A VISUAL SIMULATION SUPPORT ENVIRONMENT BASED ON A MULTIFACETED CONCEPTUAL FRAMEWORK

by

Emory Joseph Derrick

Dissertation Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Osman Balci

Osman Balci, Chairman

Richard E. Nance

James D. Arthur

Jeffrey D. Tew

C. Michael Overstreet

April 1992

Blacksburg, Virginia

A VISUAL SIMULATION SUPPORT ENVIRONMENT BASED ON A MULTIFACETED CONCEPTUAL FRAMEWORK

by Emory Joseph Derrick

Committee Chairman: Osman Balci
Computer Science

(ABSTRACT)

This thesis presents the development of a multifaceted conceptual framework for discrete-event simulation and its implementation within an integrated visual simulation support environment (VSSE). The intertwined research objectives regarding the conceptual framework and the companion VSSE are presented. A literature review of related work is conducted. The core of the thesis describes the conceptual framework (called the DOMINO), the VSSE and each of the tools from its supporting toolset, and the VSMSL (Visual Simulation Model Specification Language). Three example model applications (bus route, traffic intersection, and branch operations examples) demonstrate the use of the VSSE and the underlying DOMINO. The thesis is evaluated using the research objectives as assessment criteria.

The DOMINO is truly multifaceted. Both graphical and object-oriented, the DOMINO provides design and implementation guidance over the simulation model life cycle. The DOMINO is not restricted to specific problem domains but is independent of application domain. Several different perspectives for developing model component logic are available to modelers under the VSMSL. The VSSE demonstrates significant advances in integrated, automated support for model development which include graphical facilities for definition and specification and effective verification techniques. The VSSE underscores the contributions of the research effort and has helped to identify potential areas for future research.

ACKNOWLEDGEMENTS

Certainly, I must acknowledge the day-to-day Faithfulness and Power of a loving and personal God who has never failed me and to whom I owe everything. How Wonderful He is!

The support of my wife and her unfailing belief in me have been a constant source of strength. Ruth, how can I adequately express my thanks, love, and appreciation for all that you have selflessly done!? To my children, I also owe a special measure of gratitude. Evan and Collin, I know you have experienced sacrifices, especially in time with your Dad. Jillian, thanks for your smiles and your ability to cheer me up when I was discouraged (especially during the qualifying exams).

As a family, we could not have made it without the love, support, and financial help of my father and mother, Mr. and Mrs. Curtis Derrick, Jr. Mrs. Eleanor Williams, Ruth's mother, has also been a strong supporter and encourager with her love and help to our family. Our church family and house group have played a major role with their prayers, encouragement, and many labors of love.

A very special thanks to Dr. Osman Balci, my advisor, for his enthusiasm, energy, and excellent guidance. Dr. Balci has worked long hours on my behalf, helping me to bring this work to completion. I am grateful for Dr. Richard Nance's sound advice and technical help over the years, and for his feedback on this research. I am also indebted to my committee (Drs. Balci, Nance, Arthur, Tew, and Overstreet) for their review and screening of this work. I am especially grateful to Dr. Overstreet for travelling the many miles from Norfolk and Old Dominion University in order to serve on the committee.

I must also thank those I colabored with on the SMDE Research Project: Valerie Frankel, Lynne Barger, Dr. Bob Moose, John Bishop, Ernie Page, Jay Beams, and Fred Puthoff. All of their contributions had a positive influence upon this research. I thoroughly enjoyed working with all of them.

All of the staff of the Department of Computer Science and the Systems Research Center have been extremely supportive. In particular, thanks to Dr. Donald Allison, Dr. Verna Schuetz, Sandra Birch, Agnes Chandler, Jessie Eaves, Trish Hubble, and Sandra Griffith.

Lastly, I gratefully acknowledge the support (in part) of this research from the U.S. Navy and the IBM Corporation through the Systems Research Center.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACD	Activity Cycle Diagram
AF	Air Force
AFHRL/LRL	Air Force Human Resources Laboratory
BNF	Backus-Naur Form
CAPS	Computer Aided Programming System
CASM	Computer Aided Simulation Modelling
CF	Conceptual Framework
CLI	Command Line Interpreter
CM	Conical Methodology
CPU	Central Processing Unit
ENCOMPASS	Environment for the Composition of Programs and Specifications
FCFS	First Come First Served
FMS	Flexible Manufacturing System
GPSS	General Purpose Simulation System
GPVSS	General Purpose Visual Simulation System
GRAPHSIM	Graphical Simulation Modeling
IMDE	Integrated Model Development Environment
IMS	Internal Model Specification
IMSE	Integrated Modeling Support Environment
IPSE	Integrated Project Support Environment
ISPG	Interactive Simulation Program Generator
IVIS	Intelligent Visual Interactive Simulation
JDL	Jipc Description Language
KBMC	Knowledge-Based Model Construction
KBSim	Knowledge-Based Simulation
KBS	Knowledge-Based Simulation System
KEE	Knowledge Engineering Environment
LALR	Look-Ahead Left-To-Right Rightmost
LAN	Local Area Network
NLUS	Natural Language Understanding System
OCI	Object Class Instance
OMS	Object Management System
OOP	Object-Oriented Programming
PMV	Programmed Model Verification
PRISM	Productivity Improvements In Simulation Modeling
RESQME	RESearch Queueing Package Modeling Environment
ROSS	Rand Object-Oriented Simulation System
SES	Simulation Environment System
SIMAN	SIMulation ANalysis
SMDE	Simulation Model Development Environment
SPIF	Simulation Problem Intelligent Formulator
SPL	Simulation Programming Environment
SUNOS	Sun Microsystems Operating System
SWIRL	Simulating Warfare In The Ross Language
TESS	The Extended Simulation Support System
TWIRL	Tactical Warfare In The Ross Language
UK	United Kingdom
VIM	Visual Interactive Modeling
VIS	Visual Interactive Simulation

VM . . . . . Visual Modeling
VPI&SU . . . . . Virginia Polytechnic Institute and State University
VS . . . . . Visual Simulation
VSMSL . . . . . Visual Simulation Model Specification Language
VSSE . . . . . Visual Simulation Support Environment
WYSIWYG . . . . . What You See Is What You Get
WYSIWYR . . . . . What You See Is What You Represent
YACC . . . . . Yet Another Compiler-Compiler

CHAPTER 1
PROBLEM DEFINITION AND OVERVIEW

History provides many remarkable and challenging examples of the determination and spirit of a single individual or group of individuals in achieving objectives to meet identified needs and accomplish worthy goals. In the totality of human experience, this research is but a small part. Nevertheless, the importance of objectives in reaching a desired end rings true. Without accurate aim, one misses the mark. This chapter clearly presents the goals of the research and the defined objectives for their satisfaction in the domain of simulation model development.

As a point of introduction and backdrop for the thesis, the life cycle of a simulation study is discussed. Critical problems which modelers face while applying the principles of the life cycle are identified. Having clarified the issues to be confronted, we explain their importance and why they are worthy of research attention. We give a proposal for solutions and articulate the overall research goals. The objectives for fulfilling these goals are set forth. The organization of this thesis is also described to lend a broad perspective on the research approach. At the conclusion of this chapter, we present a summary of the contributions of the research described herein.

1.1 Introduction

Concern over high costs and limited utility of modeling projects conducted and supported by the federal government [Nance 1981b] has stimulated the search for means of controlling these costs and evaluating computer-based models. The current economical climate has only added to concerns regarding cost overruns, mismanagement, and poor planning regarding software development projects of all kinds. Methodologies formulated for the costly "software problem" are found in abundance in the software engineering domain [Nance and Arthur 1988]. The general principles of software development (software life cycle models, etc.) and the lessons learned from software engineering research can certainly be applied to the thesis problem domain of simulation model development. However, there are some very real distinctions (including, e.g., real data collection requirements for model validation, reliance on statistical

analysis, the persistent issues of model credibility, etc.) which Overstreet et al. [1986] have concisely presented. These distinctions are of sufficient significance to require continuing research in the search for more powerful tools, strategies, and environments for the simulationist. Altogether, these issues have influenced and contributed to the formulation of the simulation model life cycle [Nance 1981b; Balci 1986a], the primary focus of the next section.

1.1.1 The Simulation Model Life Cycle

The life cycle of a simulation study, shown in Figure 1.1, contains 10 phases, 10 processes, and 13 credibility assessment stages. The ten phases are designated by oval symbols. The processes are indicated by dashed arrows. Solid arrows represent the credibility assessment stages. The double-ended arrows indicate the many points of iteration and feedback; the life cycle is not a strictly sequential progression among the phases, processes, and assessment stages.

The life cycle had its beginnings as a broad group of guidelines called the "model life cycle" [Nance 1981b]. This particular form of the life cycle centered on those phases called the model development phases (the circular phases of the current form): the conceptual model, communicative model, programmed model, experimental model, and model results. Also included were other phases called integrated decision support and modified model phases. In the same paper, Nance [1981b] presented an important methodology having a significant and historical impact on the thesis research, the Conical Methodology (CM). The CM's influence on this work is a reoccurring theme throughout the thesis and is more fully discussed in other chapters.

Nance and Balci [1987] extended the early form of the life cycle to further clarify and distinguish an additional group of phases (besides the model development phases), the problem definition phases: the communicated problem, the formulated problem, and the proposed solution technique. These phases are for precisely defining the system to be studied and the study objectives. Improper problem definition increases the probability of committing the Type II error (accepting the study results when the results are not credible) or the Type III error (solving the wrong problem) [Balci 1989]. In addition, Nance and Balci

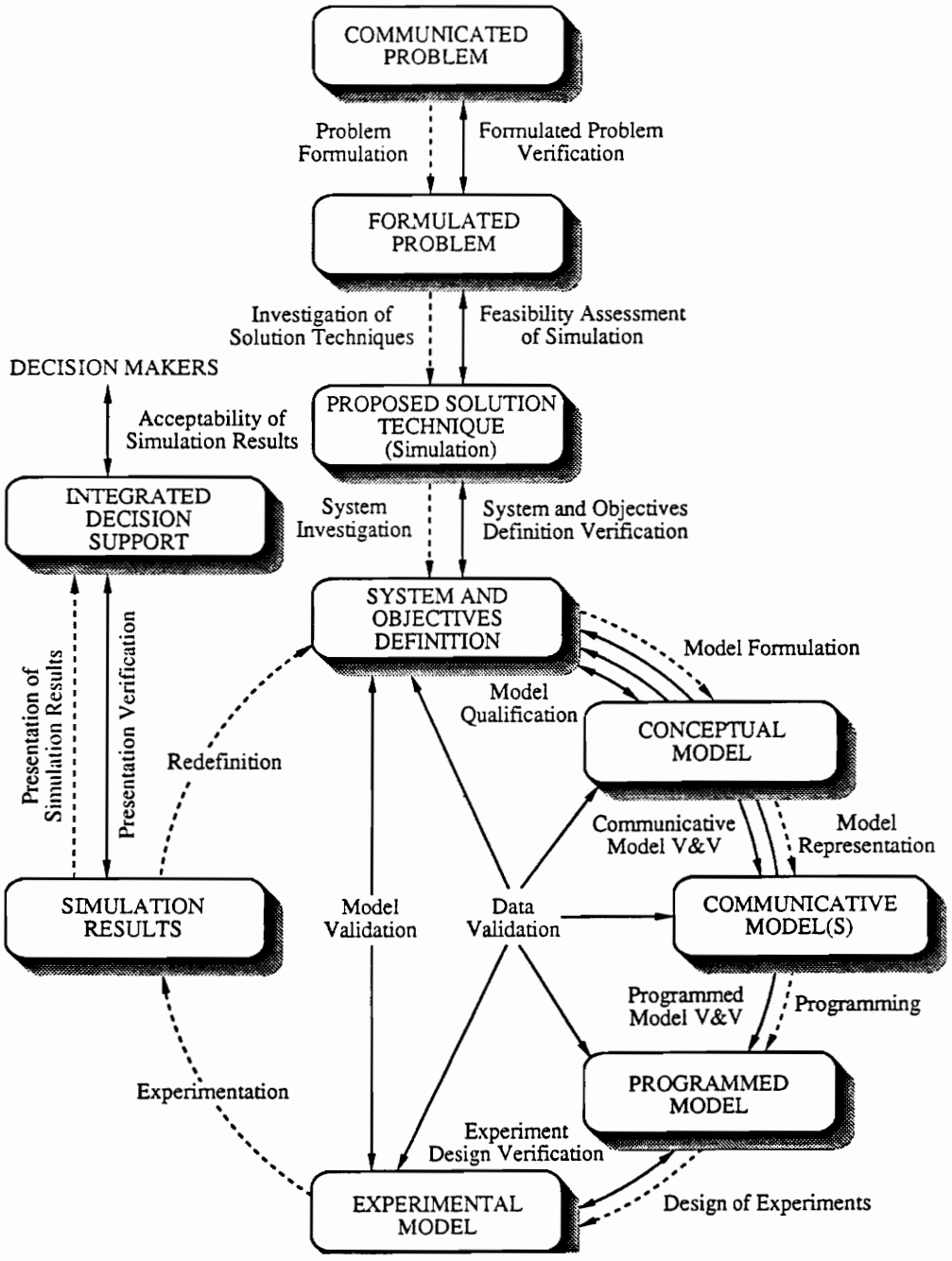

Figure 1.1 Life Cycle of a Simulation Study

[1987] set apart the integrated decision support phase (presenting study results to sponsor and decision makers for acceptance) from the model development phases. Chronologically, in the conduct of a simulation study, these three periods or groups of phases for the model life cycle are problem definition, model development, and integrated decision support.

Balci [1986a, 1989, 1990] recognized the need for assessing the acceptability and credibility of simulation results. The credibility assessment stages were added. These assess the credibility of each process as one progresses through the phases in the life cycle and are critical to the prevention of the Type II error. Model credibility is essential for a successful conclusion to integrated decision support and, ultimately, the simulation study.

1.1.2 Context of the Research Problem

Having introduced the life cycle, we now relate the research problem to the life cycle. The heart of the research problem lies within the model development phases of the life cycle of a simulation study. The model development phases cover both model design and model implementation. Model design encompasses the *model formulation* and *model representation* processes of the life cycle in creating the *conceptual model* and transferring the conceptualization to a more concrete form, the *communicative model*. The communicative model represents the specification which is to be used as the basis for implementation. Fundamentally, a modeler is guided and assisted in the creation of such specifications by a *conceptual framework*. Model implementation then occurs during the accomplishment of the *programming* and *design of experiments* processes while converting the communicative model into the *programmed model*, eventually reaching the *experimental model* phase. Earlier research [Derrick 1988, 1989a,b; Derrick et al. 1989] showed that existing conceptual frameworks fail to provide satisfactory guidance for both model design and implementation; guidance was generally found to be sufficient in one of the areas, but not in both.

Simulation Model Development Environments (SMDEs) [Balci 1986b; Balci and Nance 1987a,b] provide an integrated collection of tools for automated modeling support. (The reader can refer to Chapter 2 for a complete overview of the SMDE.) Environments, providing integrated, cost-effective, and

automated modeling support throughout the entire model development life cycle, are a necessity. The expanding complexity of systems being modeled in today's work place cannot be controlled or managed otherwise.

Balci and Nance [1987b] indicate that specifications created while using the model development environments in the performance of model formulation and representation should be: (1) analyzable, (2) fully translatable into executable code, and (3) domain independent. Each of these characteristics has important implications. The first permits the early detection of model errors and reduces the probability of committing the type II error - the error of accepting invalid model results. The achievement of the Automation-Based Paradigm [Balzer et al. 1983] is heavily dependent upon the second and allows a modeler to focus all developmental effort at the definition and specification levels without the burdens of programming and other labor-intensive tasks. Additionally, testing and maintenance are simplified. The solution of *any* discrete-event simulation problem can be attempted when the third attribute holds. A conceptual framework which promotes the formation of specifications possessing these essential attributes is critically needed.

The development of such a conceptual framework offers substantial benefit to simulationists. For example, support is extended for wider applicability and assistance throughout the simulation life cycle. Model quality is improved. The model development effort is more efficient and productive, and model development time is reduced. Simulation practitioners can tackle more complex models. Project teams need not be highly specialized and trained in certain Simulation Programming Languages (SPLs) or be tied to a wide variety of commercial, domain-dependent products when their needs are diverse. The list of benefits is extensive.

Simulation studies are too often conducted without sufficient emphasis on Programmed Model Verification (PMV) [Balci 1990; Whitner 1988; Whitner and Balci 1989]. Validation and verification techniques are rarely stressed in simulation texts. Visualization enhances these techniques [Bishop 1989; Bishop and Balci 1990]. Visual representations improve the quality and expedition of thinking [Harel 1992]. A prototype Visual Simulation Support Environment (VSSE) with an integrated toolset is needed.

A VSSE can extend our knowledge of all phases of model development. In particular, we can better understand PMV and its application within the life cycle.

1.2 Research Objectives

The overall research goals, directed at these identified needs, are to: (1) develop a new conceptual framework which is versatile and multifaceted, meeting many of the diverse needs of the simulation life cycle and model development environment support, and (2) to design and construct a VSSE with an integrated set of tools to provide the desired automated support for model development.

These two goals are intertwined. A platform is needed for the evaluation of the conceptual framework. Hence, a VSSE based upon the new conceptual framework is required for the satisfaction of the first goal. The framework can only then be evaluated concerning its potential usefulness within a model development environment and its ability to support environment functions. On the other hand, as the basis for a VSSE, the conceptual framework must provide guidance for simulation visual model design and implementation. That is, the conceptual framework must support visualization in addition to the other requirements of the support environment. The developmental and design objectives which are identified and defined below are intended to meet these symbiotic goals. Note that these objectives serve as assessment criteria for the subjective evaluation in Chapter 8.

1.2.1 The Conceptual Framework Developmental Objectives

This section delineates the developmental objectives concerning the desired traits of the new conceptual framework described in the above goals. The conceptual framework should:

Objective 1. Provide both design and implementation guidance, thereby enabling a broad range of support during the simulation model life cycle.

Objective 2. Contain the underlying organization and structure to achieve the definition and specification of visual simulation models.

Objective 3. Possess object-orientation to include inheritance of class attributes and model component logic.

Objective 4. Possess graphical-orientation to simplify the definition and specification tasks.

Objective 5. Embody a WYSIWYR (What You See Is What You Represent) philosophy, allowing modelers to represent a system and its components as they are conceptually or naturally perceived, and simplifying the transition between the conceptual model and the communicative model.

With Objective 5, the communicative model (phase) becomes a natural extension of its predecessor with an easy mapping from the mind into the appropriate high-level representation.

Objective 6. Adhere to the guiding principles of the Conical Methodology [Nance 1987]:
- Top-down definition with bottom-up specification.
- Documentation and specification are inseparable.
- Iterative refinement and progressive elaboration are essential.
- Verification begins with communicative models and continues throughout the development process.
- Specification is independent of implementation.

This objective implies adherence to the more general modeling methodology principles of abstraction, concurrent documentation, functional decomposition, hierarchical decomposition, information hiding, iterative refinement, life cycle verification, and progressive elaboration [Nance 1987].

Objective 7. Contain a rich and expressive terminology which applies to any modeling domain in the manner of general purpose "vanilla" frameworks [Harel 1992].

Objective 8. Include flexible but powerful representation schemes for the specification of model component logic and the rules for model component interaction.

Objective 9. Support the Automation-Based Paradigm by: (1) enabling modelers to focus attention and energy at the specification level (whether during design or maintenance and modification), and (2) producing specifications which are fully translatable into executable code.

Objective 9 implies the more universal objective of maintainability.

Objective 10. Produce specifications which are analyzable with maximum diagnostic capabilities in terms of analytical, comparative, and informative assistance [Nance and Overstreet 1986, 1987].

1.2.2 The Visual Simulation Support Environment Design Objectives

Design objectives for the VSSE are multi-directed. Some relate to the evaluation of the new conceptual framework; others relate more directly to VSSE capabilities, apart from the conceptual framework. Objectives are grouped by associated tool when appropriate.

VSSE (Complete toolset):

Objective 1. Provide a fully functional and highly integrated toolset. Accomplish the integration and communication among the tools using (primarily) a relational database

representation (e.g., INGRES [Sun Microsystems 1986a]) of the specification.

Objective 2. Provide a user interface that satisfies the nine usability principles for interfaces [Nielsen 1990]: simple and natural dialogue, speak the user's language, minimize the user's memory load, consistency, provide feedback, provide clearly marked exits, provide shortcuts, good error messages, and prevent errors.

Objective 2 supports usability of the human computer interface [Nielsen 1992].

Model Generator:

Objective 3. Provide means for storing model component information in a library.

Objective 4. Provide suitably implemented mechanisms for inheritance of class attributes and model component logic specifications.

Objectives 3 and 4 support the modeling methodology objective of reusability.

Objective 5. Provide sufficient capabilities for attribute access and communication between model components to include message passing and methods.

Objective 6. Provide detailed querying capabilities for displaying specification status and progress to modelers, and supporting methods of informal analysis verification techniques. Display query results in appropriate tabular or graphical forms.

Objective 7. Provide graphically based means for the flexible hierarchical definition of model static and dynamic structures. This definition should be characterized by ease and simplicity of movement between the levels of the hierarchy.

Objective 8. Provide an expressive (able to specify the various model component interactions) specification language which uses English-like expressions like the HyperTalk language [Winkler and Kamins 1990].

Objective 9. Provide ability to translate model component logic specifications directly to the target language and eliminate the need for modelers to interact at the target code level. Relate translation errors to the modeler-defined logic specification, not the target language.

