MICHIGAN STATE UNIVERSITY LIBRARIES
3 1293 01411 1276

THESIS
2

This is to certify that the

thesis entitled

An Object-Oriented Framework for Constructing Visual Formalisms

presented by

Yile Enoch Wang

has been accepted towards fulfillment of the requirements for

Master degree in Science

Betty H. C. Cheng
Major professor

Date May 26, 1995

O-7639 MSU is an Affirmative Action/Equal Opportunity Institution

An Object-Oriented Framework for Constructing Visual Formalisms

By

Yile Enoch Wang

A THESIS

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

MASTER OF SCIENCE

Computer Science Department

April 1995

ABSTRACT

An Object-Oriented Framework for Constructing Visual Formalisms

By

Yile Enoch Wang

In the past decade, formal methods have been recognized as a rigorous software development approach through which consistency, completeness, and unambiguity can be obtained. However, there is no mature methodology and development paradigm that can fully apply formal methods throughout the entire software development process. In contrast, numerous diagramming techniques that makes use of intuitive and easy to understand graphical notations, are extensively used. Our thesis asserts that the area of visual formalisms that combines the strengths of formal methods and diagramming techniques is a promising research direction, which can result in formal methodologies that are both practical and scalable. Currently, there is no generic framework or development environment that supports the construction of various visual formalisms. This thesis presents a framework developed to facilitate the development of graphical editors and the generation of formal specifications from the visual formalisms constructed using the automatic graphical editors.

To my dear wife, who makes this work meaningful.

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Dr. Betty H.C. Cheng, for her wonderful guidance and patient assistance during this research effort. I also wish to thank Dr. Anthony S. Wojcik for his constructive suggestion. Thanks also to Dr. John J. Weng for being on my commitee.

I would also like to thank Bob Bourdeau and other members in the Software Engineering Research Group for all their support and kindness.

Finally, I would like to thank my wife, Tong, for all her support, understanding, and most of all confidence in me. I appreciate the sacrifices she has made and am looking forward to more time together with her.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

People have been struggling with the *software crisis* for nearly 30 years, and significant progress has been achieved [1]. However, with the rapidly growing demand for software and the continuously increasing complexity of software, the crisis is not only unsolved, but it has become more severe. With the emergence of fourth generation techniques (4GT) [1] for software development, object-oriented technologies are quickly displacing more conventional software development approaches in many application areas [1].

Although some 4GT paradigms have been proposed and many CASE (Computer Aided Software Engineering) tools have been developed to facilitate object-oriented technologies, the new development methodologies of 4GT are still largely *ad hoc* approaches [1]. There is no significant improvement from the previous methodologies. The consistency, completeness, unambiguity, and validation of a system still cannot be guaranteed when using these techniques. The reusability, maintainability, and extensibility of existing or newly developed software are not well supported. One explanation for this situation is that the guidelines and criteria for the development approaches still use informal representations to describe the numerous, complex aspects of a software system. New methods that are able to precisely address the features of software systems must be introduced and applied in order to develop con-

sistent and unambiguous software systems, while supporting reuse and facilitating maintenance.

It is a hypothesis of this research that formal methods is a good candidate to support this goal. A *formal method* consists of a formal specification language that has a well-defined syntax and semantics and a set of rules for reasoning about the formal specifications [2]. Formal methods offer many benefits to software development, such as enabling determination of software correctness, facilitating automated processing, and minimizing the number of errors due to ambiguity, inconsistency, and incompleteness [2, 3]. Applying formal methods to software engineering is the fundamental motivation of this research. Indeed, formal methods are not intended to be a panacea [4] to all software engineering problems, but they can be used as a means to achieve a systematic approach towards software development.

1.1 Problem Description

Object-oriented analysis and development techniques, such as the *Object Modeling Technique* (OMT), are extensively used today. Most of the fourth generation technologies make use of some kind of diagramming scheme to describe systems or certain features of systems, which enable system analysts, designers, programmers, and users to better understand a system. But the informality of the diagramming notations and the lack of well-defined semantics present the potential to introduce errors to the development, particularly as the systems become more complicated. All the difficulties encountered by other development methodologies, such as maintainability, ambiguity, and system consistency, are also present. Similarly, software reusability is also not well supported with informal diagramming techniques.

Several formal specification languages, such as *Z* [5], *Lotos* [6], and *Larch* [3], are commonly used. There have been several successful attempts in applying formal

methods to large-scale software development projects [7]. Recently several papers and technical reports show that the users, as well as the respective projects, benefited greatly from the use of formal methods [8]. However, the application of formal methods to software engineering is still rare when compared to the large number of systems being developed. The advantage of formal methods, whose foundation is based largely on mathematical notation with a well-defined syntax and semantics, tends to intimidate many system analysts, designers, and programmers. Although the level of difficulty of mathematics applied in formal methods is not as great as it is often perceived, the notation may still prohibit users who are not accustomed to working with mathematical notation from taking full advantage of what formal methods have to offer.

Visual formalisms have been proposed as a means to bridge the gap between the popular informal diagramming technologies and the perceived difficult to use formal method techniques. A *visual formalism* is a diagramming technique that imposes well-defined syntax and formal semantics to the visual notations. We propose that visual formalisms may offer a new approach to software development. With this new approach, users can enjoy the ease of use of diagramming notations while also taking advantage of the automatic reasoning, consistency, and completeness checking enabled by using formal methods. Of course, there is a tradeoff between informality and formality. In our investigations, one of our objectives is to determine the balance between user-friendliness and rigor.

This thesis presents a framework developed to support visual formalisms. The application of the framework is focused on Rumbaugh's *Object Modeling Technique* (OMT) [9] comprising three diagramming techniques: object model, dynamic model, and functional model. The object model describes the static structure of a system with object diagrams; the dynamic model captures a system's behavior in terms of

state diagrams; the functional model depicts the functionality of a system using data flow diagrams. The diagrams are formalized in terms of Larch [3] specifications.

Thesis Statement: *Visual formalisms, with the strengths of formal methods and diagramming techniques, is one hopeful research direction that can bring us formal methodologies that are both practical and scalable. This thesis presents a framework developed to facilitate the development of graphical editors for diagrams and the generation of formal specifications for the visual formalisms.*

1.2 Contributions

This thesis makes several contributions to the field of software engineering. First, we have developed a software architecture to integrate a graphical diagram and its corresponding formal specifications. Second, we have constructed a development environment that uses a graphical object library that facilitates the creation of graphical diagram editors and formal specification generators. Third, we have developed a tool, VISUALSPECS, based on the above architecture and environment to support the graphical construction of OMT diagramming notations and to generate the corresponding formal specifications both in a generic algebraic specification language as well as in the Larch language. Finally, we have developed a cross-referencing utility between different types of diagrams with the same diagramming notation, which assists the system analysis process, while simplifying the formal specification process.

1.3 Organization of Thesis

The remainder of the dissertation is organized as follows. Chapter 2 gives background information on graphics utilities, specification languages, and object-oriented modeling. Chapter 3 introduces the architecture of the framework. Chapters 4, 5, and 6

illustrate how the tools developed based on the framework support the construction of diagrams and the generation of formal specifications for the object model, state schema, and dynamic model, respectively. A new modeling paradigm to perform system modeling is introduced in Chapter 7, including a sample application. Chapter 8 shows the cross-referencing mechanism developed to facilitate the modeling process. Chapter 9 gives an example that shows how the generated specifications can be used with previously developed tools. The related work is discussed in Chapter 10. Chapter 11 gives the conclusions of the thesis and addresses several future investigations.

CHAPTER 2

Background

Since the purpose of the framework is to facilitate the construction of visual formalisms, several complementary fields are integrated in order to satisfy the overall objective of the project. The creation and modification of diagrams require the use of several geometric properties, and the generation of formal specifications also has a set of corresponding issues relevant to the project. This chapter gives background knowledge, identifies obstacles encountered during the project, and overviews the respective solutions.

2.1 Geometry

Some issues of geometry were encountered during the construction of the graphical development environment. The following section addresses specific areas of concern and discusses the corresponding solutions.

2.1.1 Determine the relation between a point and an area

In general, it is straightforward to check whether a point is inside a rectangular region or not. But for some other shapes of regions, such as that of a diamond and a triangle, the number of possible cases of point location increases and is more difficult

to determine. A well-known algorithm to determine the relative position of a point to a polygon is to calculate the accumulated degrees of the between-vector angles formed by point-to-vertex vectors [10]. Figures 2.1 and 2.2 show two cases of point positions relative to a polygon area. Since the coordinates of a point and all vertices of an area are known, given a point P and a vertex P_i, the vectors can be computed by $\overrightarrow{PP_i} = P_i - P$. The angles between two vectors can be obtained by

$$cos\theta_{ij} = \frac{\overrightarrow{PP_i} \cdot \overrightarrow{PP_j}}{\left|\overrightarrow{PP_i}\right| \times \left|\overrightarrow{PP_j}\right|} \tag{2.1}$$

$$\theta_{ij} = arcos\left(\frac{\overrightarrow{PP_i} \cdot \overrightarrow{PP_j}}{\left|\overrightarrow{PP_i}\right| \times \left|\overrightarrow{PP_j}\right|}\right) \tag{2.2}$$

The summation of the vector angles can be calculated by the formula given in (2.3).

$$Total\ \ Angles = \sum_{i=1}^{n-1} \theta_{i(i+1)} + \theta_{n1} \tag{2.3}$$

If the summation of the angles is equal to 2π, then the point is inside the area. Otherwise, the point is outside. Clearly, the accumulated vector angle is 2π in Figure 2.1, whereas the accumulated vector angle is 0 in Figure 2.2.

2.1.2 End point placement

The placement of the end point of a line connecting graphical objects is a more complex issue than identifying the location of a point. The position of an end point associated to a visual object must be changed with the movement and the resizing of the object. Attaching an end point to a fixed point of the edge of an object, as shown in Figures 2.3 and 2.4, may appear odd visually.

Another concern is the circular shape of some end points. It must be placed in the proper location to make it tangential with the edge of an object. Consequently

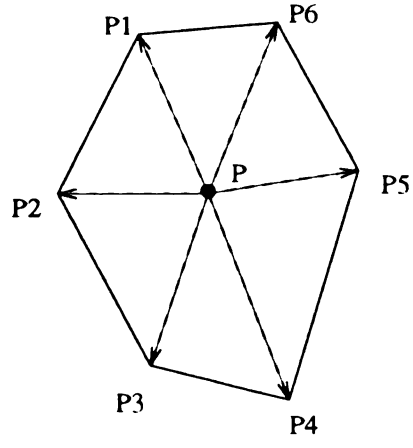

Figure 2.1. A polygon with a point inside

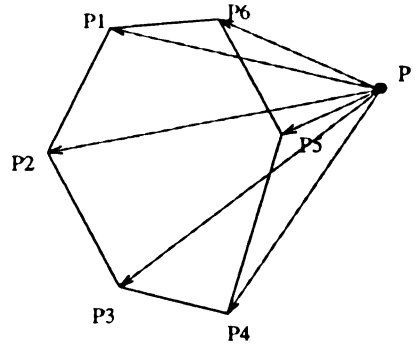

Figure 2.2. A polygon with a point outside

Figure 2.3. The diagram before the move of graphical components

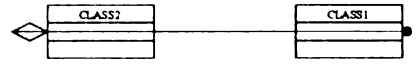

Figure 2.4. The diagram after the move of graphical components

another parameter, tangent direction between a circle and an edge, must be computed in order to achieve the best visual effect. Figures 2.5 and 2.6 illustrate the difference between drawing an end point with and without the tangent direction parameters respectively.

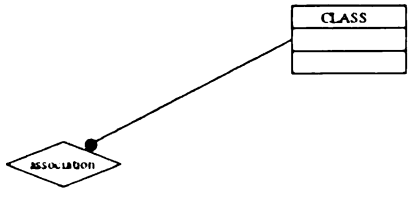

Figure 2.5. Without the tangent direction parameter

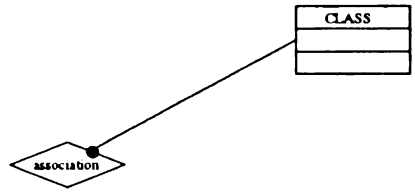

Figure 2.6. With the tangent direction parameter

2.2 Formal methods

A *formal method* is characterized by a formal specification language and a set of rules governing the manipulation of expressions in that language. A formal specification language provides [3]:

- a *syntactic domain*: the notation in which the specifications are written.
- a *semantic domain*: a universe of elements that may be specified, and
- a *satisfaction relation*: indicates which elements in the semantic domain satisfy (implement) which specifications in the syntactic domain.

Formal specifications can be checked by tools that help explore the consequences of analysis results and design decisions [2], detect logical inconsistencies [11], simulate

execution [12], execute symbolically [13], and prove the correctness of implementations steps (refinements) [14].

There exists many types of formal specifications. Formal specification languages can be partitioned according to a high-level classification [15]. The formal specifications are commonly categorized into three classes: axiomatic, operational, and denotational.

2.2.1 Axiomatic specifications:

The *axiomatic* approach implicitly defines the semantics of a programming language by a collection of axioms and rules of inference, which enable the proof of properties of programs, such as program correctness, in terms of specified input/output relations. The assertions about programs can be proven by either a mathematical or an operational definition and mathematical reasoning. The axioms are rules of inferences that can be regarded as theorems within the framework of mathematical semantics. The objective of the axiomatic formal specification is to provide a formal system that enables a proof to be constructed using only the uninterpreted specification text.

Axiomatic specifications can be further classified into predicate specifications [16] and algebraic specifications [17].

2.2.1.1 Predicate specifications:

A predicate specification explicitly describes properties of the behavior of a system that a given implementation must satisfy. The specifications describe the system's required functionality. Predicate specifications are not bound by the constraint of constructivity. The properties can be stated separately and then combined, which facilitates specification modularity. The properties include input/output constraints and other behavior conditions, such as fault tolerance, safety, security, response time, and space efficiency. Figure 2.7 contains a predicate specification describing a proce-

dure that, given an appropriate argument, x, computes an integer approximation to its square root.

Precondition S **Postcondition**
Precondition: $x \geq 0 \wedge \text{integer}(x)$;
Postcondition: $\forall i : 0 \leq i \leq x : abs(x - result \times result) \leq abs(x - i \times i)$

Figure 2.7. A predicate specification of integer square root

2.2.1.2 Algebraic specifications:

An algebraic specification is a mathematical description language, based largely on equations, commonly used to specify *abstract data types (ADT)*. An ADT is a well-defined data structure described by the available services and properties of these services. The properties of the data type are specified in terms of *equations*.

An algebraic specification normally consists of

- *Sort(s)*: the names of the abstract data types being described.
- *Operation(s)*: the services available on instances of the abstract data type and syntactically describes how they have to be invoked (*signatures*).
- *Axioms* or *theorems*: formally describes the semantic properties of the algebraic specification.

The Larch family of specification languages [18] uses a two-tiered approach to formal specifications in which one tier, the Larch Interface Language (LIL), specifies program behavior and programming language interfaces. For example, LCL [19] is a LIL for the C language that specifies program behavior in terms of predicate logic and

function and procedure interfaces in C syntax. The other tier, the Larch Shared Language (LSL), is an algebraic specification language used to specify properties that are independent of a particular programming language and paradigm. Algebraic specifications can be used to describe object-oriented software in a straightforward manner, using abstract data types as the basic unit in software specification. Accordingly, the basic unit of specification in LSL is the *trait*, which axiomatizes theories about functions and data types that are used in programs. A collection of general purpose traits that are designed for constructing application-specific traits is called a trait *handbook* [18]. In Figure 2.8, an algebraic specification describing the properties of the Abelian group is given in the Larch specification language.

Abelian: **trait**
introduces _ o _: T, T → T
asserts ∀ x, y: T
x o y ≡ y o x
implies Commutative (T for Range)

Figure 2.8. A Larch algebraic specification of the Abelian group

---

2.2.2 Operational specifications:

An operational specification [20] gives one solution that satisfies the required properties, instead of describing the required behaviors. Commonly, the operational specification is similar in format to a program. This approach has an advantage in that the operational specifications can be executed directly as a rapid prototype of the system being specified. Thus the specifiers and their clients can obtain feedback about the software system quickly. The disadvantages are that it is difficult to extract the essen-

tial properties that the system must fulfill and the specifications tend to be relatively longer than behavioral specifications. The operational specification in Figure 2.9 gives a simple implementation algorithm to compute the square root of an integer.

```
int sqrt (int x)
    requires x ≥ 0;
    effects
      i = 0;
      while i × i < x
        i = i + 1 end
      if abs(i × i - x) > abs((i - 1) × (i - 1) - x)
        then return i - 1
        else return i
```

Figure 2.9. An operational specification of integer square root

2.2.3 Denotational specifications:

The denotational specification [21] maps a specification directly to its meaning, called its denotation. The denotation is usually a mathematical value, such as a number or a function. No interpreters are used; a valuation function maps a specification directly to its meaning.

A denotational definition is more abstract than an operational definition, since it does not specify computation steps. Its high-level, modular structure makes it especially useful to language designers and users, since the individual parts of a language can be studied without having to examine the entire definition. On the other hand, the implementor of a language is left with more work. The numbers and functions

must be represented as objects in a physical machine, and the valuation function must be implemented as the processor.

Therefore, denotational semantics is more abstract than an operational specification and less abstract than an axiomatic specification. Like an algebraic specification, it can be stated in modules, which makes it especially useful to system analysts and designers.

Figure 2.10 shows a denotational specification of boolean expressions. The value denoted by an expression depends on the state because it may contain variables. **E** maps an expression onto a function from states to boolean values. For a particular expression, ε, $\mathbf{E}[\![\varepsilon]\!]$: **S** $\rightarrow$ **Bool** is a function from **S** to **Bool**, which corresponds to a set of values. States, **S**, is the set or data-type of functions from identifiers, **Ide**, to **Value**. A particular state, σ, is a particular function from variables to values. Therefore, a specific value is obtained by evaluating the expression $\mathbf{E}[\![\varepsilon]\!]\sigma$: **Bool**, where σ gives the state in terms of identifiers and their values. Suppose $\sigma[\![x]\!] =$ **true** and $\sigma[\![y]\!] =$ **false** then the boolean expression x **and not** y can be evaluated as:

$\mathbf{E}[\![x \textbf{ and not } y]\!]\sigma$

$= \mathbf{E}[\![x]\!]\sigma \wedge \mathbf{E}[\![\textbf{ not } y]\!]\sigma$

$= \mathbf{E}[\![x]\!]\sigma \wedge \neg\mathbf{E}[\![y]\!]\sigma$

$= \sigma[\![x]\!] \wedge \neg\sigma[\![y]\!]$

$=$ **true** $\wedge \neg$**false**

$=$ **true**

2.3 OMT Overview

The *Object Modeling Technique* is a methodology developed by Rumbaugh, *et al* [9], to facilitate *object-oriented analysis* and *design (OOA and OOD).* It includes three diagramming techniques to describe different aspects of a system.

Exp:
ε ::= **and** ε |
ε **or** ε |
not ε |
true |
false |
ξ

Ide:
ξ ::= syntax for identifiers

$\mathbf{E}$: **Exp** $\rightarrow$ **S** $\rightarrow$ **Bool**
$\mathbf{E}[\![\varepsilon \text{ and } \varepsilon']\!]\sigma = \mathbf{E}[\![\varepsilon]\!]\sigma \wedge \mathbf{E}[\![\varepsilon']\!]\sigma$
$\mathbf{E}[\![\varepsilon \text{ or } \varepsilon']\!]\sigma = \mathbf{E}[\![\varepsilon]\!]\sigma \vee \mathbf{E}[\![\varepsilon']\!]\sigma$
$\mathbf{E}[\![\text{ not } \varepsilon]\!]\sigma = \neg\ \mathbf{E}[\![\varepsilon]\!]\sigma$
$\mathbf{E}[\![\text{ true }]\!]\sigma$ = true
$\mathbf{E}[\![\text{ false }]\!]\sigma$ = false
$\mathbf{E}[\![\varepsilon]\!]\sigma = \sigma[\![\varepsilon]\!]$

σ: **S** = **Ide** $\rightarrow$ **Value**

Figure 2.10. A denotational specification of boolean expression

2.3.1 Complementary diagramming techniques

The essence of the *Object Modeling Technique* is to build a model of an application domain during the analysis of a system that can be augmented with implementation details during the design phase. The modeling consists of three orthogonal models, *object model*, *dynamic model*, and *functional model*, each depicting different features of a system. Each model is applicable during all stages of development and acquires implementation detail as the development progresses. The models are represented as the object diagram, state diagram, and data flow diagram, respectively.

2.3.2 Object Model

An object diagram is used to capture information about the real world that is important to an application. Thus it is the most important of the three diagrams. It describes the static objects in a system by showing their identity, their relationships to other objects, their attributes, and their operations. Therefore, it is straightforward to derive *abstract data types* (ADT) from the object diagrams. The object diagram forms a basic framework upon which the dynamic and functional models are based. The diagram provides an intuitive visual representation of a system that can be valuable in the communication between the customers and the developers since the diagrams serve to document the structure of a system. Figure 2.11 gives a high-level object diagram of the *graphical editor* implemented in the framework, where boxes represent *classes*; lines between boxes represent associations; empty, solid circles at the end of association lines represents different multiplicities.

2.3.3 Dynamic Model

A state diagram graphically represents the dynamic models that describe the behavioral aspects of a system concerned with events, time, and changes. Each state

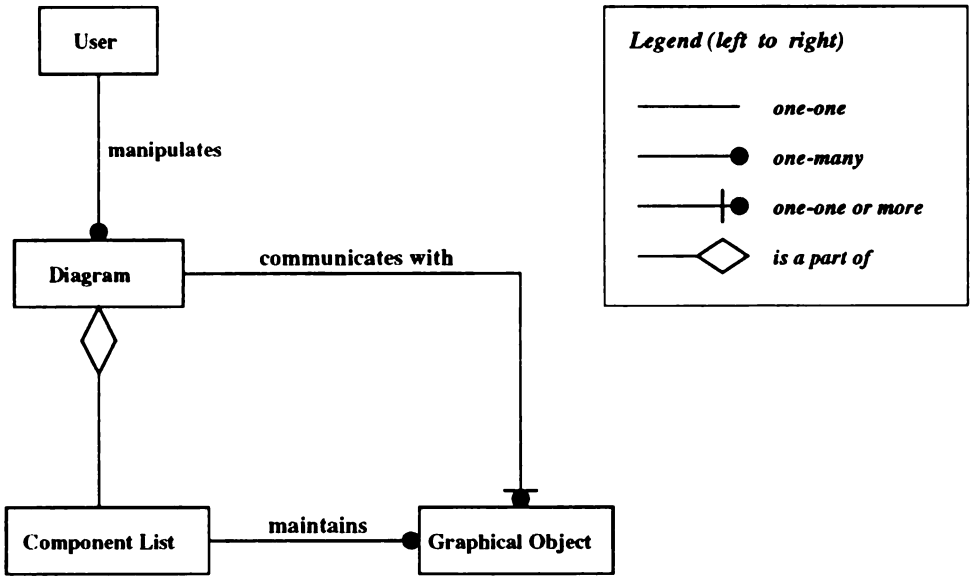

Figure 2.11. The object model of the graphical editor

diagram depicts the state and event sequences allowed in a system for one class of objects. The notation used for dynamic models is a variation of Harel's Statechart notation [22], where ovals represent states; arrow arcs represent state transitions. An example state diagram is given in Figure 2.12, where rounded rectangles represents states and labels on arrows represent events that trigger state transitions. State diagrams also reference the other diagrams. Functions in a the data flow diagram correspond to the actions from the state diagram; operations on objects in the object diagram are modeled as events in a state diagram.

