSU-CSE-02-25, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, September 2002.

[36] Eduardo B. Fernandez and Xiaohong Yuan. Semantic Analysis Patterns. In *Proceeding of the 19th International Conference on Conceptual Modeling (ER2000)*, pages 183–195, Salt Lake City, UT, October 2000.

[37] Bruce Powell Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison–Wesley, 1999.

[38] Jack Ganssle. Navigating through new development environments. *Embedded Systems Programming Magazine*, 12(5), May 1999.

[39] Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, Academic Press, 2001.

[40] Laura A. Campbell, Betty H.C. Cheng, William E. McUmber, and R.E.K. Stirewalt. Automatically detecting and visualizing errors in UML diagrams. *Requirements Engineering Journal*, 7(4):264–287, December 2002.

[41] Sascha Konrad, Laura A. Campbell, Betty H.C. Cheng, and Min Deng. A requirements pattern-driven approach to specify systems and check properties. In *Proceedings of the 10th International SPIN Workshop of Model Checking of Software (SPIN 2003)*, Oregon, May 2003. Workshop co-located with ICSE03.

[42] Sascha Konrad, Laura A. Campbell, Betty H.C. Cheng, and Min Deng. A requirements pattern-driven approach to specify systems and check properties. Technical Report MSU-CSE-02-28, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2002. Full-length technical report. Condensed workshop version of this report appears in *Proceedings of the 10th International SPIN Workshop of Model Checking of Software (SPIN 2003)*, co-located with ICSE03.

[43] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99)*, May 1999.

[44] Laura A. Campbell. Enabling automated analysis of object-oriented models. Michigan State University Department of Computer Science and Engineering Poster Workshop, April 1999.

[45] Betty H.C. Cheng, Laura A. Campbell, and Enoch Y. Wang. Enabling automated analysis through the formalization of object-oriented modeling diagrams. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, New York, NY, June 2000.

[46] Laura A. Campbell and Betty H.C. Cheng. Object-oriented modeling and automated analysis of a telemedicine application. In *Proceedings of the IEEE International Workshop on Software Specification and Design (IWSSD-10)*, San Diego, CA, November 2000.

[47] Laura A. Campbell. MINERVA: Metamodel-based intuitive editors with reports and visualization of analysis. Michigan State University Department of Computer Science and Engineering Poster Workshop, March 2001. One of 4 Best Overall Posters from a pool of 30 or more posters.

[48] William E. McUmber and Laura A. Campbell. Hydra: A general framework for formalizing UML with formal language for embedded systems. Michigan State

University Department of Computer Science and Engineering Poster Workshop, March 2001.

[49] Laura A. Campbell and William E. McUmber. A framework for detecting and visualizing structural and behavioral errors of UML diagrams. Michigan State University Department of Computer Science and Engineering Poster Workshop, March 2001.

[50] Laura A. Campbell. Visualization and interpretation of analysis results within the context of formalized UML diagrams. In *Proceedings of Doctoral Symposium for IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, May 2001. Abstract for Doctoral Symposium.

[51] Laura A. Campbell. Structural and behavioral analysis of formalized UML diagrams. In *Proceedings of Doctoral Symposium for 5th IEEE International Symposium on Requirements Engineering (RE01)*, Toronto, Canada, August 2001. Abstract for Doctoral Symposium.

[52] Betty H.C. Cheng and Laura A. Campbell. Integrating informal and formal approaches to requirements modeling and analysis. In *Proceedings of Poster Workshop for IEEE Requirements Engineering (RE01)*, Toronto, Canada, August 2001. Abstract for Poster Workshop.

[53] Sascha Konrad, Laura A. Campbell, and Betty H.C. Cheng. Adding formal specifications to requirements patterns. In *Proceedings of the IEEE Workshop on Requirements for High Assurance Systems (RHAS02)*, Essen, Germany, September 2002. Workshop affiliated with RE02.

[54] Sascha Konrad, Laura A. Campbell, and Betty H.C. Cheng. Automated analysis of timing information in UML diagrams. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE04)*, Linz, Austria, September 2004. Short paper. To appear.

[55] Sascha Konrad, Laura A. Campbell, and Betty H.C. Cheng. Automated analysis of timing information in UML diagrams. Technical Report MSU-CSE-03-17, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, Revised 2004. Short paper version to appear in the *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE04).*

[56] Laura A. Campbell. A modeling/analysis framework and process for validating embedded systems requirements with verification techniques. Michigan State University Department of Computer Science and Engineering Poster Workshop, May 2004.