Objective 10. Provide extensive use of symbol tables containing specification data from the relational database, supporting enhanced static analysis techniques during the translation process.

Objectives 6 and 10 support correctness and testability.

Model Analyzer:

Objective 11. Provide completeness and consistency diagnostic checks on the specification. Use the relational database representation as the basis for analysis. Tailor the analysis to the unique features of the framework.

This objective supports reliability and correctness.

Model Verifier:

Objective 12. Provide execution tracing with appropriate means of effectively managing the trace data and relating runtime errors to the modeler-defined specification. Use the relational database for storage of the trace data.

Objective 13. Provide the means for assertion checking as an additional facility for dynamic analysis.

Objective 14. Allow the creation of performance reporting upon runtime execution by providing execution profile reports.

Objectives 12, 13, and 14 support correctness, reliability, and testability.

Model Translator:
Objective 15. Provide automatic creation of the executable model from the modeler-defined specification, supporting the Automation-Based Paradigm.

This objective supports maintainability and adaptability.

Visual Simulator:
Objective 16. Provide for the visualization of the model from any desired runtime context. Movement between contexts should be simply executed.

Objective 17. Provide the ability to inspect model component attributes or modeler-defined performance measures at any instant during the visualization of the running model.

Objective 18. Provide the ability to run the simulation in the background without animation, producing statistical analysis reports.

Objectives 16, 17, and 18 support testability and correctness.

1.3 Thesis Organization

The thesis organization provides a comprehensive overview of the various (and many) facets of the thesis research. Altogether, the overview reveals the overall perspective taken in the research approach. Figures and examples are used extensively in each chapter to explain and to clarify the written presentation.

Beyond this chapter, Chapter 2 reviews the literature and related research. Common conceptual frameworks for simulation modeling are described. Software development environments (to include simulation support environments) are reviewed. Visualization in simulation is covered.

Chapters 3, 4, and 5 form the core of the thesis. A detailed description of the new conceptual framework (called DOMINO), including a short essay on the motivation which led to the development of the framework, is contained in Chapter 3. The DOMINO terminology and features for model design and implementation are presented, and a brief history of its development is given. Chapter 4 characterizes the VSSE in a top-down fashion and provides an indepth tool-by-tool description. Briefly, using the Model Generator tool, a user creates a logical, graphical, and activity-based specification of a simulation model.

The Model Analyzer enables the specification to be analyzed for consistency and completeness. Validation and verification techniques are available via the Model Verifier. Finally, the Model Translator and Simulator tools provide for the automatic translation of the specification into executable code and for the visualized execution of the selected simulation model. Central to the conceptual framework is the specification language for creating the specification of model logic. The specification language (called VSMSL: Visual Simulation Model Specification Language), a high level approach to logic specification, is described in Chapter 5. Here, the unique capabilities of this English-like specification language are carefully reviewed.

The next two chapters give insights into the underlying implementations and the application of the DOMINO in the context of the VSSE. Chapter 6 describes the unique implementation of the object-oriented modules and how this was accomplished within the C programming language. This chapter is of specific interest to C programmers. Three example model applications (a bus route, a traffic intersection, and a branch operations example) are explained in Chapter 7. Lessons learned from these applications serve as the basis for the evaluation of the design objectives found in Chapter 8. Finally, Chapter 9 presents the thesis conclusions and directions for future research.

1.4 Summary of Contributions

Chapter 9 presents the contributions of the thesis effort. These contributions, summarized below, are wide ranging in their scope. Gains are expected in the SMDE research effort, and simulation model development has been positively affected.

The DOMINO:

- The DOMINO is truly multifaceted and represents a step forward in conceptual frameworks for the design and implementation of simulation models.
- The incorporation of visualization as a prominent element of the DOMINO enhances the verification and validation tasks.
- The DOMINO is application domain-independent in all aspects of terminology and representation supplying significant advantages to project teams with diverse modeling needs.
- The DOMINO has achieved marked progress toward achieving the Automation-Based

Paradigm, with potential gains in efficiency and productivity in the model development effort, and reductions to development time.

- The DOMINO fully supports the guiding principles of the Conical Methodology and is a strong candidate for use as the framework for the SMDE.

The VSSE:

- The VSSE with its fully functional and highly integrated toolset demonstrates significant advances in the SMDE prototyping effort and in automated support for model development.

- The VSSE provides a wide range of effective verification techniques. This contributes to an area in the life cycle of a simulation study where emphasis is currently lacking.

- The VSSE graphical facilities for structure definition and contextual visualization clearly offer help to the modeler in overcoming the complexity problems associated with large modeling efforts. The screen designs demonstrate simplicity and ease of use.

- The VSSE enables the effective evaluation of the DOMINO and underscores the contributions of the research effort.

CHAPTER 2
LITERATURE REVIEW

Related research spans several fields of interest. This section gives the necessary background for each of these fields and presents each within three broad categories. The first category is that of conceptual frameworks. The discussion includes a brief description of conceptual frameworks which are applicable to discrete-event model representation techniques. Another important category is that of simulation model development environments. Influences from a broad spectrum of software engineering topics are discussed: software development environments, software engineering environments, and simulation support environments. Verification and validation aspects and, in addition, the impact of automatic programming and program generators are included. Visualization in simulation is the final pertinent category.

2.1 Conceptual Frameworks for Simulation Modeling

A *Conceptual Framework* (CF) is an underlying structure and organization of ideas which constitute the outline and basic frame that guide a modeler in representing a system in the form of a model. "Simulation strategy", "world view", and "formalism" are other terms used in lieu of CF. CFs possessing applicability to discrete-event simulation are reviewed. Derrick et al. [1989] compare thirteen CFs from four categories: (1) the classical CFs (Event Scheduling, Activity Scanning, the Three-Phase Approach, Process Interaction, and Transaction Flow), (2) other discrete-event CFs (System Theoretic Approach, Conical Methodology, and Condition Specification), (3) more generic CFs (Entity-Relationship-Attribute, Entity-Attribute-Set, and the Object-Oriented Paradigm), and (4) CFs with potential applications (Structured Modeling and Process Graph Method). Brief descriptions of the CFs which are applicable to the concerns of the proposed research are given below. Further details of these and the other mentioned CFs can be found in an extensive review presented by Derrick [1988].

2.1.1 Event Scheduling

Under this CF, the modeler considers the system of interest to be described in terms of *events*. An

event routine or set of actions following a state change in the system is associated with each event [Pidd 1984]. Kiviat [1969] describes the approach as one which seeks to execute the event routine "only when a state change occurs." Lackner [1964] states that the representation of the duration and sequence of events is a key issue. Regarding duration, events may be *determined* or *contingent* [Nance 1981a]. Determined events occur at a known future time. Event routines associated with determined events can be explicitly scheduled with the implementation of an *event list*, a list of event notices or records which are ordered by time. The event list helps to solve the sequencing problem identified by Lackner [1964]. Event routines are selectable for execution when appropriate. Conditional (contingent) events occur at the satisfaction of some set of conditions which cannot be predicted in advance. The checks for contingent events are contained within the event routines.

2.1.2 Activity Scanning

This approach requires that the modeler identify the types of objects in the system of interest, the *activities* which they perform, and the conditions under which these activities occur. A test set of boolean conditions or *testhead* [Pidd 1984] pinpoints the state change that initiates an activity. The state transitions and interactions among the model objects are produced and linked together by the testheads and their associated activity descriptions. Activity scanning executives are two-phased, containing a time scan for defining the time increment and update of the system clock and an activity scan for checking the testheads and for performing the necessary actions.

2.1.3 The Three-Phase Approach

The Three-Phase Approach is a modification of Activity Scanning which improves execution efficiency by reducing the amount of scanning. Activities are categorized as B-activities or C-activities [Tocher and Owen 1961]. The B-activities are the bound-to-occur or book-keeping activities that represent unconditional state changes which can be scheduled in advance. The C-activities are conditional or cooperative activities that represent state changes which are conditional upon the cooperation of different

objects or the satisfaction of specific conditions. B-activities can be scheduled using an event list technique, eliminating unnecessary scans from the activity scan phase. C-activities with testheads must enter the usual, repetitive activity scan. Now, instead of two phases, the executive is three phases with the addition of another phase (to execute the B-activities which are due) just before the scan phase.

2.1.4 Process Interaction

Instead of the event or activity, Process Interaction uses the *process* as its basic building block [Kiviat 1969; Fishman 1973; Pidd 1984]. The process represents a sequence of events and interspersed activities through which a specific object moves. As the object moves through its process, it may experience certain delays and be blocked in its movement. Delays can be time-based and determined (e.g., service times, arrival times) or state-based and contingent (e.g., wait-until situations). Objects experience periods of activity during process execution and periods of inactivity or delay. When an object experiences a delay in its process and becomes "passive", another model object is allowed to become "active" [Franta 1977] initiating or resuming its process. Such delays are incurred and execution is shifted (to another object) at *interaction points* [Kiviat 1969]. An object process returns to an active state following such interaction at its *reactivation points*. Process Interaction enables the modeler to clearly grasp a model's structure since each object or class of objects can be represented by a single, coherent process rather than by multiple event routines [Kiviat 1969; Fishman 1973].

2.1.5 Transaction Flow

Transaction Flow handles the time and state relationships of the model in exactly the same manner as Process Interaction. Several different and distinct characteristics are noted. "Transactions" are created and moved through the blocks, executing specialized actions that are "associated" with each block. The block structure generates a rigid structure which limits the "examination and communication" among system components [Shub 1980]. In addition, as objects (transactions) pass through these blocks, "predefined processes" are activated which are hidden to the modeler. Statement languages with their lower level

primitives provide generality and flexibility to the modeler.

Tocher [1965] characterizes simulation programming languages as machine or material-oriented. He further defines transactions to be material or temporary objects. In a machine-oriented view, servers (machines) are the dominating and active influence in the model [Kreutzer 1986]. They obtain the material objects (transactions), operate on them, and place them in (or remove them from) queues. Transaction Flow promotes material-oriented models in which the transactions are the dominant objects. Servers, now passive, are "acquired, held, and released again" by the transactions which flow from machine to machine [Kreutzer 1986].

2.1.6 Conical Methodology

The Conical Methodology [Nance 1981b, 1987] divides model representation tasks into two stages: model definition and model specification. The Conical Methodology is based on the Object-Oriented Paradigm and seeks to achieve five primary objectives: model correctness, testability, adaptability, reusability, and maintainability. The Conical Methodology mandates a strict separation between model representation and model execution. *Top-down definition* and *bottom-up specification* techniques are at the core of its procedural guidance. Top-down model definition produces a static model representation and is accomplished through a hierarchical decomposition of the model into successive submodels. At each level of decomposition, attributes, including attribute dimensionality and range of values, are assigned (to a particular submodel associated with that level) and are classified by type in accordance with Nance's [1987] taxonomy tree. Bottom-up specification produces a model representation which contains the necessary information for model dynamics. Specification, "...the process of describing system behavior so as to assist the system designer in clarifying his conceptual view of the system" [Barger 1986], is started at some base-level submodel in the decomposition hierarchy and is performed at successively higher levels until the model level is reached. The time flow mechanism to be used in building the model is not dictated.

2.1.7 Condition Specification

Condition Specification produces a model specification that can be analyzed to: locate problems with the specification, help build an executable model representation, and produce meaningful model documentation [Overstreet and Nance 1985; Nance and Overstreet 1987]. The Condition Specification, attributed to Overstreet [1982], is the principal target form for bottom-up specification within the Conical Methodology. The analytic and diagnostic capabilities of the Condition Specification are an extremely desirable and significant strength. The principal components of the Condition Specification are the *interface specification, the specification of model dynamics*, and the *report specification*. The input and output attributes of the model are described within the interface specification. The specification of model dynamics consists of a set of *object specifications* and a *transition specification*. The object specification is a complete listing of all objects and their attributes. A value range is given for each attribute. The transition specification contains the description of model dynamics in the form of condition and action pairs (CAPS). The report specification is defined for the data output or model results.

2.1.8 Object-Oriented Paradigm

The Object-Oriented Paradigm is viewed as a framework for system design. According to Meyer [1987], "object-oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs." Functions tend to change in order to adapt to changing needs whereas objects remain more or less constant [Meyer 1987]. The paradigm is principally characterized by two features: (1) *encapsulation* of data and operations and (2) an *inheritance* mechanism for developing object hierarchies. An object can be considered to be "encapsulated", an "armor-plated" entity [Cox 1986] with "private data and a set of operations that can access that data." The use of objects therefore improves the reliability and maintainability of system code. Additionally, by inherent abstraction, the object improves the view of the system by introducing a high level perspective and promotes reusability of code. The principles of *modularity, abstract data typing*, and *information hiding* are accommodated. Inheritance

is the ability to define new objects from existing objects by extending, reducing, or otherwise changing their functionality. New instances of an object class can be easily created which automatically inherit the attributes of that class definition. Inheritance supports hierarchical structures that are commonly found in the real world and provides substantial benefit to the user by improving his understanding and view of the system.

2.2 Software Development Environments

Sommerville [1989] states that an *environment* "is used to encompass all of the automated facilities that the software engineer has available to assist with the task of software development." This includes tools like electronic mail, documentation helps, etc. The structure of an environment is most often composed of layers for the support of each function. Software engineering environments support all stages of the software process and are sometimes called *integrated project support environments* (IPSEs). In the context of simulation support environments, Balci et al. [1990] define an *environment* as "integrated set of hardware and software tools that provide cost-effective, automated support throughout the entire development life cycle." The above definitions put a clear emphasis upon support throughout the complete life cycle, not just a portion of it. Agreement among definitions and typing of environments is not so easily found.

Sommerville [1989] categorizes software development environments as programming environments (language dependent or independent) and as software engineering environments. Programming environments do not support a life-cycle emphasis and are, generally, not considered in this coverage.

Dart et al. [1987] are not as rigid in their classification scheme. Software development environments are placed into four categories: language-centered, structure-oriented, toolkit, and method-based. The range of life-cycle support varies across these categories. CEDAR [Teitelman 1985] is an example of the language-centered environment with seemingly little life cycle support beyond the programming task. GANDALF [Habermann and Notkin 1986] and the Cornell Program Synthesizer [Reps and Teitelbaum 1984] are structure-oriented, stressing syntax-directed editors, user-interactive tools for building programs

in terms of language constructs. Structure-oriented environments seem to support a wider range of life-cycle activities. The toolkit environments operate primarily as a collection of tools geared to programming support activities, attempting to maintain language independence while bringing in further life cycle support. ARCADIA [Taylor et al. 1989] is an example of the toolkit environment. Method-based environments support particular software development approaches or frameworks, like Entity-relationship diagrams [Chen 1976] and Petri-nets [Peterson 1977]. Software Through Pictures [Wasserman and Pircher 1987] is one such environment.

Finally, Hendersen and Notkin [1987] group environments by *activity* (range of life cycle support), *domain* (class of software being supported), and *mechanism* (the concepts and methods characterizing the environment's internal and external structure). This is a particularly enlightening perspective regarding the proposed research. Our interests lie in environments which offer a broad range of software activities (over the entire life cycle), over any discrete-event simulation domain, and centering upon a single mechanism (conceptual framework).

Research in software engineering environments (following Sommerville's [1989] bent) is briefly discussed since this most closely fits the research direction. Of particular interest, object-oriented technologies are expected to play a major role in the definition of integrated support environments in the 1990's [Basili and Musa 1991]. Two notable software engineering environments are introduced. Simulation environments are then reviewed with special emphasis on the Simulation Model Development Environment (SMDE).

2.2.1 Software Engineering Environments

IStar [Dowson 1987a,b; Stenning 1986], developed by Imperial Software Technology, is an excellent example of an integrated, language-independent, project-support environment which provides comprehensive support for "every aspect of software and system development throughout the life cycle, encompassing project management, data and configuration management, and technical development" [Dowson 1987a]. Software projects can be hierarchically decomposed into layers of tasks which

aggregately support project completion. In IStar, these tasks are called *contracts*; IStar is therefore designed to support the *contractual approach*. IStar uses a layered approach, like that discussed above, consisting of a contracts database, database interface, communications and tools interface, and user interface. Tools are grouped by workbenches for the different functions which are needed (e.g., project and resource management, data and configuration management, technical development, quality management, and office automation and system administration functions). IStar does not enforce the sequencing of developmental activities.

The SAGA project [Campbell 1986] is another notable research effort into software engineering environments. This project is specifically aimed, however, at life-cycle support for developing and maintaining aerospace systems and applications software. ENCOMPASS is the prototype system which implements the SAGA concepts. The environment is based upon three basic components for meeting requirements: (1) a configuration management component for maintaining consistency and integrity of the resulting products during software development, (2) a software development component supporting the life-cycle model, methodologies, etc., and (3) a tools component. The project is investigating several different design methodologies. An important area of focus is the prototyping and use of executable specification languages.

2.2.2 Simulation Support Environments and Related Work

This section focuses on several efforts which are fueling the fires of simulation support environment research. The coverage is not comprehensive. Those research efforts which are most stimulating to simulation support environment issues are presented. Where applicable, object-oriented facilities, the method of creating specifications, the underlying conceptual framework, and automatic translation capabilities are discussed.

2.2.2.1 ROSS and KBSim

ROSS [Klahr 1986; McFall and Klahr 1986; Rothenberg 1988; McArthur et al. 1984] was developed

by RAND Corporation. The goal of ROSS (Rand Object-oriented Simulation System) is "to provide a programming environment in which users can conveniently design, test, and modify large knowledge-based simulations of complex mechanisms" [McArthur et al. 1984]. ROSS, an object-oriented language, is touted as the first to provide a multiple inheritance mechanism. Implemented in LISP, ROSS is interactive. A ROSS simulation "can be interrupted while it is running, the state can be queried or the code modified, and the simulation can then be resumed" [McArthur et al. 1984]. Alternative model designs can therefore be quickly and easily explored. A sophisticated screen-oriented editor, a color graphics package, and facilities for browsing and editing objects and their behaviors (rules for object interaction) are interfaced with ROSS producing a powerful simulation development environment. ROSS is built upon LISP, a symbolic AI language. No special capabilities exist for specification creation. Once coded, the ROSS program is the specification which is automatically translated into the underlying LISP implementation. ROSS is event-oriented. Most applications are created for military battle simulations. Two such simulations, TWIRL and SWIRL, implement an expert knowledge-base in which domain specific knowledge is represented within simulations in terms of the different object classes and their behaviors [Klahr 1986].

The KBSim (Knowledge-Based Simulation) project, also developed at RAND, contains a logic-based modeling environment called MODL (written in PROLOG) and enables users to conduct what-if, why, when, how, ever/never, and goal-directed questions with their simulations. In addition, RAND is researching the use of intelligent exploration techniques in which a user will be able to interact with the simulation to "run it to a certain point and then try excursions, changing assumptions, backing up, and trying different paths" [Rothenberg 1988].

2.2.2.2 KBS

KBS (Knowledge-Based Simulation System) is similar in approach to ROSS in that it is object-oriented, contains attribute and behavioral descriptions, and provides interactive access and display [Reddy et al. 1986]. Model specification is done via a window-based editor in which the modeler creates *schema*,

the basic unit for representing objects, processes, ideas, etc. KBS is event-oriented and manages a calendar of event notices. Recently renamed Simulation Craft [O'Keefe and Roach 1987; Nielsen 1986], KBS provides knowledge frameworks under which some model validation, reduction (abstraction), and analysis are performed.

2.2.2.3 SIMKIT

SIMKIT [Harmon and King 1985], produced by IntelliCorp, is written on top of KEE (Knowledge Engineering Environment) which in turn is written in LISP. "Frames representing simulation objects are developed and then used to build a simulation"[O'Keefe and Roach 1987]. SIMKIT inherits the object-oriented programming facilities of KEE [Nielsen 1986]. SIMKIT has a graphics editor through which the modeler specifies the model frame structure. The graphics editor enables the manipulation of icons to accomplish this. SIMKIT allows the construction of a template library that can be used to assemble simulation model objects in a particular problem domain. The graphics editor can be used to retrieve and store objects in this library. The graphics editor also lets the modeler view model objects at different levels of abstraction down to some base level. In addition, users are able to specify relationships and movement by pointing at the *from* object and then to the *to* object on a graphical depiction of the system to be simulated. Once the model structure is complete, SIMKIT automatically converts the graphical representation "into the internal form of knowledge representation required for simulating the model's dynamic behavior" [Nielsen 1986]. SIMKIT implementation is event-oriented (with a central event calendar). The graphical facility promotes rapid model development and easier model modification. Similar to KBS, SIMKIT allows some generalized analysis capabilities when such restricted domain expertise is appropriately represented in the knowledge base.

2.2.2.4 SES

SES (Simulation Environment System) [Adelsberger et al. 1986] has several goals, to integrate: "functionality, ease of use, ease of model creation, dynamic run-time interaction, model extendibility, and

interactive interactions." SES, built by the simulation group at Texas A&M University, is a rule-based, object-oriented simulation environment. Objects are created via graphical input, using templates if desired. An intelligent assistant supports this process with natural language dialogue. Specification language input is possible, however. Objects are viewed as active or passive. Their graphical, static, and dynamic behavior can be defined. Active objects monitor the system and their actions are triggered when certain conditions are met. Passive objects respond to messages sent by other objects and initiate their actions accordingly. Goal-directed simulation provides support in experimentation and automatic modification of models. Graphics displays not only assist in model creation and modification but also provide visual display of simulation dynamics. A knowledge-based interpreter executes the model.