2.3.4 Functional Model

The functional model, depicted in the form of data flow diagram, is the third dimension of the three orthogonal modeling techniques of OMT. Data flow diagrams consist of nodes and arcs, which correspond to processes and data flows, respectively,

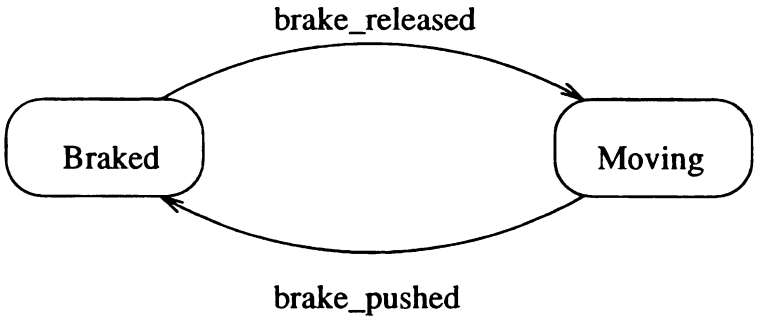

Figure 2.12. An example state diagram

that specify and implement the control aspects of a system (rectangles represent external entities). Figure 2.13 is a high-level Data Flow Diagram showing the overall architecture of the framework. The data flow diagram also specifies the meaning of the operations in the object model and the actions in the dynamic model, as well as constraints for values within an object model.

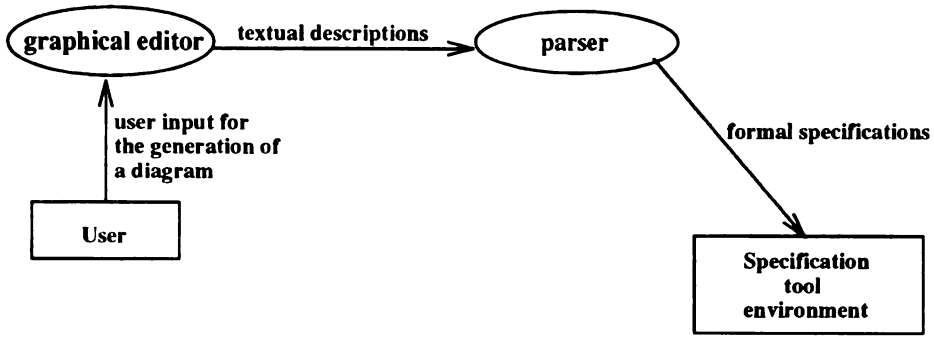

Figure 2.13. A high-level Data Flow Diagram of the framework

2.4 Requirements Analysis and Design

The three complementary diagrams depict different aspects of a system and are attractive to analysts and designers initially. However, it is not an easy task to obtain an accurate description of a system using the three diagramming schemes. Typically, several iterations are necessary in order to refine the diagrams to obtain a useful depiction of the system.

2.4.1 Analysis

The object diagrams that describe the types of objects existing in the system and the allowable relationships among the objects are the first diagrams obtained from the requirements analysis process. Requirements analysis typically focuses on the static architecture of the system. Only the types of objects and the relationships among them are depicted. Object *attributes* and *operations* are left to the design phase to be resolved. Although different people have different perspectives about the same system, the inherent object-oriented properties of a system will largely limit this kind of variation. Our experience shows that the final diagramming results tend to be similar after several iterations. Furthermore, analysis facilitates the communication between system analysts and customers during the analysis phase, which is critical to obtaining a good understanding of the system. A systematic analysis process serves to reveal miscommunication between the customer and the analyst, and to achieve a well-defined object model for later analysis and design.

The essence of computation of the Turing machine is to deal with the changes of state; a system without state changes typically provides no functionality. The dynamic model is also critical in gaining a thorough understanding of a system. Based on the object diagrams, a system may be further decomposed in order to model the basic state changes of components. This process is much more complicated

than object diagramming. The states of a system might increase exponentially with respect to the size of the problem. How to control the modeling process is a key issue in obtaining an appropriate depiction of the state changes.

Data flow diagrams can be largely based on information object and state diagrams in order to bridge the gap between the static and dynamic models respectively. The functional aspect described in a data flow diagram answers *how* the state changes in state diagrams are realized. It resolves the implicit behavior exhibited by state transitions that appear in the dynamic model and provides a framework upon which the design phase may be built.

2.4.2 Design

Design is the second iterative process that refines the three diagrams, because more detailed (implementation-specific) information will be added to the corresponding diagrams. The *attribute* and *operation* properties of the static object diagram are available from the statechart and data flow diagrams. In this stage, *abstract data types (ADTs)*, the kernel of object-oriented software development methodology and described by algebraic specifications, are available for refinement and use. In turn, the dynamic model and the functional model can be further analyzed and revised.

CHAPTER 3

Architecture of Graphical Environment

Since the objective of the project is to develop a graphical environment to facilitate the development of visual formalisms, the architecture of the environment plays an important role in the flexibility, usability, and extensibility of the environment. The environment is divided into three components: *a graphical editor*, *a textual description* of the diagrams, and *a parser* to output formal specifications from the textual descriptions. Figure 3.1 is a high-level Data Flow Diagram showing the overall architecture of the three components. The *graphical editor* is constructed by a graphical component library. The diagrams edited by the *graphical editor* then output their corresponding intermediate textual descriptions to be processed next by a *parser*. It is the *parser* that takes the textual description of a diagram and finally generates the formal specifications. The remainder of this chapter describes the *diagram*, *textual description*, and *parser*.

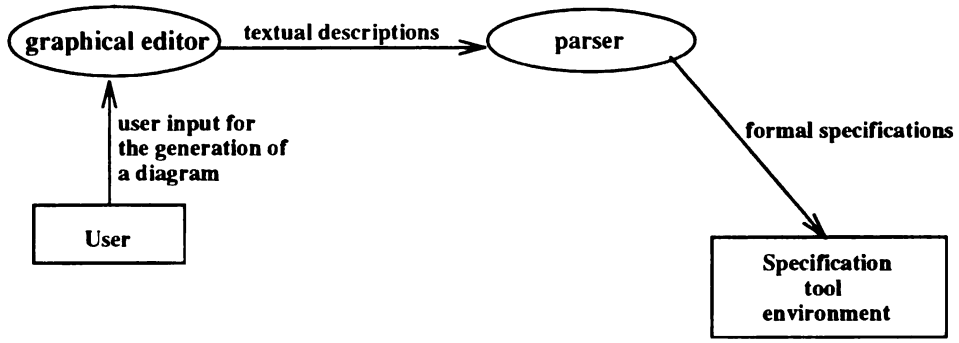

Figure 3.1. A high-level Data Flow Diagram of the overall architecture

3.1 Graphical Editor

Figure 3.2 gives the high-level object model of the *graphical editor*. The graphical editor is composed of a graphical icon library that contains a set of graphical component objects. Since the *diagram* graphical object in the library is so special, it will be discussed separately in this section.

3.1.1 Graphical Icon Library

The graphical icon library contains a set of icons that can be used as graphical components and visual notations in a diagram. The icons are designed and implemented as objects that encapsulate the data structure with their operations. The hierarchical inheritance relationships between the objects are also taken into consideration and designed into the objects. Thus new icons can easily be constructed and added into the library or derived from one or more existing icons. Treating the icons as objects is straightforward and intuitive, and the development framework benefits much from such a design decision. Figures 3.3, 3.4, 3.5, and 3.6 depict the inheritance relationships among the graphical icons.

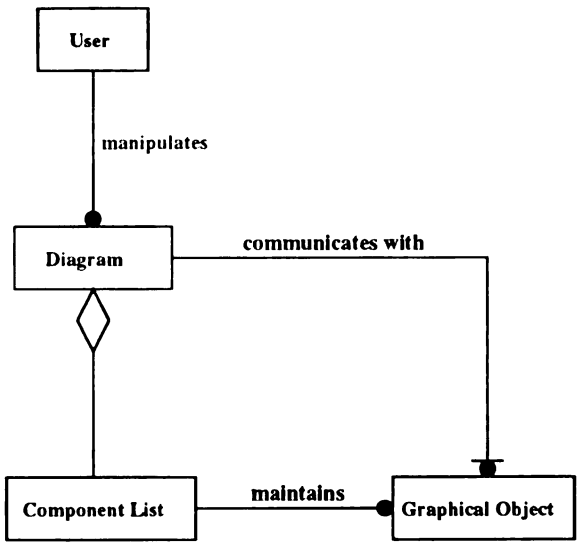

Figure 3.2. The object model of the graphical editor

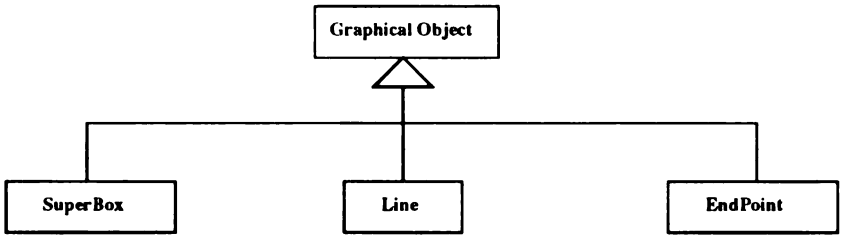

Figure 3.3. Level 0 inheritance relationship among the graphical icons

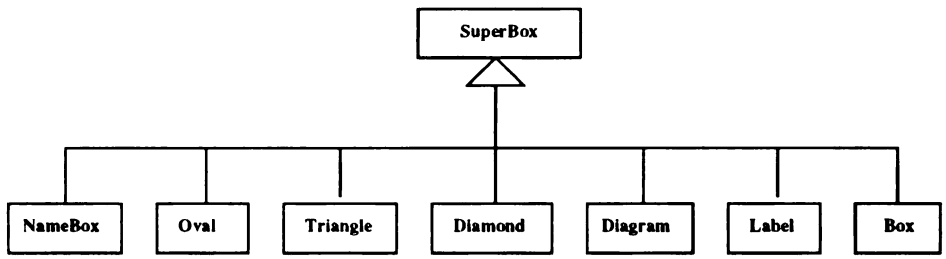

Figure 3.4. Subclass hierachy for SuperBox class

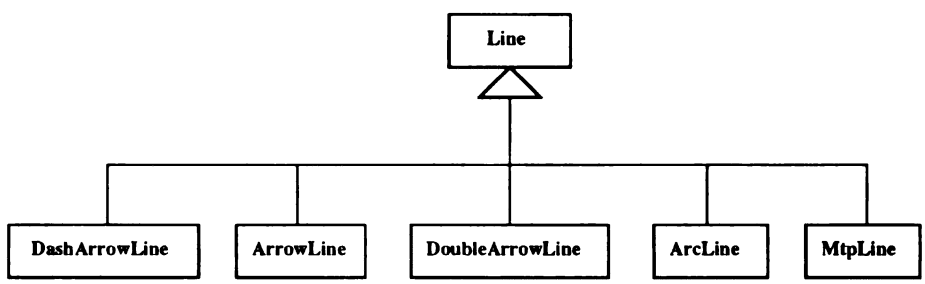

Figure 3.5. Subclass hierachy for Line class

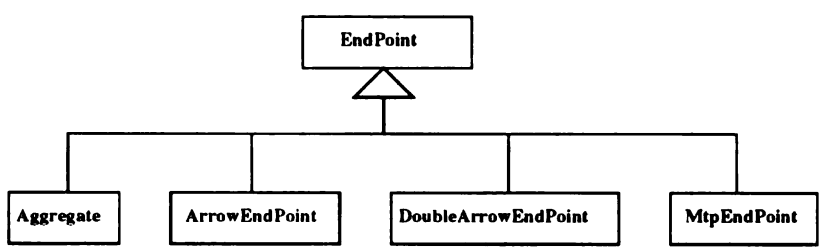

Figure 3.6. Subclass hierachy for EndPoint class

3.1.1.1 Set of icons:

The library currently has the following high-level graphical icons for constructing object diagrams:

- *Box*: the *class* visual notation in an object diagram.
- *Oval*: the *instance* visual notation in an object diagram.
- *Diamond*: the *association* notation in an object diagram.
- *Triangle*: the *super/subclass* notation in an object diagram.
- *Binary*: the *binary association* notation in an object diagram.

Although the above icons were designed for the *object diagram*, they are composed of the following general-purpose graphical objects, which can be reused through composition, modification, or both to construct new icons.

Some of the utility graphical objects can be used to construct graphical icons are also included in the library:

- *Graphical object*: the superclass of all graphical objects (if there exists another class inherited from class ψ, we say ψ is a superclass). It declares the basic data structure needed in a graphical icon object and specifies the essential operations common to all objects or diagram.
- *Super box*: the superclass from which *box*, *oval*, *diamond*, and some other classes are derived. It declares several of the common data structures as well as operations shared by the above-mentioned objects.
- *String box*: a box containing a string.
- *Label*: a variation of the string box, which makes the rectangular boundary invisible and only allows one line of string of text.

- *String dialog box*: a variation of the string box with a method that performs interactive modification of the strings through a dialog box.
- *Line*: a generic line connecting other objects.
- *Rubber line*: a rubber-banding line that changes its length with the move of the mouse pointer. It is used to connect objects.
- *End point*: An object links objects, such as a line and a box or a diamond and passes information between the two objects connected for communication purposes.

3.1.1.2 Operations applicable to icons:

Though there are many different icon classes that exist and can potientially be added into the library, the common operations applicable to these graphical objects are not as numerous. Several commonly owned operations are:

- *Create*: to create a new graphical object from either scratch or a file containing a previously drawn diagram.
- *Delete*: to delete an existing object from a specific diagram.
- *Draw*: to draw the boundary as well as the label and some other attributes of an object.
- *Move*: to move an object around the diagram.
- *Resize*: to enlarge or shrink an object.
- *Save*: to save information relevant to an object into a file.
- *Message*: to communicate with other graphical objects or a diagram.

3.1.2 Diagram

Diagram is a special graphical object containing other graphical objects, such as boxes and lines. From the view of formal specifications, a diagram is a specification composed of visual formal notations. In order to impose the least number of constraints on the diagram for generality and reusability purposes, the formal semantics of the diagram is decoupled from the graphical objects and moved to the parser component. The diagram for a specific visual formalism with its visual notation set and formal semantics can be constructed by composing or modifying the existing graphical objects in the graphical library and reconstructing the parser. Only the syntax checking of the entire diagram (specification) is delegated to the parser. However, some simple errors, such as linking two objects with an improper line, could be eliminated by adjusting the graphical environment to a specific specification language.

3.1.2.1 Functionality of the diagram:

The purpose of the diagram is to serve as a repository to accommodate graphical components and to support their corresponding manipulations. It serves as an event dispatcher and object container to convey users' requests, such as moving an object, to the individual object, and performs some general diagramming activities, e.g., saving a diagram.

3.1.2.2 Structure and working mechanism of diagram:

To a programmer, a diagram is a graphical object container and a message dispatcher. Since all the graphical object components are subclasses of the *GraphicalObject* superclass, they have a uniform method, *message*, to communicate information from another object. Thus, a diagram maintains a general-purpose *list* that contains all the graphical components created and several iterators that deliver messages to the

objects. To some extent, the list is a stack that allows an iterator to process the graphical objects it contains in a first in last out order. This process enables the most recently drawn objects to remain on the top of the objects. All the relevant windowing events (button press, pointer motion, button release, ...) are passed to graphical objects, such as boxes and lines, by the iterators. The objects, in turn, determine if the event or message is valid and behave accordingly. A diagram has no global view of the objects and their inter-relationships. It is only aware of graphical components, boxes, lines, diamonds, etc., but not their inter-relationships, such as association or aggregation.

3.1.2.3 Communication model between objects:

Since neither the diagram nor the individual graphical components have a global understanding of the inter-relationships between graphical objects, the project uses a message passing communication mechanism to exchange information between objects. Of course, the lack of a global view necessitates message passing, which incurs a heavy communication cost. However, the modularity of this design greatly benefits the environment in terms of flexibility, extensibility, reusability, and maintenance. Figure 3.7 shows a communication example that is used to control the layout of the diagram if one object is moved or resized. The numbers enclosed in circles indicate the order of events; a dashed line represents a binary relationship; dashed arrows indicates an ordered binary relationship; solid arrows indicates the direction of communication or message passing; the bold line represents a line object.

3.2 Textual description

Besides serving as a repository for graphical objects, a diagram is considered to be a graphical representation of specifications. Once a visual formalism with well-defined

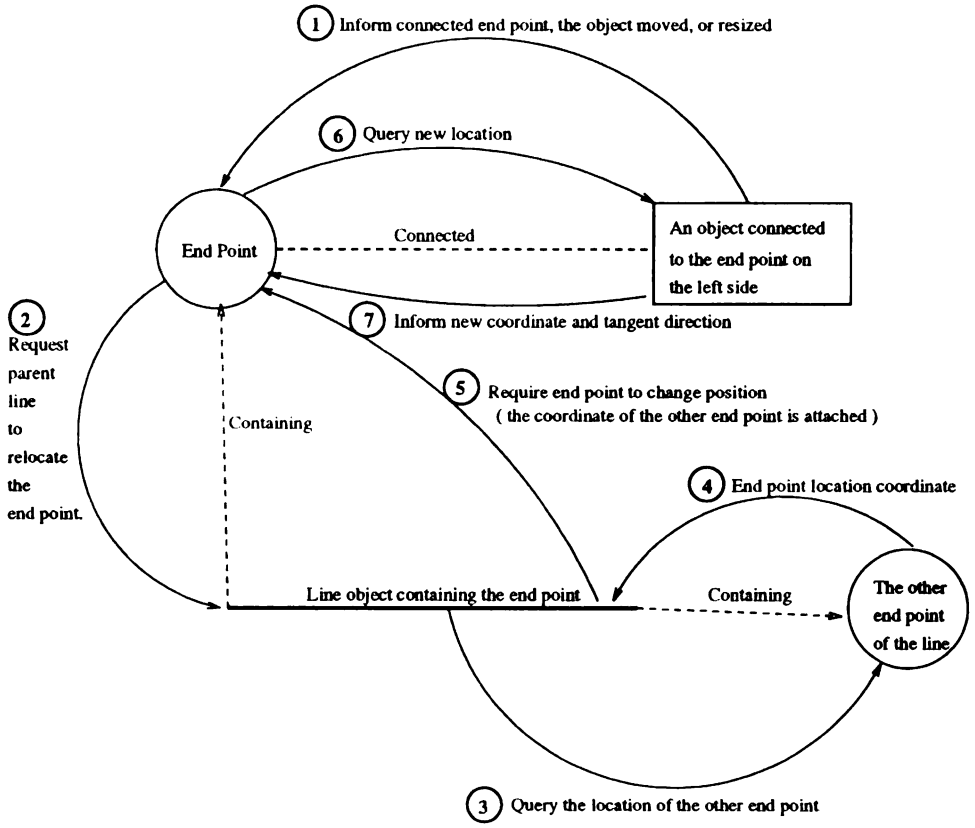

Figure 3.7. The communication among *end points*, *connected object*, and *line* to repose the end point

formal semantics is given, formal specifications can be derived, which can be used to facilitate the system analysis and design processes. For a given formalization of the diagrams, there may potentially be more than one algebraic language that can be used to represent the semantics (e.g., OBJ3 [23], Larch). Since the project is not restricted to a specific formal specification language, the component used to extract formal specifications from the diagram is separated from the diagramming module. Instead, intermediate text descriptions of the diagram, written in a well-defined language, is analyzed and processed by a parser to generate particular formal specifications.

3.2.1 Icon level attributes

Since there is no global view of the diagram in the diagramming phase, the textual descriptions derived from the visual representation also largely describe the attributes of those individual graphical icons. The textual attributes include *object type*, *name*, *identification number*, and *identification numbers* of those objects that are connected, where identification numbers play a key role in obtaining a global view of the diagram.

3.2.2 Diagram level attributes

One of the diagram level attributes is the *diagram name*. Graphical object identification numbers can also be regarded as a global attribute. A parser specific to a certain formal language relates the connected objects in its global tables obtained from the textual descriptions of diagrams to achieve a global understanding of the inter-relationships among the objects, and generates formal specifications.

3.3 Parser

Unlike the graphical editor, which deals with the syntactic aspect of a visual formalism, the parser is the crucial component in our framework that interprets the

diagrams and generates the corresponding formal specifications. More succinctly, the formal semantics of a visual formalism is imposed by the parser. As a consequence, the parser becomes the component that requires the most effort when constructing a visual formalism, using components of a graphical icon library to build a graphical editor.

The complete BNF grammar that describes the diagrams is given in Figures 3.8 and 3.9. The BNF grammar forms the syntactic part of a parser, while the formal semantics of a certain visual formalism is manually coded into the parser. At present, the parser is implemented with LEX and YACC. However, users have the flexibility to implement the parser with other tools.

$$
\begin{aligned}
< diagram > &\Rightarrow BEGIN\ DIAGRAM\ < sign > < state\ name > \\
&\quad < object >^{+}\ END \\
< state\ name > &\Rightarrow\ < identifier > < parameter\ part > \\
< object > &\Rightarrow\ < class > \mid < n\_ary\ association > \mid \\
&\quad < binary\ association > \mid < aggregate > \mid \\
&\quad < super\ sub\ class > \mid < state > \mid < instance > \mid \\
&\quad < line > \mid < endpoint > \\
< class > &\Rightarrow\ BEGIN\ CLASS\ < object\ id > < identifier > \\
&\quad < attributes > < operations > < endpoint\ ids > \ END \\
< n\_ary\ association > &\Rightarrow\ BEGIN\ N\_ARY\_ASSOCIATION\ < object\ id > \\
&\quad < sign > < identifier > < attributes > < operations > \\
&\quad < endpoint\ ids > \ END \\
< binary\ association > &\Rightarrow\ BEGIN\ BINARY\_ASSOCIATION\ < object\ id > \\
&\quad < sign > < identifier > < attributes > < operations > \\
&\quad < object\ id > < object\ id > \ END \\
< aggregate > &\Rightarrow\ BEGIN\ AGGREGATE\ < sign > < object\ id > \\
&\quad < multiplicity > < object\ id > < endpoint\ ids > \ END \\
< super\ sub\ class > &\Rightarrow\ BEGIN\ SUPERSUBCLASS\ < object\ id > \\
&\quad < super\ subclasses > \ END \\
< super\ subclasses > &\Rightarrow\ < superclass\ ids > < subclass\ ids > \\
< superclass\ ids > &\Rightarrow\ < superclass\ ids > < superclass\ id > \\
< superclass\ id > &\Rightarrow\ SUPER\ < object\ id > \\
< subclass\ ids > &\Rightarrow\ < subclass\ ids > < subclass\ id > \mid < subclass\ id > \\
< subclass\ id > &\Rightarrow\ SUB\ < object\ id > \\
< state > &\Rightarrow\ BEGIN\ STATE\ < object\ id > < identifier > \\
&\quad < endpoint\ ids > \ END
\end{aligned}
$$

Figure 3.8. The BNF grammar that describes the diagrams

$$
\begin{aligned}
< instance > &\Rightarrow BEGIN\ INSTANCE\ < object\ id >< identifier > \\
&\quad < endpoint\ ids >\ END \\
< line > &\Rightarrow BEGIN\ LINE\ < object\ id >< identifier > \\
&\quad < object\ id >< object\ id >\ END \\
< endpoint > &\Rightarrow BEGIN\ ENDPOINT\ < object\ id >< types > \\
&\quad < object\ id >< object\ id >\ END \\
< types > &\Rightarrow MULTIPLICITY\ < multiplicity > \mid ARROW \mid \\
&\quad DOUBLE\_ARROW \\
< multiplicity > &\Rightarrow ONE \mid MANY \mid ONE\_OR\_MORE \mid OPTIONAL \\
< attributes > &\Rightarrow BEGIN\ ATTRIBUTE\ < strings >\ END \\
< operations > &\Rightarrow BEGIN\ OPERATION\ < strings >\ END \\
< strings > &\Rightarrow < strings >< string > \mid < string > \\
< string > &\Rightarrow BEGIN\ < identifier >^{*}\ END \\
< parameterpart > &\Rightarrow BEGINBRACE\ < parameters >\ ENDBRACE \\
< parameters > &\Rightarrow < parameters >< identifier > \mid < identifier > \\
< endpoint\ ids > &\Rightarrow < endpoint\ ids >< object\ id > \\
< identifier > &\Rightarrow IDEN \\
< object\ id > &\Rightarrow NUMBER \\
< sign > &\Rightarrow NUMBER
\end{aligned}
$$

Figure 3.9. The BNF grammar that describes the diagrams (continued)

CHAPTER 4

Object Diagrams

The object model is the most important of the three diagrams of OMT. The first tool of VISUALSPECS, based upon the graphical environment, facilitates the diagramming of the object model and generates formal specifications from the object diagrams. The remainder of this chapter discusses the object model notation, including examples. We present an overview of the formalization of the object model developed by Bourdeau and Cheng [24, 25, 26] in terms of a well-defined syntax and the corresponding semantics. It is precisely this formalization that is used as the basis of the specifications that we generate automatically for diagrams constructed by the user.