[57] Anish Arora and Sandeep Kulkarni. Detectors and correctors: A theory of fault-tolerance. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, May 1998.

[58] Enoch Y. Wang, Heather A. Richter, and Betty H.C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proceedings of IEEE International Conference on Software Engineering (ICSE97)*, Boston, MA, May 1997.

[59] Enoch Y. Wang and Betty H.C. Cheng. Formalizing and integrating the functional model into object-oriented design. In *Proceedings of International Conference on Software Engineering and Knowledge Engineering*, June 1998. Nominated for Best Paper.

[60] Enoch Y. Wang and Betty H.C. Cheng. A rigorous object-oriented design process. In *Proceedings of International Conference on Software Process*, Naperville, Illinois, June 1998.

[61] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[62] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[63] National Institute of Standards. Dictionary of algorithms and data structures. `www.nist.gov/dads/`.

[64] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* ETACS, Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[65] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1996.

[66] Santiago Pavon and Martin Llamas. The testing functionalities of LOLA. In Juan Quemada, Jose A. Manas, and Enrique Vazquez, editors, *Formal Description Techniques*, volume III, pages 559–562. IFIP, Elsevier Science B.V. (North-Holland)., 1991.

[67] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A protocol validation and verification toolbox. In *Proceedings of the 8^{th} Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440, August 1996. `www.inrialpes.fr/vasy/cadp.html`.

[68] IEEE. *IEEE Standard VHDL Language Reference Manual*, June 1994. ANSI/IEEE Std 1076-1993.

[69] Gerald J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[70] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[71] B. Cheng, P. Fraley, G. Gannod, J. Kusler, S. Schafer, J. Sharnowski, and E. Wang. A distributed, object-oriented multimedia environmental information system: A development document. MSU Technical Report CPS-94-60, Michigan State University, East Lansing, Michigan, November 1994.

[72] The teleservices and remote medical care system (TRMCS). `www.ics.uci.edu/iwssd/case-study.pdf`, 2000.

[73] Robin Milner. A calculus of communication systems. In *Lecture Notes in Computer Science*, volume 92. Springer Verlag, 1980.

[74] Jean-Claude Fernandez and Laurent Mounier. Verifying bisimulations "on the fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3^{rd} International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[75] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[76] Tom Sato and Brian V. Smith. Xfig user manual, version 3.2.3c. `file:/soft/sparc/xclients/lib/X11/xfig/html/index.html`, July 2000.

[77] Anthony Torre. Project specifications for adaptive cruise control system, 2001. `http://www.cse.msu.edu/~cse470/Public/F01/Projects/`.

[78] Willie D. Jones. Keeping cars from crashing. *IEEE Spectrum*, September 2001.

[79] Wilhelm Grimm and Jacob Grimm. *Hansel and Gretel.* Penguin USA, 1986.

[80] Free Software Foundation. GNU Compiler Collection, 2004. `gcc.gnu.org`.

[81] John McLean and Constance Heitmeyer. High assurance computer systems: A research agenda. In *America in the Age of Information, National Science and Technology Council on Information and Communication Forum*, Bethesda, MD, 1995.

[82] Peter Alan Lee and Thomas Anderson. *Fault Tolerance - Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag/Wien, New York, second, revised edition, 1990.

[83] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[84] Anthony Torre. Project specifications for diesel filter system, 2000. `www.cse.msu.edu/~cse470/Public/F00/Projects/air-filter.html`.

[85] Anthony Torre. Project specifications for integrated starter/generator, 1999. `www.cse.msu.edu/~cse470/F99/Projects/`.

[86] Jeremy Bowers. Project specifications for anti-lock brake system, 2001. www.cse.msu.edu/~cse470/Public/F01/Projects/ABS/.

[87] Honeywell Corporation. *DOME Guide*, version 5.2.1 edition, 1999. www.htc.honeywell.com/dome.

[88] Honeywell. DoME. www.htc.honeywell.com/dome.

[89] MetaCase Consulting. Domain-specific modeling: 10 times faster than UML. White paper, MetaCase Consulting, January 2001.

[90] Tigris. ArgoUML features. argouml.tigris.org/features.html, 2001.

[91] Corin Gurr and Perdita Stevens. A cognitively informed approach to describing product lines in UML. Technical Report (unpublished), University of Edinburgh, Edinburgh, Scotland, 1999.

[92] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Bogdan Franczyk. Extending the UML to model system families. In *Proceedings of the Fifth International Conference on Integrated Design and Process Technology (IDPT) 2000*, Dallas, TX, June 2000. Society for Design and Process Science.