2.2.2.5 JADE

JADE [Unger et al. 1986a,b] is a supporting environment for the development of distributed system software and has been used to develop simulation models. JADE was developed at the University of Calgary. JADE is uniquely designed to support the development of distributed software in that its prototyping facility allows the developer to simulate the target systems [Unger et al. 1983]. The model of the target system is then used for debugging, testing, and performance prediction of the target system and developed software. JADE has three major focuses: (1) use of simulation for modeling the target system, (2) automation of the program development process by generating programs from specifications, and (3) improved human interface, using interactive graphics, animation tools, and speech input and output tools [Unger et al. 1983]. Prototyping of systems is possible through the use of a formal specification language, JDL (Jipc ["gypsy"] Description Language). A run-time verifier, written in PROLOG, translates JDL and enables run-time checking of the specification to ensure consistent and expected behavior.

JDL also serves as documentation and a means of communication between developers who may be working on different aspects of a project. Developed systems may be programmed in any of five different languages: ADA, C, LISP, PROLOG, or SIMULA via excellent interface facilities at the programming environment level. Automatic translation of the specification to other of the languages is apparently not

available. Programming development appears to be separate from the prototyping effort.

2.2.2.6 CASM

The CASM (Computer Aided Simulation Modelling) Project [Balmer 1987; Balmer and Paul 1986] is ongoing at the London School of Economics. Researchers hope to automate the simulation modeling process. Before CASM, the program generator approach [Mathewson 1984a,b, 1985] had been successfully used to produce programmed models from flowchart, ACD (Activity Cycle Diagram), or other symbolic notation which described the formulated problem. CASM goals, in comparison, are to automate the remaining manual and labor-intensive actions of problem formulation and output analysis. Artificial intelligence techniques are used to help formulate the problem via natural language dialogue with the user. Limited progress has been made in embedding analyst expertise in an intelligent output analyzer which determines errors in the model and changes the model as necessary.

Initial CASM implementations include a program generator (after Clementson's [1978] CAPS system) written on top of eLSE (extended Lancaster Simulation Environment), a comprehensive collection of PASCAL simulation routines. The latest ISPG (Interactive Simulation Program Generator) is AUTOSIM [Balmer and Paul 1990] which also accepts model specifications in terms of ACDs, like CAPS. CASM produces implementations which are based on the Three-Phase Approach. Problem formulation was first attempted using an expert system with interactive dialogue [Doukidis and Paul 1985] to produce an ACD representation of the problem and system to be modeled. The use of expert systems for problem formulation was found to be inappropriate. A programming-by-questionnaire approach had been taken. Although easier than manually forming the ACD, this way was slow, not modular, menu-driven, and required a great deal of abstraction on the part of the modeler [Doukidis and Paul 1986]. The development of a natural language understanding system (NLUS) is deemed more appropriate. Doukidis [1987] and Doukidis and Paul [1986] describe this aspect of the CASM research. SPIF (Simulation Problem Intelligent Formulator) is a product of this effort.

In general, an NLUS requires the use of a dictionary to perform the semantic and syntactic analysis.

This dictionary may be small at first but grows as the system "learns". SPIF is shown to be feasible for a wide domain of applicability. Initially constrained to accept only action sentences, SPIF now understands complex statements which use the attributes and conditions of the model objects. SPIF directs problem formulation and enables the client to participate with the modeler in creating the specification. SPIF does require a structure for entering the sentences (to assist in resolving semantic and syntactic ambiguity) but is not too restrictive. Since the client and modeler are assumed to be working together while using SPIF, the imposed structure is not a problem. Such a cooperative effort, though, represents a significant constraint.

Graphics-driven systems are the latest development in the series of ISPG prototypes. MacGraSE [Au 1990] uses the ACD concept; users define the system in entity life cycle terms which are graphically represented by an associated ACD. The ACD is produced in the Apple Macintosh iconic environment by assembling the icons on the screen in a descriptive fashion. MacGraSE generates a Pascal source code file [Balmer and Paul 1990].

More recently, more formal simulation model specification methods combining system-theoretic formalisms [Zeigler 1984], formal languages (Z and VDM), and diagrammatic techniques like ACDs and Petri nets have been explored [Balmer and Paul 1990].

2.2.2.7 KBMC

KBMC (Knowledge-Based Model Construction) System [Murray and Sheppard 1987, 1988] was developed to automate model construction. KBMC "utilizes a knowledge-based approach to automatic programming to build a simulation model and extends the knowledge-based approach to include model specification acquisition" [Murray and Sheppard 1987]. KBMC research explores three areas: (1) simulation modeling, (2) automatic programming, and (3) expert systems. KBMC is applied to a restricted problem domain, queueing systems. The target language of the system is SIMAN which is event- and process-oriented. Specification is performed via a structured, interactive dialogue with the user. Extraction rules using domain and modeling knowledge guide this process. The underlying principal component of

the specification process is the Internal Model Specification (IMS) which facilitates model specification. IMS generic data structures are formed and permit the specification of static and dynamic components of queueing systems. Automatic translation is conducted by goal and data-driven transformations "where the internal specification is transformed first into a basic model and then into SIMAN structures" [Murray and Sheppard 1987]. The knowledge bases for specification extraction and model construction are built using an expert system building tool.

2.2.2.8 Lehman's Conceptual Modeling Approach

Largely unimplemented, this research effort [Lehman et al. 1986] shows promise. The approach is characterized by dialogue-oriented specification. The dialogue uses alphanumerical communication and contains graphical and menu-driven support for the user. Once the initial specification has been created, an underlying expert system fits it to a conceptual modeling method which seems to be most appropriate for the specified problem and the goals of the user [Lehman et al. 1986]. A knowledge base containing the characterization of these conceptual modeling methods is the basis for the consultation. Such an approach widens the domain of applicability and allows the user to describe the initial specification in terms which he understands.

2.2.2.9 IMDE

IMDE (Integrated Model Development Environment) [Ozden 1991] is a research system for the graphical programming of simulation models in an object-oriented environment. The IMDE is part of the PRISM (Productivity Improvements in Simulation Modeling) at the Logistics and Human Factors Division of the Air Force Human Resources Laboratory (AFHRL/LRL) at Wright Patterson AFB. The IMDE is a prototype system for the AF logistics support systems. The methodology derived from the IMDE will be expanded and serve as the basis for a broader model development environment.

The IMDE, still under study, will contain an integrated toolset for model development support, to include model specification and verification techniques. The target programming language for the

development of the IMDE and its graphical programming methodology is the object-oriented, Smalltalk-80 language.

The IMDE graphical programming methodology has six defined criteria. The methodology intends to: (1) facilitate the easy use of the simulation environment, (2) do (1) in a straightforward manner, (3) increase modeler productivity, (4) minimize programming errors, (5) enhance easy visualization of the problem, and (6) serve as a good communication medium. A prototype graphical programming facility, GraphSIM (Graphical Simulation Modeling) has been developed. GraphSIM contains three components: a graphical programming editor, a model builder and interpreter for translating the graphical models into executable versions, and a view builder for creating the runtime displays. The view builder uses the Activity Cycle Diagram as the graphical representation of execution. GraphSIM is currently able to handle simple resource utilization, production, and activity synchronization among model objects. Future research will extend expressiveness and enrich the modeling capability for tackling more complex modeling problems, to include concurrent activities and the notion of intelligent simulation objects that can make independent decisions. In addition, further study will be directed toward improving computational efficiency within the object-oriented environment using RISC and parallel processing technologies.

2.2.2.10 The IMSE Project

The primary goal of the IMSE (Integrated Modeling Support Environment) [Hillston et al. 1990] is to bring integrate a set of tools, exploit current workstation technology, and assist performance modelers in all stages of the performance modeling aspect of performance evaluation. This would include the actual model construction and workload generation stages as well as experimentation and final report generation.

The architecture of the IMSE contains a WorkBench (the graphical interface to the environment), an OMS (Object Management System) which holds all system objects and the links between them, and an integrated assortment of environmental tools. Models are built using three graphical editors, each based upon a modeling paradigm: stochastic Petri networks, queueing networks, and the process-oriented view. The diagrammatic form of the model is automatically converted into a textual form. These graphical

editors allow the graphical forms to be composed hierarchically. The modular construction of models allows existing submodels to be combined to produce models of more complex systems. Tools are also included in the environment for workload analysis and the static modeling of systems.

Three particular tools are worthy of separate consideration. The Experimenter tool enables naive users to develop experimental plans (solutions address stated objectives to produce integrated reports) [Zeigler 1976] in terms of the workloads and results of interest, independently of the model itself. The Experimenter's model independent interface is a valuable feature; yet, much of the work on this tool is preliminary. An Animator tool visualizes the execution. A Reporter tool creates and collates results and reports.

2.2.3 The Simulation Model Development Environment

Much of the following description of the SMDE research project at VPI&SU is taken from the overview of Balci et al. [1990]. The SMDE can be characterized as a simulation support environment or a computer-aided simulation engineering environment. The SMDE project has addressed a complex research problem: prototyping a domain-independent discrete-event simulation support environment to provide a comprehensive, integrated collection of computer-based tools to:

(1) offer cost-effective, integrated and automated support of model development throughout the entire model life cycle;

(2) improve the model quality by assisting in the quality assurance of the model;

(3) significantly increase the efficiency and productivity of the project team; and

(4) substantially decrease the model development time.

Guided by the fundamental requirements identified by Balci [1986b], prototyping techniques have been used to develop the prototypes of SMDE tools. The Object-Oriented Paradigm enunciated by the Conical Methodology [Nance 1987] has furnished the underpinnings of the SMDE research prototype [Balci and Nance 1987b].

2.2.3.1 SMDE Architecture

Figure 2.1 depicts the architecture of the SMDE in four layers: (0) Hardware and Operating System, (1) Kernel SMDE, (2) Minimal SMDE, and (3) SMDEs.

The hardware constituting *Layer 0* is composed of a Sun 3/160 computer workstation running under an MC68020 CPU with 8 megabytes of main memory, 380 megabytes of disk subsystem, a 1/4-inch cartridge tape drive, and a 19-inch color monitor with 1152-by-900 pixel resolution. A laser printer and a line printer accessible via an Ethernet local area network produce high quality documents and hard copies of Sun screens and files. The software environment at layer 0 contains principally, the UNIX SunOS 4.0 operating system and utilities, SunView (a graphical human-computer user interface) [Sun Microsystems 1988], SunCore and SunCGI (device-independent graphics library and computer graphics interface), the Sun programming environment, and the INGRES (SunINGRES) relational database management system [Sun Microsystems 1986a].

The kernel, *Layer 1*, integrates all SMDE tools into the software environment described above. SunINGRES databases, communication and run-time support functions, and a kernel interface are provided. Three SunINGRES databases (labeled project, premodels, and assistance) are at this level. Each are administered by a corresponding manager in Layer 2. All SMDE tools are required to communicate through the kernel interface. Direct communication between two tools is prevented to make the SMDE easy to maintain and expand. The kernel interface provides a standard communication protocol and a uniform set of interface definitions. Security protection is imposed by the kernel interface to prevent any unauthorized use of tools or data.

Layer 2 provides a "comprehensive" set of tools which are "minimal" for the development and execution of a model. "Comprehensive" implies that the toolset is supportive of all model development phases. "Minimal" implies that the toolset is basic and general. It is basic in the sense that this set of tools enables modelers to work within the bounds of the minimal SMDE without significant inconvenience. It is general in the sense that the toolset is generically applicable to various simulation modeling tasks. There are two categories of minimal SMDE tools: (1) tools specific to simulation modeling (Project

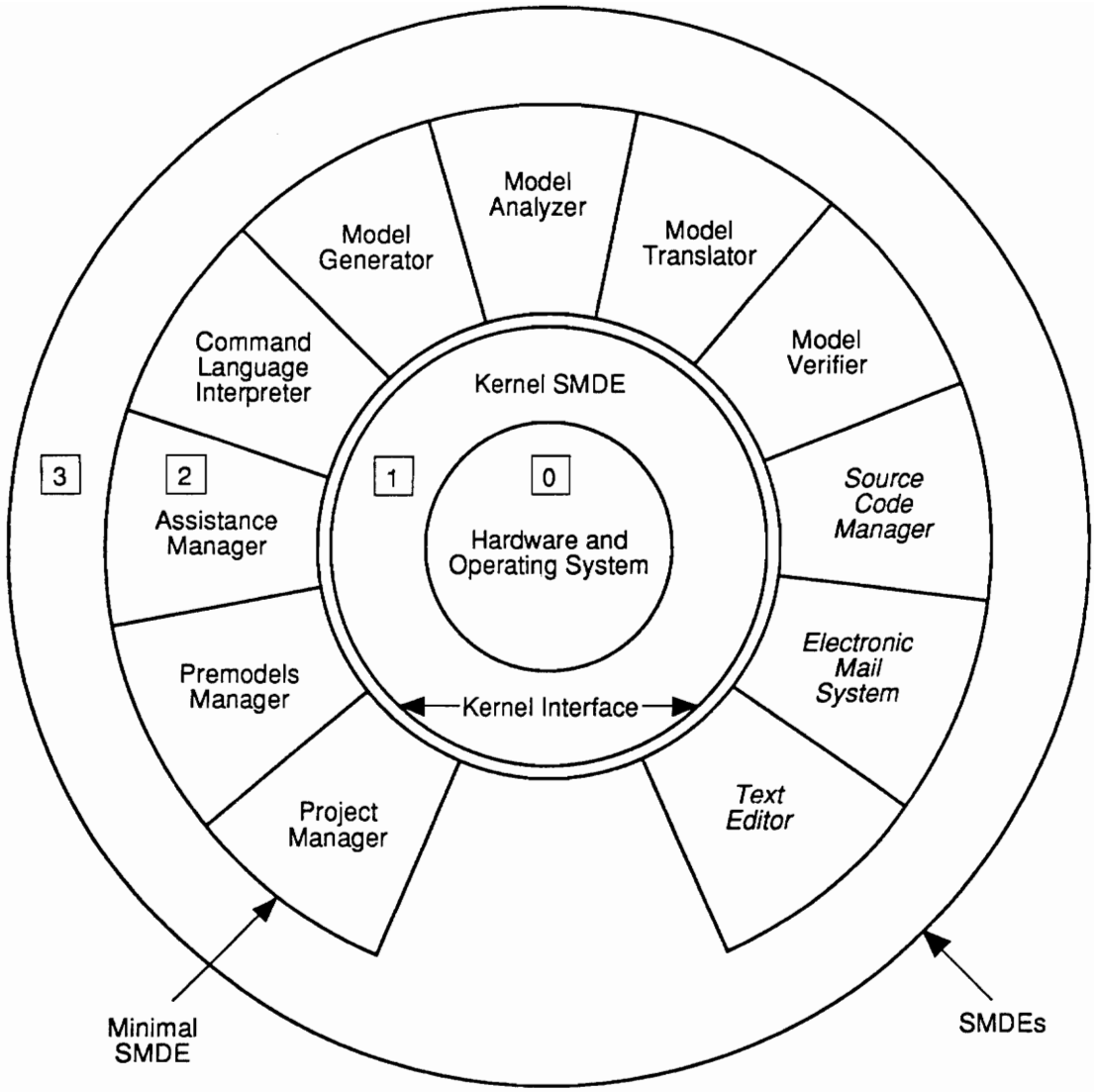

Figure 2.1 Architecture of the SMDE

Manager, Premodels Manager, Assistance Manager, Command Language Interpreter, Model Generator, Model Analyzer, Model Translator, and Model Verifier) and assumed/library tools (at layer 0 - Source Code Manager, Electronic Mail system, and Text Editor). Current prototypes of the minimal SMDE tools are described in Section 2.2.3.2 below.

The highest layer of the environment, *Layer 3*, expands on a defined minimal SMDE. The SMDEs incorporate tools that support specific applications and are needed either within a particular project or by an individual modeler. If no other tools were added to a minimal SMDE toolset, a minimal SMDE would be a SMDE. The tools at this layer are in two categories. There are tools specific to a particular area of application (requiring further customizing for a specific project or needed to meet specific project requirements) or assumed/library tools (needed for statistical analysis of simulation output data, for designing simulation experiments, for animation, for input data modeling, etc.). SMDE tools at layer 3 are integrated with other SMDE tools and with the software environment of layer 0 through the kernel interface. The provision for this integration is indicated in Figure 2.1 by the opening between Project Manager and Text Editor.

2.2.3.2 Description of Current Prototypes of Minimal SMDE Tools

Project Manager is a software tool which: (1) administers the storage and retrieval of items in the project database; (2) keeps a recorded history of the progress of the simulation modeling project; (3) triggers messages and reminders (especially about due dates); and (4) responds to queries in a prescribed form concerning project status. Other than the preliminary design decisions, little work has been done on the Project Manager. Its development is dependent on tougher design decisions taken for the other minimal SMDE tools; therefore, high-level design is being delayed until sufficient progress is achieved on the other tools.

Premodels Manager is a software tool intended to facilitate the reusability of earlier developed models or model components. Using this tool, a modeler searches the premodels database to identify earlier developed model components for reuse in the development of a new simulation model. An early prototype

has been developed by Box [1984]. An advanced prototype of the Premodels Manager software tool [Beams 1991; Beams and Balci 1992] uses the SunView graphical user interface software. Storage and retrieval of Premodels data are achieved via the INGRES relational database management system. Components of completed simulation studies are locatable and reusable.

The **Assistance Manager** software tool has been prototyped [Frankel 1987; Frankel and Balci 1989] to provide: (1) information on how to use an SMDE tool; (2) a glossary of technical terms; (3) introductory information about the SMDE; and (4) assistance for tool developers in supplying "help" information.

Command Language Interpreter (CLI) is the language through which a user invokes an SMDE tool. An early prototype, based on the design by Moose [1983], is described in [Humphrey 1985]. The SunView graphical user interface now serves the CLI function.

Model Generator (the simulation model specification and documentation generator) is a software tool which assists the modeler in: (1) creating a model specification in a predetermined form which lends itself to formal analysis; (2) creating multi-level (stratified) model documentation, and (3) performing model qualification. Three research prototypes of the Model Generator based on the Conical Methodology have been developed. The first [Hansen 1984] implements the definition stage of the Conical Methodology [Nance 1987]. The second builds on the first by adding the specification stage (using the Condition Specification) of the Conical Methodology [Barger 1986; Barger and Nance 1986; Overstreet and Nance 1985; Overstreet et al. 1986]. The third [Page and Nance 1989; Page 1990] takes advantage of an improved user interface with windows and mouse-driven input (of the SunView graphical user interface) and builds on the work done by Barger. Furthermore, it improves the modeler's developmental context with hierarchical displays and maintains strong adherence to the Conical Methodology concepts of definition and specification. Improved prototypes of the Model Generator are needed since it is the most crucial tool of the SMDE. Chapter 4 discusses new developments with the Model Generator. Achieving the automation-based software paradigm [Balci and Nance 1987a] in the simulation modeling domain relies heavily on improved understanding of contributions by conceptual frameworks to knowledge extraction and

model representation.

The **Model Analyzer** diagnoses the model specification created by the Model Generator and effectively assists the modeler in communicative model verification. The first prototype is described by Moose and Nance [1987]. Another prototype version implements a control and transformation metric for measuring model complexity [Wallace 1985; Wallace and Nance 1985], and provides diagnostic assistance using digraph representations of simulation model specifications [Nance and Overstreet 1987; Overstreet and Nance 1983, 1986]. Thus far, the Condition Specification has been the only specification form subjected to formal analysis. The current prototype [Puthoff 1991] provides parsing of the Condition Specification and storage of parse data in a relational database. Graphical presentations of the diagnosis are provided. Chapter 4 presents a description of another prototype of this tool which performs diagnosis upon the relational database representation created under the DOMINO, the new conceptual framework of the thesis research.

Model Translator translates the model specification into an executable representation after the quality of the specification is assured by the Model Analyzer. Chapter 4 describes a working prototype of the Model Translator.

Model Verifier is intended for programmed model verification [Whitner and Balci 1989]. Applied to the executable representations, it provides assistance in substantiating that the simulation model is programmed from its specification with sufficient accuracy. Extensive groundwork [Balci 1987, 1990; Whitner and Balci 1989] has been conducted to prototype the Model Verifier, but development is delayed until a standard executable model representation is adopted for the SMDE. A prototype of this tool using the DOMINO is presented in Chapter 4.

Source Code Manager is a software tool which configures the run-time system for execution of the programmed model, providing the requisite input and output devices, files, and utilities. Its development is being delayed until a standard executable model representation is adopted for the SMDE.

Electronic Mail System facilitates the necessary communication among project personnel. Primarily, it performs the task of sending and receiving mail through computer networks. The Sun workstation's

MailTool is currently used as the Electronic Mail System of the SMDE. The Sun computer workstation is a node on the Internet computer network with the node name of *mdesun.cs.vt.edu*.

Text Editor is used for preparing technical reports, user manuals, system documentation, correspondence, and personal documents. Currently, the vi editor serves as the text editor for the SMDE.

2.3 Visualization in Simulation

The visualization of simulation model dynamics is becoming a popular adjunct to the use of simulation for the purpose of problem solving. Hardware developments related to graphics display and processing are making visualization today's reality rather than yesterday's dream. In this section, the background of this important field is given. Critical terms are defined. Methods of representation, display, and interaction are highlighted. The advantages and disadvantages of visualization are covered, with special emphasis on areas of needed research. For completeness, recent systems applying visual techniques for simulation are noted.