4.1 Graphical depiction and formal specification generation

Bourdeau and Cheng [24] identified a subset of the object model notation appropriate for describing requirements. In object diagrams, rectangles enclose the names of *classes*, where a class describes a particular type of object in the system. Relationships between objects, called *associations* in OMT, are specified by connecting the classes involved in the relationship by a line with the name of the association centered on the line. Object models expressed in the context of requirements analysis are referred

to as A-schemata to represent *analysis object schemata.* An A-schema describes the static structure of a system. It consists of a set of classes and associations among those classes.

In Bourdeau and Cheng's formalization, the semantics of the A-schemata and instance diagram notations are described by an algebraic formalization. A graphical overview of this formalization process is given in Figure 4.1.

In this figure, the arrow labeled "OMT semantics" represents the currently informal concept of consistency between an instance diagram and an A-schema. The arrow labeled "algebraic semantics" represents the formal concept of consistency between an algebra and an algebraic specification that has been well-developed in the literature [27]. In Bourdeau and Cheng's formalization, A-schemata are formalized as algebraic specifications, and instance diagrams are formalized as algebras. As a result, the OMT semantics, which were previously not well-defined, can now be described in terms of an algebraic semantics.

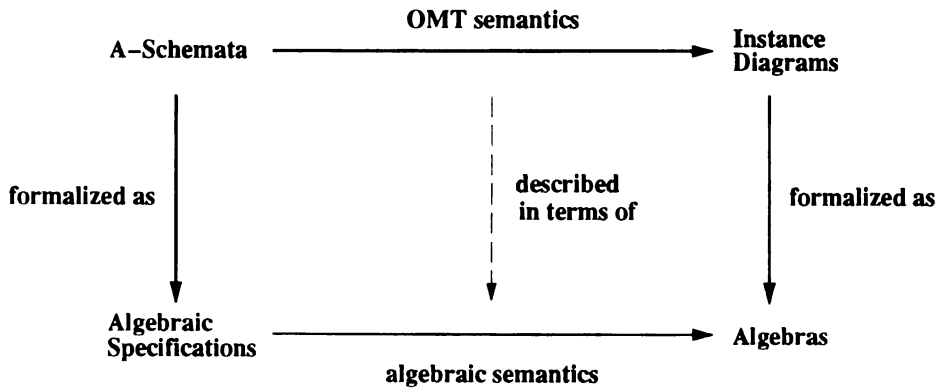

Figure 4.1. Basic approach to formalization.

First, the semantics of classes, objects, and object-states will be given. Next, the semantics of associations will be addressed, and finally, the combination all of the formalizations will be presented in order to describe the semantics of A-schemata.

Bourdeau and Cheng use the *Larch Shared Language* (LSL) [28] to illustrate how these basic formalisms are incorporated into a structured, algebraic specification.

SPECNAME (*parameters*) : `trait`
`includes`
list of pre-existing specification modules to be used
`introduces`
syntax declarations for functions are listed here
`asserts`
axioms are listed here

SPECNAME is the name of the specification module. The sorts that are to be considered as the parameters of the module are given in the parameter list following the name of the trait. *Includes* indicates other traits on which the given trait is built. The *introduces* section itemizes *function signatures*, each of which gives the number and types of input arguments and result type of a function. *Asserts* defines the constraints for the specification. When using LSL, one assumes that a basic axiomatization of Boolean algebra is part of every trait. This axiomatization includes the sort *BOOL*, the Boolean constants *true* and *false*, the connectives '$\wedge$' and '$\vee$', implication '$\Rightarrow$', and negation '$\neg$'.

4.1.1 Semantics for Classes and Object States

Let $\mathcal{S}$ be an A-schema, and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of class names given in $\mathcal{S}$. Formally, each class name $C_i \in \mathcal{C}$, where $1 \leq i \leq n$, is considered to be the name of a *sort* (type). For each class name C_i, a sort C_i-*STATES* is introduced, which characterizes the set of states that are possible for any C_i-object. For each object-state s of class C_i, s is specified with the signature (syntax and type specifications for input

arguments and output value)

$$s: \rightarrow C_i\text{-}STATES \quad .$$

States are nullary functions with no input arguments, therefore they are considered to be constants. For every class C_i, the set of possible states defined by C_i-*STATES* must include a state $undef_{C_i}$, in which case the state of C_i is *undefined.* The corresponding signature is

$$undef_{C_i}: \quad \rightarrow C_i\text{-}STATES \quad .$$

For every pair of object-states s_1 and s_2 of C_i (including $undef_{C_i}$), $s_1 \neq s_2$. In order to bind a C_i-object to one of its possible states, a *valuation function*, \$, is introduced with the signature

$$\$: C_i \rightarrow C_i\text{-}STATES \quad ,$$

for each class name $C_i \in \mathcal{C}$. Figure 4.2 contains a user constructed A-schema $\mathcal{S}_3$, containing object-states s_1 and s_2, where VISUALSPECS generates the corresponding specification according to the above formalizations.

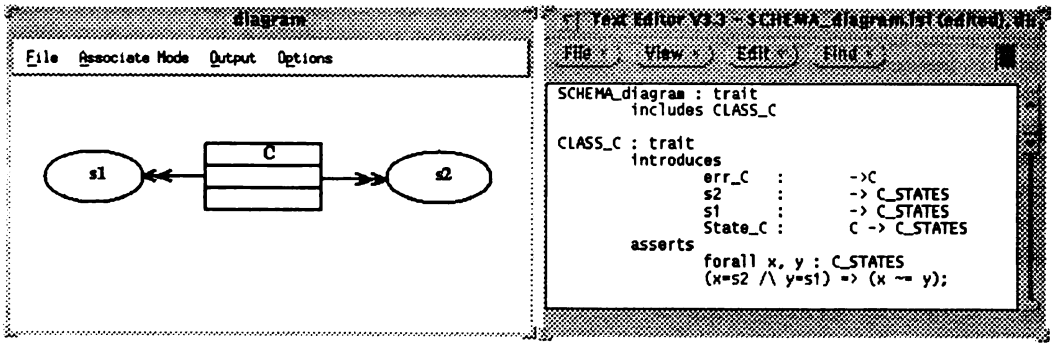

Figure 4.2. Schema $\mathcal{S}_3$ containing object states and its corresponding specification

When using VISUALSPECS [29], the user is presented with a single diagram canvas or sheet [9] to begin the graphical depiction of the requirements of a system. There are four types of options for the user. First, the "file" options allow the user to load an existing diagram, save the current diagram, or generate the (algebraic) formal specifications for the current diagram. Second, two different categories of associations may be selected when two graphical icons are related: binary or aggregate. In both cases, the default is a one-to-one relationship, but the user may change each endpoint individually. Third, the user has several different output options for documentation preparation or integration with other tools, such as a postscript representation of the diagram, a Latex [30] representation of specifications (a text formatting language), and the Larch specification language [3]. VISUALSPECS has been implemented such that most algebraic languages can be generated to describe the diagrams, given that the language has a well-defined grammar. Larch is specifically provided as an option in the current prototype in order to facilitate the integration of VISUALSPECS with other tools that have been developed by the Software Engineering Research Group in Michigan State University, such as LDE, an integrated environment for Larch specifications, including a syntax and type checker [31], theorem prover [11], and a graphical browser for existing specifications. Fourth, users have the options to the specific font and font size for the text in a diagram. The default font is *Roman-Times*; the default font size is 12 point.

Since there were three object-states in the A-schema, $undef_C$, s_1, and s_2, three inequality axioms are needed to establish uniqueness among object-states. (The name of a specification is constructed by adding a prefix 'CLASS-' to the name of the respective class.)

4.1.2 Semantics for Associations

An instance of an n-ary association R consists of a set of *R-links*, where an R-link is an n-tuple of object names. For example, the subgraph in Figure 4.3 depicts a single *stops*-link, whose 2-tuple can be formally represented by the newly created two-argument *stops* predicate.

$$\begin{aligned} stops(b_1, w_1) &\stackrel{def}{=} true \\ stops(b_2, w_2) &\stackrel{def}{=} true \\ stops(b_3, w_3) &\stackrel{def}{=} true \\ stops(b_4, w_4) &\stackrel{def}{=} true \\ stops(x, y) &\stackrel{def}{=} false \quad \text{when the pair } (x, y) \text{ is not one of those above.} \end{aligned} \tag{4.1}$$

Similarly, the *stops* predicates for Figure 4.4 are :

$$\begin{aligned} stops(b_2, w_2) &\stackrel{def}{=} true \\ stops(b_3, w_2) &\stackrel{def}{=} true \\ stops(x, y) &\stackrel{def}{=} false \quad \text{when the pair } (x, y) \text{ is not one of those above.} \end{aligned} \tag{4.2}$$

The predicate for the *stops* association has the signature:

$$stops : Brake, Wheel \rightarrow BOOL \quad .$$

If an association name R is used to name several different associations, then VISUALSPECS uses a naming convention that includes associated class names as part of the name given to its specification.

Multiplicity constraints are described in terms of four relational properties: *functional*, *injective*, *surjective*, and *total*, whose properties have been formalized as predicates and incorporated into VISUALSPECS. Figure 4.5 contains a summary of the formalization of the endpoints. A reference to any of the four relational predicates, for example

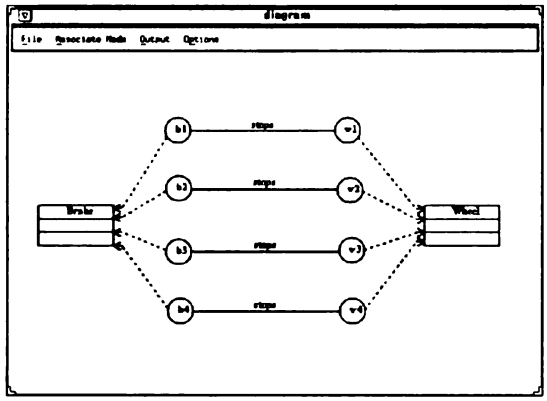

Figure 4.3. An instance diagram $\mathcal{I}_1$ of the A-schema $\mathcal{S}_1$

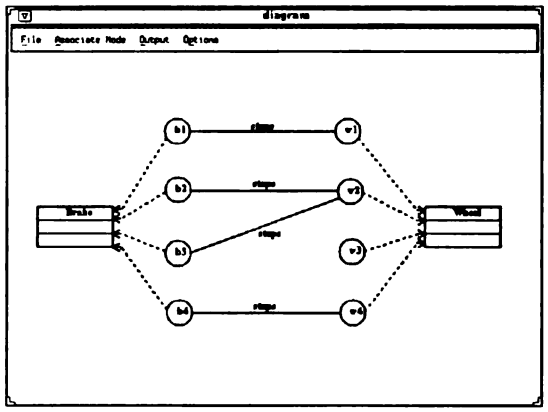

Figure 4.4. An instance diagram, $\mathcal{I}_2$, inconsistent with the A-schema $\mathcal{S}_1$.

$$\textbf{total}(stops, Brake, Wheel) \quad , \tag{4.3}$$

is replaced with the corresponding predicate definition to obtain the given expression,

$$(\forall b_1 : Brake \ . \ (\exists w_1 : Wheel \ . \ stops(b_1, w_1))) \quad , \tag{4.4}$$

which is "skolemized" (removal of the existential quantifier '$\exists$') to obtain

$$(\forall b_1 : Brake \ . \ stops(e_1, skolem(b_1))) \quad ,$$

where the signature of the Skolem function *skolem* is given by

$$skolem : Brake \rightarrow Wheel \quad .$$

Given the formalization techniques presented thus far, it is possible to obtain several equivalent specifications for this diagram, which differ in the way they are modularized. As a convention, VISUALSPECS extracts the specifications corresponding to

classes first, followed by the specifications for the associations, which are extracted by importing the relevant class specifications. The specification for a class C is named *CLASS-C*, and that for an association R is named *ASSOCIATION-R*. In addition to the specifications for the constraints on the endpoints of the association, the specifications for the associated classes are included via the "**extends**" keyword and the '+' operator.

Let R be a predicate (denoting an association) with signature

$$R : A, B \rightarrow BOOL \quad .$$

The statement $(\forall x : X \ . \ P)$ is read "for any element x of sort X, the statement P evaluates to true," and a similar reading applies when the existential quantifier '$\exists$' is used in place of '$\forall$'.

R **is** *functional* **from** A **to** B: if every element of A is related to at most one element of B:

$$\textbf{functional}(R,A,B) \stackrel{def}{=} (\forall a : A,\ x,y : B \ . \ (R(a,x) \wedge R(a,y) \Rightarrow x = y)) \quad . \tag{4.5}$$

R **is** *injective* **from** A **to** B: if every element of B is related to at most one element of A:

$$\textbf{injective}(R,A,B) \stackrel{def}{=} (\forall x,y : A,\ b : B \ . \ (R(x,b) \wedge R(y,b) \Rightarrow x = y)) \quad . \tag{4.6}$$

R **is** *surjective* **from** A **to** B: if every element of B is related to some element of A:

$$\textbf{surjective}(R,A,B) \stackrel{def}{=} (\forall b : B \ . \ (\exists a : A \ . \ R(a,b))) \quad . \tag{4.7}$$

R **is** *total* **from** A **to** B: if every element of A is related to some element of B, or formally:

$$\textbf{total}(R,A,B) \stackrel{def}{=} (\forall a : A \ . \ (\exists b : B \ . \ R(a,b))) \quad . \tag{4.8}$$

Figure 4.5. Formalization of relations that characterize endpoints

---

An aggregation association between two classes A and B is defined to be a general binary association between A and B having the name *hasPart*. A part name is appropriately formalized as a mapping from a set of aggregate objects to a set of their parts. Axioms are added to ensure consistency between aggregate parts and the aggregation association. Each part name of an aggregate object must correspond to a distinct part. Figure 4.6 summarizes the A-schemata semantics.

Definition (Semantics of A-schemata, part 1) :
Let $\mathcal{S}$ be an A-schema. The semantics of $\mathcal{S}$ is an algebraic specification satisfying the following data.

(SS1) Each class C in the schema $\mathcal{S}$ is denoted by a sort of the same name.

(SS2) For each class C, a sort *C-STATES* is introduced as well as a nullary function $undef_C$ having signature $undef_C : \rightarrow C\text{-}STATES$.

(SS3) Each object-state s, for which a double-headed arrow leads from a class C to the oval containing s, is denoted by a function with signature $s : \rightarrow C\text{-}STATES$, and for every pair of object-states s_1 and s_2, the axiom $s_1 \neq s_2$ is included.

Definition (Semantics of A-schemata, part 2) :
Let $\mathcal{S}$ be an A-schema, and let R be a k-ary association in $\mathcal{S}$ relating objects from classes $D_1, \ldots, D_k$. The semantics of $\mathcal{S}$ is an algebraic specification satisfying requirements (SS1)-(SS3), given earlier, and the requirements below:

(SS4) Association R is denoted by the predicate $R : D_1, \ldots, D_k \rightarrow BOOL$.

(SS5) The multiplicity constraints of R are denoted by a set of axioms derived from R using the basis schemata, relational predicates, unfolding, and skolemization.

Figure 4.6. The semantics of A-schemata

4.2 Consistency between Class and Instance Diagrams

Next, the relationships between algebraic specifications and algebras are discussed [24]. When defining an *algebra* for an instance diagram, VISUALSPECS uses the types and function signatures from the corresponding original schema diagram as specified by its algebraic specification. An algebra differs from an algebraic specification in that a set of elements are specified for each sort name in an algebra, and a function is defined on these sets for each function symbol of the algebraic specification. An algebra $\mathcal{A}$ is said to be *consistent* with its specification $\mathcal{S}$, if the axioms of $\mathcal{S}$ are satisfied by the functions defined in the algebra $\mathcal{A}$. In general, any instance diagram can be denoted by an algebra. Due to space constraints, the detailed descriptions of the formalization of instance diagrams in terms of algebras are omitted, but can be found in [24]. Figure 4.7 summarizes the instance diagram semantics.

4.3 An example of a simplified automobile system

We illustrate the use of the notation and its corresponding formalization with a simplified automobile transportation system. Figure 4.8 depicts a simple transportation system.

In order to construct the specification for a given A-schema, VISUALSPECS performs a union of the specifications, as indicated by the operator '+", of its classes and associations.

Figure 4.9 contains the corresponding formal specification of the more complex object model for the automobile transportation system modeled in Figure 4.8. The axioms of this specification are those that are imported from the respective class and association specifications.

Definition (Semantics of Instance Diagrams, part 1) :
Let $\mathcal{S}$ be an A-schema, and let $\mathcal{I}$ be an instance diagram. The algebraic denotation of $\mathcal{I}$, with respect to schema $\mathcal{S}$, is given (in part) by the following data.

(IS1) For each class C in the schema $\mathcal{S}$, the data value set $C^{\mathcal{I}}$ is the set of objects $\{x_1, \ldots, x_k\}$ given in $\mathcal{I}$ for which a dashed arrow leads from each x_i to C.

(IS2) The data value set $C\text{-}STATES^{\mathcal{I}}$ is the set of object-state names, including $undef_C$, given in $\mathcal{S}$ for w which a double-headed arrow leads from C to the object-state.

(IS3) For each object e of class C in $\mathcal{I}$, if there is a double-headed arrow leading from e to some object-state s, then $\$^{\mathcal{I}}(e) \stackrel{def}{=} s$, otherwise, $\$^{\mathcal{I}}(e) \stackrel{def}{=} undef_C$.

(IS4) For each object-state s, $s^{\mathcal{I}} \stackrel{def}{=} s$.

Definition (Semantics of Instance Diagrams, part 2) :
Let $\mathcal{S}$ be an A-schema, and let R be a k-ary association in $\mathcal{S}$ relating objects from classes $D_1, \ldots, D_k$. Let $\mathcal{I}$ be an instance diagram. The algebraic denotation of $\mathcal{I}$, with respect to schema $\mathcal{S}$, is given by (IS1)-(IS4), and the following data.

(IS5) For each R-link in $\mathcal{I}$, relating objects $d_1 : D_1, \ldots, d_k : D_k$, $R^{\mathcal{I}}(d_1, \ldots, d_k) \stackrel{def}{=} true$.

(IS6) For each tuple of objects $d_1 : D_1, \ldots, d_k : D_k$, if $\mathcal{I}$ has no R-link relating these objects, then $R^{\mathcal{I}}(d_1, \ldots, d_k) \stackrel{def}{=} false$.

Figure 4.7. The semantics of simple instance diagrams.