[93] Nima Kaveh and Wolfgang Emmerich. Deadlock detection in distributed object systems. In *Proceedings of the 8th European Software Engineering Conference (ESEC'01)*, Vienna, Austria, September 2001.

[94] Mario Dal Cin. Extending UML towards a useful OO-language for modeling dependability features. Technical Report (unpublished), Friedrich-Alexander Universität de Erlangen-Nürnberg, Germany, 2003.

[95] Grant Martin. UML for embedded systems specification and design: Motivation and overview. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. IEEE Computer Society, 2002.

[96] João M. Fernandes, Ricardo J. Machado, and Henrique D. Santos. Modeling industrial embedded systems with UML. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, 2000. ACM Press.

[97] Object Management Group (OMG). UML 2.0 working draft. www.uml.org, August 2004.

[98] Rational. Using UML for modeling complex real-time systems. www.ibm.com/developerworks/rational/library/content/03July/1000/1155/11%55_umlmodeling.pdf, March 1998.

[99] Rational/IBM. Mapping object to data models with the UML. www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/%tp185.pdf, 2003.

[100] Rational. Modeling web application architectures with UML. www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/%webapps.pdf, June 1999.

[101] Rational. Effective business modeling with UML: Describing business use cases and realizations. www.ibm.com/developerworks/rational/library/905.html, June 2003.

[102] Object Management Group (OMG). UML profile for schedulability, performance and time. www.omg.org/cgi-bin/doc?ptc/2002-03-02, 2002.

[103] Object Management Group (OMG). UML profile for CORBA. www.omg.org/technology/documents/formal/profile_corba.html, April 2002.

[104] Bruce Powell Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley, 2003.

[105] Martin Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[106] Mary Shaw. Some patterns for software architectures. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design*, volume 2, pages 255–269. Addison-Wesley Longman Publishing Co., Inc., 1996.

[107] Wolfgang Keller. Object/relational access layers - a roadmap, missing links and more patterns. In *Proceedings of EuroPLoP 1998*, Bad Irsee, Germany, July 1998.

[108] Michael Adams, James Coplien, Robert Gamoke, Robert Hanmer, Fred Keeve, and Keith Nicodemus. Fault-Tolerant Telecommunication System Patterns. In *Proceedings of the 2nd Conference on Pattern Language of Programs*, Monticello, Illinois, September 1995. http://www.bell-labs.com/user/cope/Patterns/PLoP95_telecom.html.

[109] *Proceedings of the Patterns in Distributed Real-Time and Embedded Systems Workshop as part of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tampa, FL, October 2001.

[110] *Proceedings of the 9^{th} Conference on Pattern Language of Programs*, Monticello, IL, September 2002.

[111] *Proceedings of the Patterns in Distributed Real-Time and Embedded Systems Workshop as part of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, November 2002.

[112] Antonio Rito Silva. Dasco project - development of distributed applications with separation of concerns, 2000. http://www.esw.inesc.pt/~ars/dasco/.

[113] Doug Lea. Design Patterns for Avionics Control Systems. Technical Report ADAGE-OSW-94-01, DSSA Adage Project, 1994.

[114] EventHelix.com. Realtime mantra, 2004. `http://www.eventhelix.com/RealtimeMantra`.

[115] Sascha Konrad, Betty H. C. Cheng, Laura A. Campbell, and Ronald Wassermann. Using security patterns to model and analyze security requirements. In *Proceedings of the Requirements for High Assurance Systems Workshop (RHAS03) as part of the IEEE Joint International Conference on Requirements Engineering (RE03)*, Monterey, CA, September 2002.

[116] Eduardo Fernandez. Good analysis as the basis for good design and implementation. Technical Report TR-CSE-97-45, Florida Atlantic University, September 1997. Presented at OOPSLA'97.

[117] Peter Coad, David North, and Mark Mayfield. *Object Models (2nd ed.): Strategies, Patterns, and Applications.* Yourdon Press, 1997.

[118] Andreas Geyer-Schulz and Michael Hahsler. Software engineering with analysis patterns, 2001. `http://wwwai.wu-wien.ac.at/~hahsler/research/virlib_working2001/virlib/%`.

[119] Fonda Daniels, Kalhee Kim, and Mladen A. Vouk. The reliable hybrid pattern: A generalized software fault tolerant design pattern. In *Proceedings of Pattern Languages of Programming*, 1997.

[120] Lucianne Lamour Ferreira and Cecilia Mary Fischer Rubira. Reflective design patterns to implement fault tolerance. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, Vancouver, B.C., Canada, October 1998.