2.3.1 Background

R.D. Hurrion conducted the first significant work in Visual Simulation (VS) at the University of Warwick in England [Bell and O'Keefe 1987]. While simulating a job shop manufacturing system, a visual display (letters representing entities) was created so that a human scheduler could determine the current state of the model and, as a result, make realistic scheduling decisions. Although the idea of visualization and interaction was not new (there were, indeed, earlier graphical facilities for simulation [Hurrion 1986]), Bishop and Balci [1989] note that the term Visual Interactive Simulation (VIS) was introduced by Hurrion. The groundwork had been established for further research.

2.3.2 Terminology

The following group of terms represent the critical terminology: animation, visual simulation, visual interactive simulation, visual modeling, and visual interactive modeling. Each term is defined in turn.

Animation refers to graphic art which occurs over time and which imparts information to the viewer via image changes [Baeker 1974]. This obviously includes many types of graphics displays which are clearly outside the scope of simulation. There is some wide variance in this definition among simulationists [Bishop and Balci 1989; O'Keefe 1987; Standridge 1986]. Two important typings are given by Standridge regarding animation: *simulation-concurrent animation* (dynamic displays produced by model state) and *post-simulation animation* (those driven by simulation trace).

Visual Simulation (VS) is the dynamic display of a modeled system where the display covers some or all of the following: model input, internal behavior, output, and experimentation data. Visual Simulation which includes capabilities for user interaction with the running model is *Visual Interactive Simulation* (VIS). Here, model parameters may be changed or inputs made which modify model state and alter model behavior during continued execution. Hurrion [1989] notes that the technique of VIS consists of: (1) developing the simulation model of the system of interest, (2) incorporating a method for animating the model, and (3) interacting with the model (following observation) to explore alternative decision strategies. *Visual Modeling* (VM) and *Visual Interactive Modeling* (VIM) seem to be used interchangeably with VS and VIS, respectively [Bishop and Balci 1989].

2.3.3 Graphical Symbology and Display Types

Paul [1989] classifies the graphical symbology into three types: keyboard character, iconic image, and three dimensional rendering. The first is simplest in form and implementation but lacks versatility. The second provides a suitable likeness of objects being modeled but is more expensive to implement. Model quality is enhanced. The third brings significant realism to the presentation at high computational cost.

Hurrion [1986] classifies two major types of display. The *schematic display* mimics "the operational characteristics of the system under study." This is useful for conveying physical or logical dynamics. The other principal display type is the *logical display* for the representation of summary measures of performance in the form of bar charts, histograms, etc.

2.3.4 *Interaction with the Model*

Model interaction can be *model-prompted* or *user-prompted*. Such interactions may effect changes to entities and/or their attributes, priorities, or model algorithms [Hurrion and Secker 1978]. Interactions may be handled by incorporating display generating code into the model itself (*embedded programming*), with a library of *standard interactions* to speed development time, or by *stopping interpretation* [O'Keefe 1987]. This last method requires the model to be interpreted; once execution is interrupted, the model code may be altered.

Hurrion makes the observation that simulationists in North America tend to use a passive graphics approach with no interaction. The user only watches the simulation animation display. On the other hand, the UK approach is the interactive one which has been described. In a more recent development, expert interactive components (often based on PROLOG) are being added to visual simulation models, allowing the model to interact on its own in search of possible solutions. This represents *Intelligent Visual Interactive Simulation*, or IVIS [Hurrion 1989].

2.3.5 *Benefits and Drawbacks*

VS/VIS improves model presentation of results by providing a display which allows a client to readily grasp simulation results. Gaming capabilities can be more easily incorporated into complex system models. Furthermore, users of VS/VIS software experience enhanced learning capabilities while playing with the system [O'Keefe 1987]. In a general context, Nielson [1991] notes that the use of color, intensity, transparency, texture, and other visualization techniques can convey large amounts of information in a short period of time, if the display is properly prepared. Thus, visualization enables modelers to review large quantities of data with greater efficiency and comprehension.

Model verification, validation, and testing are also enhanced. The visualization of model execution offers valuable assistance to the modeler [Bishop and Balci 1989]. A modeler may place too much confidence in the model, however, just because the display appears correct [Paul 1989].

Building VS/VIS models incurs increased costs which may not offset the above benefits. Hardware

requirements can be limiting. The additional complexity derived in the creation of the dynamic displays can be difficult depending upon model characteristics. Corey and Clymer [1991] offer assistance in the form of a set of routines for specifying object movements and interactions. These are only however, for low-level use at the programming level. Most of the VS/VIS systems of the next section attempt to harness the complexity by providing system facilities which support display creation.

The model can be made more detailed than necessary in order to support the graphics display. Human factors issues (like color selection, etc.) must be carefully considered. Two important facts are: (1) good screen design methodologies for simulation displays are lacking [Bishop and Balci 1989], and (2) VS/VIS displays are application dependent. This is a disadvantage impacting the general and more widespread use of VS/VIS.

2.3.6 VS/VIS Systems

VS/VIS systems can be placed in three categories [Vujosevic 1990]. Each category contains many representative examples. A few of the more recent are given here. Mathewson [1989] provides an excellent overview of many of these systems.

The first group are those general purpose systems that require the modeler to have knowledge of the target programming or simulation language. CINEMA (Siman) [Poorte and Davis 1989], SIMGRAPHICS (Simscript II.5) [Bryan 1989], and SEE WHY (Fortran) [Gilman and Billingham 1989] are examples.

Systems which provide graphical capabilities for model development but in which the animation layout is developed separately form the second group. Most often, some graphical representation scheme is used. TESS [Standridge 1986; Grant and Starks 1988], INSIGHT [Roberts and Flanigan 1989], and WITNESS [Gilman and Billingham 1989] are in this category of systems which can be general purpose or special purpose.

The last group of systems for VS/VIS are those which are special purpose and contain graphical capabilities (icon-based) for both model and animation layout creation. XCELL+ [Conway and Maxwell 1986], SIMFACTORY II.5 [Rohrbough 1989] and SIMFLEX [Vujosevic 1990] are representative examples

of this final group. These are oriented toward FMS (Flexible Manufacturing Systems) and are domain-dependent. Another example, RESQME [Kurose et al. 1986; Gordon et al. 1990], is a graphical workstation environment for iteratively constructing, running and analyzing hierarchical models of resource contention systems.

CHAPTER 3
THE CONCEPTUAL FRAMEWORK

Advances in computer chip and display technology are constantly providing fertile ground for the growth of new and powerful software applications. Processing speed, display resolution, or imaging capabilities are rarely limitations. A recent development which has revolutionized document processing has been the advent of desk top publishing and the **WYSIWYG** (What You See Is What You Get) display. This display allows the user to view and manipulate a replica of the actual document representation. Text formats and character fonts are displayed on the computer screen in the same likeness as they appear in the final hardcopy. Graphics are easily included within the text for on-screen viewing. In this chapter, we extrapolate this concept for the design and creation of high quality, professional documents to a similar capability in a new conceptual framework for the design and implementation of simulation models.

3.1 Motivation

Non-trivial problems combined with the presence of technological advancements in software and hardware are inducing modelers to build models of greater and greater complexity. This skyrocketing complexity dictates the need for a conceptual framework which will guide the modeler, harness the complexity, and make the tasks of model design and model implementation manageable. Chapter 1 highlights ten developmental objectives for a next-generation conceptual framework. Each objective strikes to the heart of the complexity issue. Achieving substantive progress toward the satisfaction of these objectives produces a framework which, indeed, is versatile and multifaceted. As stated in the overall goal for a new conceptual framework (Section 1.2.1), such a framework then meets many of the diverse needs of the simulation life cycle and model development environment support.

An important, if not pervasive, influence during the development of the new conceptual framework has been the desire for an approach which allows the modeler to represent a model and its components exactly as they are conceptually or naturally perceived. Like **WYSIWYG**, this could be described as a **WYSIWYR** (What You See Is What You Represent) capability. The modeler needn't have to contort his

own view of the model in order to fit the requirements of the conceptual framework under which he is guided. For example, consider a machine repairman problem where a repairman services dozens of randomly failing machines in a first-come first-served (FCFS) order. In modeling this problem under the GPSS conceptual framework (Transaction Flow), the repairman would be defined as a facility since he provides service upon demand. Then machines would be represented as transactions that would queue up in front of the **"SEIZE RMAN"** block where repairman (**RMAN**) would be captured whenever available in a FCFS order. Although this produces a working and efficient model, the representation of the components is not conceptually true to the real system. Machines are made to dynamically move throughout the model as transactions when, in actuality, they are very attached to the production floor. Hence, the modeler is coerced to twist what he sees in the system and specify a model that is conceptually very different than the actual system operation. This significantly increases the model complexity and potentially causes errors in model representation especially for large and complex systems. The influence of the **WYSIWYR** approach is clearly reflected in the description of the conceptual framework which follows and is particularly evident in the chosen terminology.

Considering the goal for developing a multifaceted conceptual framework for the design and implementation of visual simulation models, the name chosen for the new conceptual framework is the **DOMINO**, derived from:

--> multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling <--

Multifaceted is defined as *having many aspects or phases, possessing many talents* [Webster's 1989]. Versatility is implied. As the proceeding presentation demonstrates, the DOMINO fits the label.

The DOMINO is presented from three perspectives: (1) model composition and general structure, (2) features of the framework, in particular its object orientation, graphical orientation, and multiview specification of model component logic (in the context of model design and implementation), and (3) an historical perspective. The first two perspectives are presented in detail. Each discussion provides a description and explanation of the terminology and the application of framework principles in their current form. Rationale for design decisions is included when appropriate. The historical perspective, the last

section of the chapter, traces the DOMINO through its evolutionary stages of development.

3.2 Model Composition and General Structure

A DOMINO *model* of a system is comprised of model components (submodels, static objects, dynamic objects, subdynamic objects, and base dynamic objects) and the interactions among these components. In keeping with the WYSIWYR philosophy, model components are "naturally" classified as real or virtual, and dynamic or static.

> *Real* components have a direct correspondence to a component in the system being modeled. Real components are visualized.
>
> *Virtual* components of the model, on the other hand, do not have a direct correspondence in the system and are not visualized. Virtual components are typically linked to model features not having a "real" representation such as components for statistics collection, random variate generation, model startup, etc.
>
> *Dynamic model components* are movable. This movement is categorized in Section 3.2.3. Dynamic objects, subdynamic objects, and base dynamic objects (all explained later) are dynamic model components.
>
> *Static model components* are physically at rest (if real) and immovable (whether real or virtual). Furthermore, these components are permanently within the model, staying within the model boundaries for the duration of model execution. Submodels and static objects (also explained later) are static model components.

Model components (much like model "building blocks") are defined, specified, and joined or related in various configurations to form the model static structure or a model dynamic structure.

> The *model static structure* is the architecture of the model formed from its static components.
>
> A *model dynamic structure* is the architecture of a dynamic component.

A model has only one model static structure while there can be many model dynamic structures depending on the number of dynamic components. Static structure and dynamic structure are more fully explained in the succeeding sections. In addition, the five types of model components (submodel, static object, dynamic object, subdynamic object, and base dynamic object) are defined and classified. Their use is described. Examples (e.g., of the various model components) are given for clarification at the end of each discussion.

3.2.1 Static Structure

We carefully distinguish between the model static structure and a simple static structure. As previously mentioned, there is one model static structure with the model itself as the root. Submodels and static objects are the primary components of the model static structure. The model can be decomposed into zero or more submodels and zero or more static objects. These member submodels can also be further decomposed. The member static objects CANNOT be decomposed. The model static structure is therefore a hierarchy of potentially many simple static structures. Informally:

> A *simple static structure* is a component hierarchy of submodels and static objects having a submodel as its root. The hierarchy is extended at points where interior submodels are decomposed.

More formally, modifying the definition of a tree [Knuth 1973]:

> A *simple static structure* is a finite decomposition set *SS* of one or more model component nodes such that (1) there is one specially designated submodel node which is the root of the simple static structure; and (2) the remaining component nodes in the decomposition set (excluding the root) are partitioned into $m \geq 0$ disjoint sets $SS_1,\ldots,SS_m$, (each of these sets in turn is a simple static structure) and $n \geq 0$ static object nodes, $so_1,\ldots,so_n$.

The difference between the model static structure and a simple static structure is that the model is the root in the first case. A submodel is the root of a simple static structure. Figure 3.1 displays a simple static structure. Following the formal definition, the submodel at the root can be decomposed into 0 or more submodels and 0 or more static objects. Subsequently, if m is not zero, then the *mth* submodel can be decomposed similarly, where $j \geq 0$ and $k \geq 0$. Figure 3.2 gives a practical example, a model static structure, taken from the Branch Operations application in Chapter 7.

The decomposable/non-decomposable character of submodels and static objects gives a hint to their respective definitions:

> A *submodel* is the root of a simple static structure with children of zero or more submodels and zero or more static objects.
>
> The *static object* is the most basic model component of interest in a simple static structure and, as such, cannot be decomposed.

The choice to represent a system component which is static as a submodel or static object is based primarily on the expected need for a decomposition point in the model hierarchy. The greatest flexibility in

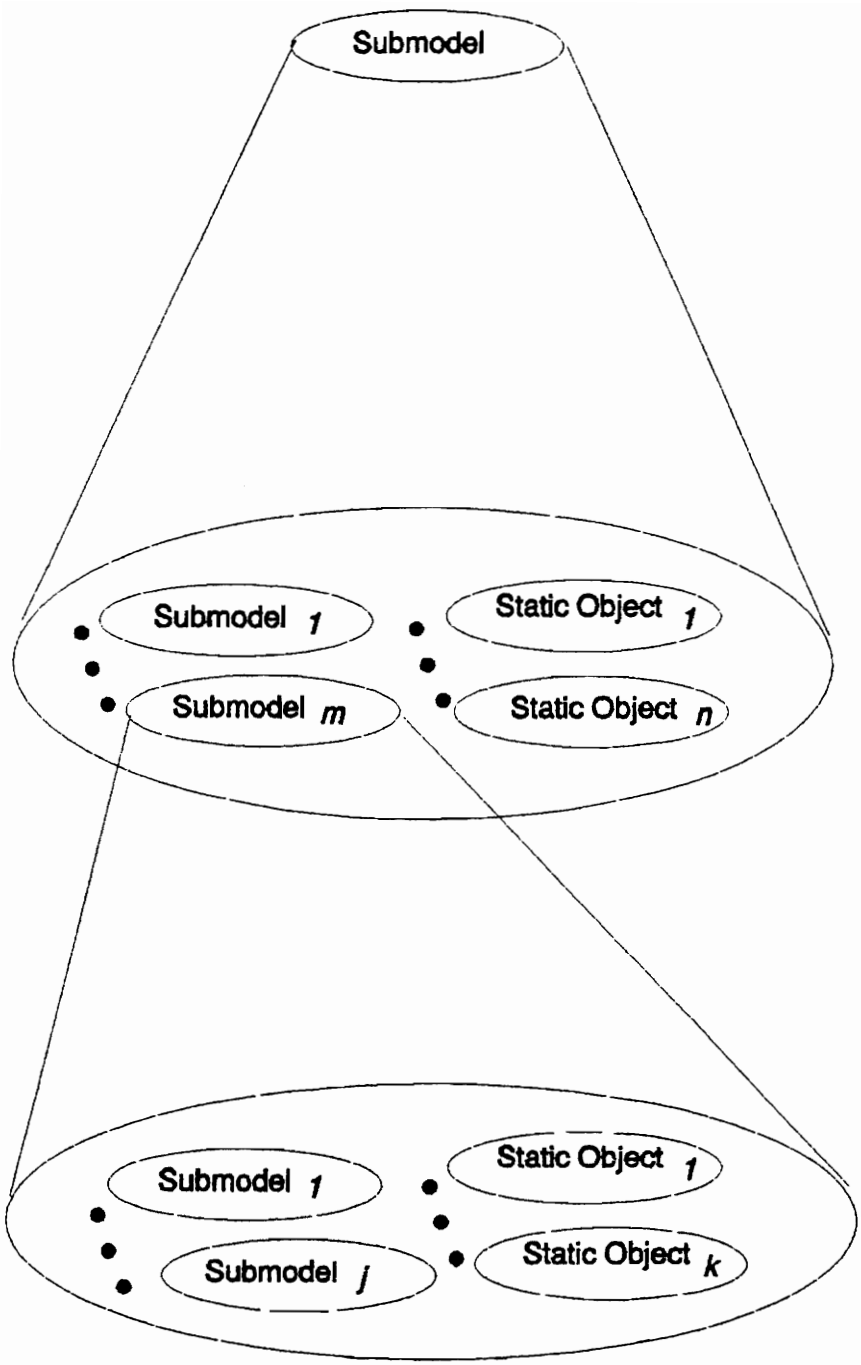

Figure 3.1 Simple Static Structure

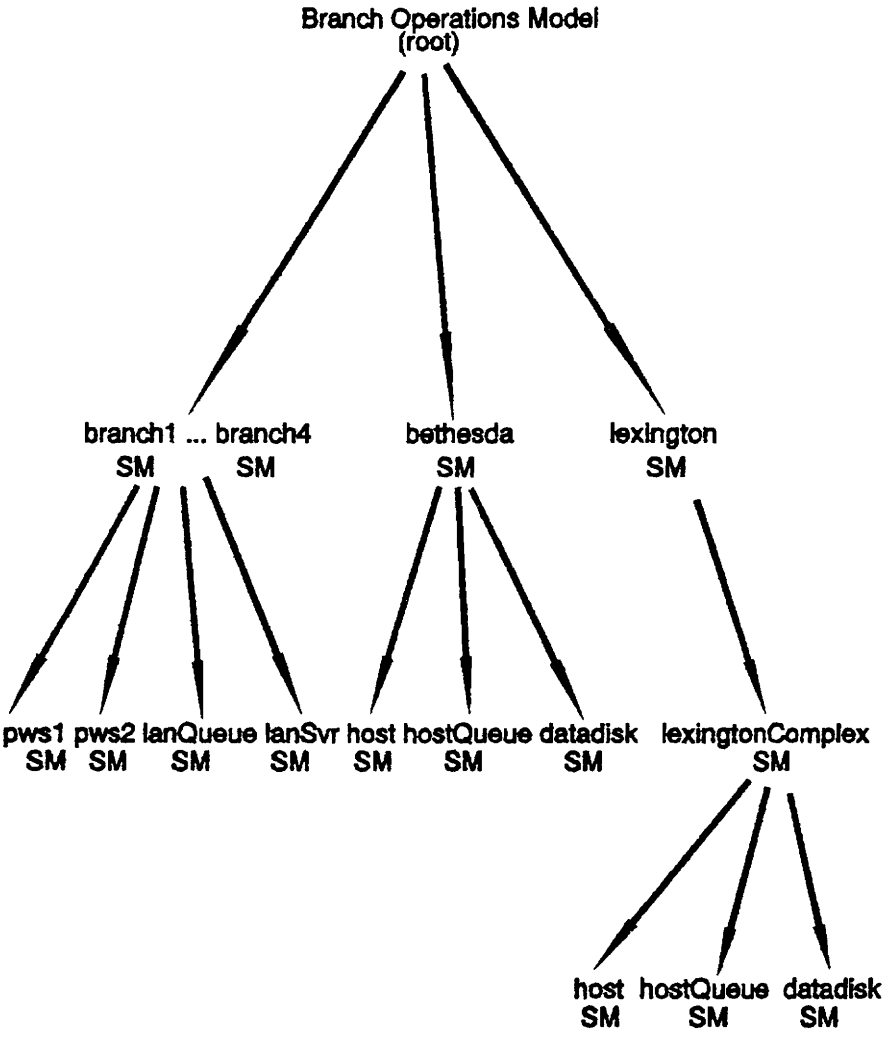

Figure 3.2 Model Static Structure of the Branch Operations Model

development is retained by modeling these system components as submodels. However, other considerations (see Section 3.3.2) related to visualization/animation requirements could dictate otherwise. Decomposition, such as that associated with submodels, enables a modeler to break a large modeling problem into smaller, more manageable parts. This enriches the framework and produces other important implications for visualization (also discussed in Section 3.3.2).

The following are examples of real and virtual static model components:

Real submodels: a cpu, a waiting line, a traffic lane, a terminal, a service area, a combat zone, an electrical circuit, etc.

Virtual submodels: a random number generator, a gamma random variate generator, a statistics collection routine, a printing routine, a graphical display routine, and experimental design routine, etc.

Real static objects: arithmetic unit of a cpu, a machine, a block of a traffic intersection, a traffic light, a stop sign, etc.

Virtual static objects: a compound logical condition, a data structure, abstract data types, program modules, etc.

Most often, submodels represent objects within the model (e.g., a robot on the production floor of an automobile assembly plant). In fact, all model components usually represent system objects. The definition and specification of model components assume an object orientation and follows the Object Oriented Paradigm (see Section 3.3.1). However, submodels may be regions or spaces within the model (e.g., an operating area at sea for a Naval task force, where each quadrant in the operating area is a submodel). In this case, a submodel is an "abstract object" taking Booch's [1986] viewpoint.

3.2.2 Dynamic Structure

Again, we distinguish between a model dynamic structure and a simple dynamic structure. Dynamic objects, subdynamic objects, and base dynamic objects are the dynamic model components which are the basis for a model's dynamic structure(s). At the root of every model dynamic structure is a dynamic object. Dynamic objects can be decomposed into zero or more subdynamic objects and zero or more base dynamic objects. Like submodels, the member subdynamic objects are decomposable. And like static objects, base dynamic objects are NOT decomposable. A model dynamic structure is, therefore, a

component hierarchy of possibly many simple dynamic structures.

> A *simple dynamic structure* is a component hierarchy of subdynamic objects and base dynamic objects having a dynamic object as its root. The hierarchy is extended at points where interior subdynamic objects are decomposed.