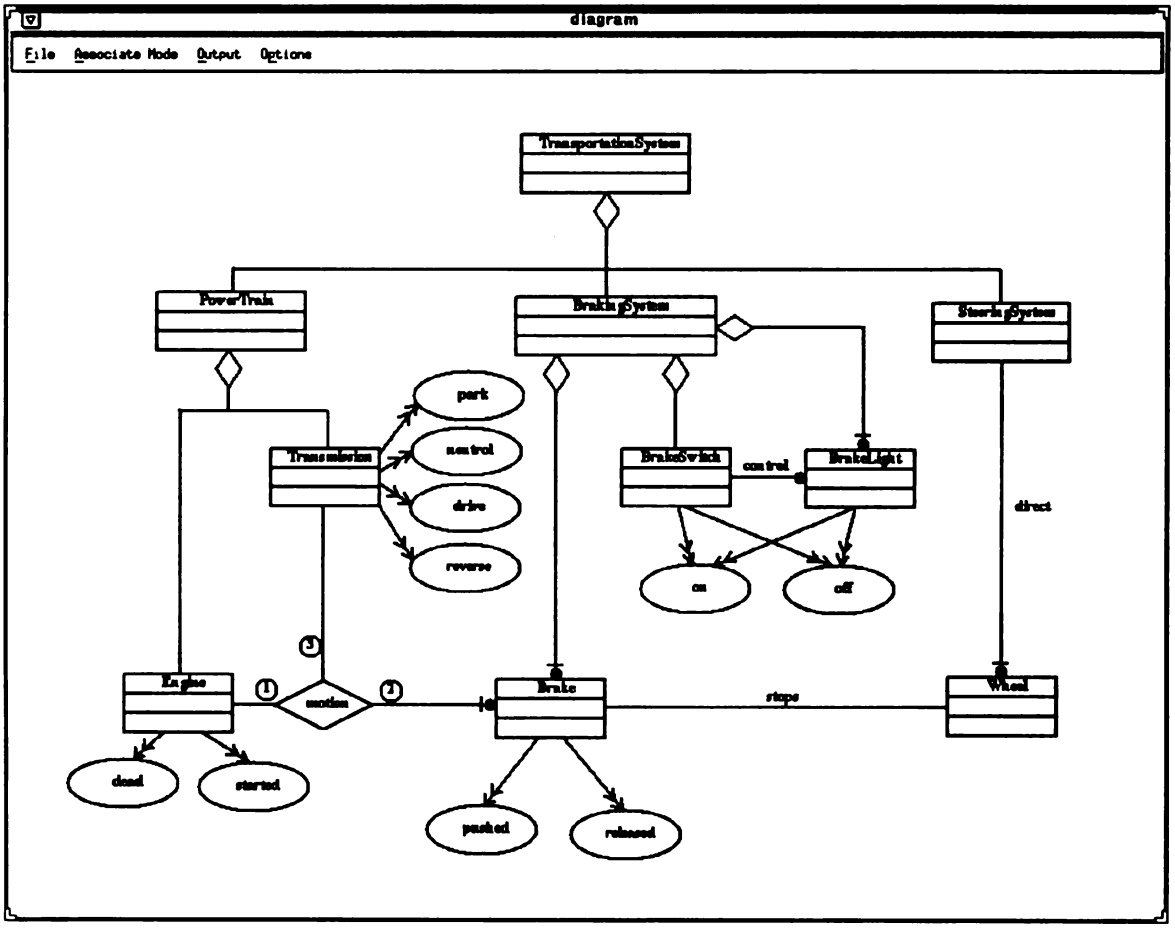

Figure 4.8. A simplified automobile transportation system

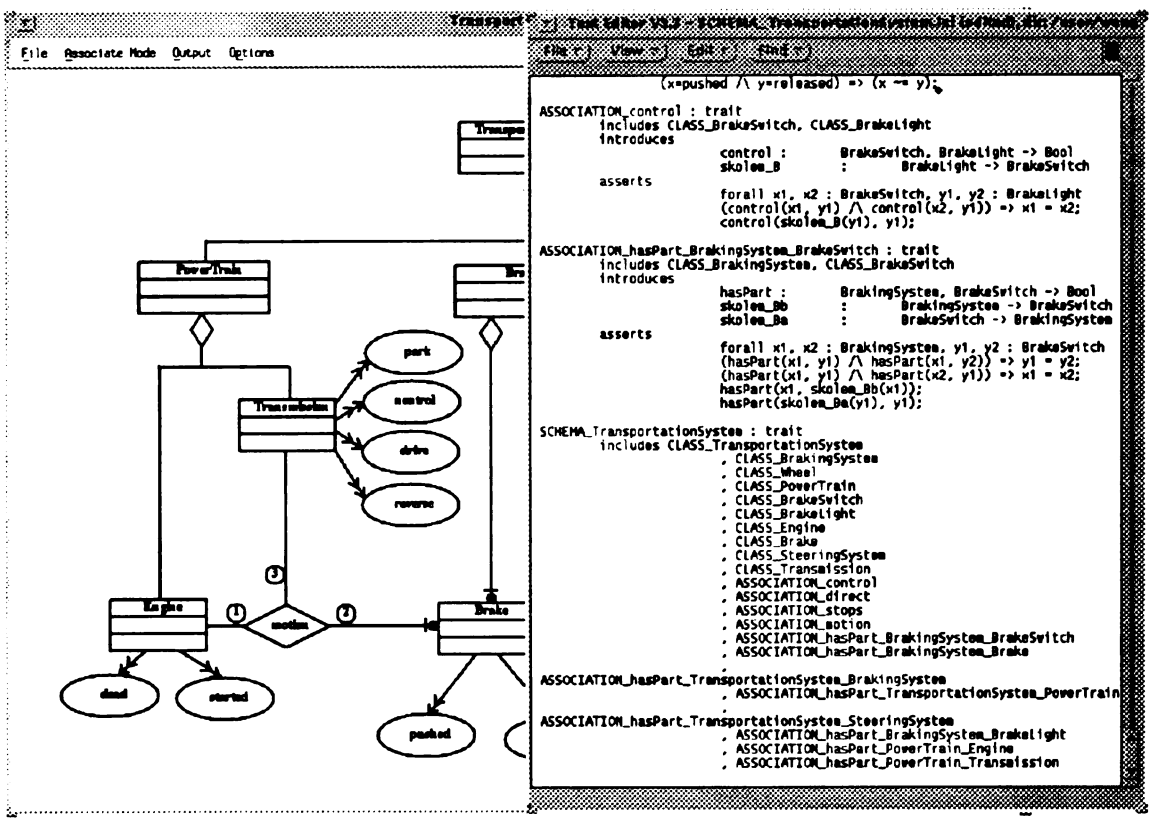

Figure 4.9. Formal specifications of the simplified automobile object model

4.3.1 Classes and Binary Associations

Figure 4.10 contains a sample session with VISUALSPECS, where the *stops* association relates the *brake* and *wheel* components of the transportation control system. This diagram is intended to express that a single *brake* controls a single *wheel*. Its corresponding Larch specification is given in Figure 4.11.

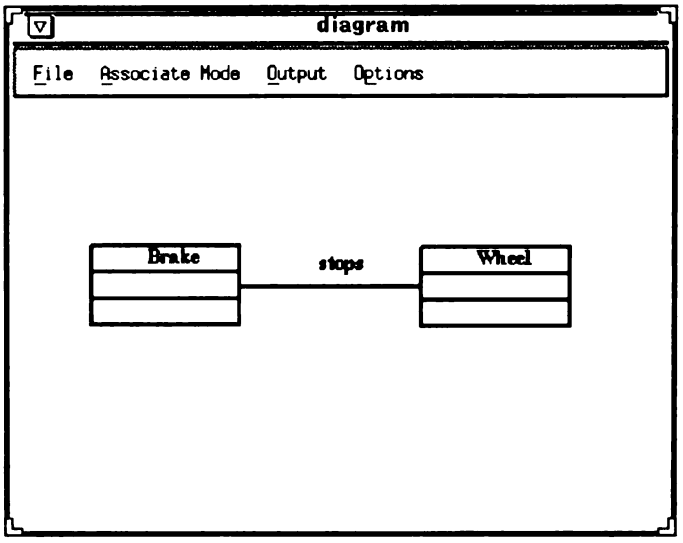

Figure 4.10. A simple A-schema, $\mathcal{S}_1$

4.3.1.1 Diagram Consistency

Given an A-schema, it is possible to construct examples of consistent and inconsistent *instance diagrams* for that schema, where an instance diagram describes how a particular set of objects relate to each other. An instance diagram is said to be consistent with an A-schema if it represents a valid system state. For each class C of the A-schema, the instance diagram has a set of named objects represented by

```
ASSOCIATION_stops : trait
    includes CLASS_Brake, CLASS_Wheel
    introduces
            stops   :       Brake, Wheel -> Bool
            skolem_W    :       Brake -> Wheel
            skolem_B    :       Wheel -> Brake
    asserts
            forall x1, x2 : Brake, y1, y2 : Wheel
            (stops(x1, y1) /\ stops(x1, y2)) => y1 = y2;
            (stops(x1, y1) /\ stops(x2, y1)) => x1 = x2;
            stops(x1, skolem_W(x1));
            stops(skolem_B(y1), y1);
```

Figure 4.11. The corresponding Larch specification of the simple A-schema, $\mathcal{S}_1$

circles with each object's name inscribed. Continuing with our example, the instance diagram $\mathcal{I}_1$ shown in Figure 4.3 represents four brakes (b_1, b_2, b_3, and b_4) and four wheels (w_1, w_2, w_3, and w_4). As shown, a dashed arrow connects each object to its class. In a consistent instance diagram, each association specified by the schema must be satisfied. For example, since the *stop* association is a *one-to-one* correspondence between *Brake* and *Wheel* objects, Figure 4.10, the instance diagram, $\mathcal{I}_2$, in Figure 4.4 is inconsistent with the A-schema $\mathcal{S}_1$ (wheel $w2$ is controlled by brakes $b1$ and $b2$, thus violating the one-to-one constraint on the *stops* association). Due to the simplicity of this example, we are able to detect the inconsistency visually; however, for more complex diagrams, relying on visual techniques is clearly inadequate for detecting inconsistencies.

4.3.2 Aggregation

An *aggregation association* is a relationship between two objects, where one of the objects is considered to be a part of the other. For example, parts of a brake system would include brakes, a brake switch, and brake lights. The notation used to specify this type of association is illustrated in Figure 4.12, where the diamond-headed line denotes the aggregation association; the *Associate Mode* menu is displayed indicating that the user can select either a binary or an aggregate association between two classes. This figure asserts that the class *BrakeSystem* has at least one brake as its parts. Endpoints are used to specify *multiplicity constraints* on an association, which describe a restriction on the number of objects from one class that may be associated with any one object from another class. Figure 4.13 contains its Larch specification.

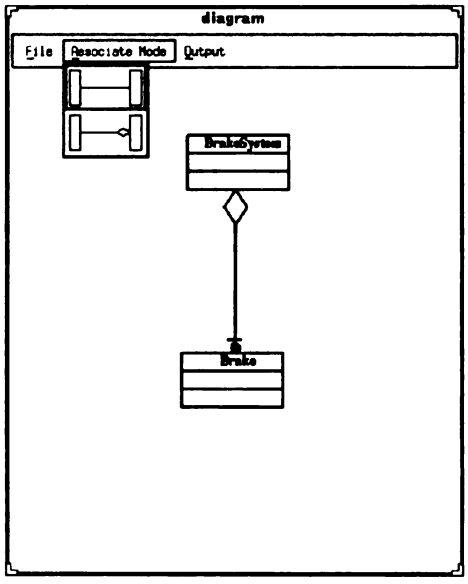

Figure 4.12. An aggregation association.

```
ASSOCIATION_hasPart_BrakingSystem_Brake : trait
    includes CLASS_BrakingSystem, CLASS_Brake
    introduces
            hasPart :       BrakingSystem, Brake -> Bool
            skolem_Bb       :       BrakingSystem -> Brake
            skolem_Ba       :       Brake -> BrakingSystem
    asserts
            forall x1, x2 : BrakingSystem, y1, y2 : Brake
            (hasPart(x1, y1) /\ hasPart(x2, y1)) => x1 = x2;
            hasPart(x1, skolem_Bb(x1));
            hasPart(skolem_Ba(y1), y1);
```

Figure 4.13. The corresponding Larch specification of the aggregation association.

4.3.3 Object States

The OMT notation describes the system in terms of the relationships between objects. For some classes, however, it may be necessary to describe allowable states for their objects. Consider the brake light, which should indicate that the automobile is slowing down when the brake pedal is pressed. Such a light must be turned on whenever the brake is in effect. These states are expressed pictorially in Figure 4.14; the double-headed arrows leading away from the class *BrakeLight* specify possible states for objects of that class. The text enclosed by an oval gives the name of an allowable object-state.* Figure 4.15 exhibits an instance diagram for the A-schema given in Figure 4.14, in which the current state of the BrakeLight-object named i is *on*.

*This notation is an extension to the OMT definition.

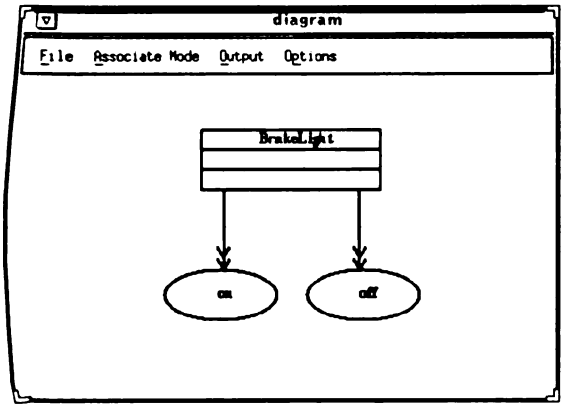

Figure 4.14. Notation for object states

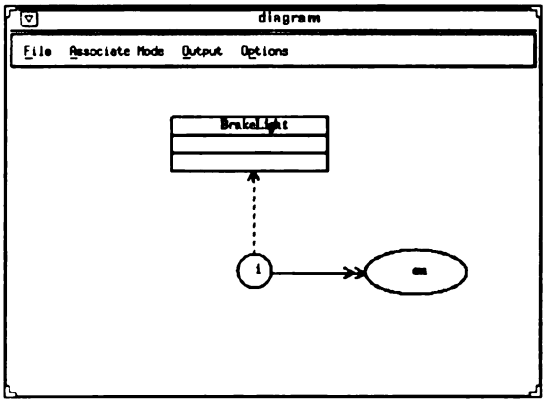

Figure 4.15. An example instance diagram

4.3.4 N-ary Associations

The notation used to express n-ary associations is slightly different than that used to express a binary association. For each class of the association, there is an edge connecting the class to a diamond enclosing the name of the association; the multiplicity constraints may be specified on either end of an edge. Figures 4.16 and 4.17, respectively, contain an example of a ternary association and its corresponding Larch specification. In this example, the multiplicity constraints on the edge connecting the class *Engine* to the *motion* association are interpreted as follows. Given any arbitrary *Engine* object e, there are one or more pairs of *Brake* and *Transmission* objects that are associated with e. Also, given any one pair of *Brake* and *Transmission* objects (b, s), there is at most one *Engine* object associated with (b, s). An analogous interpretation applies to the other edges of the association. The circled numbers around the *motion* diamond are designed to represent the order of the signature for the association *motion*.

Each edge in an n-ary association is read as a binary association relating objects of one class to tuples of objects of the remaining classes.

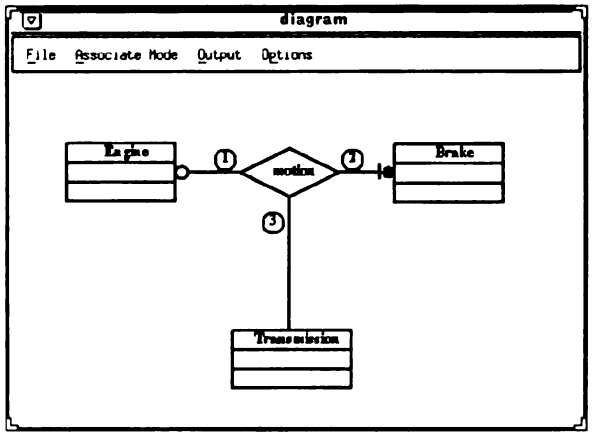

Figure 4.16. An A-schema $\mathcal{S}_2$ consisting of a simple ternary association.

The instance diagram notation for instances of n-ary associations is essentially the same as that used for binary associations. Figure 4.18 gives an instance diagram, $\mathcal{I}_3$, that is consistent with the A-schema $\mathcal{S}_2$ given in Figure 4.16. This instance diagram depicts a single engine, a transmission, two brakes, all of which are related by the respective *motion* relationships.

```
ASSOCIATION_motion : trait
    includes CLASS_Engine, CLASS_Brake, CLASS_Transmission
    introduces
            motion  :       Engine, Brake, Transmission -> Bool
            skolem_B    :       Engine -> Brake
            skolem_T    :       Engine -> Transmission
            skolem_E    :       Brake, Transmission -> Engine
            skolem_E    :       Brake -> Engine
            skolem_T    :       Brake -> Transmission
            skolem_B    :       Engine, Transmission -> Brake
            skolem_E    :       Transmission -> Engine
            skolem_B    :       Transmission -> Brake
            skolem_T    :       Engine, Brake -> Transmission
    asserts
        forall a1, a2 : Engine, b1, b2 : Brake, c1, c2 : Transmission
            %Multiplicity constraints for Engine-edge
            (motion(a1, b1, c1) /\ motion(a1, b2, c2))
                                        => (b1 = b2 /\ c1 = c2);
            (motion(a1, b1, c1) /\ motion(a2, b1, c1)) => a1 = a2;
            motion(a1, skolem_B(a1), skolem_T(a1));
            motion(skolem_E(b1, c1), b1, c1);
            %Multiplicity constraints for Brake-edge
            (motion(a1, b1, c1) /\ motion(a2, b1, c2))
                                        => (a1 = a2 /\ c1 = c2);
            motion(skolem_E(b1), b1, skolem_T(b1));
            motion(a1, skolem_B(a1, c1), c1);
            %Multiplicity constraints for Transmission-edge
            (motion(a1, b1, c1) /\ motion(a2, b2, c1))
                                        => (a1 = a2 /\ b1 = b2);
            (motion(a1, b1, c1) /\ motion(a1, b1, c2)) => c1 = c2;
            motion(skolem_E(c1), skolem_B(c1), c1);
            motion(a1, b1, skolem_T(a1, b1));
```

Figure 4.17. The corresponding Larch specification of the A-schema $\mathcal{S}_2$

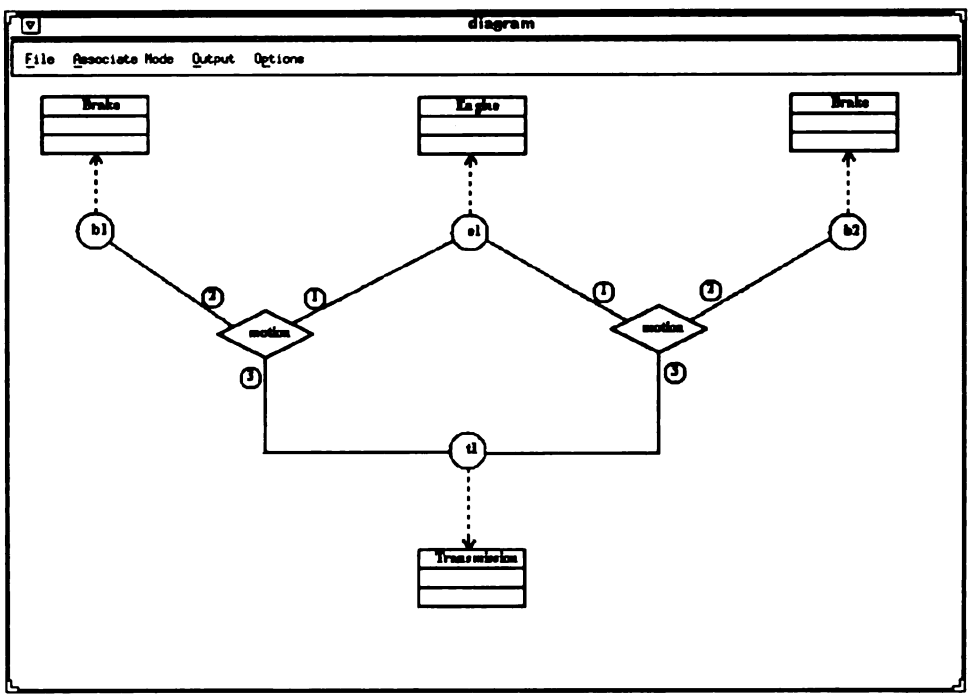

Figure 4.18. An instance diagram, $\mathcal{I}_3$, consistent with the A-schema $\mathcal{S}_2$

CHAPTER 5

State Schemas

In Chapter 4, we introduced the object model and instance diagram to depict the static aspects of a system. Instance diagrams, based on object models, are a set of possible states that a system may enter. Chapter 9 will show that the consistency between the formal specifications of an object model and that of its corresponding instance diagrams can be checked by Larch tools. This consistency checking capability enables a means for validating the object models obtained in the analysis phase. However, an instance diagram only represents one out of many possible system states. Since the dynamic model is another very important model for describing the behavioral features of a system with respect to state changes, we need a more powerful and general method other than the instance diagrams to describe the primitive system states.

The state schema developed by Bourdeau and Cheng [24] is a predicative definition of state that delineates a family of possible system states by predicates. Instance diagrams may be checked against several state schemas to determine whether the state schemas are satisfied. A state schema represents the primitive states in the dynamic model which is discussed in the next chapter. Thus, the state schemas provide a crucial link between object models and dynamic models. We divide the discussion on state schema into two sections. Section 5.1 introduces the syntax and

semantics for the state schema. Section 5.2 discusses how state schemas were added into VISUALSPECS.

5.1 Syntax and semantics

The state schema introduces a hybrid graphical notation to define the state-predicates. It is composed of the name of the schema with parameters attached and an instance diagram. The major syntactic difference between instance diagrams and state schemas is the naming mechanism and the negation symbol. Since the state schema is intended to use predicates to describe a family of states, parameters are allowed in the name structure of a state schema. These parameters can later be instantiated by the names of real objects to obtain a specific state description, which can be referenced to verify if a certain system state satisfies a specific state schema. Because state schemas correspond to predicates, there is a need to negate diagrams that represent sate schemas. The negation symbol '×' is used to negate either the whole state schema or associations within a state schema.

Figure 5.1 shows a simple state schema depicting a *braked* state of the transportation system presented in Chapter 4. b is the variable in the state schema that represents a predicate depicting the *braked* state. Figure 5.2 illustrates a scenario that the transportation system has a bad engine.