[121] World Wide Web Consortium (W3C). Extensible markup language (XML) 1.0 (second ed.). `www.w3.org/TR/REC-xml`, October 2000.

[122] Francis Schneider, Steve Easterbrook, John Callahan, and Gerard Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proceedings of the 3rd International Conference on Requirements Engineering*, pages 4–13, Colorado Springs, Colorado, April 1998.

[123] Gerald Luettgen and Victor Carreno. Analyzing mode confusion via model checking. In *Proceedings of the 6th International SPIN Workshop on Practical Aspects of Model Checking (SPIN99)*, Toulouse, France, September 1999.

[124] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mongardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety-critical software with SPIN: An application to a railway interlocking system. In *Proceedings of the 17th International Conference on Computing Safety, Reliability, and Security*, pages 284–295, Heidelberg, October 1998.

[125] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[126] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[127] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE99)*, Cocoa Beach, FL, October 1999.

[128] Stefan Leue and Gerard Holzmann. v-Promela: A visual, object-oriented language for SPIN. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society Press, May 1999.

[129] ARTiSAN Software. Real-time Studio. `www.artisansw.com`.

[130] Telelogic. TAU Generation2. `www.telelogic.com`.

[131] Jozef Hooman. Towards formal support for UML-based development of embedded systems. In *Proceedings of the 3rd Workshop on Embedded Systems (Program for Research on Embedded Systems and Software: PROGRESS 2002)*, pages 71–76. Technology Foundation STW, 2002.

[132] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proceedings of the 10th International Computer Aided Verification Conference*, Vancouver, Canada, June 1998.

[133] Jeffery M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification-based prototyping for embedded systems. In *Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, September 1999.

[134] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[135] Jerry Krasner. UML reduces embedded design costs and speed time-to-market. `www.embeddedstar.com/press/content/2004/4/embedded14153.html`, April 2004.

[136] Werner Damm and David Harel. Live sequence charts (LSCs): Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[137] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951. *Object Constraint Language Specification*, 1.1 edition, September 1997.

[138] Angelo Gargantini and Elvinia Riccobene. Automatic model driven animation of SCR specifications. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2003)*, Warsaw, Poland, April 2003.

[139] Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, St. Petersburg, FL, March 1998.

[140] Constance Heitmeyer, James Kirby, and Bruce Labaw. Applying the SCR requirements method to a weapons control panel: An experience report. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, St. Petersburg, FL, March 1998.

[141] Anthony Torre. Project specifications for electronically controlled steering system, 1999. `www.cse.msu.edu/~cse470/Public/F00/Projects/steering.html`.

[142] Cadence Berkeley Laboratories. Cadence SMV model checker. `www-cad.eecs.berkeley.edu/~kenmcmil/smv`, 2000.

[143] Il Centro per la ricerca scientifica e tecnologica (ITC-irst), Carnegie Mellon University, University of Genova, and University of Trento. NuSMV: a new symbolic model checker. `nusmv.irst.itc.it`.

[144] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[145] Sascha Konrad, Laura A. Campbell, and Betty H.C. Cheng. A pattern-based approach to providing user guidance for specifying and analyzing properties. Technical Report MSU-CSE-04-3, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2004.

[146] Meyer C. Tanuan. Automated analysis of unified modeling language UML specifications. Master's thesis, University of Waterloo, August 2001.

[147] Wuwei Shen, Kevin Compton, and James Huggins. A toolset for supporting UML model checking based on abstract state machines. In *Proceedings of ASE 2001*, Loews Coronado Bay, San Diego, CA, November 2001. Short paper.

[148] Lin Hsin-Hung. A research of model checking UML statechart diagrams. Master's thesis, School of Information Science, Japan Advanced Institute of Science and Technology, Japan, August 2003.

[149] Heather J. Goldsby. Specifying compositional semantic functions for non-hierarchical languages using natural deduction systems. Master's thesis, Michigan State University, East Lansing, Michigan, 2004.

[150] Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, Stanford University, 1991.

[151] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[152] Dragan Bošnački and Dennis Dams. Discrete-Time Promela and Spin. *Lecture Notes in Computer Science*, 1486:307+, 1998.

[153] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348, 1996.

[154] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10^{th} International Conference on Computer Aided Verification*, volume 1427, pages 546–550, Vancouver, Canada, 1998. Springer-Verlag.

[155] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[156] Betty H.C. Cheng, Enoch Y. Wang, and Robert H. Bourdeau. A graphical environment for formally developing object-oriented software. In *Proceedings of the IEEE 6^{th} International Conference on Tools with Artificial Intelligence*, November 1994.