More formally, using the same conventions as the simple static structure:

> A *simple dynamic structure* is a finite decomposition set *DS* of one or more model component nodes such that (1) there is one specially designated subdynamic object node which is the root of the simple dynamic structure; and (2) the remaining component nodes in the decomposition set (excluding the root) are partitioned into $i \geq 0$ disjoint sets $DS_1,\ldots,DS_i$, (each of these sets in turn is a simple dynamic structure) and $j \geq 0$ base dynamic object nodes, $bdo_1,\ldots,bdo_j$.

The distinguishing difference between a model dynamic structure and a simple dynamic structure is that a dynamic object stands at the root of a model dynamic structure. In contrast, simple dynamic structures have subdynamic objects at their root. Figure 3.3 displays a simple dynamic structure and Figure 3.4 gives a model dynamic structure of the city bus from the Bus Route example model application in Chapter 7. The Bus Route model static structure is included in Figure 3.4.

The dynamic model components are now defined; notations concerning their decomposable/non-decomposable nature are included.

> The *dynamic object* is the dynamic model component which is the basis (root) for model dynamic structures and is therefore decomposable.
>
> The *subdynamic object* is the root of a simple dynamic structure and can be decomposed into zero or more subdynamic objects and zero or more base dynamic objects.
>
> The *base dynamic object* is the most basic model component of interest in a simple dynamic structure and cannot be decomposed.

Dynamic objects (or model dynamic structures) reside in the model, most often temporarily but possibly permanently. If temporarily within the model, they have been created during model execution. At some instant during model execution, they can exit the model boundaries and be subsequently destroyed. Permanent dynamic objects exist in the model throughout execution, from simulated time zero and onward.

Noteworthy relationships exist (1) between the model static structure and the model dynamic structures and (2) between the components of the model static structure (submodels, static objects) and those of a model dynamic structure (dynamic objects, subdynamic objects, base dynamic objects). Relative to the

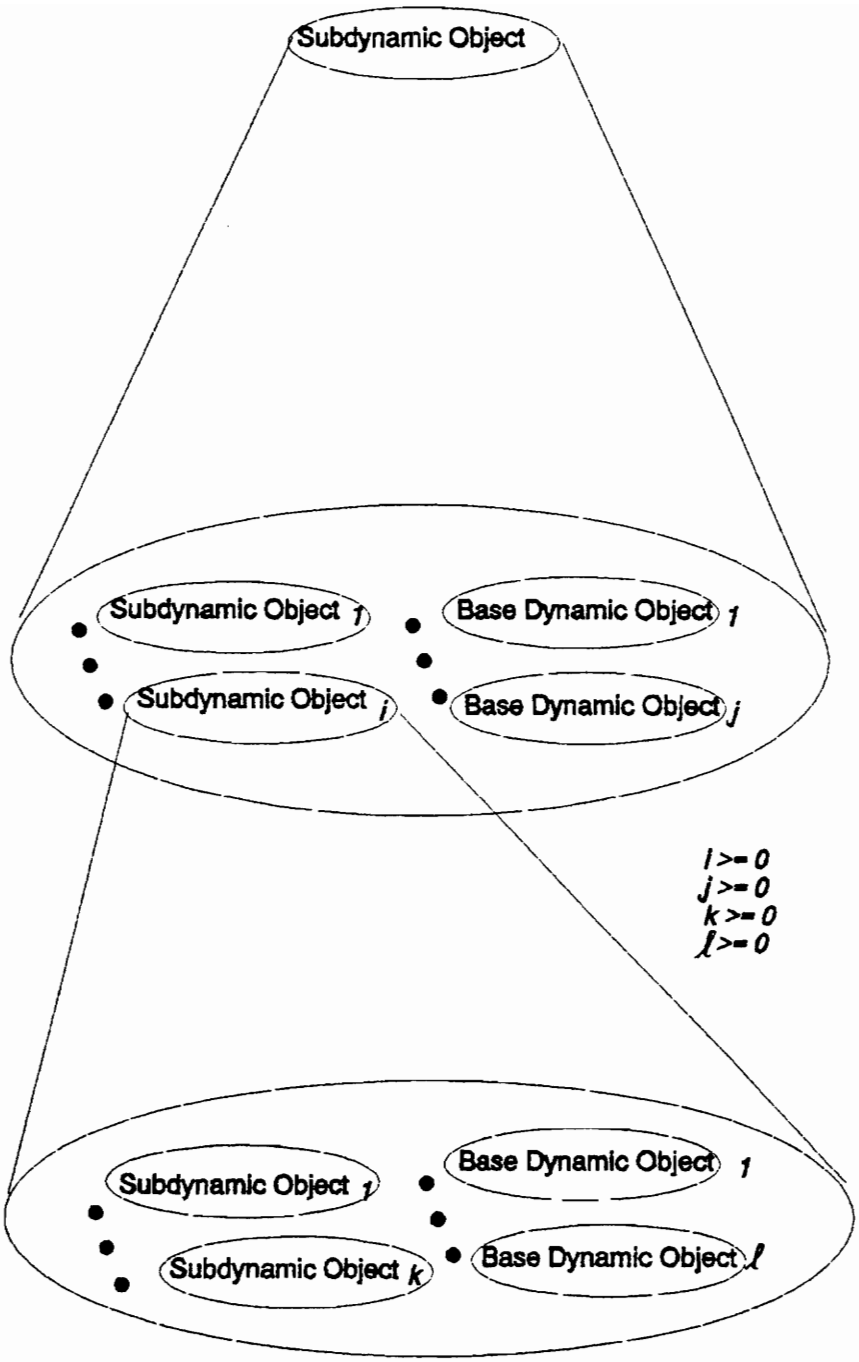

Figure 3.3 Simple Dynamic Structure

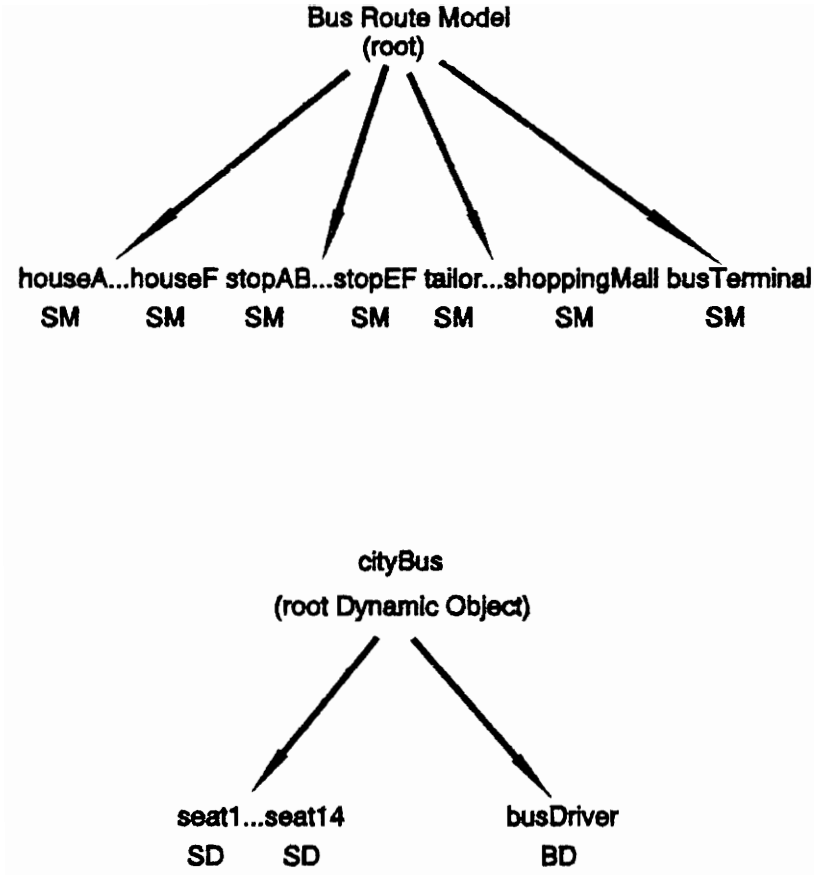

Figure 3.4 Model Static and Dynamic Structures of the Bus Route Model

model itself, subdynamic objects and base dynamic objects are dynamic since they are component members of a model dynamic structure. However, to their parent (the root dynamic object of their model dynamic structure), they are relatively static. In a sense, each model dynamic structure has a structural kinship to the model static structure. Note that the VSSE (Chapter 4) enforces the top-down definition of these component hierarchies (both static and dynamic). The component hierarchies of a model's static structure and its dynamic structure(s) are separate and distinct from the class inheritance hierarchies originating from the object-orientation of the DOMINO (Section 3.3.1).

The subdynamic object serves the same functions and has the same form in a model dynamic structure as the submodel in the model static structure. Submodels own (as the root) a simple static structure and are, therefore, decomposition points. Each subdynamic object owns a simple dynamic structure and is decomposable. Subdynamic objects, like submodels, can represent objects within the system being modeled or they can represent regions or spaces. Similarly, base dynamic objects are non-decomposable cousins to their static object counterparts. The decision to represent a system component as a subdynamic object or a base dynamic object concerns the same issues as the choice between submodels and static objects.

The following are examples of real and virtual dynamic model components:

Real dynamic objects: a car, an aircraft, a war ship, a missile, a bus, a bicycle, a travelling repairman, a transaction, etc.

Virtual dynamic objects: a virtual dynamic object having two attributes, one containing the current time and another holding the arriving unit's name, sent to the statistics collection virtual submodel for recording the arrival time; a virtual dynamic object sent to an output routine to activate printing; a virtual dynamic object used by the system to create the initial model components which are present at simulation time of zero, etc.

Real subdynamic objects: a flight deck of an aircraft carrier, a bus seat, a car engine, a baggage area onboard a plane, etc.

Virtual subdynamic objects: submessages of a decomposed virtual (dynamic object) message, which handle statistics collection based on various conditions; each submessage corresponds to a particular condition which can be further decomposed, perhaps in accordance with a range of values which the activating condition could assume (e.g., if the conditions were based on attribute values, then TRUE/FALSE or RED/GREEN/BLUE, etc.).

Real base dynamic objects: a bus driver, a cook on the mess decks of war ship, a loading crane on a passenger ship, an oven in the galley of a passenger ship, etc.

Virtual base dynamic objects: submessages of a decomposed virtual (dynamic object) message, which

handle statistics collection based on various conditions; each submessage corresponds to a particular condition which CANNOT be further decomposed, etc.

3.2.3 Movement of Model Components within Model Structures

Only dynamic model components undergo "movement." This section describes the various forms this movement can take. Movement can be spatial, temporal, or logical, depending on the situation.

> *Spatial* movement is reflected as physical movement of real dynamic components between model components during animation.

For example, a customer dynamic object can be moved into a grocery store submodel. During animation, the customer is visualized as moving into the store. Another example of spatial movement is cars moving through an intersection. A system, not normally considered a physical one, can be modeled and visualized with real components undergoing spatial moves. Consider these situations: human thought processes moving within the brain; the execution control point of a computer program moving through the various modules and subroutines of the program logic.

> *Temporal* movements are movements in time that can be accomplished by virtual or real dynamic components.

An example of a temporal move is the delay of a customer while being serviced by a clerk at a cash register. No movement in space occurs, but there is movement (passage) in time. Temporal moves (or delays) are "determined" or "contingent" [Nance 1981a].

> *Logical* movements are changes in the logic decision path of the dynamic component. Both real and virtual dynamic components can make logical movements.

For real dynamic objects, spatial moves are often associated with a logical move. If the logic specification for a dynamic object's actions is contained at the possible locations that the dynamic object could be (Section 3.3.3 explains more fully) then a move (spatially) to a different location is also a logical move. For virtual dynamic objects being moved to virtual submodels (e.g., a virtual dynamic object for retrieval of random variates being moved to a virtual random variate generator submodel), this is strictly a logical move. Due to its "virtualness", there is no movement in space.

Dynamic objects can move spatially, temporally, or logically. Base dynamic and subdynamic objects

can only move spatially as part of a decomposed dynamic object. The terms "movement" or "move", although possibly referring to a temporal move, generally indicate a logical or spatial move.

Dynamic objects move throughout the model's static and dynamic structures. Dynamic objects can move among the submodels and static objects of the model static structure. Movement up and down the model static hierarchy is via decomposed submodels. Within a submodel, a dynamic object can utilize the resource of a static object. Besides moving among submodels and static objects in the static structure, dynamic objects can also move into decomposed dynamic objects a (model dynamic structure) and among its member subdynamic objects and base dynamic objects. Movement up and down the model dynamic structure is via the decomposed subdynamic objects. Thus, dynamic object decomposition not only provides a means of managing model complexity, but it also simplifies the modeling of such things as a bus carrying passengers or an aircraft carrier which carries planes. This is in faithful keeping of the WYSIWYR philosophy.

Table 3.1 summarizes the terminology which has been presented concerning the composition and structure of DOMINO models.

3.3 Features for Model Design and Implementation

The DOMINO has evolved to have certain characteristics and guides a modeler to accomplish key steps while constructing a model. First, while taking an object-oriented design approach, a modeler defines and specifies the classes of model components as explained in Section 3.3.1, Object Orientation. Also, in order to animate the model, the images associated with the model components must be created and assembled in the manner described in Section 3.3.2, Graphical Orientation. This section shows how the modeler manipulates model components (objects) of the various classes to produce the model component hierarchies, e.g., static structure and dynamic structure(s). The modeler must also specify the dynamics and interactions among model components in accordance with the principles of Section 3.3.3, Multiview Specification of Model Component Logic. This section gives an overview of class logic specification under the framework and its specialized specification language. Flexibility is maintained for the modeler to

Table 3.1 Composition and Structure Terminology

TERM	DEFINITION
Real model component	Has a direct correspondence to a component in the system being modeled; Visualized or represented.
Virtual model component	Has no direct correspondence in the system; Not visualized.
Dynamic model component	Model component which is movable; temporary or permanent.
Static model component	Model component which is not movable; permanent.
Model static structure	Architecture of model formed from its static model components; There is one model static structure with the model itself as the root.
Model dynamic structure	Architecture of a dynamic component; Has dynamic object as its root.
Simple static structure	A component hierarchy of submodels and static objects having a submodel as its root in which the hierarchy is extended at points where interior submodels are decomposed.
Submodel	The root of a simple static structure with children of zero or more submodels and zero or more static objects; Decomposable.
Static Object	The most basic component of interest in a simple static structure; Not decomposable.
Simple dynamic structure	A component hierarchy of subdynamic objects and base dynamic objects having a subdynamic object as its root; Extended at points where interior subdynamic objects are decomposed.
Dynamic object	Basis (root) for model dynamic structures; Decomposable
Subdynamic object	Root of a simple dynamic structure with children of zero or more subdynamic objects and zero or more base dynamic objects; Decomposable
Base dynamic object	The most basic model component of interest in a simple dynamic structure; Not decomposable

switch back and forth between definition and specification. The modeler uses top down definition and bottom up specification, following the Conical Methodology. While upholding the principles of the Conical Methodology, the DOMINO promotes individual creativity and freedom in the design process. The above features (Object Orientation, Graphical Orientation, and Multiview Specification of Model Component Logic) are now broadly discussed in the context of model design and implementation. These features of the DOMINO are fostered by several different means through the VSSE toolset (Chapter 4), the English-like specification language (Chapter 5), and the underlying OOP (Object Oriented Programming) implementation (Chapter 6). This section deals with these features of the DOMINO only at the highest levels. The individual chapters mentioned above and the figures within them should be consulted for further clarification, understanding, and demonstration. When appropriate, applicable sections and figures from other chapters (including Chapter 7 covering the example applications) are referenced as a help and for amplification.

3.3.1 Object Orientation

Derrick [1988] suggests that object-oriented capabilities are desirable in the next generation conceptual frameworks. Balci and Nance [1989] affirm this. Indeed, the benefits of an object-oriented framework for model design and implementation have been long recognized within the SMDE project given the object-oriented characteristics of the Conical Methodology [Nance 1987]. The DOMINO supports the Object Oriented Paradigm. Therefore, the attendant benefits of the Object Oriented Paradigm which promote good model design are available when building models under the new conceptual framework (modularity, information hiding, weak coupling, strong cohesion, abstraction, extensibility, and reusability [Korson and McGregor 1990]. In concert with these benefits, perhaps the most significant "plus" is that it provides a vehicle for a direct and natural correspondence between the system (reality) and the model [Booch 1986]. The following OOP facilities are available: class and object creation; the use of methods; message passing among object instances; a full, single inheritance mechanism; name conflict resolution for methods and attributes in the class hierarchy; and a limited capability for dynamic binding of references

to an object and its attributes. Multiple inheritance is not allowed and polymorphic referencing among objects is not possible as explained in Chapter 6.

3.3.1.1 Class Definition and Specification

Under the DOMINO, modelers specify class information (for real and virtual objects; Section 4.1.2.1) such as attributes (name, type, initial values) for model objects of the class; they also specify class inheritance hierarchies (class-subclass data). Model component logic specifications in the form of class methods, supervisory logic, and self logic (See Section 3.3.3) are developed using the specification language. Modelers associate a set of images to the class. That is, object instances of the class can assume a single image at a time from the set; the image can be shifted during runtime. If class objects are decomposable in the model hierarchy, a layout image set is also associated with the class. (The term layout is defined in Section 3.3.2). Component object and layout image sets are created and linked to their appropriate classes.

Using sets of images for the components and their layouts holds important implications. Objects of the same class can take on different images from the identified class set of images. This is helpful, for example, in the case of a traffic intersection light which can be red, green, or yellow during its life. A set of images for the light accomplishes this. Also consider a server object which is sometimes busy, sometimes idle. Thus, a set of images for the server object's class can be created which contains two images: one representing the busy state, and one representing the idle state. Generally speaking, anytime an object needs additional attributes or needs a different behavior via a new logic specification, a new class for the object must be created. Yet, because of the use of sets of images for a class, model object instances do not need to be created from different classes in order to look differently. However, should it be necessary to create a new class for an object, the new class can inherit and use the image set of a parent class. The impact of having a set of images for layouts of a class is discussed in Section 3.3.3.

3.3.1.2 Object Creation

Object instances in the model which are present at the initiation of model execution are created and identified. These instances are instantiated by a modeler during a graphical traversal of the model static and dynamic hierarchies as reviewed in the next section. Object instances which are generated during model execution are accommodated by the runtime creation facilities of the specification language. Graphical instantiation is required only for real objects. Virtual objects are created and instantiated by textual input rather than graphical manipulations.

3.3.2 Graphical Orientation

In this section, we cover the terminology of the DOMINO as it relates to the graphical definition and specification. The creation of images for model components and model layouts is presented. The importance of the class layout concept is given. The various configurations of layouts are discussed. Finally, the top-down definition of model component hierarchies and real object instantiation throughout the hierarchy is explained.

3.3.2.1 Layouts, Paths, and Connectors

In Section 3.2, model static structure and dynamic structure(s) were discussed. The movement of dynamic objects throughout these model component hierarchies was described. In order to understand this movement and the DOMINO features which permit its visualization, two important terms of the DOMINO, layout and path, are now defined.

The decomposition points at each level require the existence and association of a background image over which the dynamic objects travel. These background images are called *layouts*.

As described earlier, the root of the model static structure is the model itself; a single top level layout must be in place for the model. The root of any model dynamic structure is a dynamic object; if decomposed, the dynamic object also must have a single top level layout in association. The decomposition points down either of these hierarchies (i.e., at decomposed submodels in the model static structure or at

decomposed subdynamic objects in a model dynamic structure) must also "own" and have an associated layout. The layout is required only when the components (submodels or subdynamic objects) are decomposed. If not decomposed, then the layout in which the submodel or subdynamic object resides suffices for dynamic object movement at that level.

Dynamic object movement between or among model components is specified for each layout by the creation of roadways or *paths*. These paths connect the model components that exist within each layout. These components, before setting up paths, must first be created and instantiated as later described.

Connectors are also created within the layouts to facilitate the movement of dynamic objects into the layout (as the dynamic objects descend into the hierarchy, i.e., *entry* connectors) and out of the layout (as the dynamic objects ascend up the hierarchy, i.e., *exit* connectors). Top level layouts (for the model and decomposed dynamic objects) do not have connectors.

3.3.2.2 Creation of Layout and Component Images

Before objects are instantiated on the layouts and before paths or connectors may be created, the layout images must be drawn or created (Section 4.1.1). Once drawn, these images (full screen) are associated with the model's top level object (root of the model static structure), a dynamic (most likely decomposed) object (root of a model dynamic structure), or other decomposable component objects (submodels, subdynamic objects). The model object is allowed only one top level layout. However, other layout images can be members of a set of images, associated with a decomposable component class. At this point the layout images have no meaning to the model other than as simple (raster) images; there are no identifiable model components within them.

Model component classes (submodels, static objects, subdynamic objects, base dynamic objects, and dynamic objects) have individual images created for them. Like layouts, these individual component images can be grouped as members of a class set. The images created for dynamic object class instances are used to visualize/animate the spatial movements of the dynamic objects over the layout image. The images for submodels, subdynamic objects, static objects, and base dynamic objects are useful (via "cut

and paste" type operations) for constructing the various portions of layout images.

Figure 7.66 and 7.75 show completed examples of a layout image and dynamic object component images of the Branch Operations Model (Chapter 7).