The semantics of a state schema is a typed predicate whose definition describes the situation depicted by the instance diagram enclosed in the state schema [26]. A state schema P with parameters $x_1 : X_1, x_2 : X_2, ..., x_n : X_n$, is expressed by a parameterized predicate as:

$$P : X_1, X_2, ..., X_n \longrightarrow BOOL \quad .$$

A formal semantics for state schemas has been defined in [26]. The definition of P can be obtained by systematically applying the following set of rules to the state

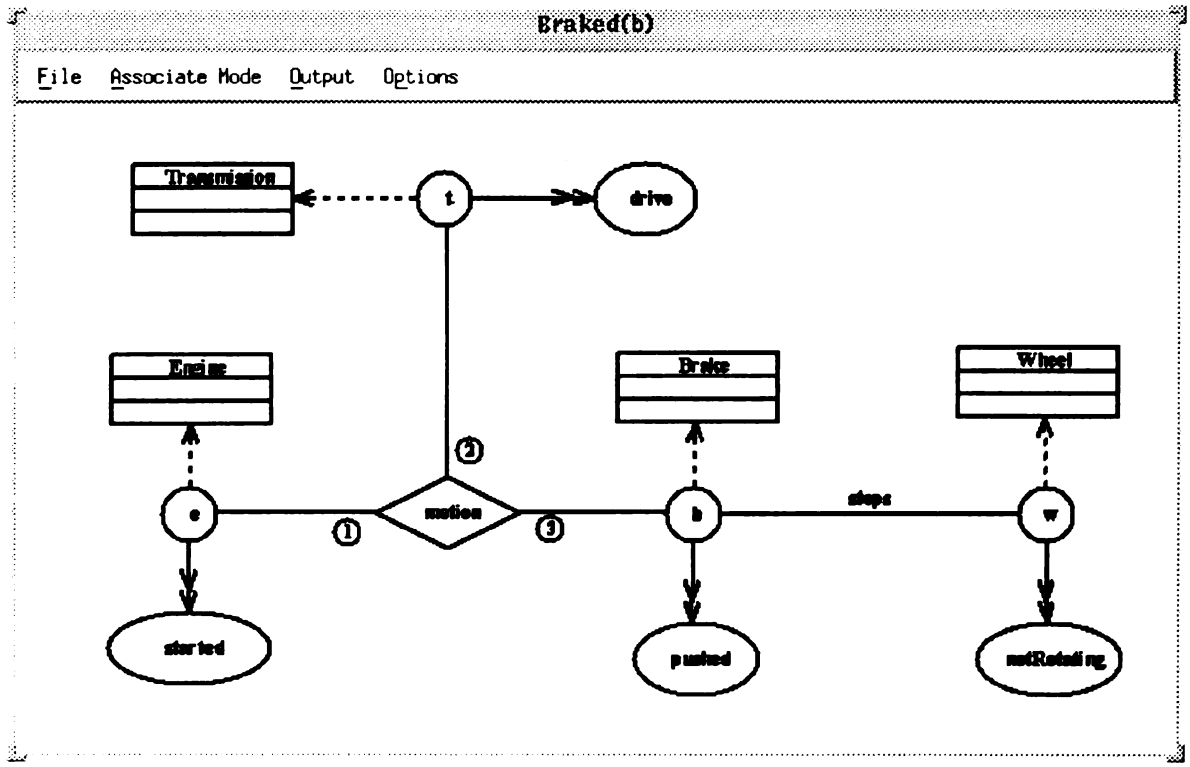

Figure 5.1. A state schema showing a braked state

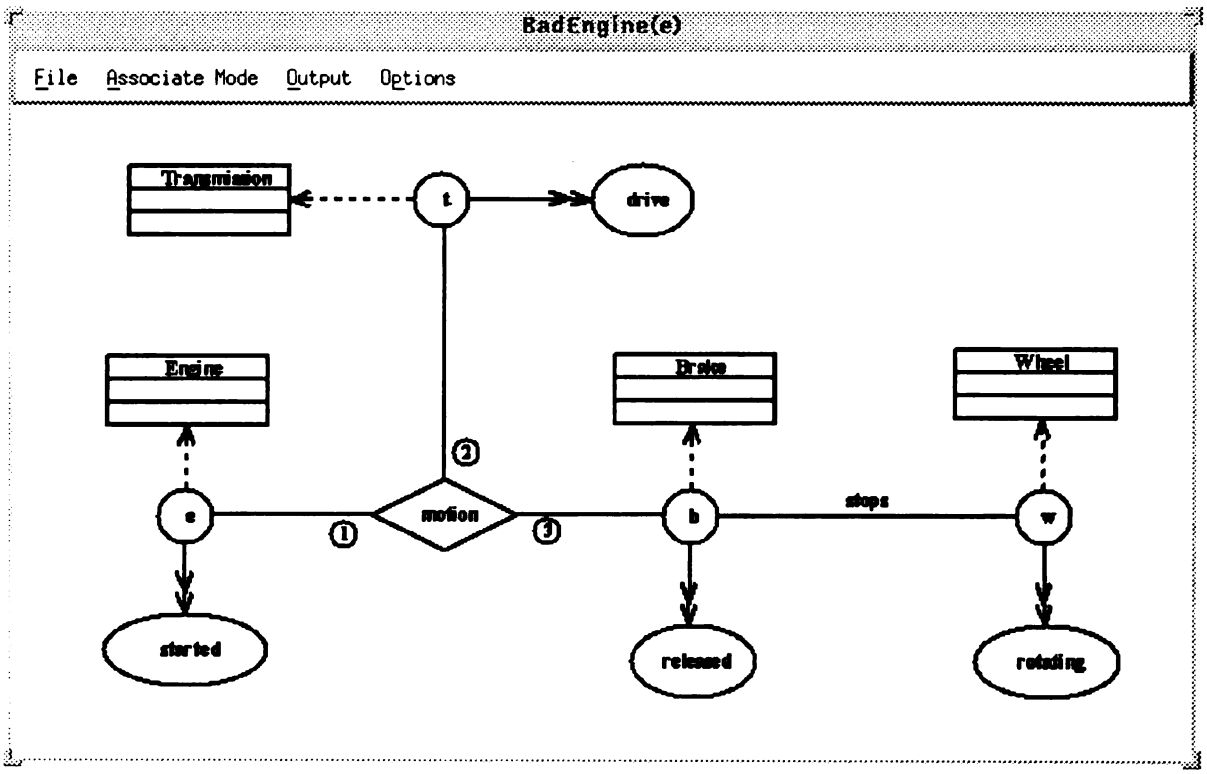

Figure 5.2. A state schema specifying a bad engine

schema. Generally, the definition of the state schema predicate P is presented in one of two forms

$$(\forall x_1 : X_1, x_2 : X_2, ..., x_n : X_n . P(x_1, x_2, ..., x_n) == \psi) \quad or \tag{5.1}$$

$$(\forall x_1 : X_1, x_2 : X_2, ..., x_n : X_n . P(x_1, x_2, ..., x_n) == \neg\psi) \tag{5.2}$$

which represent the state schemas without and with negation, respectively. The statement ψ is based on the instance diagram enclosed in the state schema. The format of ψ is:

$$prefix \ . \ matrix$$

where *prefix* is of the form:

$$(\exists y_1 : Y_1, y_2 : Y_2, ..., y_m : Y_m) \ ,$$

and *matrix* is a conjunction of literals. Figure 5.3 contains three rules to determine the *prefix* and *matrix* [32].

The Larch specifications of the state schemas in Figures 5.1 and 5.2 are shown in Figure 5.4 and 5.5 respectively. The binary association *stops* and the ternary association *motion* in both of the state schemas yield conjuncts *stops(b, w)* and *motion(e, t, b)*, respectively. The instances are evaluated to their corresponding states by operation *State_X* for *x: X*. Instance *b* of class *Brake* is universally quantified in Figure 5.4, while instance *e* of class *Engine* is universally quantified in Figure 5.5. The negation in state schema *BadEngine* creates a *\neg* (negation) in the definition of predicate *STATE_BadEngine*.

(SR1) For every free variable (appearing in the state schema but not in the parameters of predicate P) y of class Y in the state schema P, y: Y is in the prefix of ψ (y is existentially quantified).

(SR2) For each association and valuation (object states and attributes) in state schema P, a conjunct is contributed to the matrix of ψ. A visual object (association or valuation) with a negation symbol '×' derives a literal preceding with a '-' sign.

(SR3) For every object name x: X that is a free variable in the state schema P, the expression $x \neq err\_X$ is a conjunct of the matrix of ψ.

Figure 5.3. Rules to obtain the *prefix* and *matrix* parts of a specification from a state schema.

5.2 Implementation

Since there are only minor changes in the state schema with respect to what is contained in an instance diagram, the addition of state schema to VISUALSPECS was easily accomplished by reusing the design and code for the object model and instance diagrams. A few minor changes were made to the *graphical object* classes and the grammar used for the parser in order to meet the new syntax of the requirements of state schemas and to obtain the corresponding specifications.

A data field *sign*, representing the negation when it is set and its corresponding drawing operation have been added to the graphical object classes, *binary* and *n_ary* associations, *aggregate*, *diagram*, and *state evaluation*, addressed in Chapter 4. Figures 5.6, 5.7, and 5.8 demonstrate a procedure for negating the binary association *stops*.

Figures 5.9, 5.10, and 5.11 illustrates a similar procedure for negating a ternary association *motion*. Figure 5.9 gives the state schema before negating the association.

```
STATE_Braked(b): trait
  includes ASSOCIATION_stops
      , ASSOCIATION_motion
      , CLASS_Transmission
      , CLASS_Wheel
      , CLASS_Brake
      , CLASS_Engine

  introduces
    Braked : Brake-> BOOL

  asserts
    \forall b: Brake

      STATE_Braked(b) ==
        \exist w: Wheel, e: Engine,
               t: Transmission
        stops (b, w)
        \and motion (e, t, b)
        \and State_Brake (b) = pushed
        \and State_Wheel (w) = notRotating
        \and State_Engine (e) = started
        \and State_Transmission (t) = drive
        \and w \= err_Wheel
        \and e \= err_Engine
        \and t \= err_Transmission
```

Figure 5.4. The corresponding Larch specification of the *Braked* state schema

```
STATE_BadEngine(e): trait
  includes ASSOCIATION_stops
      , ASSOCIATION_motion
      , CLASS_Engine
      , CLASS_Wheel
      , CLASS_Transmission
      , CLASS_Brake

  introduces
    BadEngine : Engine-> BOOL

  asserts
    \forall e: Engine

      STATE_BadEngine(e) ==
        \neg (
        \exist w: Wheel, b: Brake,
               t: Transmission
        stops (b, w)
        \and motion (e, t, b)
        \and State_Wheel (w) = rotating
        \and State_Brake (b) = released
        \and State_Engine (e) = started
        \and State_Transmission (t) = drive
        \and w \= err_Wheel
        \and b \= err_Brake
        \and t \= err_Transmission
        )
```

Figure 5.5. Larch specification of the *BadEngine* state schema

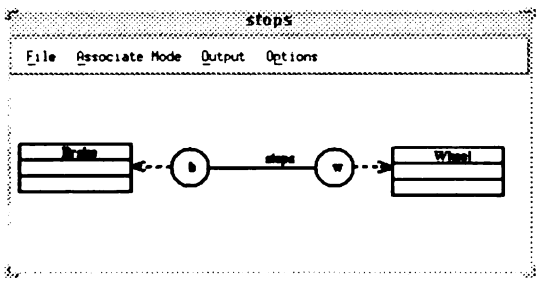

Figure 5.6. Before negating the binary association *stops*

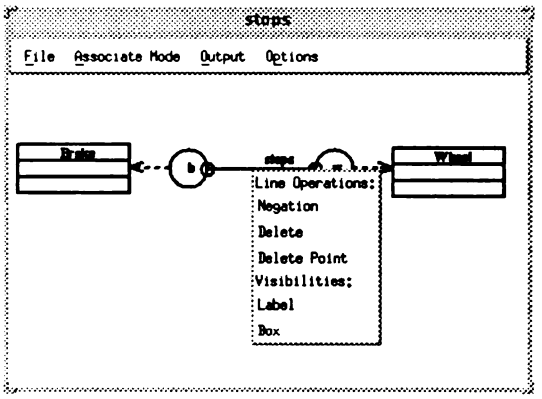

Figure 5.7. Negating the binary association *stops*

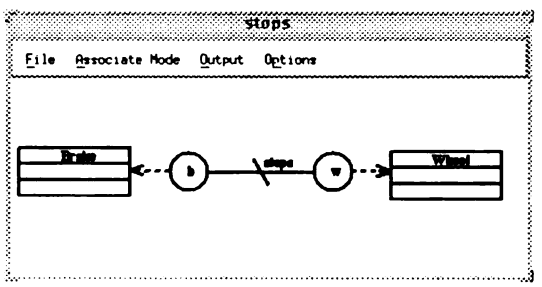

Figure 5.8. After negated the binary association *stops*

Figure 5.10 and 5.11 demonstrate that by selecting the *Negation* item in the popup menu of association *motion*, the association can be negated.

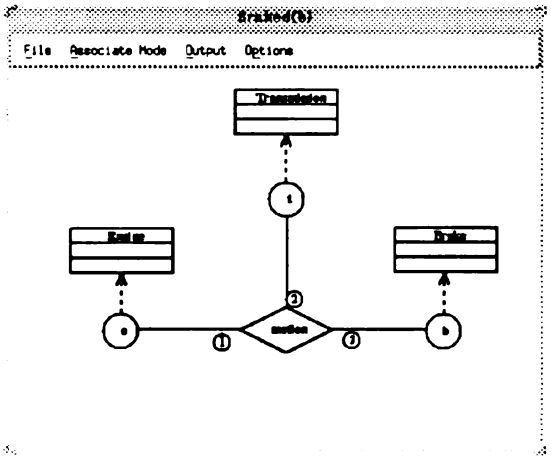

Figure 5.9. Before negating the ternary association *motion*

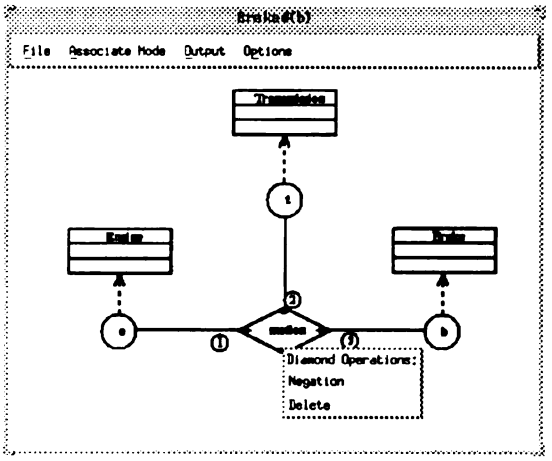

Figure 5.10. Negating the ternary association *motion*

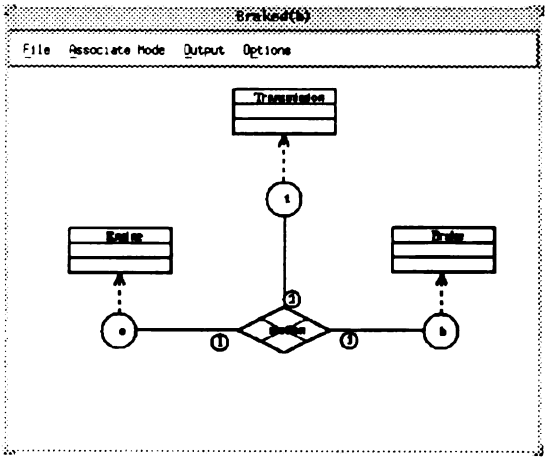

Figure 5.11. After negated the ternary association *motion*

The grammar for the state schemas are based on that of the instance diagrams. Some modifications were made to accommodate parameters for the names of state

schemas and to incorporate the negation of associations or state schemas. The parser has been further refined to enable the identification of the variables that appear in the parameters of the predicate as universal variables and the remaining variables as free variables that are considered to be existentially quantified. As shown in Figure 5.1, variable b is a universal variable while w, e, t are existentially quantified.

CHAPTER 6

Dynamic Models

The dynamic model that describes the behavior of a system has been studied extensively [9, 22, 33, 34]. Both informal and formal approaches that currently exist attempt to present the dynamic aspects of systems easily and rigorously. In this chapter, we discuss the formalization of the OMT's dynamic model [9] and the tool support that VISUALSPECS environment can provide. Section 6.1 introduces the formal semantics imposed on the state diagram of OMT. Section 6.2 illustrates the editor developed for VISUALSPECS and shows how it supports the diagramming of state diagrams and the generation of formal specifications.

6.1 Formalized Dynamic Model

The dynamic model of the OMT has been formalized previously [22, 35, 25]. However, we have developed the formalization of both state and transition in terms of algebraic specifications, thus facilitating the integration between the object and dynamic models via their specifications. The state diagram, unlike the object diagram, whose major contribution is the description of the static system structure delineates the dynamic aspects of a system through state transitions that alter the system states in response to various events. The *state* and *event arc* are two primitive graphical

notations that are used in composing a state diagram. A *state* is referred to as a certain condition or a mode that a system may be in at a given time; the *event arc* represents a stimulus that provokes system state changes.

Furthermore, the states are categorized into *partitioning* and *orthogonal* states in accord with the decomposition rules. In the following subsections, we discuss the syntax and semantics of states and state transitions.

6.1.1 States

6.1.1.1 Syntax

The syntax of states is a tuple $\langle \Sigma, \gamma, \phi \rangle$, where

1. Σ: a set of states,
2. γ: $\Sigma \rightarrow 2^{\Sigma}$,
3. ϕ: $\Sigma \rightarrow \{PART, ORTH\}$.

Functions γ and ϕ are decomposition and type functions respectively. $\gamma(x)$ represents the set of immediate successive states (substates) that composes the state x. In addition, we introduce two terms, *superstate* and *substate*, for convenience. Given states x and y, if $y \in \gamma(x)$, we say y is x's substate and x is y's superstate. The type function ϕ categorizes states into $PART$ and $ORTH$ state types. Informally, being in an *orthogonal* state, P, means being in each of P's substates; being in a *partitioning* state, P, means being in exactly one of P's substates. In terms of graphical notations, the state and the functions γ and ϕ are denoted as:

- State x and type function ϕ: a state x is represented by a solid oval or a dashed rectangle labeled with the name of the state. The label is placed on the outside of the solid oval or dashed rectangle, but enclosed in an attached rectangle.

The state of type $ORTH$ is represented by an oval in which the substates are separated by dashed lines, whereas the state of type $PART$ is denoted as an oval that encloses its substates.

- State decomposition function γ: the set of states directly enclosed in a solid oval x are considered substates of x, and are denoted as $\gamma(x)$.

Figure 6.1 is an example that shows a *moving* partitioning state of an automobile. The *moving* state consists of *accelerating*, *constantly moving*, and *decelerating* substates among which transitions can be triggered when certain events occur. Figure 6.2 is a state diagram that contains an *accelerating* orthogonal state comprising the *engine_started*, *brake_released*, *gas_pedal_pushed*, and *wheel_rotating* substates. For a system in an orthogonal state, the exact state of the system is actually determined by the encapsulated substates, which, in turn, are determined by the subsystem states of the system.

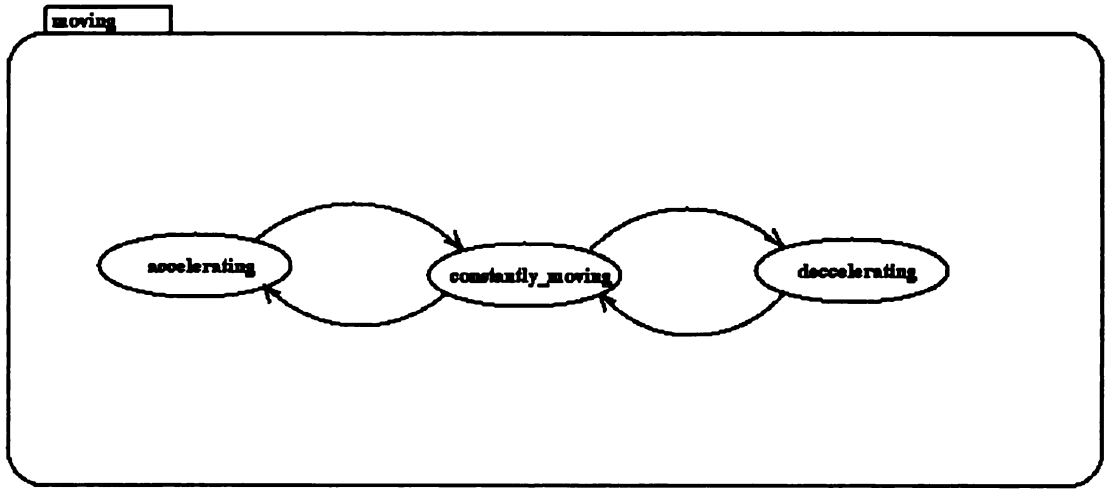

Figure 6.1. The partitioning *moving* state

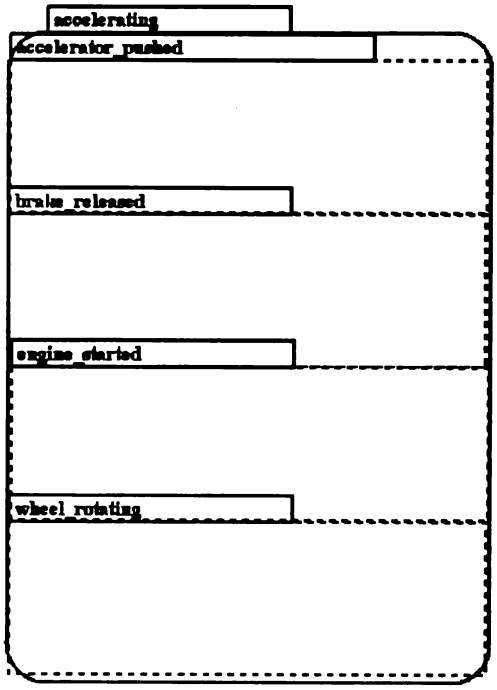

Figure 6.2. The orthogonal *moving* state

6.1.1.2 Formal Semantics

Since Σ contains only a set of state names that do not have any specific meanings, the formal semantics of the dynamic model mainly comes from the interpretation of the decomposition and type functions, γ and ϕ, respectively. The formal semantics of these two functions are defined by the following rules:

1. $(\forall x \in \Sigma.\ (\phi(x) = PART) \Longrightarrow ((\forall y \in \gamma(x).y \rightarrow x) \wedge (\forall y, z \in \gamma(x).\ y \neq z \rightarrow (y \rightarrow \neg z)) \wedge (x \rightarrow (\bigvee \gamma(x)))))$.

2. $(\forall x \in \Sigma.\ (\phi(x) = ORTH) \Longrightarrow (x \rightarrow (\bigwedge \gamma(x))))$.

Rule (1) says that if a system, at a certain time, is in a partitioning state x, it must also be in one and only one substate of x; a substate of x refines state x. Rule (2) implies that whenever a system is in an orthogonal state, it is also in all of its substates simultaneously, which indicates that an orthogonal state of a system depends on the states of its subsystems. Thus the orthogonal state is analogous to the system decomposition hierarchy.

6.1.2 State transitions

6.1.2.1 Syntax

Since the purpose of state diagrams is to describe the dynamic aspects of a system, which changes in response time and events, state transitions become crucial in achieving this objective. A state diagram would convey very little information without state transitions. The syntax of the state transition is an extension of the $\langle \Sigma, \gamma, \phi \rangle$ tuple and is defined as $\langle \Omega, \delta \rangle$, where

1. Ω: a set of events.

2. δ: $\Sigma_{PART}, \Omega \rightarrow \Sigma_{PART} \cup \{\emptyset\}$, where $\Sigma_{PART} = \{x | x \in \Sigma \wedge \phi(x) = PART\}$.

Set Ω contains events that trigger transitions between partitioning states. There is no state transition associated with orthogonal states. The set δ actually defines transitions, fired by specific events, between partitioning states. The state on the left side of the arrow "$\rightarrow$" is called the departure state; the state on the other side is called the arrival state. Given a state s and an event e, if $\delta(s, e) = \{\}$, then we say there is no transition from s when event e occurs.

In terms of graphical notations, every transition is denoted by a labeled arrow that connects two ovals that represent partitioning states. The states can be further identified as departure or arrival states in accordance with whether the state is associated with an arrow. Thus the syntax of a state diagram is extended to become the combination of the state and transition tuples, $\langle \Sigma, \gamma, \phi, \Omega, \delta \rangle$.

6.1.2.2 Formal Semantics

Because neither timing factor nor time constraints were introduced into the formalisms of the dynamic model, a transition happens instantaneously when the triggerring event occurs and the system is in the appropriate departure state. The formal semantics of the transition is

$$(\forall x \in \Sigma, \forall e \in \Omega.\ (x \wedge \phi(x) = PART \wedge e) \rightarrow \delta(x, e)),$$

which means a system in state x immediately changes to state $\delta(x, e)$ once event e occurs. If $\delta(x, e)$ is $\{\}$, the system then remains in the current state. This occurrence implies that the given event has no effect on the (sub)system.

6.2 Tool support

We have developed a graphical editor based upon the VISUALSPECS environment and its graphical icon library in order to support the formalized dynamic model. The architecture of DMTOOL is exactly the same as that introduced in Chapter 3.

Only slight changes to the *graph editor* and *parser* were made in order to support the construction of the dynamic model. The remainder of this section discusses the details of the changes to the graph editor and the parser.

6.2.1 Graphical editor

The architecture of the graphical editor of DMTOOL is exactly the same as that of the object model. The graphical editor maintains a list of graphical components, such as ovals and arcs, and is responsible for passing information to the graphical objects. A part of the syntax of the state diagram is also incorporated into the graphical editor in terms of context-sensitive diagramming in order to eliminate some invalid drawings.

The primary graphical constructs of the state diagram are relatively simple in comparison to those used by the object diagram. *Solid oval*, *dashed rectangle*, and *arc arrow* are the only notations that are used in the dynamic model of a system, whereas, in contrast, a description of the object model needs *rectangle*, *oval*, *triangle*, *diamond*, *segment*, etc.

Since the properties of an *arc arrow* are very similar to that of a *line* object class previously defined in the object model, the *arc arrow* object class was derived from the *line* class with the *draw* member function modified to draw the shape of an arc. Unlike the object model and state schema, the state diagram introduced modularity as a means to facilitate the decomposition of states. Thus the DMTOOL has to provide the corresponding facilities to support the modularity characteristics of the dynamic model. A new graphical construct, *floating diagram*, was implemented to enable users to enclose a subdiagram as a graphical component in a diagram.

Partitioning diagram and *orthogonal diagram* are two subclass derivations of the *floating diagram* as a means to represent the partitioning and orthogonal states, respectively. The user is free to incorporate either a normal state or a subdiagram, typed as *PART* or *ORTH*, which may be further decomposed into other states, during the

construction of a state diagram. As we mentioned in Section 6.1.1, the ovals with substates separated by solid or dashed lines represent the partitioning and orthogonal substates respectively.

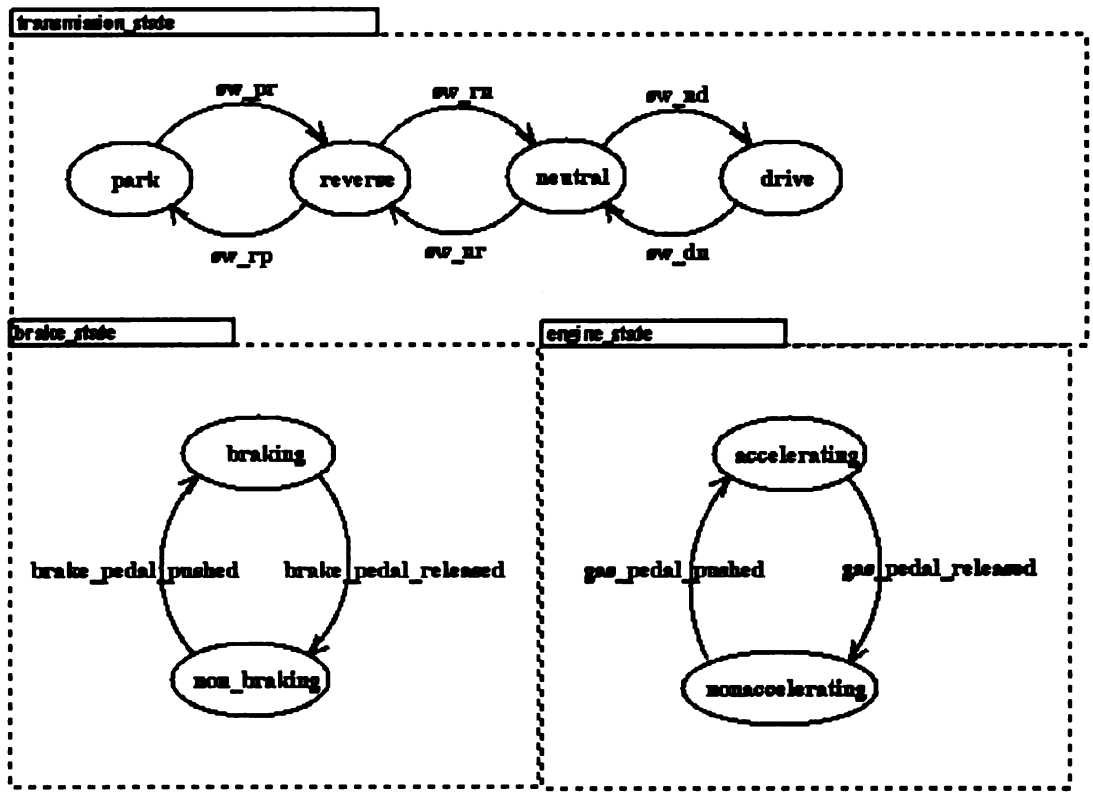

Figure 6.3. The *motion_state* orthogonal state

In order to support modular construction of specifications, the *floating diagram* object has operations that support the viewing of the contents of a superstate with a mouse click. Then another diagram, whose content is exactly that of the clicked state, will be displayed in another pop-up window.

Figure 6.3 shows an orthogonal state, *motion_state*, of the simplified automobile system that was introduced in Chapter 4. As illustrated in the figure, the *motion_state*

state is composed of *transmission_state*, *brake_state*, and *engine_state* substates. Furthermore, each substate is itself a superstate for other states. For example, *transmission_state* can be further decomposed into *park*, *reverse*, *neutral*, and *drive* substates.

6.2.2 Larch specification generation

As with the object models, the Larch specification language is used to describe the formal semantics of states and state transitions introduced in Section 6.1. The *trait* construct provided by Larch is used to describe the functions γ, ϕ, and δ.

Since the semantics of the state diagram is twofold, we need two different schemas to describe state diagrams. First, a Larch *trait* is generated for each superstate along with its corresponding substates with respect to the *partitioning* and *orthogonal* state decomposition rules. The distinct semantics of the two types of states is reflected by the different axioms in the *assert* portion of the traits. Figure 6.4 is the specification generated for the state *motion_state* shown in Figure 6.3. The specification indicates that a system is in the *motion_state* state only if it is also in *engine_state*, *brake_state*, and *transmission_state*.

Figure 6.5 contains the specification derived from *transmission_state*, which is also a substate of *motion_state*. Although the *transmission_state* state is enclosed by *motion_state*, which is an orthogonal substate, it, itself, is actually a partitioning state. The specification implies that an object in *transmission_state* state must be in one and only one of the *park*, *reverse*, *neutral*, and *drive* states.

Second, every transition represented by an arc and two solid ovals should also produce a Larch specification trait in order to depict a system state transition triggered by the transition event when the system is in the *departure* state. The generated specifications describe the system changes in accordance with events. Figure 6.6 shows a transition that changes the state of a transmission from the *drive* state to the *neutral* state given the *sw_dn* event which represents "switch from drive to neutral".

```
STATE_motion_state: trait
    includes STATE_engine_state
             , STATE_brake_state
             , STATE_transmission_state
    introduces
             motion_state :  Object -> Bool

    asserts
             \forall a: Object
             motion_state (a) == (engine_state (a) ) \and (brake_state (a) )
             \and (transmission_state (a) )
```

Figure 6.4. Larch specification of the *motion_state* orthogonal state

```
STATE_transmission_state: trait
    includes STATE_drive
        , STATE_neutral
        , STATE_reverse
        , STATE_park
    introduces
        transmission_state : Object -> Bool

    asserts
        \forall a: Object
        drive (a) => transmission_state (a)
        neutral (a) => transmission_state (a)
        reverse (a) => transmission_state (a)
        park (a)  => transmission_state (a)
        drive (a) => ~ neutral (a)
        drive (a) => ~ reverse (a)
        drive (a) => ~ park (a)
        neutral (a) => ~ reverse (a)
        neutral (a) => ~ park (a)
        reverse (a) => ~ park (a)
        transmission_state (a) == (drive (a)) \or (neutral (a))
        \or (reverse (a)) \or (park (a) )
```

Figure 6.5. Larch specification of the *transmission_state* partitioning state

```
STATE_TRANSITION_transmission_state_drive_neutral_sw_dn : trait
    include STATE_drive
        , STATE_neutral

    introduces
        sw_dn     : -> EVENT
        Trans_drive_neutral_sw_dn: Bool, EVENT -> Bool

    asserts
        forall a : OBJECT, e : EVENT
        ( drive (a) /\ e = sw_dn ) => (Trans_drive_neutral_sw_dn(drive(a), e)
        => ( ~ drive (a) /\ neutral (a) )
```

Figure 6.6. Larch specification of the transition from state *drive* to *neutral*

CHAPTER 7

A New Modeling Paradigm

The statechart [36] allows a given state to be decomposed into *XOR (orthogonal)* and *AND* substates, where the *XOR* substates can be considered as a refinement of states whereas the *AND* substates are normally a result of splitting a state in accordance with its physical subsystems [36].

Since the statechart is state-oriented and does not address the properties of system objects, the purpose of decomposition is solely to reduce the complexity of traditional state machines. However, within the OMT framework, for each object diagram, there is a corresponding statechart, thus defining the integration between the two modeling approaches. No modification was applied to statecharts to facilitate its integration with the structure of the object model which favors the object-oriented development paradigm. Therefore, the inter-relationship between the object and dynamic models in the OMT has a loose coupling that is not well-defined. Additionally, there are no explicit guidelines for integrating the two models.

Also, no detailed guidelines were given to conduct the *XOR* state decompositions in statecharts. On the one hand, the *XOR* substate is a refinement of their superstate. On the other hand, the superstate is a clustering of its respective *XOR* substates in order to make the statechart concise. Thus the *XOR* is, in fact, a type of abstraction of the state space. Nevertheless, how to perform the abstraction process in order

to take advantage of the benefits of applying *XOR* decomposition is not explicitly described and greatly depends on the experience of the individual software engineers.

We propose a new approach to perform system state decomposition, which is designed to tightly couple the object and dynamic models. More specifically, our approach to the integration of object and dynamic modeling techniques enables us to depict the static and dynamic aspects of a system in the same modeling framework.

In this chapter, we introduce the concept of substate decomposition and present critera that such decompositions must satisfy. Statecharts so structured will be of much greater use in the specification process due to their modularity. With partitioning states and state schemas, we are able to conduct a state space decomposition that is tightly coupled with the construction of object diagrams. In our approach, the object diagrams and the statecharts are constructed alternately in an iterative process. There are four steps in every iteration.

1. Develop the object model
2. Develop the dynamic model
   (a) State refinement
   (b) Transition assignment
3. Depict the states in the dynamic model by state schemas
4. Refine object model to include states from state schemas

The remainder of this chapter discusses the four steps along with examples from the *ENFORMS* [37, 38, 39, 40] system. *ENFORMS* is a distributed object-oriented multimedia decision support system for environmental science and global change, being developed by the Software Engineering Group at Michigan State University [41].

7.1 Develop the Object Model

The first step of modeling in every iteration is to develop an object model based on the previous iteration. The exception is the first iteration where no previous models can be referenced. In this case, object modeling is used to decompose the whole system into subsystems and to model the system and its corresponding subsystems with aggregation, binary, and n-ary associations. For convenience, the object or (sub)system being decomposed is referred to as the *current* object or (sub)system; the other objects or (sub)systems are called component classes or (sub)systems. The decomposition is only applied to the current (sub)system level. No other decomposition and modeling is allowed for the component classes. Thus, through iterations, the system can be decomposed and modeled with increasing detail at each iteration. Figure 7.1 shows the level one object model diagram of the *ENFORMS* system.

Since Figure 7.1 is the first level object model, the states, *initiating*, *idle*, and *querying*, associated with the *Enforms* class are not derived from the previous iteration of modeling, but are directly constructed by the user. For object modeling other than that performed for level one modeling, the object states are acquired in step 4 (decomposition) of the previous iterations.

In this example, the *Manipulator*, *NameServer*, *ArchiveServer*, and *Client* classes will be decomposed and modeled in the next iteration of modeling.

7.2 Develop the Dynamic Model

The dynamic model of the (sub)system that is decomposed and modeled in step one is constructed in this step. The modeling procedure consists of two sub-steps: state refinement and transition assignment.

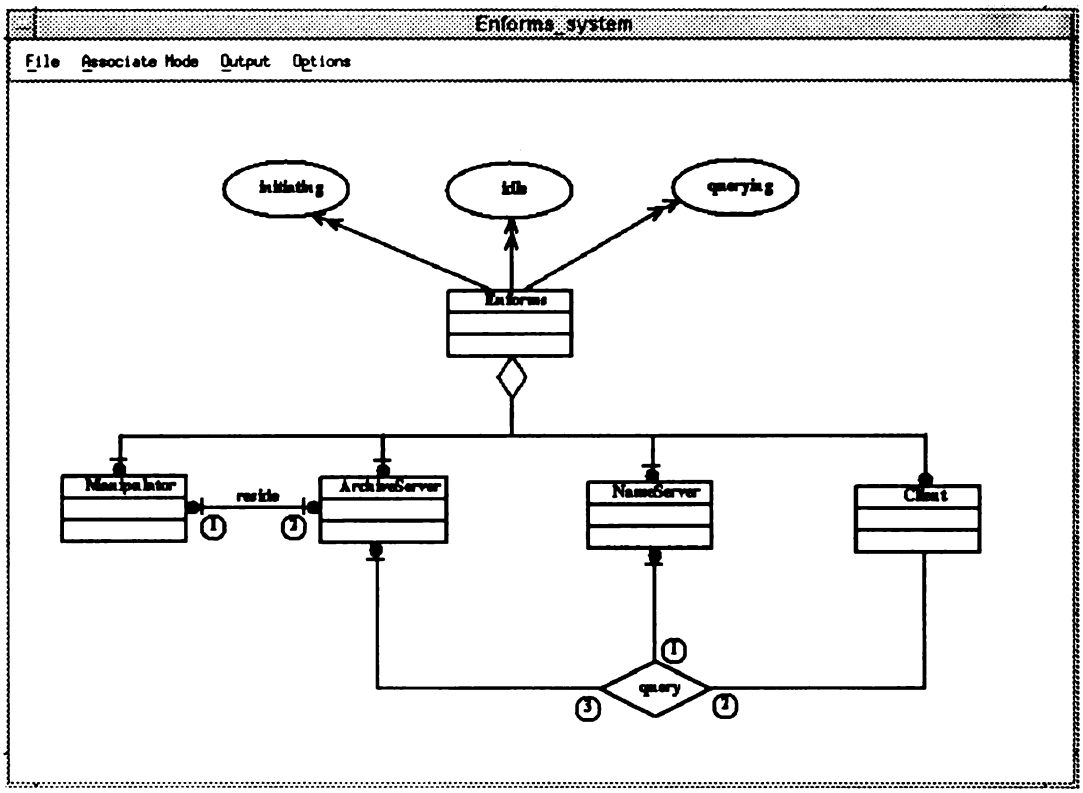

Figure 7.1. The level one object model diagram of the ENFORMS system

7.2.1 State refinement

We have noticed that the current class is the only object associated with states. During the first iteration of modeling, these states are directly constructed by the users; for the other iterations, these states are acquired in step four of the previous iterations. The states associated with the current class, if possible, are further refined into substates, called refining substates, in this substep.

The strategy to conduct the refinement has been discussed in the literature [9, 42, 43], and is also one of our ongoing research topics. However, the refinement strategy still heavily depends on the users' preference and experience.

Figure 7.2 gives one possible refinement of the *querying* state. The state *ArchReso* means that the system is resolving the internet address of the archive server on which a query is to be performed; the state *QueryPerfo* is the state in which the querying is taking place. In contrast, the state *querying* is a clustering or a higher level abstraction of the *ArchReso* and *QueryPerfo* states. In order to maintain simplicity and support abstraction, the states associated with the current object or (sub)system in the object diagram obtained in step one will not be changed, even though they may be refined in this step. In step one of the next iteration of modeling, the refinement to the object model will be captured.

Since it did not appear necessary to refine the dynamic model, Figure 7.3 is kept as our level one dynamic model for *ENFORMS* system.

7.2.2 Transition assignment

Given a set of states associated with the current class, transition and trigger event between the states or the refining substates must be assigned such that a dynamic model of the current class can be constructed. In both Figures 7.2 and 7.3, transitions between the states *initiating*, *idle*, and *querying* as well as between the refining sub-

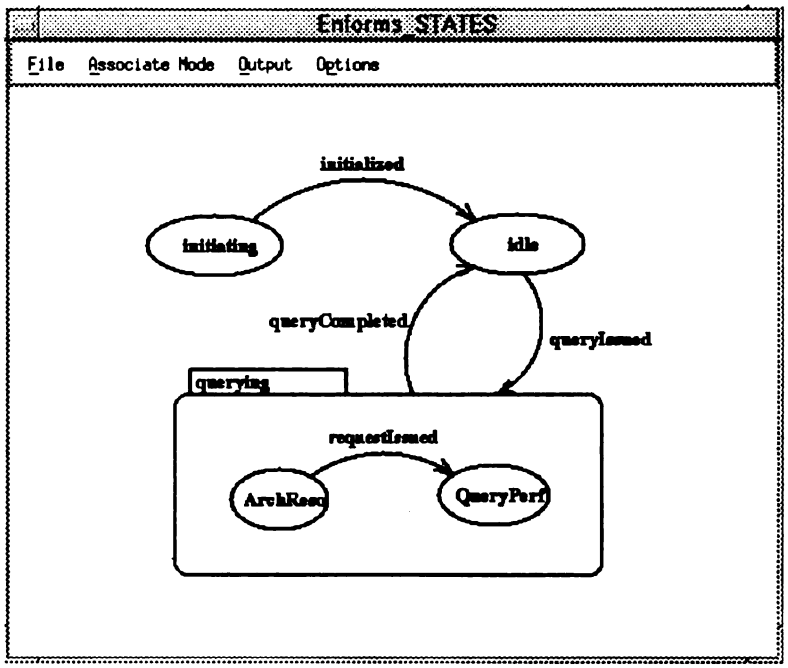

Figure 7.2. A possible refinement of the state *querying*

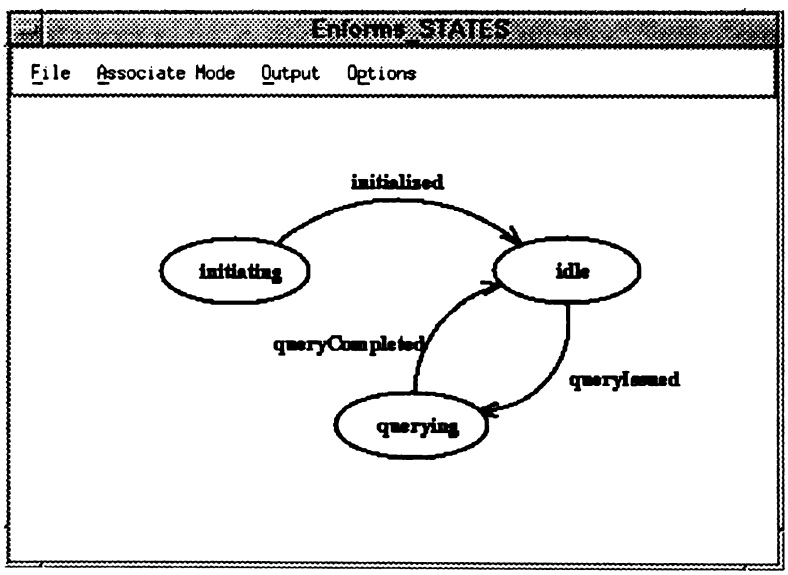

Figure 7.3. The level one dynamic model of the *ENFORMS* system

states *ArchReso* and *QueryPerfo* are shown. Although we are not giving any specific guideline to determine the transitions and triggering events, the constraint that every state must be reachable must be satisfied in the constructed dynamic model.

7.3 Depict the States in the Dynamic Model by State Schemas

In this step, we try to give a state schema for every state that appears in the dynamic model constructed in step two. The result of this step provides another perspective on the system states. The state schema is intended to reveal the relationship between the state of the current object and the states of its component class. Since there are no states associated with the component classes yet, we also need to determine in what states the components object classes should be during the construction process of the state schemas associated with the states of the current class.

Figure 7.4 illustrates a state schema depicting the *querying* state of class *Enforms*. There are four instances *e*, *ns*, *c*, and *as* typed as *Enforms*, *NameServer*, *Client*, and *ArchiveServer*, respectively. The *ns*, *c*, and *as* instances are related by association *query*, which means that the three, as a whole, carry out query activities. The *e* instance of class *Enforms* is composed of *ns*, *c*, and *as* instances. In addition, the *as* instance is associated with state *doQuerying*; and *c* is associated with state *waitQuerying*. Therefore, these relationships imply that *doQuerying* is a state of class *ArchiveServer* and *waitQuerying* is a state of class *Client*. Since there is no state associated with the *ns* instance, the *querying* state of class *Enforms* will not be affected by the states of its *NameServer* components.

In Figure 7.5, a Larch specification generated from the state schema is given. The meaning of the specification is: if there are at least three components, archive server,

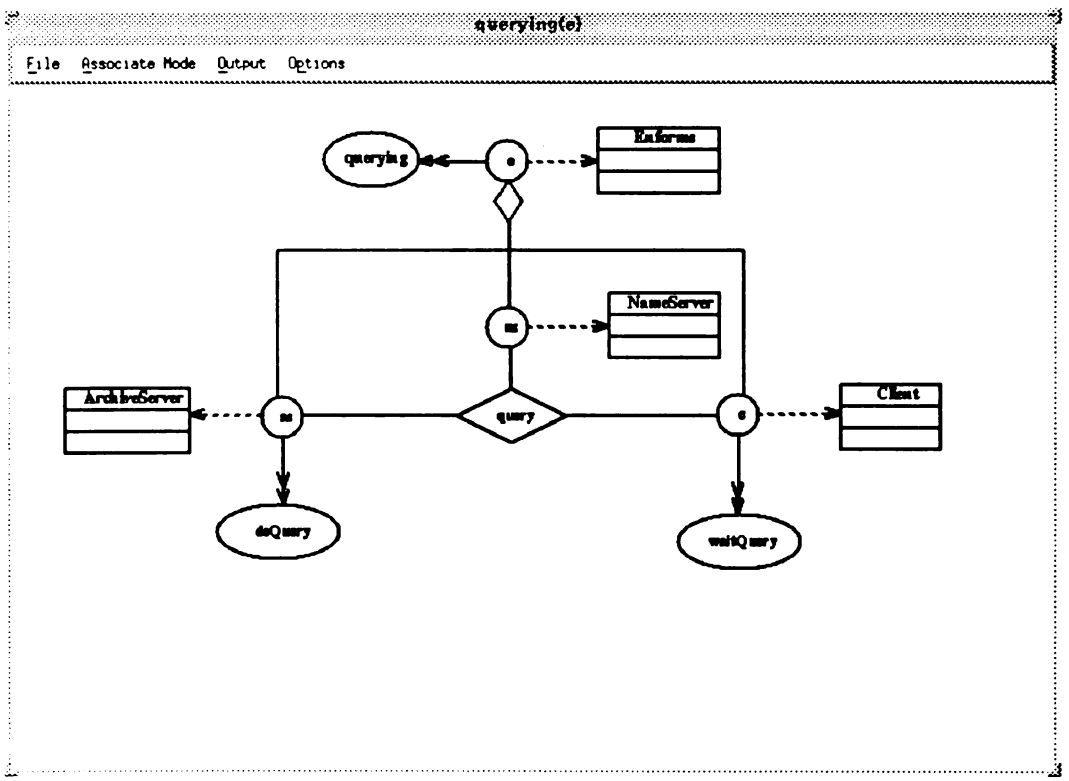

Figure 7.4. The state schema of the *querying* state of the *Enforms* object

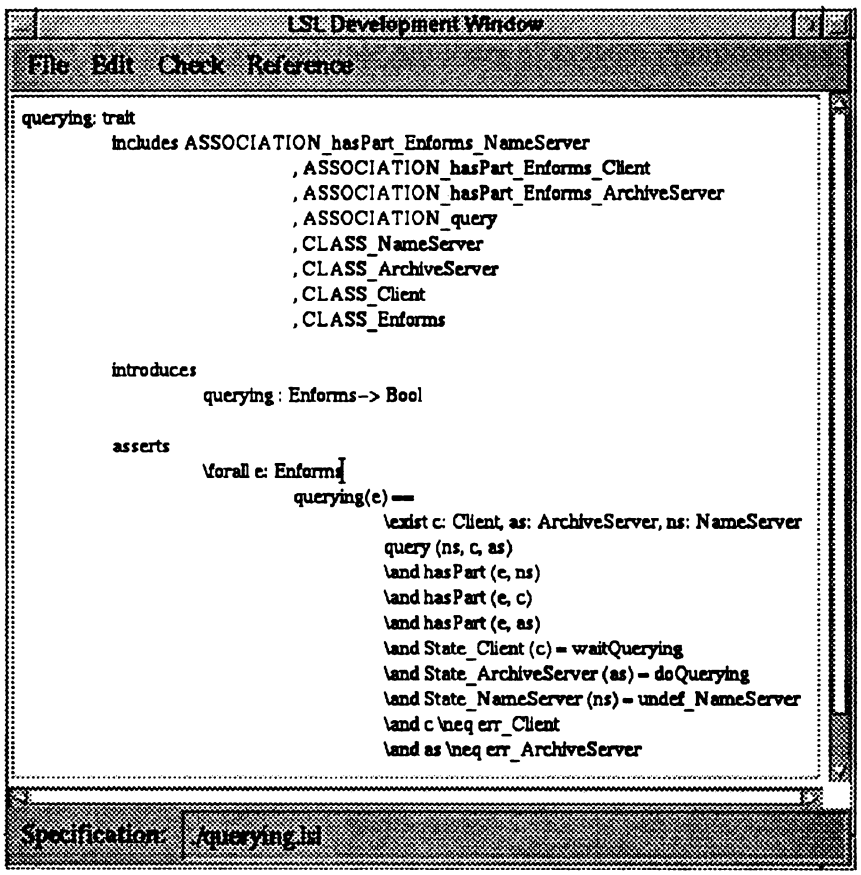

Figure 7.5. The state schema of the *querying* state of the *Enforms* object

name server, and client, in an ENFORMS system working together to implement query activities, and the archive server is performing a query while the client is waiting for the result of a query, then the ENFORMS system can be said to be in a *querying* state. Although a typical statechart is also able to depict the aggregate association between the system and its components through the orthogonal decomposition, it cannot describe the relationships between the components and does not have the means to depict the states of multiple numbers of system components. Clearly, the state schema provides a more precise semantics. The *query* association in Figure 7.5 shows the strength of the state schema. It eliminates the abnormal system states that may be otherwise acceptable in a statechart. Consider the following scenario.