3.3.2.3 Class Layout Creation and Layout Definition

Once the layout image is created, the class layouts are formed one at a time from their associated layout images. Each component within the layout image is identified by graphically bounding the image portion corresponding to the component. For example, submodels and static objects are identified in layouts for the model top level or for decomposed submodels. Subdynamic objects and base dynamic objects are located in layouts for decomposed dynamic objects or decomposed subdynamic objects. The designation of component locations on a layout image instantiates the class layout (Section 4.1.2.2). As each component is identified, it is given a variable name as a member of that layout. The variable name has meaning only within the context of the layout, not the model as a whole. Components are connected with paths indicating the pathways for dynamic object movement; entry and exit connectors are also indicated as well. Dynamic objects move directly into decomposable components (submodels, subdynamic objects); therefore, a path may lead into or out of any of these. However, static objects and base dynamic objects are not decomposable and do not have incoming or outgoing paths. Interaction points, *interactors*, are created which permit dynamic object interaction with static and base dynamic objects. Thus, dynamic objects move *into* decomposable components (submodels, subdynamic objects) but *to* the interactors associated with non-decomposable components (static objects, base dynamic objects). In Section 3.2, we noted that the use of submodels and subdynamic objects promotes design flexibility. That is, the model static structure or dynamic structure can be extended through these component types if further decomposition becomes warranted in the course of the design. As stated above, during animation, dynamic objects move *into* these decomposable components but *to* the non-decomposable ones. Hence, the visualization of an interaction (e.g., customer dynamic object and server static object, bus passenger dynamic object and bus driver base dynamic object) is more meaningful with a movement to (the

interaction point of the non-decomposable component) than movement into.

The spatial description of a layout's image which is derived from the aforementioned process (designating model component, connector, and interactor locations, as well as paths for dynamic object movement) is called a layout *definition*. Each layout image must have a layout definition. Figure 7.67 is the completed class layout definition of the top level for the Branch Operations Model of Figure 7.66. Note the submodels are bounded by rectangles and are connected by paths.

All definitions of a class layout image set must be *compositionally equivalent*. That is, there must be an equal number of each component type, connectors, and interaction points (if applicable). And there must be a one-to-one correspondence between the variable names among the layout definitions of a class set. These conditions being satisfied, the spatial configuration of the components, paths, etc. may be different among definitions. Using the power of the Object Oriented Paradigm, compositional equivalence among layout definitions of a class set enables several class layouts and definitions to be created for a single class. For example Figures 7.68 through 7.71 depict the set of class layouts and definitions for the submodel class **branch** from the Branch Operations Model. Each of the four is compositionally equivalent to the others. The important implications of compositional equivalence among class layouts is given in Section 3.3.3 on Multiview Specification of Model Component Logic. A final note: the configuration requirements of the next section must be followed and kept in consideration during layout image drawing and class layout creation and definition.

Table 3.2 provides a summary of all terms related to graphical orientation of the DOMINO.

3.3.2.4 Requirements for Layout Configurations

Considerable flexibility is provided for establishing the component configurations of class layouts. However, certain restrictions do apply to the association of components with connectors (entry and exit) and interaction points. The following five rules govern the restrictions of linkages in the layout configurations which are made using paths between the components:

Rule 1. A model component can be linked to an entry point, exit point, and (possibly) an interaction point, but not more than one of each. (General Rule)

Table 3.2 Graphical Terminology

TERM	DEFINITION
Layouts	Background images associated with the decomposition points of each level of the model static structure or any model dynamic structure.
Paths	Roadways for dynamic object movement between model components.
Connectors	Entry and exit points which facilitate the movement of dynamic objects between layouts.
Interactors	Interaction points for dynamic objects with non-decomposable components (static objects and base dynamic objects).
Layout definitions	Spatial descriptions of class layouts which include applicable component, connector, interactor, and path locations superimposed on the layout image; Definition lines are not visible during animation.
Compositional equivalence	Exists between a set of layout definitions which have: (1) an equal number (by type) of components, connectors, and interactors and (2) a one-to-one correspondence between the variable names among the definitions of the set.

Rules 2 through 4 clarify the General Rule depending upon the type of model component:

Rule 2. Submodels and subdynamic objects can be linked to entry or exit points or to both. The "not more than one of each" restriction (Rule 1) still holds.

Rule 3. Static objects and base dynamic objects can be linked to entry or exit points or both (as in Rule 2) *but this linkage is via an associated interaction point which must be present.*

Rule 4. Only one associated interaction point is permitted for each static object or for each base dynamic object.

Rule 5 covers the basic requirements regarding model component-to-model component linkages.

Rule 5. In a single direction of movement, only one linkage can exist between components.

Consider the following configurations (Figures 3.5 through 3.15) for clarification of these rules. All figures are submodel layouts with submodels and static objects. Simple substitution of subdynamic object (for submodels) and base dynamic objects (for static objects) converts each figure for applicability to dynamic component layouts. Figures 3.5 (A), (B), and (C) display configurations which support Rule 2. The submodels are connected to both entry points and exit points, but not more than one of each. Rule 2 is violated in Figure 3.6 on two counts. There are two entry points for the submodel and two exit points. Figure 3.7 is a valid configuration for a static object supporting Rule 3. Two entry points connected to a single interaction point of the static object violate Rule 3 in Figure 3.8. Increasing the complexity, Figure 3.9 conforms to all rules, particularly 3 and 4, with only one interaction point per static object. Rule 4 is violated in Figure 3.10 with multiple interaction points for the static object. Considering Rule 5, Figure 3.11 presents a legal configuration. For example, only one link exists from submodel B to submodel C and one link from submodel C to submodel B. Having two links from submodel A to submodel B in Figure 3.12 invalidates that layout. The linkages between submodels B and C in that figure are, however, satisfactory. In the same manner, static object A is appropriately linked to submodels A, B, and C in Figure 3.13. On the other hand, the configuration is invalid due to the presence of two links from static object B to submodel D.