Two client programs c_1, c_2, two archive servers as_1, as_2, and a name server are running in an ENFORMS system. The clients c_1, c_2 issued two queries to the archive servers as_1, as_2 respectively and are waiting for the query results while the two archive servers are performing the queries. Accidently, client c_1 and archive server as_2 die and leave c_2 waiting for as_2 and as_1 serving for c_1.

In a typical statechart, this state is considered a legal state; the potential system hazard cannot be detected. However, the state schema in Figure 7.5 does not accept the state as a *querying* state and will clearly reveal the problem.

7.4 Refine Object Model to Include States from State Schemas

Based on the state schemas developed in step three, we are able to associate states to the component class in the object diagram constructed in step one. However, not all the component classes need to have corresponding states. If there are no states associated with the instances of a certain component class in the state schemas, the

state of that object class is considered insignificant for the behavior of the system and no state needs to be attached to the component class in this step. VISUALSPECS will automatically associate a default state to the object during the specification generation process.

For a component class that is already given states in the state schemas, we need to collect the states that may be scattered in the state schemas in order to form a complete set of states for that component class. Unfortunately, the set of collected states (from all the state schemas) of a component class usually does not cover all the possible states. The users need to carefully study the component class and the states it already has, then determine whether there are missing states and add them into the state space. Thus a relatively more complex and informative version of the object diagram obtained in step one is derived. We consider those component classes that are associated with states to be likely candidates for decomposition in the next iteration of modeling.

Figure 7.6 is the revised object diagram obtained by adding states to component classes. The difference between Figures 7.6 and 7.1 is that classes *ArchiveServer*, *NameServer*, and *Client* are now associated with their corresponding available states, which are collected from the state schemas derived for states ***idle***, ***initiating***, and *querying* of the *Enforms* class. In the next iteration of modeling, classes *ArchiveServer*, *NameServer*, and *Client* will be decomposed and modeled in a similar fashion.

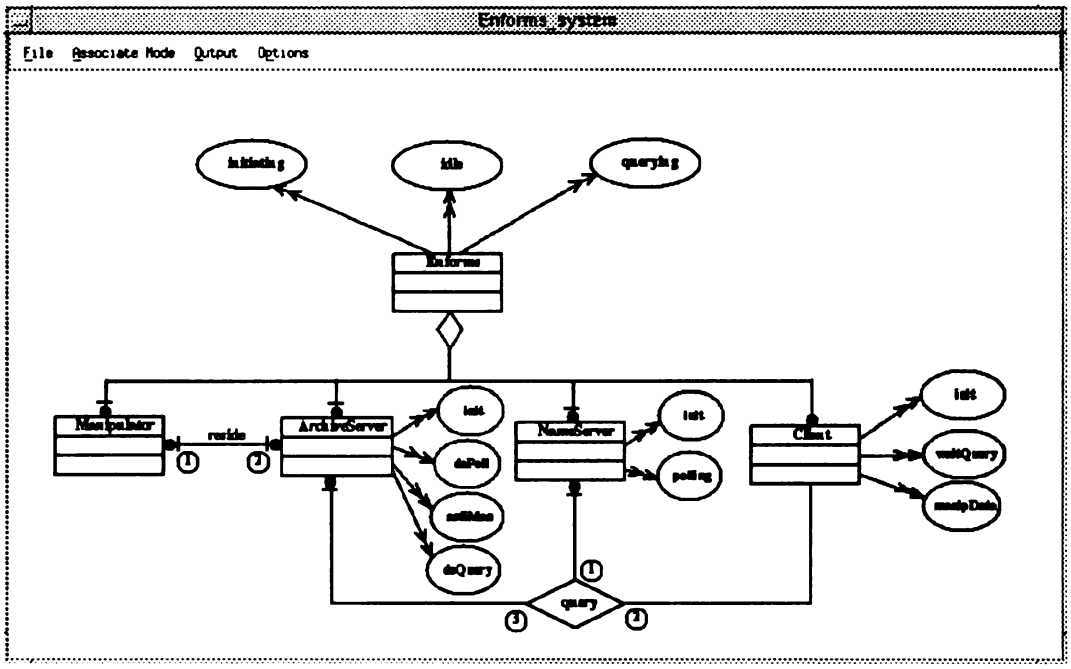

Figure 7.6. The revised level one object model diagram of the ENFORMS system

CHAPTER 8

Intra- and Inter-model Referencing

Visual formalisms are intended to model systems that are potentially quite complex. However, it is unrealistic to draw everything in one diagram. A system of reasonable complexity is always decomposed into smaller pieces and modeled separately. As a consequence, a graphical object may occur in multiple diagrams. To complicate things further, several modeling techniques that describe different aspects of a system may need to be integrated into one modeling framework [44]. Thus a cross-referencing mechanism that supports both diagram and graphical object references is necessary as a means to facilitate the modeling process.

Diagram-level referencing involves techniques and concerns, relevant to configuration management, which is beyond visual formalisms. Currently only graphical object referencing is supported. The reference mechanism supports both intra- and inter-model referencing. Section 8.1 illustrates the intra-model referencing we have implemented for the object model. Section 8.2 shows the inter-model referencing between the object and dynamic models. Section 8.3 introduces the mechanism that we developed to support cross-referencing.

8.1 Intra-model Referencing

Figure 8.1 shows an indexed object model of the ENFORMS system in which class *ArchiveServer* is found of interest and highlighted. By selecting *reference* from the popup menu with the cursor in the *ArchiveServer* class, the reference collaborator can be activated.

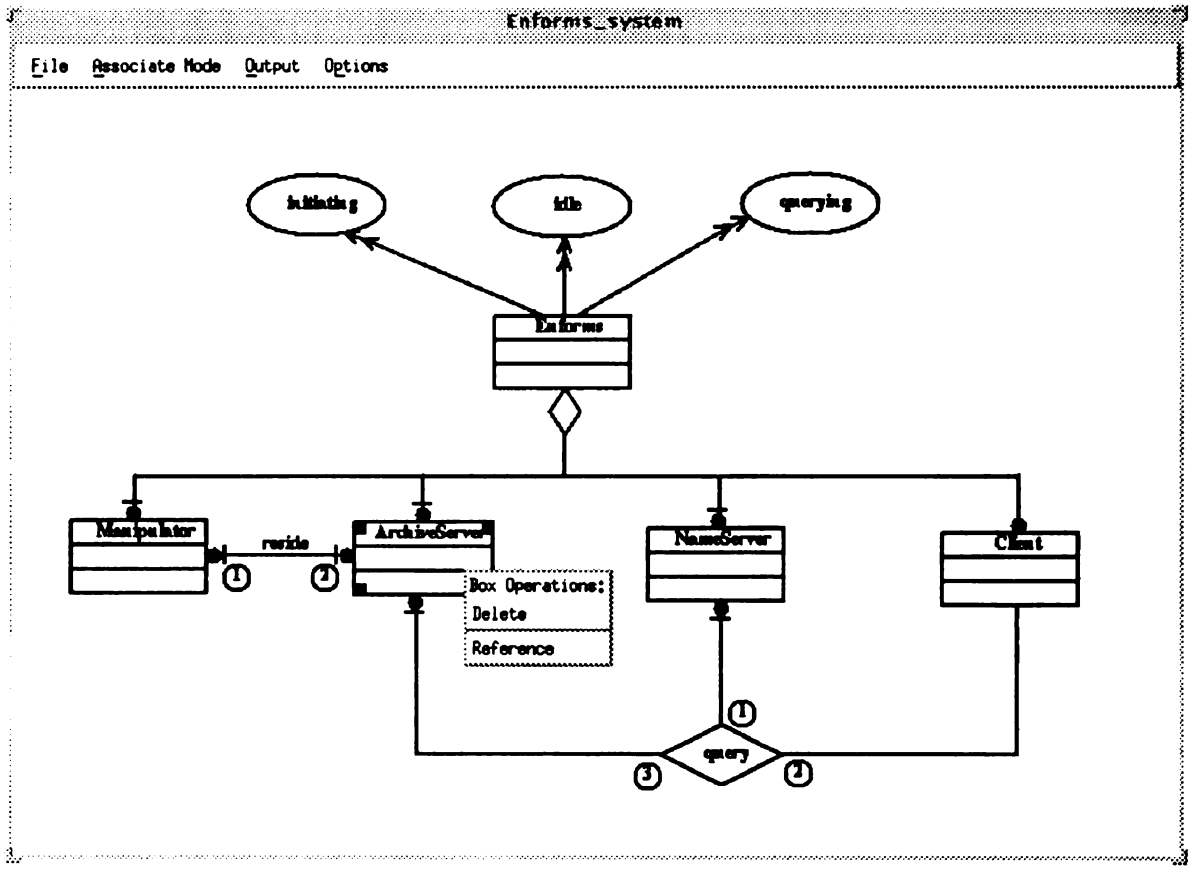

Figure 8.1. The object diagram of ENFORMS system

In Figure 8.2, the reference collaborator is displayed. It contains three categories of available diagram references—object diagrams, instance diagrams, and state schemas. After checking all the diagram index files, the collaborator lists *om1.dgm* as an avail-

able object diagram reference; *query_ins2.dgm* and *query_ins1.dgm* as available instance diagram references; *query_ss1.dgm* as an available state schema reference. The file name, *ins1.dgm*, in the text field at the lower part of the reference collaborator is the selected referenced diagram for viewing. According to our classification of diagram types, the object and instance diagrams both belong to the object model, hence the referencing between the two diagrams is considered intra-model referencing.

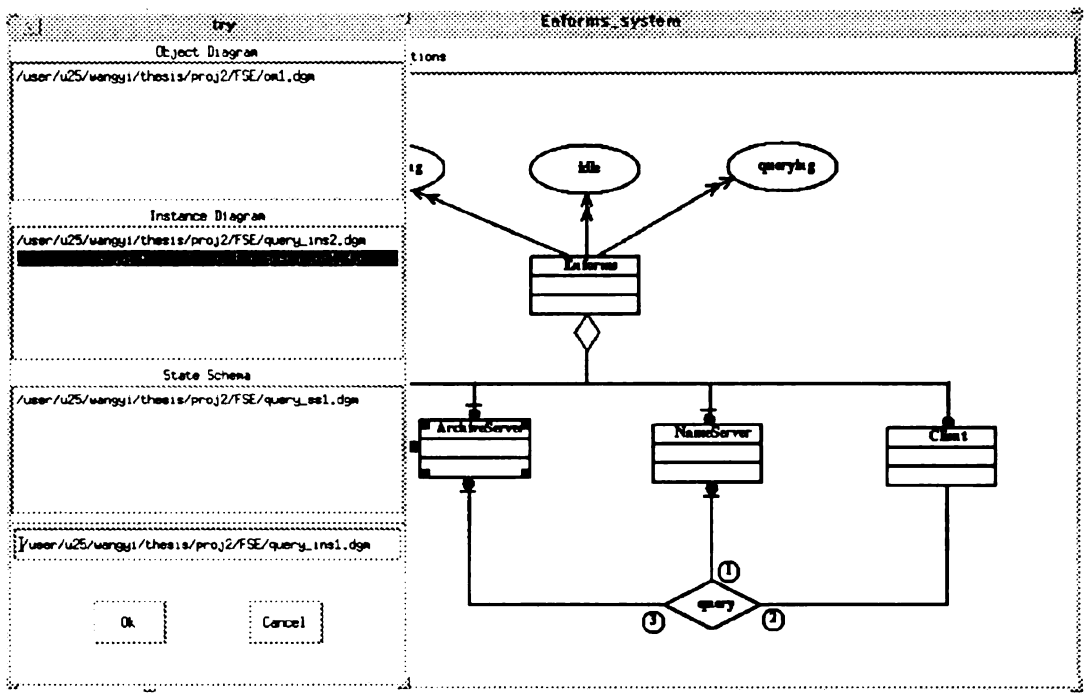

Figure 8.2. An instance diagram of interested is selected to display

The referenced instance diagram, *query_ins1.dgm*, is finally displayed at the upper-left corner of Figure 8.3. The displayer, *diagram.viewer*, is a read-only diagram viewer. No modifications or other manipulations are allowed in the *diagram.viewer*.

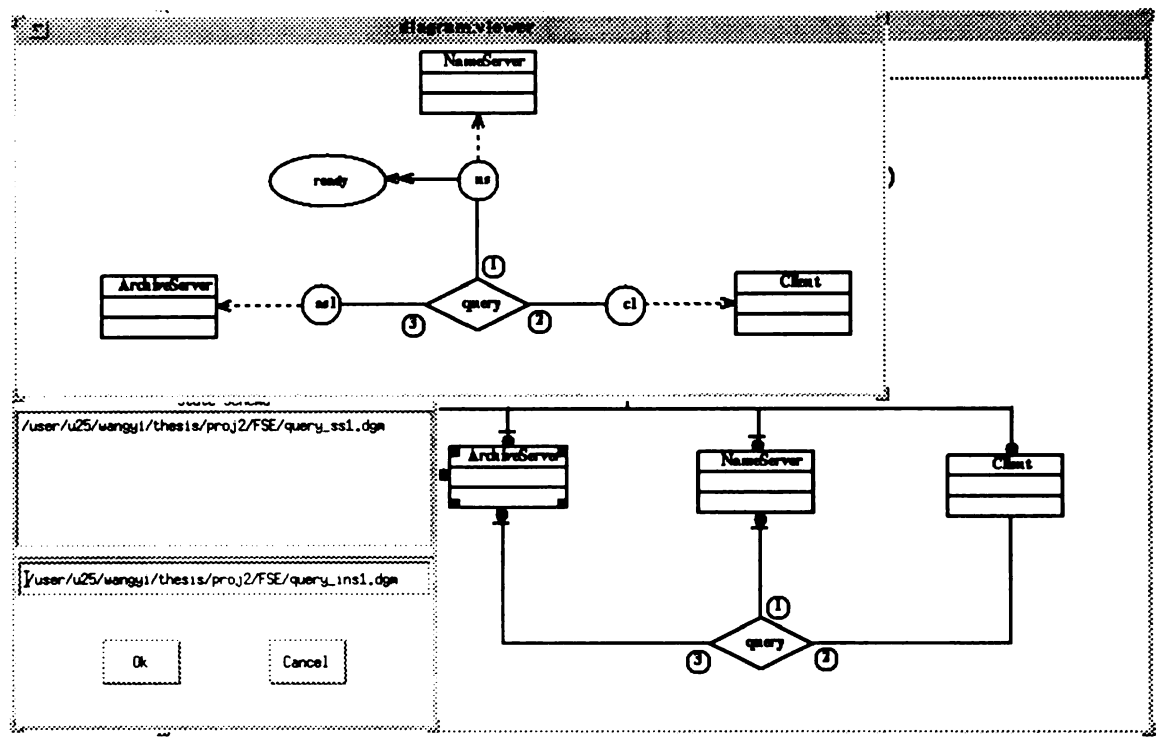

Figure 8.3. The referenced instance diagram is displayed

8.2 Inter-model Referencing

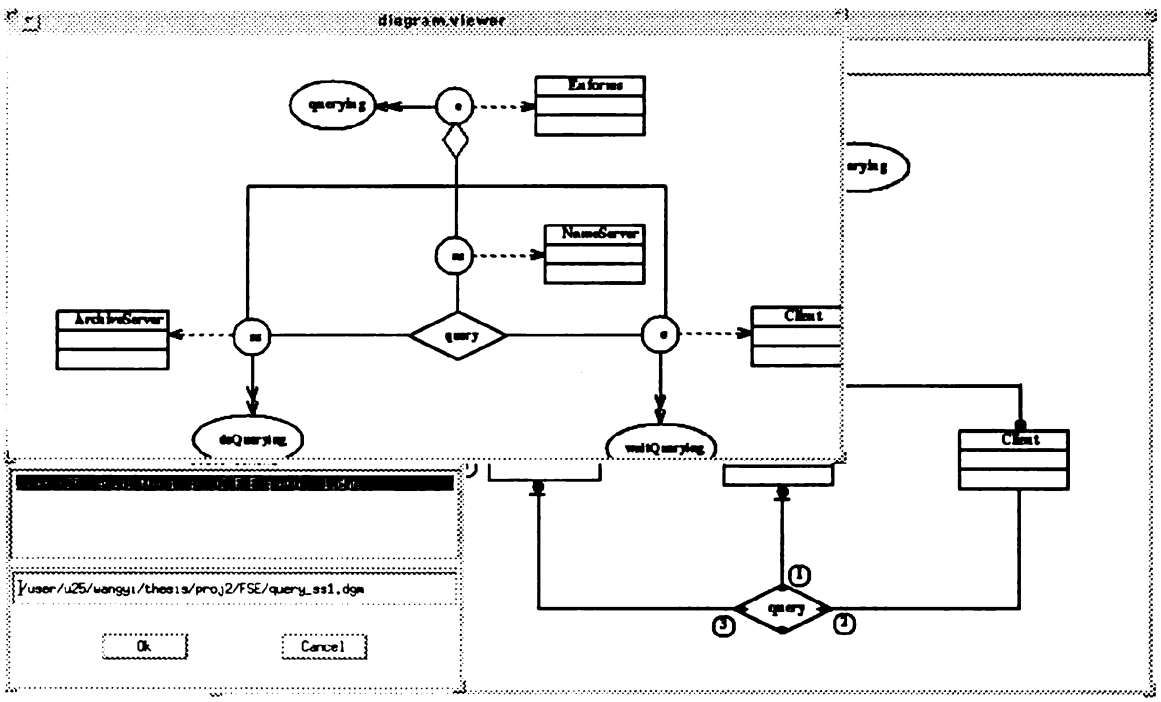

Figure 8.4. An inter-model referencing

Similar to intra-model referencing, the inter-model referencing is achieved by selecting the available reference diagrams from the different types of models. Figure 8.4 is the result of performing referencing on association *query* in the object model of ENFORMS. But this time, the state schema *query_ss1.dgm* is referenced and displayed at the upper-left corner of Figure 8.4.

8.3 Referencing Mechanism Implementation

The cross-referencing mechanism is implemented through an index generator, a reference collaborator, and a reference displayer. Figure 8.5 is a high level data flow diagram that shows how the index generator, the reference collaborator, and the reference displayer cooperate to provide the numerous referencing services.

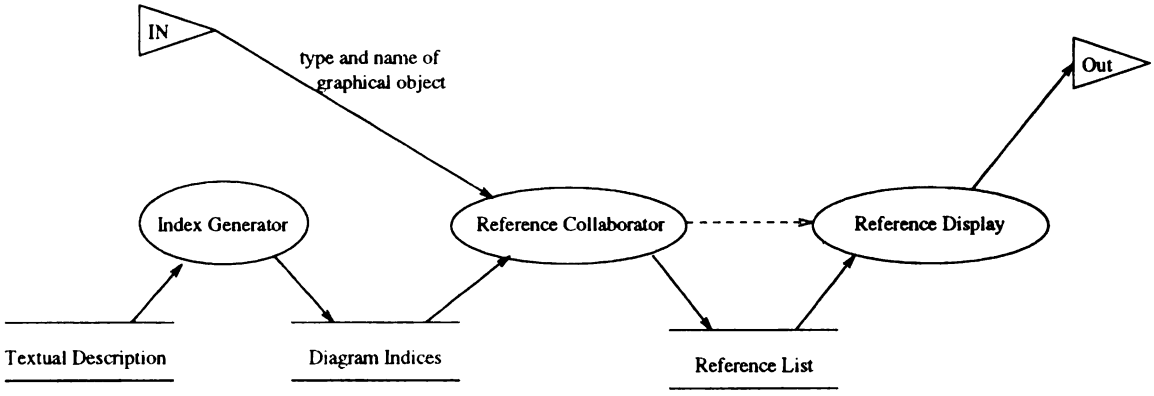

Figure 8.5. A high data flow diagram of the referencing sub-system

The index generator is a parser that takes the textual description of a diagram as input and creates an index file. Each valid diagram, where valid means syntactic correctness, has a corresponding index file, that contains information about the diagram and its graphical objects. The information stored in the index files includes the names and full paths of the files containing the indexed diagrams, the types, names, and signatures of the indexed graphical objects, etc. By modifying the index generator, a user can add new graphical objects of interest to be indexed.

During the construction or browsing of diagrams, if a user is interested in viewing the references related to a certain graphical object, the reference collaborator can be launched from the popup diagram menu associated with the graphical object of

interest. The reference collaborator then searches through the diagram index files to find the diagrams that contain the graphical object whose type, name, and signature (only for associations) match those of the graphical object from which the reference collaborator was activated. The selected diagrams forms a reference list and are loaded to a reference displayer. The reference displayer, in turn, displays a dialog box to show the reference list. If a diagram in the reference list is found of interest, a read-only diagram viewer can be triggerred to show the contents of that diagram.