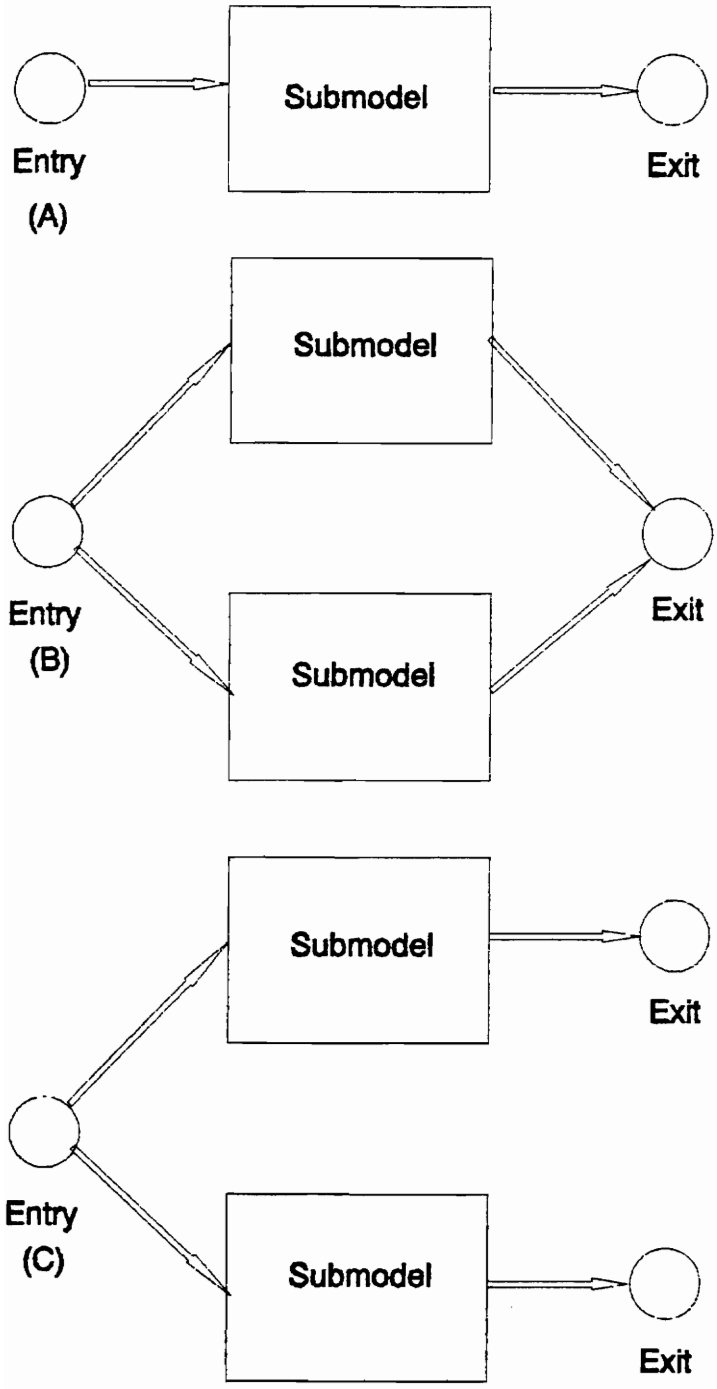

Figure 3.5 Layout Configurations Supporting Rule 2

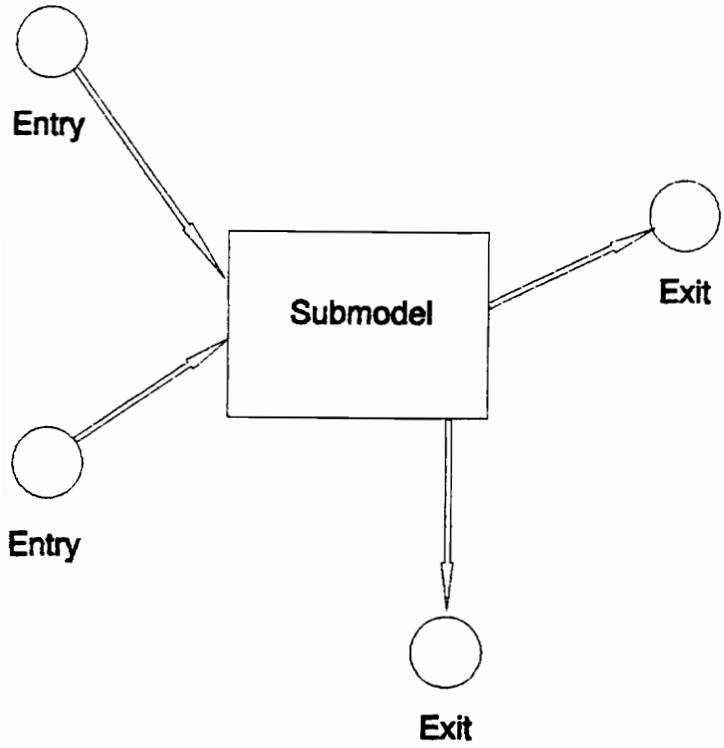

Figure 3.6 Layout Configuration Violating Rule 2

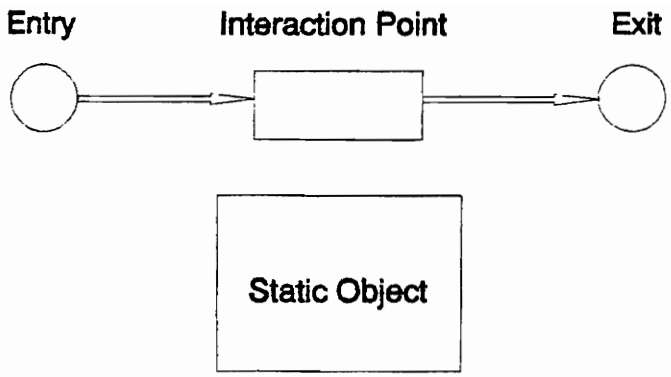

Figure 3.7 Layout Configuration Supporting Rule 3

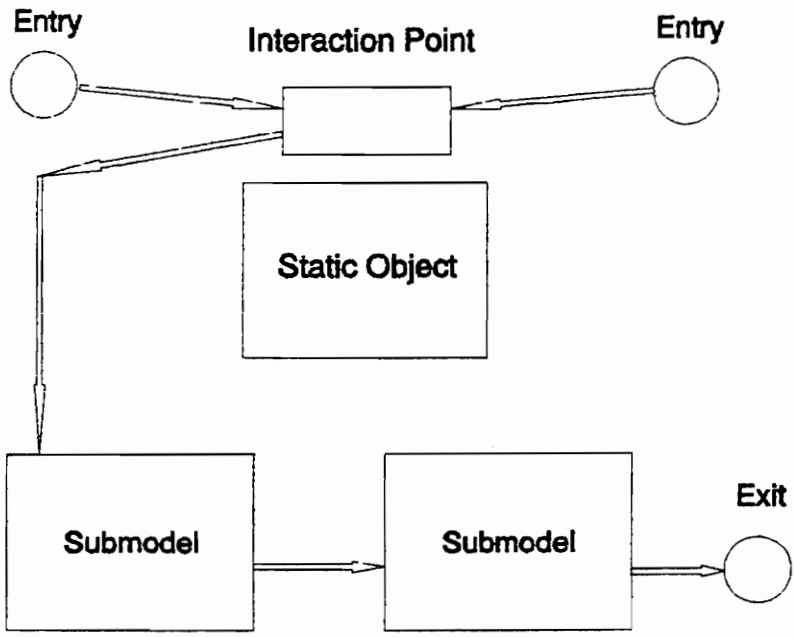

Figure 3.8 Layout Configuration Violating Rule 3

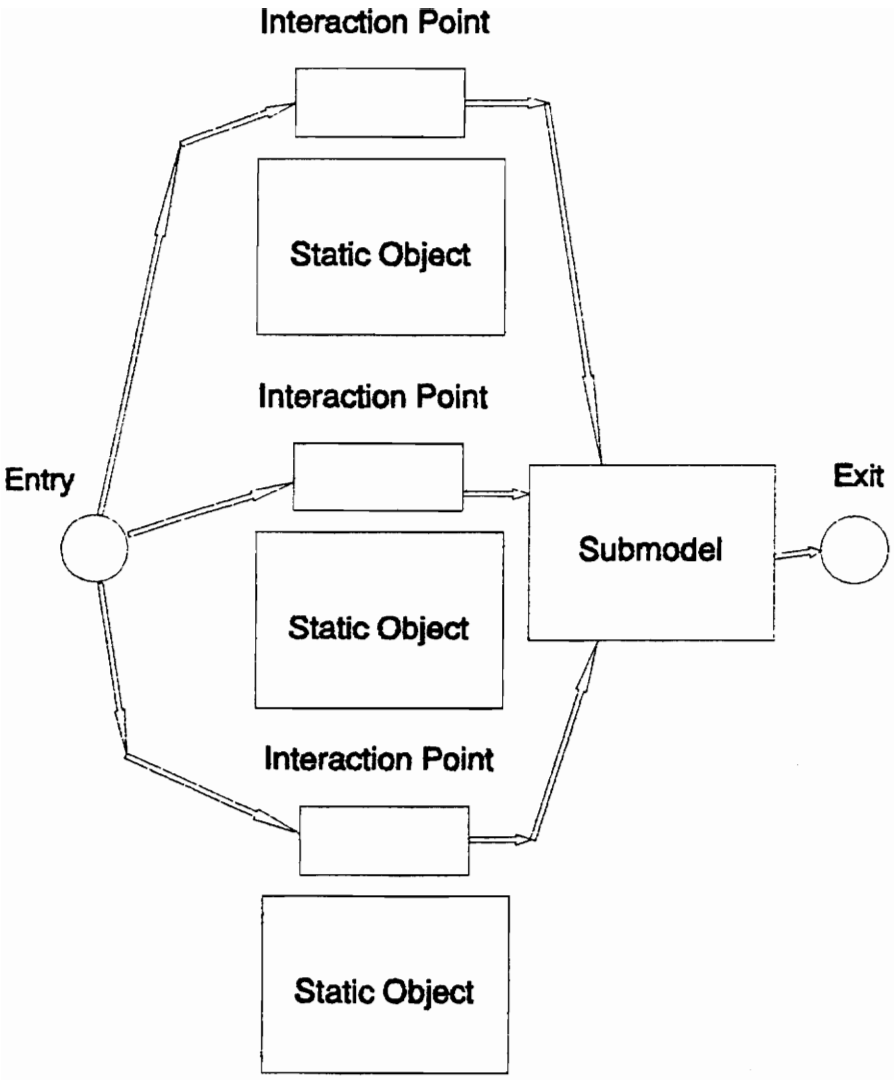

Figure 3.9 Layout Configuration Supporting Rules 3 and 4

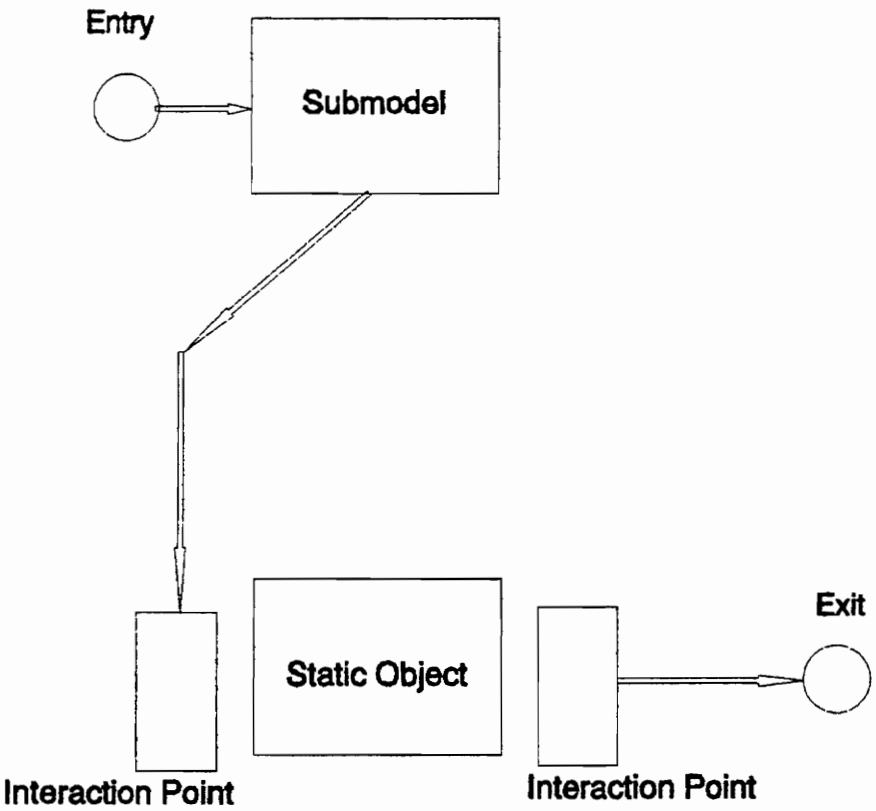

Figure 3.10 Layout Configuration Violating Rule 4

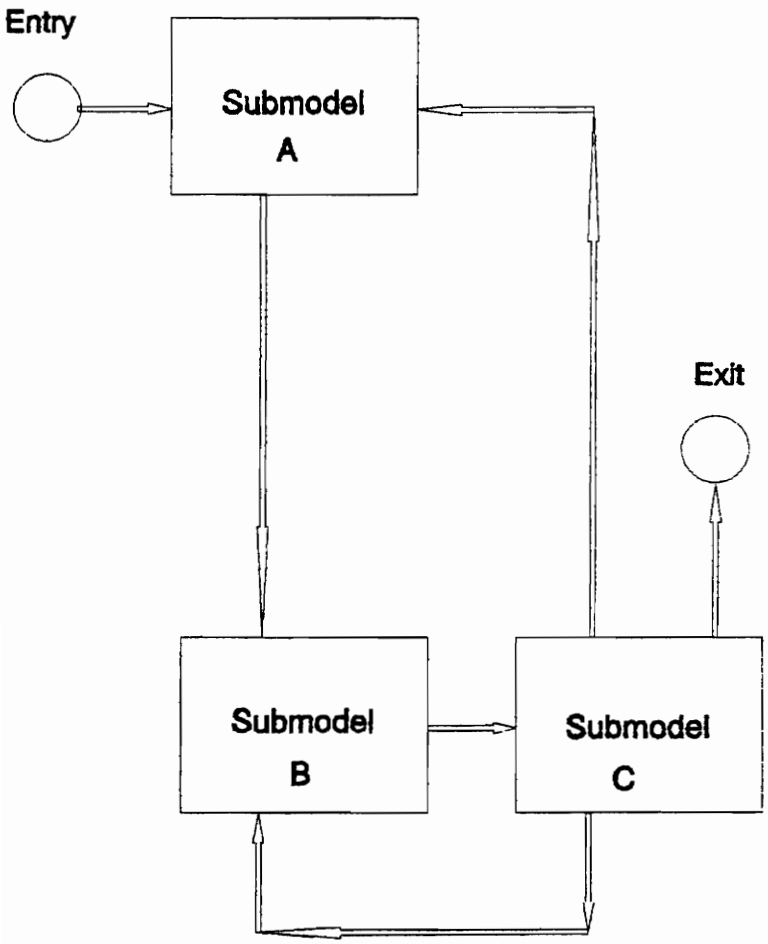

Figure 3.11 Layout Configuration Supporting Rule 5

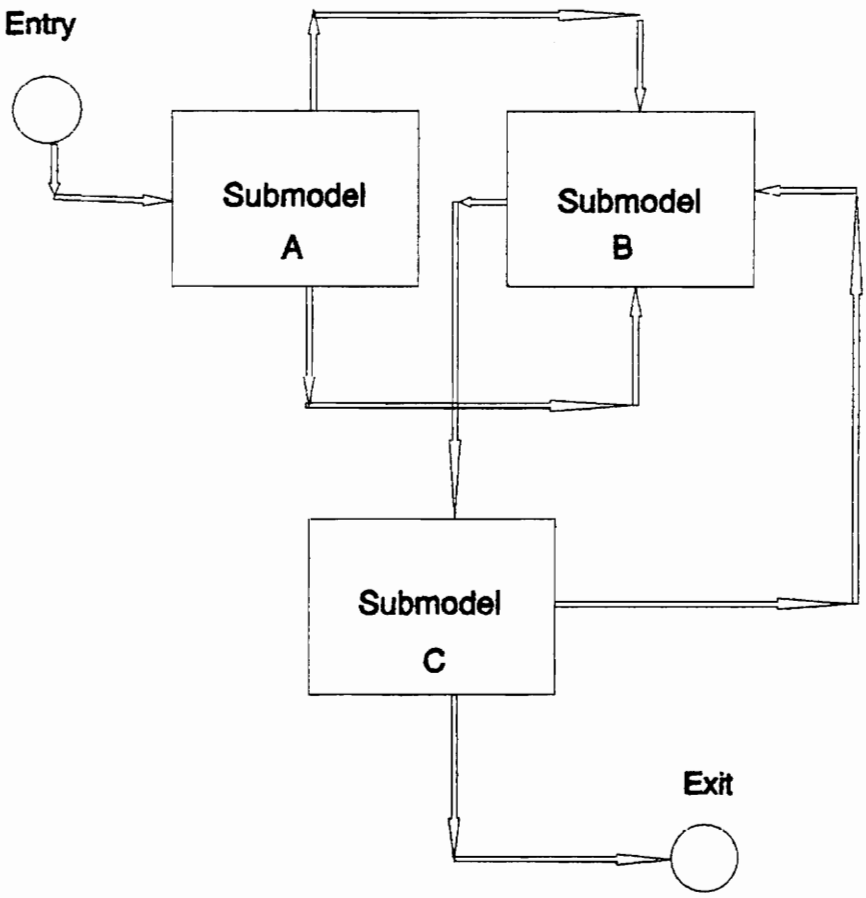

Figure 3.12 Layout Configuration Violating Rule 5

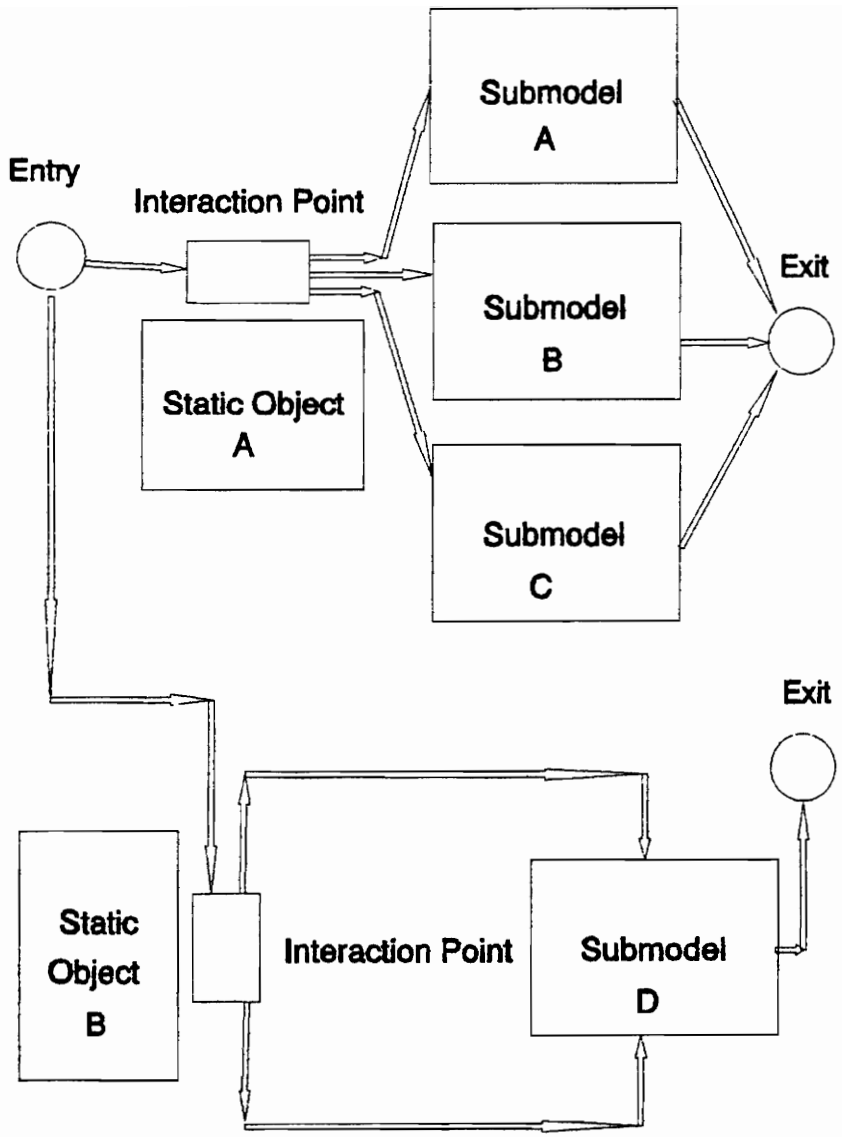

Figure 3.13 Layout Configuration Violating Rule 5

3.3.2.5 Top-Down Definition and Hierarchical Traversal

Once the class layouts and their definitions have been created with a general instantiation, a further specific instantiation of components can be performed. This is done in a top-down manner. Each class layout is used to create an instance layout. We start with the model top level class layout and its definition to instantiate the model static structure. Similarly, to instantiate a model dynamic structure of more than one level, we begin with the class layout and definition associated with a decomposed dynamic object.

The instantiation process (described in Section 4.1.2.3 in much greater detail) converts the variable names of class layout components to their literal names which uniquely identify them from among all other components in the model. For each component in the class layout (and definition), beginning from the top level) three actions for instantiation are possible: **create**, **decompose**, and **descend**. The create action is available for all modeler-defined components. The decompose and descend actions are only available for submodels and subdynamic objects. A class layout component must first be created and instantiated before any decomposition or descending of the hierarchy can occur.

Create performs the object instantiation of a class layout component; the modeler uses create to assign the unique literal name to the component. Note that this instantiation is not required for connectors and interactors. Once created, a submodel (or subdynamic object) can now be decomposed (if desired).

The **decompose** action allows the modeler (should decomposition be desired) to associate a new class layout image and definition with the submodel or subdynamic object component decomposition. The instantiation process as just described can be continued in the newly chosen class layout. During instantiation, the modeler can choose to ascend the hierarchy, a level at a time, or jump to the root layout if deep into the hierarchy. Similarly, once a decomposable component has been created and decomposed in a particular level of the hierarchy, the **descend** action becomes available. The modeler can now graphically traverse down the hierarchy during the instantiation and definition of model components or upward as deemed necessary. The modeler essentially "stacks" the instantiated class layouts during the traversal, effectively building the model static structure or a dynamic structure. Top-down definition is enforced but the modeler has extreme flexibility in his decomposition choices. Complete definition at each

level (before proceeding deeper decomposition) is not mandated. Later, in Chapter 4 on the VSSE, we describe how the same hierarchical traversal facility is available during model execution and animation.

3.3.3 Multiview Specification of Model Component Logic

There are three different types of logic specification for use in building models under the DOMINO: supervisory logic, self logic, and method logic. These specifications provide the rules for model component interaction and model dynamics. Each type of specification is created using the DOMINO's specification language. This is done within the Model Generator with the same options that cover class specification. As such, each type of logic is tied to an object class. In this section, we give an overview of logic specification. The details of the specification language are covered separately in Chapter 5.

We have already presented that model design under the DOMINO is object oriented and that a modeler may take a natural view of the system. The three types of specification under the DOMINO provide additional multiple views or modeler perspectives of the system for implementation purposes. The specification language and system implementation make these multiple views possible. Modelers can design the traditional material-oriented or machine-oriented models. (Tocher [1965] originally characterized SPLs as material or machine-oriented. Although these terms are dated (i.e., the use of terms permanent entity and temporary entity is commonly favored over machine and material respectively), recent literature [Kreutzer 1986] and preference has prompted their use throughout the thesis.) The DOMINO also allows a modeler to combine these two views in a hybrid view. The multiple views and the relationship of each to the three types of specification are described. First, the types of logic specification are defined. The implementation views are then covered. Relevant examples of model component logic are covered in detail in the model applications of Chapter 7 rather than here. Finally, the promised implications of compositional equivalence among class layout images and definitions are presented.

3.3.3.1 Types of Logic Specification

Models built exclusively with supervisory logic are machine-oriented. The supervisors (machines)

are the principal influencers for model execution. The dynamic objects (material, transactions) are manipulated and moved from component to component. Logic attached to the class of the supervising component is called *supervisory logic*. Supervisors can be any of the component types: submodels, static objects, subdynamic objects, base dynamic objects, or dynamic objects (decomposed only). The dynamic objects, as they are passed along from component-to-component, execute the various supervisory logics. The logic specification is composed as directions to each dynamic object that moves into or to the supervising component. The earliest prototypes of the VSSE utilized only this logic approach.

Self logic is the logic attached to a dynamic object (material, transaction). The dynamic object now executes its own logic and determines its own destiny. Models built entirely around self-logic are called material-oriented models. In this case, the supervisors (machines) are now passive, and they are acquired, held, and released by the dynamic objects. The self logic reads as a "process". Self logic is available only to dynamic objects. Any existing supervisory logic which is tied to component class is either bypassed (turned off) or is completely missing. The ability to turn on and turn off the logic suggests some interesting possibilities which have been investigated in part and are later described relative to the hybrid view. The self logic of the executing dynamic object remains its sole source of execution direction. The dynamic object directs itself from one component to another; the logic specification is a set of directions to itself.

The logic attached to a component class and activated by sending a message to an owning component in the class is called a *method*. Methods typically encapsulate operations on a component's own data. Therefore, no executing dynamic object exists for this logic. Table 3.3 summarizes the terminology of logic specification.

3.3.3.2 Implementation Views

Several versions of a Bus Route model (Chapter 7 describes) were built which use the various combinations of the logic specification types as follows:

All Supervisory Logic: The logic is spread among the supervising components and is executed by the

Table 3.3 Model Component Logic Specification Terminology

TERM	DEFINITION
Supervisory Logic	Logic attached to the class of the supervising component; The supervisors are the primary influencers for model execution, manipulating and moving dynamic objects from component to component.
Self Logic	Logic attached to a dynamic object class; The dynamic object executes its own logic and determines its own destiny when its self logic is activated.
Method Logic	Logic attached to a component class which encapsulates operations on its own data; Activated by sending a message to an owning component in the class.
Machine-oriented	Descriptive term for models built exclusively with supervisory logic.
Material-oriented	Descriptive term for models built exclusively with self logic.
Hybrid	Descriptive term for models built using a combination of supervisory and self logic.

various moving dynamic objects.

Example: In the Bus Route model, there are person and city bus (decomposed) dynamic objects which execute supervisory logic of a bus terminal, bus stop, and house submodels. Once on the bus, the person dynamic objects execute the supervisory logic of the bus driver base dynamic object and the seat subdynamic objects.

All Self Logic (except for the virtual initializing logic): All dynamic objects take on self logic. No supervisory logic of the static components is utilized. Although supervisory logic may be present (that is, specified), it is turned off.

Example: In the Bus Route model, the person and city bus dynamic objects execute their own self logic. The self logic code, in this case, gets to be quite voluminous since queueing and servicing logic must be included in the self logic. The logic is complicated and hard to follow.

Hybrid Logic: There are two types of hybrid logic. First, there may be all self logic except that the executing move statements (See Chapter 5) within the existing self logic allow some supervisory logic to be executed. This removes the complexity mentioned just above. The self logic now reads more like a process, and the queueing and servicing logic can be encapsulated in the supervisory logic of the queue and service components.

Example: In the Bus Route model, the city bus executes only its self logic, however, person dynamic objects execute self logic AND the supervisory logic of the bus driver and the seats.

In the second type, which is much more the hybrid, all temporary dynamic objects execute the supervisory logic of their visited components, but any decomposed dynamic object executes its own self logic.

Example: In the Bus Route model, the city bus executes its self logic, while the person dynamic objects execute supervisory logic as in the earlier "all supervisory logic" versions.

Model components can retain their self and supervisory logic and the model can be compiled and run under any of the above hybrid versions. The version is determined by the setting of a boolean variable which will turn on or off the various self and supervisory logics into the desired combination.

3.3.3.3 Implications of Compositional Equivalence

Previous sections have promised to describe the implications of compositional equivalence of class layout set images and their definitions. Figures 7.68 through 7.71 show such a set for the submodel class branch. Using the variable names for the components of the branch layout (which are the same among all

layout definitions of the set) within the specification language, a single supervisory logic specification is all that is necessary for the class branch to direct dynamic object movement (the variable names of components are used, not the literal names). Visually during animation, the movements will be different among the four branches due to the different images and definitions. But since compositional equivalence holds among them, there is no need to produce multiple specifications for supervisory logic. Previous versions of the DOMINO and VSSE prototypes required this duplication of specification effort. Now, due to the ability to use a single supervisory logic specification with variable component referencing in tandem with multiple layout configurations, the specification effort has been considerably shortened; the coding requirements have been reduced. Chapter 7 discusses this in greater detail.

3.4 History

This section presents the evolutionary development of the DOMINO and its associated prototype systems. The evolutionary development has spanned between 1984 and March 1992. Three separate phases of development can be identified. The final version of the DOMINO and the VSSE (which incorporates the improvements and modifications from each phase) are chronicled in this thesis work. Thus, the details of the final phase are only briefly mentioned here.

The earliest concepts centered upon the dynamic object, submodel, and static object components in a model. Objects were classified as real or virtual. Component logic specifications were attached to submodel instances (essentially as supervisory logic) and were organized on the basis of six identifiable activities. These activities were driven by three conditions (entry condition, activity start, and exit condition). The first prototype for testing these initial concepts was the GPVSS [Bishop 1989; Bishop and Balci 1990]. Without the benefit of a specification language, model component logic was specified in the C programming language with the help of some macro definitions. GPVSS could only handle simple models like the M/M/1 queueing model. The INGRES relational database system was used to store model specification information. Under GPVSS, submodel and object attributes were global. OOP was not directly supported since inheritance was not possible. Visualization of the model execution was

implemented using the concepts of path and backgrounds (layouts). Some concepts (static objects, submodel decomposition, and dynamic object decomposition) were conspicuously missing from the GPVSS implementation.

The development of the second version of the DOMINO included simple OOP with the class concept and limited inheritance but no ability for inheritance hierarchies (i.e., instances were created from classes; the class hierarchies were only one level). Inheritance of model component logic was not possible; the logic was still attached to each object instance. An initial version of the specification language was created. The term background was renamed layout. The prototype (not named) implementing this version of the DOMINO incorporated the decomposition of submodels. For this, the concept of connectors was introduced. All INGRES database relations were redesigned. The six-level activity based approach for logic specifications was scrapped in favor of the new English-like specification language. This prototype also included statistical analysis facilities (confidence intervals, etc.) and limited capabilities for contextual visualization.

In the final phase, the terminology was further extended with subdynamic objects and base dynamic objects. These resulted from the implementation of the DOMINO in the VSSE (Chapter 4) via considerations for the decomposition of dynamic objects. Also, with the implementation and use of static objects and base dynamic objects (in the VSSE), the concept of the interactor was derived. Full single inheritance of attributes and model component logic was incorporated. As described in Chapters 3 and 6 inheritance hierarchies are now possible. Methods and message passing are included. The specification language (Chapter 5) has been completely revamped; a powerful translation facility using symbol tables has been developed. Multiple modeler perspectives for model component logic (supervisory, self, method) are possible. A complete toolset has been implemented in the VSSE providing analysis and verification facilities. The contextual visualization facilities have been significantly restructured to include a runtime inspection facility (at any context) of model component attributes and their values.

CHAPTER 4
THE VISUAL SIMULATION SUPPORT ENVIRONMENT

This chapter presents a comprehensive description of the Visual Simulation Support Environment (VSSE) and its facilities. The VSSE is another in the series of evolutionary prototypes built for the SMDE (Simulation Model Development Environment) Research Project. Unlike previous prototypes, the VSSE is based on a new conceptual framework (the DOMINO) for the design and implementation of visual simulation models. This chapter is not intended to be a user's guide but does contain many explanations which modelers should find helpful. Additionally, this chapter does not attempt to describe the VSSE implementation. Instead, since the VSSE represents the culmination of many years of research (1984-1992) on the DOMINO, the VSSE is a platform from which the DOMINO can be evaluated. (The VSSE was used to build the example model applications in Chapter 7.) This chapter intends, therefore, to present the wide range of VSSE capabilities for automated modeling support and, in so doing, to help demonstrate the utility of the DOMINO. Naturally, the VSSE underscores the contributions of the research effort.

The VSSE is developed by using the C programming language, the SunView Graphical User Interface [Sun Microsystems 1988], and EQUEL/C [Sun Microsystems 1986b]. It encompasses approximately 50,000+ lines of documented code and runs on a SUN 3/160 color workstation. Tools are able to effectively communicate with one another primarily on the basis of a common model specification representation as stored in the INGRES Relational Database [Sun Microsystems 1986a]. Some secondary information is kept in VSSE system files. Figure 4.1 is the top level menu of the VSSE. As shown, the VSSE has a fully working minimal set of tools, each indicated by an icon on the top level display: Model Generator, Model Analyzer, Model Verifier, Model Translator, and Visual Simulator. The description of each tool, given below, uses several illustrative figures and examples taken from the various application examples in Chapter 7.

4.1 The Model Generator

The Model Generator is activated from its icon at the top level display. With any of the tools, one

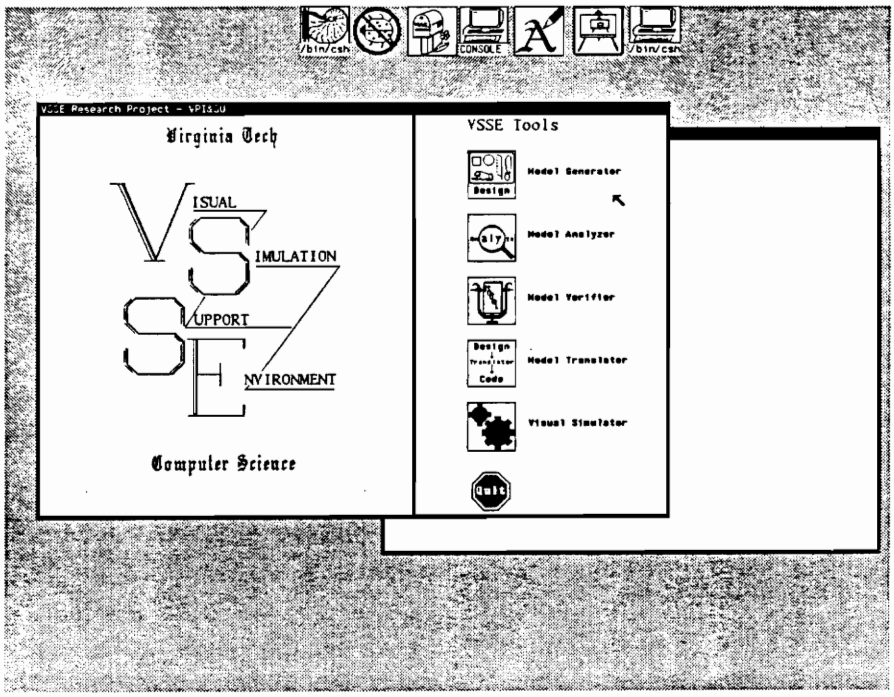

Figure 4.1 VSSE Top Level Menu

can retrieve an existing model (which may be in any intermediate or final stage of development). This is done by selecting the **Retrieve Existing Model** button or by typing in the model name at the caret (Figure 4.2). The Model Generator provides for model creation as well. Only one tool can be "in use" at any time.

The Model Generator contains two principal parts: the Image Editor and the Model Editor. Both parts are graphically-oriented and maximize the use of the direct manipulation interface. Generally speaking, the Image Editor is used to create the pictures and images for the visualization of models. Specifically, the component object images (which must include dynamic object images if animation is desired) and decomposition layout images are created. The Model Editor allows the complete definition and specification of a model to include the model's static and dynamic structures and object class information (attributes and logic specification). Thus, much of the Model Generator utilizes object-oriented features in addition to the graphical displays.

4.1.1 The Image Editor

The Image Editor (Figure 4.3) is purely a "draw" facility. The various draw operations are available with the buttons from left to right across the top of the Image Editor window. In the **PEN** mode the mouse can be used for freehand drawing. The **LINE** and **RECT** modes produce lines and rectangles. Text of various font styles can be entered or set on the canvas using the **TEXT** mode. Circles and arcs are possible with the **CIRCLE** and **ARC** modes. The **SELECT** mode allows a modeler to define a rectangular portion of the drawing canvas (usually over previously drawn images) and then duplicate the same defined graphic in another location on the canvas. An eraser (with point, small, and large erasure areas) is available using the middle and right mouse buttons. The right button sets the erasure area size; the middle button activates the eraser. The **TO MODEL EDITOR** button toggles the Model Generator to the Model Editor facility. Using the **SCREEN OPERATIONS** button, a modeler can (1) save a new drawing, (2) load a previous drawing, and (3) clear the drawing screen.