CHAPTER 9

Integration with Other Tools

Because VISUALSPECS can generate multiple types of formal specifications, it can be integrated with other tools. Currently, VISUALSPECS is used with LDE (Larch Development Environment), an integrated environment supporting the construction and manipulation of Larch specifications. The remainder of this section gives an overview of the integration between VISUALSPECS and LDE.

9.1 LDE tool

A related project developed an integrated environment that facilitates the construction of LSL specifications, including a graphical interface to theorem proving and syntax checking tools. The tool, LDE (*Larch Development Environment*) [45], provides simplified access to trait handbooks, editing facilities for traits, graphical interfaces to the LSL syntax checker [3, 18, 46], the Larch theorem prover (LP) [11], and a graphical rendering of trait hierarchies based on trait inclusion.

While LDE contains features found in many browsers, it also contains functions to assist the specifier in the development of traits, including creation, modification, and saving traits as well as traversal of their dependencies. LDE provides capabilities such as the graphical display of traits, operators, sorts (data types), and dependencies,

all of which can be generated with respect to the context of trait dependencies. Figure 9.1 contains a sample session with the LDE, where a specification for the `dictionary` trait is displayed in the "Larch Trait Browser" window. The operators and sorts are declared within the section headed **introduces**. Axioms are included in the **asserts** section. Any traits that have been included, as delimited by the **includes** keyword, can be selected (highlighting) for display in a "Trait Reference Window," which cannot be edited. Figure 9.1 also contains the interface to the syntax checker [31] and LP [11], which supports many types of verification tasks, including consistency and completeness checks. Finally, the user may select from the list of "Known Traits" for modifying or viewing available traits.

9.2 Integration with Larch tools

Next, we consider the generation of Larch specifications and illustrate how VISUALSPECS can be integrated with tools that support the processing of Larch specifications. Recall the class diagram given in Figure 4.10, which depicts the *stops* relationship between the *Brake* and *Wheel* classes. Further, recall the two instance diagrams for this class diagram given in Figures 4.3 and 4.4. Figure 9.2 contains all three diagrams, their corresponding Larch specifications, and the LDE. The corresponding specifications are all correct syntactically, but upon invocation of the Larch Prover (LP), it is discovered that the second instance diagram (Figure 4.4) is inconsistent with the class diagram. More specifically, the *stops* relationship is one-to-one, and yet wheel *w2* has brake *b2* and *b3* traveling in it. Notice that the error messages in the LP window describe the inconsistencies with respect to the axioms in the specification for the schema in Figure 4.10. In other words, when checking for inconsistencies, LP takes the *stops* relation axioms from the *INSTANCE_ASSOCIATION_stops* specifica-

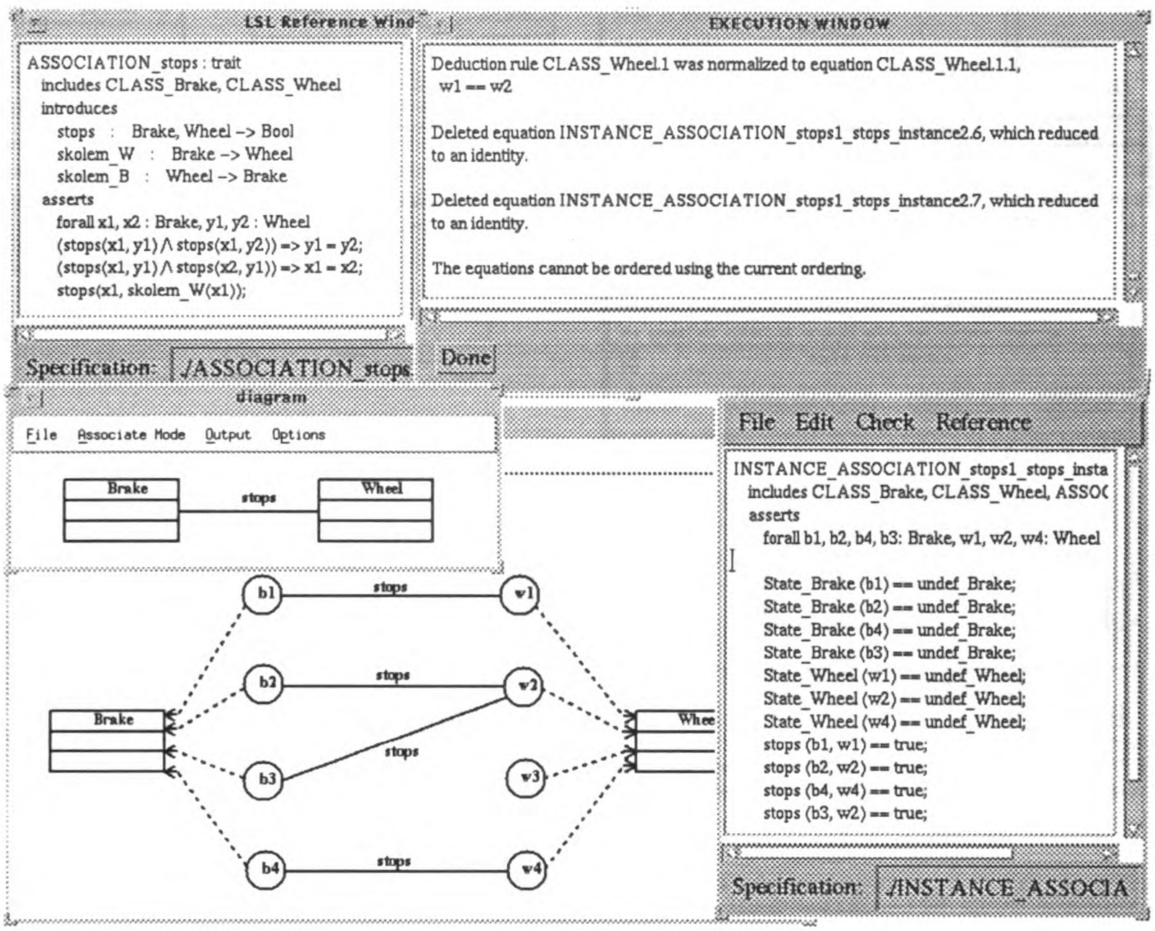

Figure 9.1. Sample session with Larch Shared Language Browser

tion and checks them against the axioms in the *ASSOCIATION_stops* specification.

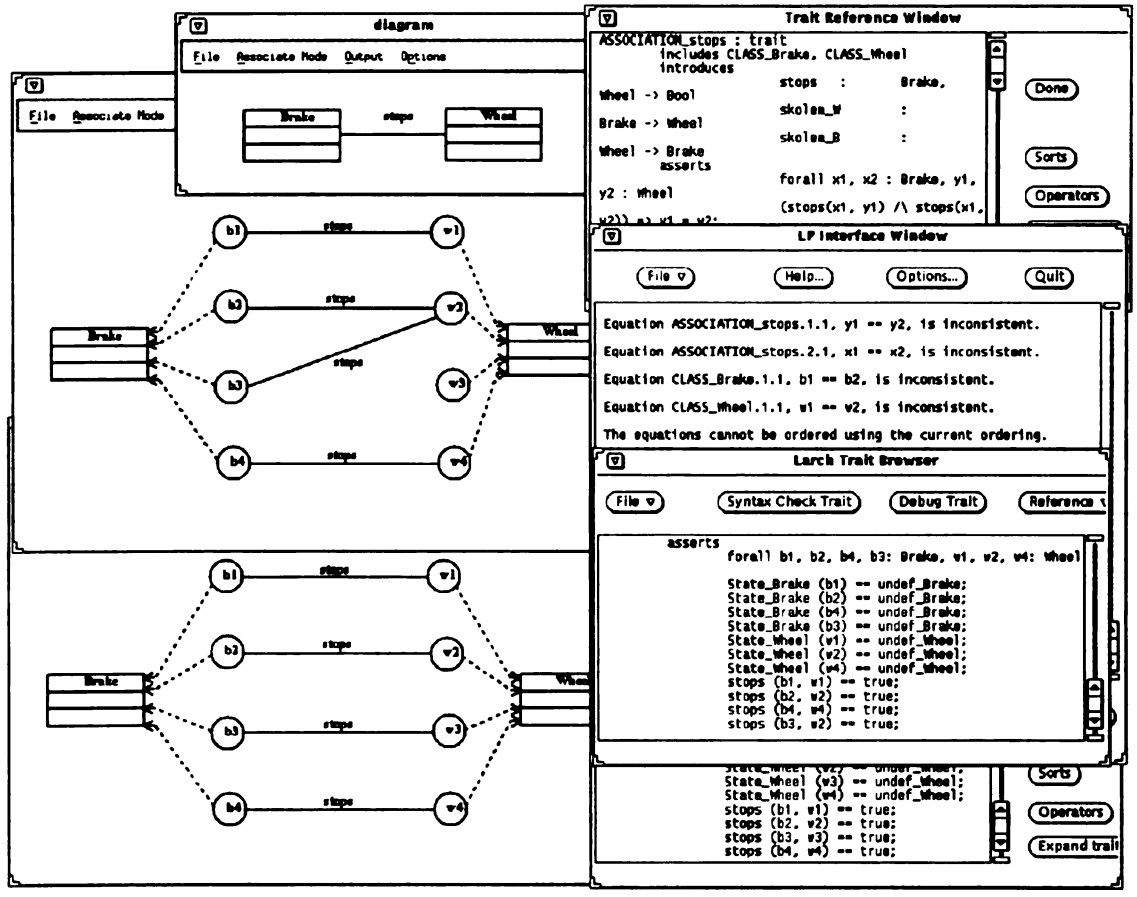

Figure 9.2. Class and instance diagrams with their respective Larch specifications

CHAPTER 10

Related Work

This chapter describes several categories of related work, including system modeling techniques that use more than one diagramming notation as well as projects that involve adding formalisms to existing diagramming notations.

10.1 Object-Oriented Modeling

Rumbaugh *et al.* [9] and Shlaer and Mellor [47] proposed two similar object-oriented modeling techniques that model the real world in terms of three orthogonal models: object, dynamic, and functional models. The modeling procedure is also divided into three phases in order to construct the three models. However, there is no well-defined guideline to direct system analysts to achieve the goal. The informal graphical notations allow the possibility of incompleteness, inconsistency, as well as ambiguity. More studies are needed to extend and understand the inter-relationships among the three models, and more specific guidelines for the analysis and design process are needed to illustrate the benefits of OMT and other graphical modeling techniques.

The remainder of the discussion describes other projects that have addressed the formalization of graphical modeling approaches.

10.2 Statechart

David Harel introduced *statecharts* [22, 36, 48, 42] as a visual formalism to describe the dynamic behaviors of complex systems. Statecharts extend conventional state transition diagrams with essentially three elements dealing, respectively, with the notions of hierarchy, concurrency, and communication. Thus state diagrams are transformed into a highly structured and concise description language that is expressive as well as compositional and modular.

STATEMATE, based on Harel's statecharts, is a commercial product intended for the specification, analysis, design, and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software or hardware. STATEMATE enables a user to prepare, analyze, and debug diagrammatic, yet precise, descriptions of the system from three complementary perspectives that capture *structure, functionality*, and *behavior.*

However, the relationship and information exchange between the three types of diagrams are complicated and potentially difficult to manipulate. The complex and intricate charts also make changes, modifications, and maintenance challenging.

10.3 Requirement State Machine Language (RSML)

Leveson, *et al.* [35, 49] developed a formal approach to using a state-based model to specify the requirements for process-control systems and successfully applied it to an industrial aircraft collision avoidance system (TCAS II). Unlike other systems that use multiple different and incompatible models within the same specification (e.g., STATEMATE [42], Hatley/Pirbhai [50], and Ward/Mellor [51]) to specify the information needed in a system requirements model, the proposed approach builds

one state-based model that includes all of the information needed to describe the behavior of the components of the systems, including the computer and the interface between the components.

The RSML approach models the required software black-box behavior along with the assumptions about the behavior of the other components of the system. Formal analysis procedures can be applied to the model in order to ensure that the software requirements model satisfies required system functional goals and constraints.

10.4 SAZ Method

The SAZ [52, 53] method was developed at the University of York. It addresses some of the problems of structured and formal methods, by integrating SSADM (Structured Systems Analysis and Design Method) version 4 and SAZ.

The SAZ approach is to use SSADM to specify the whole system. The SSADM method includes five modules: feasibility, requirements analysis and specification, logical and physical design. The system state of the logical data model and parts of the functional requirements that are particularly complex or critical are then presented in specification language Z. SAZ can be used either as a pure quality assurance tool, or as an integral systems analysis technique. It comprises three main elements: the specification of the system state (or a sub-model of the state), the specification of critical processing, and the specification of selected inquiries.

The principal benefits of the integrated method are,

- the ability to express features of the functional system requirements that are not well documented or which have been omitted from the SSADM requirements specification;

- the expression of all functional requirements in a common, mechanically-checkable notation;
- the ability to provide (formal or informal) reasoning about, for instance, pre-conditions to operations;
- the facility for concise expression of error processing.

However, SAZ is currently still a proposed methodology and, so far, there is no sophisticated tool support.

10.5 OMT and Z

Hartrum and Bailor give a high-level Z formalization of the three OMT notations, including a subset of the OMT object model notations [54]. Their formalization of object models is designed to *guide* the development of formal specifications from object models, where it is intended for the specifier to modify the specification directly to include details about analysis and design information. Our approach, in contrast, assumes that much of the analysis process will proceed at the diagram level, and our formalization techniques can be used to obtain a formal specification of the explicit and implicit information in the diagrams.

CHAPTER 11

Conclusions and Future Investigations

Software development, especially for complex systems has been a great challenge for the academic and industrial environment, thus prompting research in many disciplines of computer science, such as software engineering, artificial intelligence, knowledge engineering, etc., dealing with different aspects of the software development process. One of the fundamental problems that makes software development difficult is the large gap between the formal world of computation and the informal, real world. Formal methods are considered to be a means to bridge the gap by providing formal approaches throughout the software development process [2]. Although many formal specification languages for various application domains have been developed so far, the lack of methodologies that can incorporate formal methods into the development process hinders the widespread acceptance and use of formal methods.

In contrast, numerous diagramming techniques, due to their intuitive and easy to understand graphical notations, are broadly adopted in industry. The diagramming techniques are intended to visualize information, especially information that is complex and intricate in nature. Naturally, formal semantics has been one of the research subjects for visualizing information. Visual formalisms [36, 48], with well-defined

graphical syntax and formal semantics carried by graphical notations, are recognized as an approach to potentially bridge the gap between the formal and informal methods. In the past decade, considerable progress has been achieved in formal methods and diagramming techniques. There have also been several attempts to incorporate visual formalisms into the software development process [36, 52].

This dissertation presents a framework to support visual formalisms. The architecture and graphical library of the framework facilitates the construction of graphical editors for various diagram notations and the generation of formal specifications from the the visual formalisms. The visual formalisms developed by Bourdeau and Cheng were used as a testbed of the framework. The graphical editors for four different diagramming notations, the object diagram, the instance diagram, the state schema, and the statechart, were constructed respectively using the framework. The four parsers to create Larch specifications from their corresponding diagrams were developed. We have also shown how the formal specifications derived from their graphical representation can be used with other tools to perform system consistency checking. A cross-referencing mechanism has also been developed to facilitate the intra- and inter-model referencing during the diagram construction or browsing process.

Future investigations will focus on formalizing the Data Flow Diagram (DFD) for integration into the visual formalisms developed for the object and dynamic models. The overall objective of our work is to

1. Further investigate the integration of the three models.

2. The proposed paradigm for performing modeling with OMT will be refined and studied more extensively.

3. We will also investigate the development of transformations that can be applied to the diagrams and their specifications in order to obtain design information.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* McGraw Hill, third ed., 1992.

[2] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, pp. 8–24, September 1990.

[3] J. V. Guttag and J. J. Horing, *Larch: Language and Tools for Formal Specification.* New York, New York: Springer-Verlag, 1993.

[4] A. Hall, "Seven myths of formal methods," *IEEE Software*, pp. 11–19, September 1990.

[5] J. Spivey, *The Z Notation A Reference Manual.* Prentice Hall International (UK) Ltd., 1989.

[6] E. Brinksma, G. Scollo, and C. Steenbergen, "LOTOS specifications, their implementations and their tests," in *Protocol Specification, Testing, and Verification, VI* (B. Sarikaya and G. V. Bochmann, Eds.), pp. 349–360, Elsevier Science Publishers B.V, 1987.

[7] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software*, vol. 11, January 1994.

[8] S. L. Gerhart, "Applications of formal methods: Developing virtuoso software," *IEEE Software*, vol. 7, pp. 7–10, September 1990.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[10] M. E. Mortenson, *Geometric modeling.* New York: Wiley, 1985.

[11] S. Garland and J. Guttag, "A guide to lp, the larch prover," Technical Report TR 82, DEC SRC, December 1991.

[12] P. van Eijk, "The design of a simulator tool," in *The Formal Description Technique LOTOS* (P. van Eijk, C. Vissers, and M. Diaz, Eds.), pp. 351–390, Elsevier Science Publishers B.V., 1989.

[13] A. Fantechi, S. Gnesi, and C. Laneve, "An expressive temporal logic for basic LOTOS," in *Formal Description Techniques, II* (S. Vuong, Ed.), pp. 261–276, Elsevier Science Publishers B.V., 1990.

[14] C. B. Jones, *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, Prentice Hall International (UK) Ltd., second ed., 1990.

[15] J. Wing, "Role of formal specifications (discussion group symmary in NRL invitational workshop on testing and proving, 1986)," vol. 11, pp. 65–67, Oct. 1986.

[16] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[17] J. Bergstra, J. Heering, and P. Klint, *Algebraic Specification.* Addison Wesley, 1989.

[18] J. Guttag, J. J. Horning, and J. M. Wing, "Larch in Five Easy Pieces," tech. rep., Digital Equipment Corporation, Systems Research Center, Palo Alto, California, July 1985.

[19] J. Guttag and J. Horning, "Introduction to LCL, a Larch/c interface language," tech. rep., Digital Equipment Corporation, Systems Research Center, July 29 1991.

[20] S. Fickas, "Automating the transformational development of software," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1268–1277, November 1985.

[21] J. Stoy, *The Scott-Strachey Approach to Programming*, ch. Denotational Semantics. Cambridge: MIT Press, 1977.

[22] D. Harel, A. Pneuli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *Proceedings of the 2nd IEEE symposium on Logic of Computer Science*, pp. 54–64, 1987.

[23] J. A. Goguen and T. Winkler, "Introducing OBJ3," Tech. Rep. SRI-CSL-88-9, Computer Science Labratory, SRI International, August 1988.

[24] R. H. Bourdeau and B. H. C. Cheng, "A formal semantics of object models," Technical Report MSU-CPS-94-6, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, 48824, January 1994. (under revision for *IEEE Trans. on Software Engineering.*).

[25] R. H. Bourdeau, *A Graphical Method for Algebraic Specification and Design of Object-oriented Systems.* PhD thesis, Michigan State University, Department of Computer Science, (in preparation) 1995.

[26] R. H. Bourdeau, E. Y. Wang, and B. H. C. Cheng, "An integrated approach to developing diagrams as formal specifications," Technical Report MSU-CPS-94-26, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, 48824, September 1994.

[27] M. Wirsing, "Algebraic specification," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, Ed.), ch. 13, Amsterdam: Elsevier Science Publishers and MIT Press, 1990.

[28] J. V. Guttag, J. J. Horning, K. J. S.J. Garland, A. Modet, and J. Wing, *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[29] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A graphical environment for formally developing object-oriented software," in *Proc. of IEEE 6th International Conference on Tools with Artificial Intelligence*, November 1994.

[30] L. Lamport, *LaTeX, A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[31] J. Horning and J. Guttag, "Larch shared language checker." Private communication.

[32] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[33] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose, "Mt: A toolset for specifying and analyzing real-time systems," *IEEE Software Engineering*, pp. 12–22, 1993.

[34] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," *IEEE Transactions on Software Engineering*, vol. 19, pp. 24–40, January 1993.

[35] N. G. Leveson, M. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 684–707, September 1994.

[36] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, July 1987.

[37] R. H. Bourdeau, B. Pijanowski, and B. H. C. Cheng, "A decision support system for regional environmental analysis," in *Proc. of 25th International Symposium on Remote Sensing and Global Environmental Change: Tools for Sustainable Development (Vol. II)*, (Graz, Austria), pp. 223–233, 1993.

[38] R. H. Bourdeau, B. H. Cheng, and B. Pijanowski, "A regional information system for environmental data analysis," *accepted to appear in Photogrammetric Engineering & Remote Sensing*, 1995.

[39] B. H. C. Cheng, R. H. Bourdeau, and G. C. Gannod, "The object-oriented development of a distributed multimedia environmental information system," in *Proc. of IEEE 6th International Conference on Software Engineering and Knowledge Engineering*, pp. 70–77, June 1994.

[40] J. L. Sharnowski, G. C. Gannod, and B. H. C. Cheng, "A distributed multimedia environmental information system," in *IEEE International Conference on Multimedia Computing and Systems (accepted to appear)*, (Washington, D.C.), May 1995.

[41] 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 48824, November 1994.

[42] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, pp. 403–413, April 1990.

[43] F. Jahanian and A. K. Mok, "Modechart: A specification language for real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 933–947, December 1994.

[44] X. Song, "A framework for understanding the integration of design methodologies," *Software Engineering Notes*, vol. 20, pp. 46–54, January 1995.

[45] M. R. Laux, R. H. Bourdeau, and B. H. C. Cheng, "An integrated development environment for formal specifications," in *Proc. of International Conference on Software Engineering and Knowledge Engineering*, (San Francisco, California), pp. 681–688, July 1993.

[46] J. Guttag, J. Horning, and A. Modet., "*Report on the Larch Shared Language: Version 2.3*," Technical Report 58, DEC Systems Research Center, Palo Alto, CA, April 1990.

[47] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis.* Prentice Hall Building, Englewood Cliffs, New Jersey 07632: Yourdon Press, 1988.

[48] D. Harel, "On visual formalisms," *Communications of the ACM*, vol. 31, pp. 514–530, May 1988.

[49] M. P. E. Heimdahl, "Completeness and consistency analysis of state-based requirements," Tech. Rep. CPS-94-53, Department of Computer Science Michigan State University, East Lansing, MI 48824, October 1994.

[50] D. Hatley and I. Pirbhai, *Strategies for Real Time System Specification.* New York: Dorset House Publishing, 1987.

[51] P. Ward and S. Mellor, *Structured Development for Real-Time Systems.* New York: Yourdon Press, 1985.

[52] F. Polack, M. Whiston, and K. Mander, "The saz project: Integrating ssadm and z," in *First International Symposium of Formal Methods Europe* (J. Woodcock and P. Larsen, Eds.), (Denmark), pp. 540–557, Springer-Verlag, April 1993.

[53] F. Polack and K. Mander, "Software quality assurance using the saz method," in *Proceedings of the Eighth Z User Meeting* (J. Bowen and J. Hall, Eds.), (Cambridge), pp. 230–249, Springer-Verlag, June 1994.

[54] T. C. Hartrum and P. D. Bailor, "Teaching formal extensions of informal-based object-oriented analysis methodologies," in *Proc. of Computer Science Education*, pp. 389–409, 1994.

MICHIGAN STATE UNIV. LIBRARIES
31293014111276