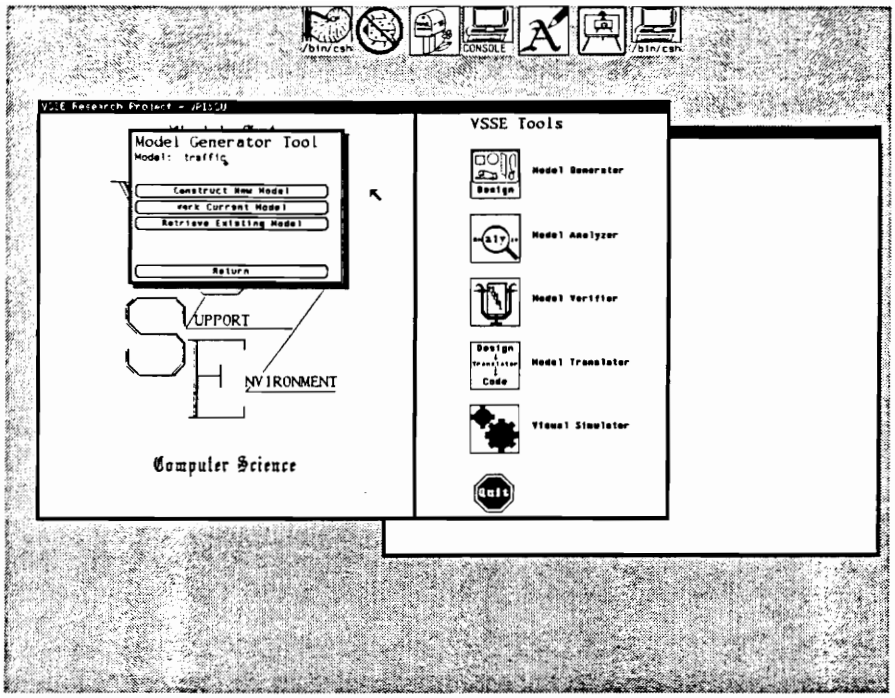

Figure 4.2 Model Generator Menu

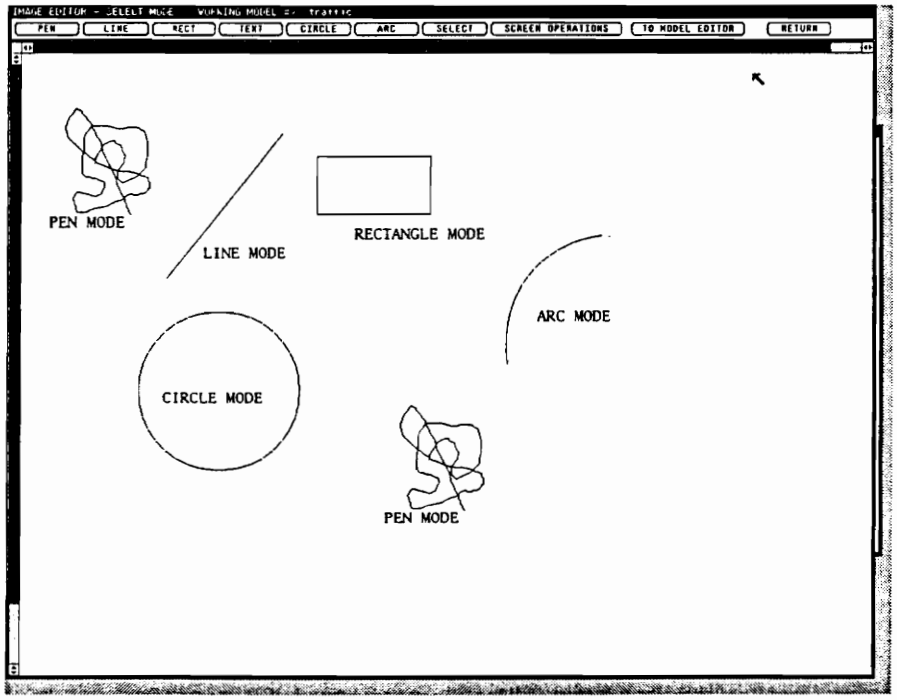

Figure 4.3 Image Editor

4.1.1.1 Saving Images

Figure 4.4 shows that the **SAVE** operation can be applied under screen operations to save a particular component object type (submodel, static object, etc.) image (that has been previously "selected"), a decomposition layout image, or a library image. Saving component object and layout images to the library permits the reusability of the images. Images are saved to file (if, for some reason, the modeler is not yet ready to modify the database with the new image) or to database. Library saves are only to file. Images saved to file are saved by name. For images saved to the database, both name and the class information associated with the image is supplied by a modeler.

During saves to the database, images are named and can be stored as the default image for a component class or as one image in a set associated with the component class (See Figure 4.5; e.g., saving a submodel component image to database). Note that the particular class association must be identified, although the class may possibly not be yet defined. Decomposition layout images are associated with the class of a possible owning component instance (i.e., model top level, dynamic object, submodel, or subdynamic object). Figure 4.6 shows how this association is made. Class associations for component object images are set in a similar fashion. The set of images/default image option is available for saves of component object images and for decomposition layouts. Having a set of images available for a class enables instances of that class to take on any of the representations from within the set. This greatly enhances visual flexibility.

4.1.1.2 Loading Images

Loading an image (Figure 4.7) is handled by the same conventions as saving. They may be loaded from file or from the database, and always by component type (submodel, etc.). Figures 4.8 and 4.9 show the load of a submodel component object image from file and a submodel decomposition layout from the database. The canvas is cleared during the loading of a decomposition layout image which is then automatically placed on the clean canvas. Loading component object images does not clear the canvas. Once the component object image is retrieved, it is placed on the canvas at the location pointed to by the

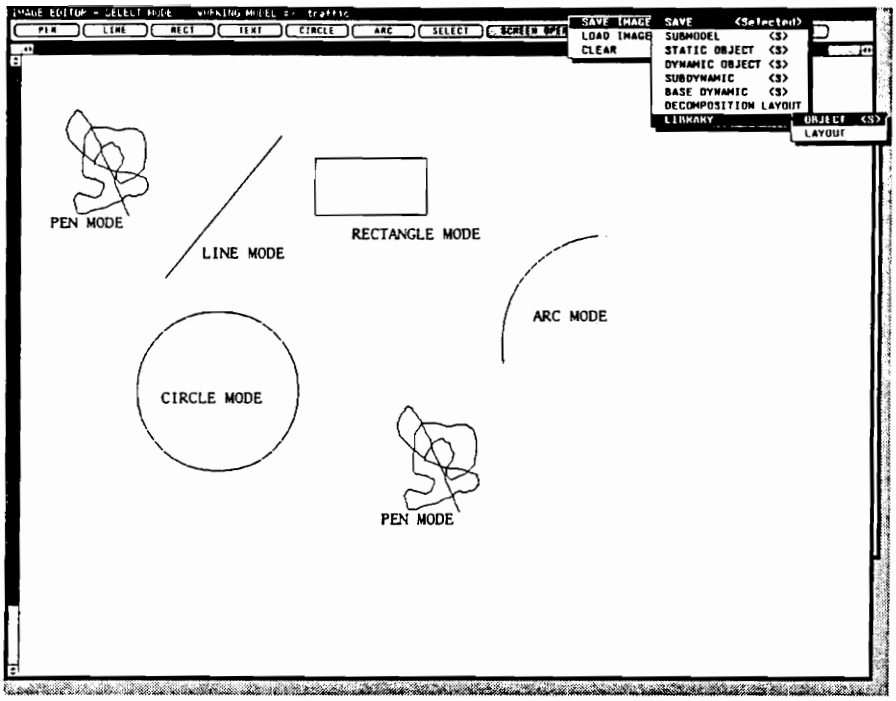

Figure 4.4 Screen Operations (Save) of Image Editor

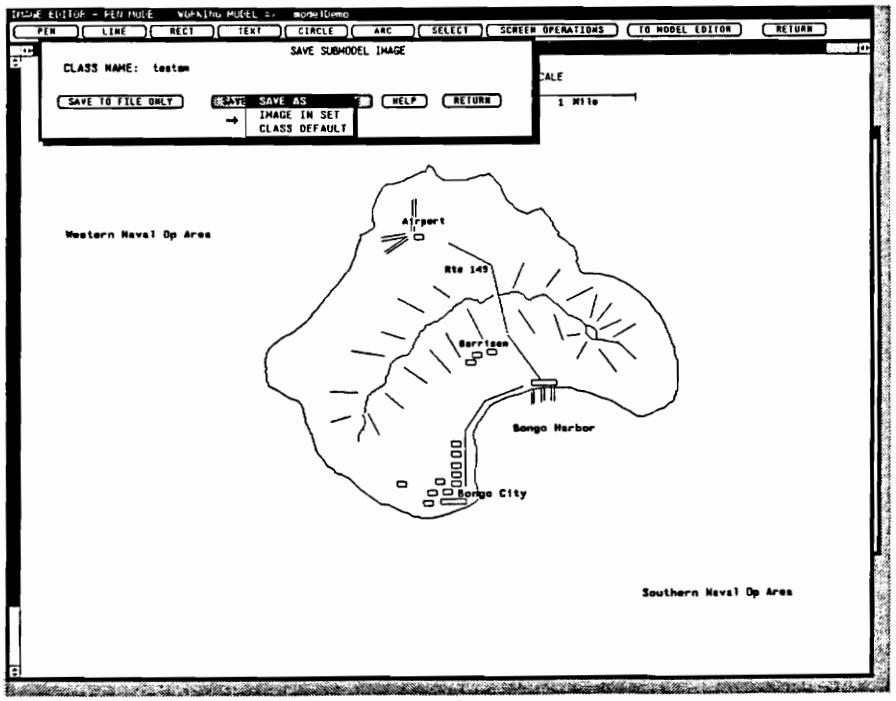

Figure 4.5 Saving Component Image

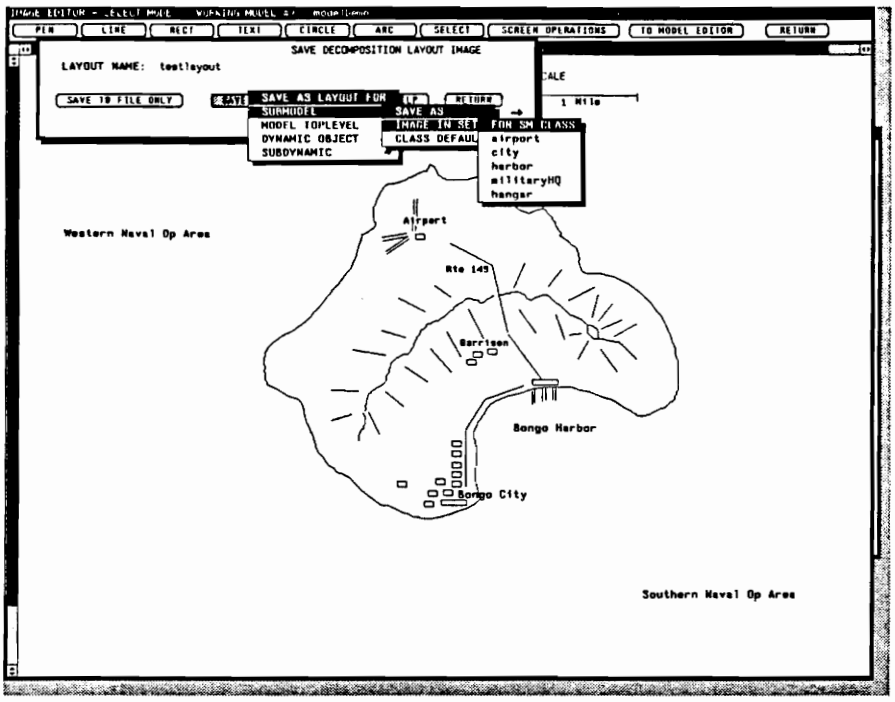

Figure 4.6 Associating Class to Decomposition Layout Image

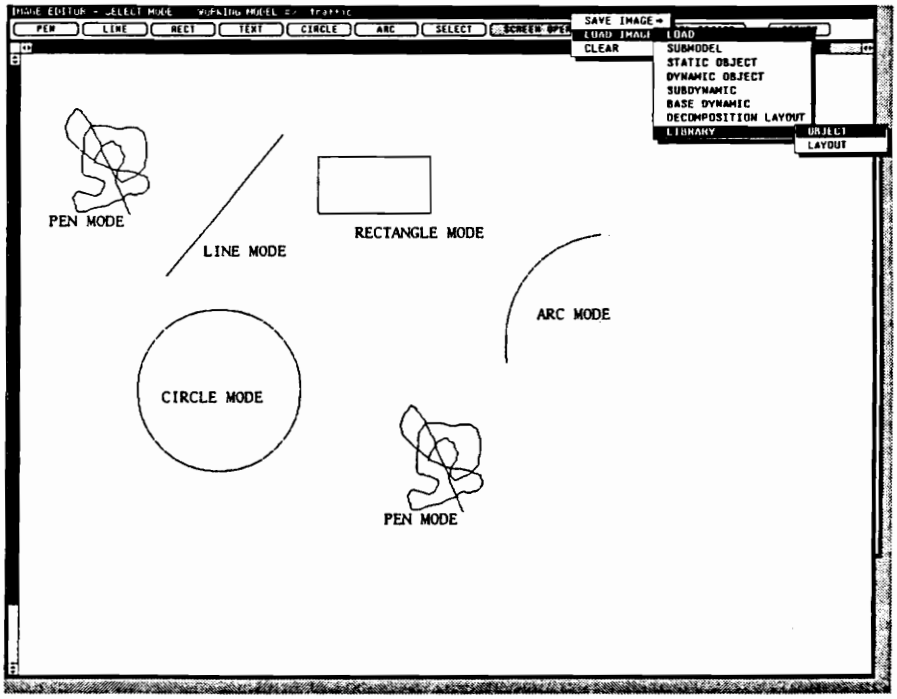

Figure 4.7 Screen Operations (Load) of Image Editor

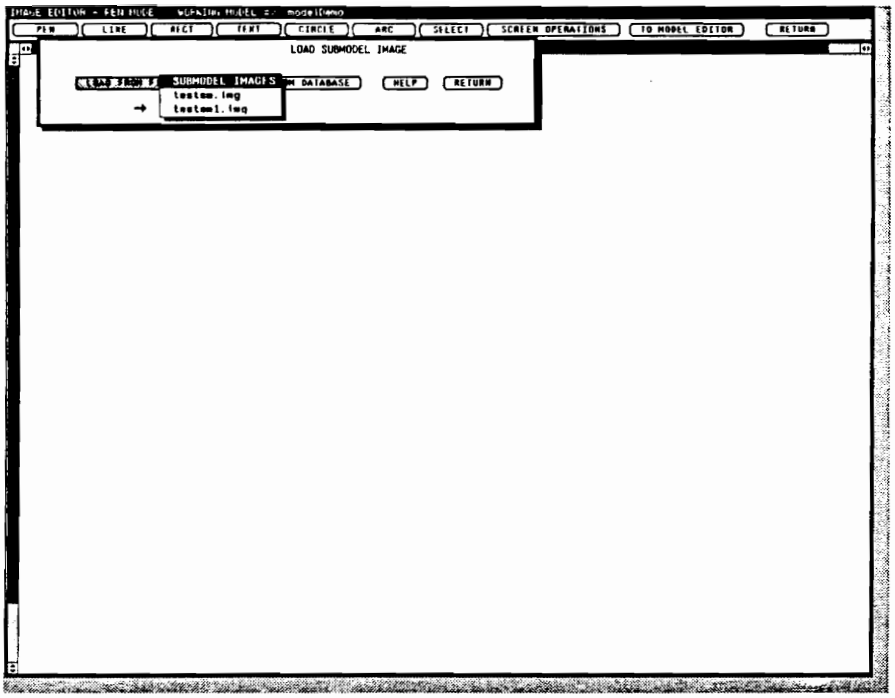

Figure 4.8 Loading Component Image from File

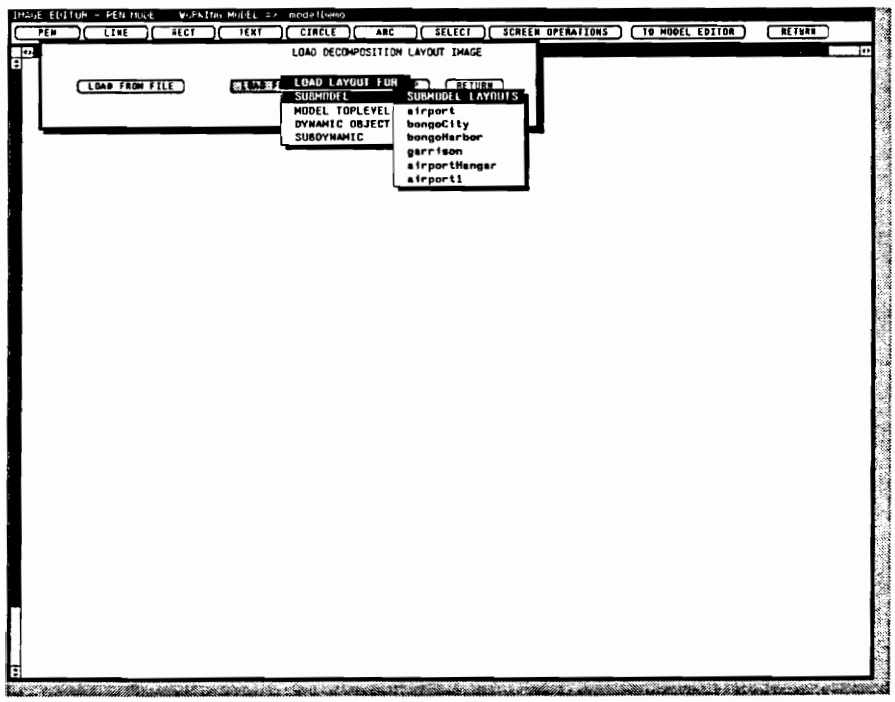

Figure 4.9 Loading Decomposition Layout Image from Data Base

mouse and "dropped" by clicking the middle mouse button.

4.1.2 The Model Editor

Definition and specification are completed from within the Model Editor. All images do not have to be drawn before entering any Model Editor function. Certain aspects of definition and specification (like class attribute and logic specification) can be (but not necessarily) performed prior to any drawing. Flexibility is retained for performance order. Like the Image Editor, the ease of accomplishing editor functions is achieved through the graphical-oriented features of the direct manipulation interface. Now, four separate and primary actions must be accomplished in the Model Editor (not necessarily in this order) to complete the definition and specification of a model:

Action 1. The specification of the object class information (attribute and component model logic),

Action 2. The general instantiation of the components in each class layout using variable names,

Action 3. The identification of paths, connectors, and interactors (as applicable) in each class layout, and

Action 4. The specific instantiation of the components in each class layout using literal names while "stacking" these layouts to form the model's static and dynamic structures.

Four principal operations (buttons) are available and are used to satisfy the above actions. The **SPECIFY** operation is applied toward Action 1. The **CREATE** operation supports Actions 2 through 4. **MODIFY** and **QUERY** operations are available to assist in any of the actions. Each of the four actions is discussed in the following sections. The four principal operations are also covered.

4.1.2.1 Specifying Class Attributes and Model Component Logic

Under **SPECIFY**, class information is specified (Action 1) by first identifying the component type of the particular class. Figure 4.10 displays how classes are specified as submodel classes, static object classes, etc. The class specification window (Figure 4.11) becomes available to a modeler. The class is named by typing in the name. Error checking on the name is performed (by pressing the <return> or <enter> key) to determine if the class name has been previously used for the component type. (If already

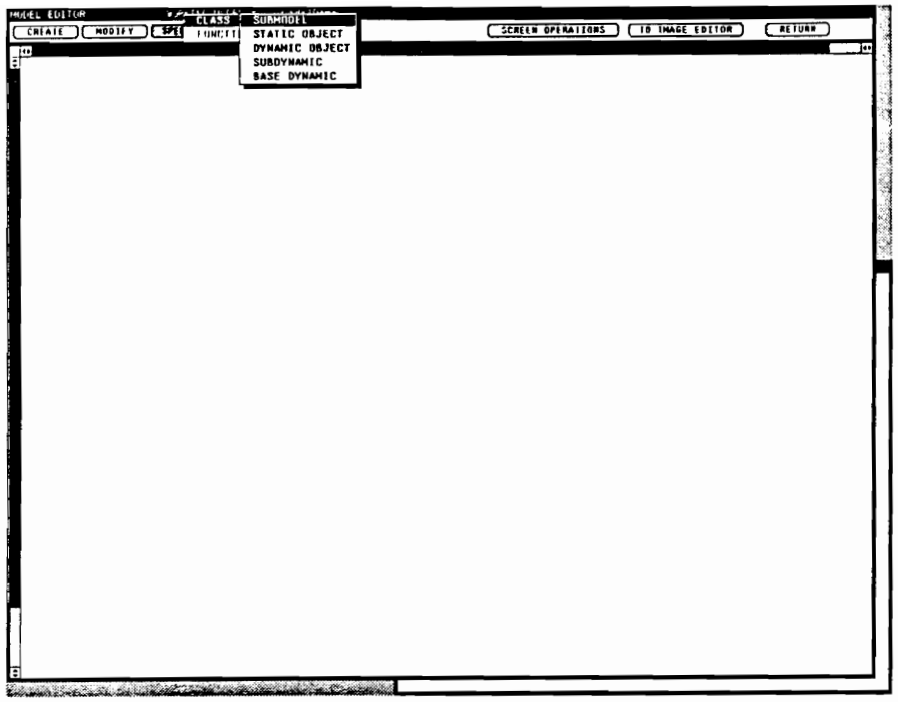

Figure 4.10 Specify Operation (Class by Component Type) of Model Editor

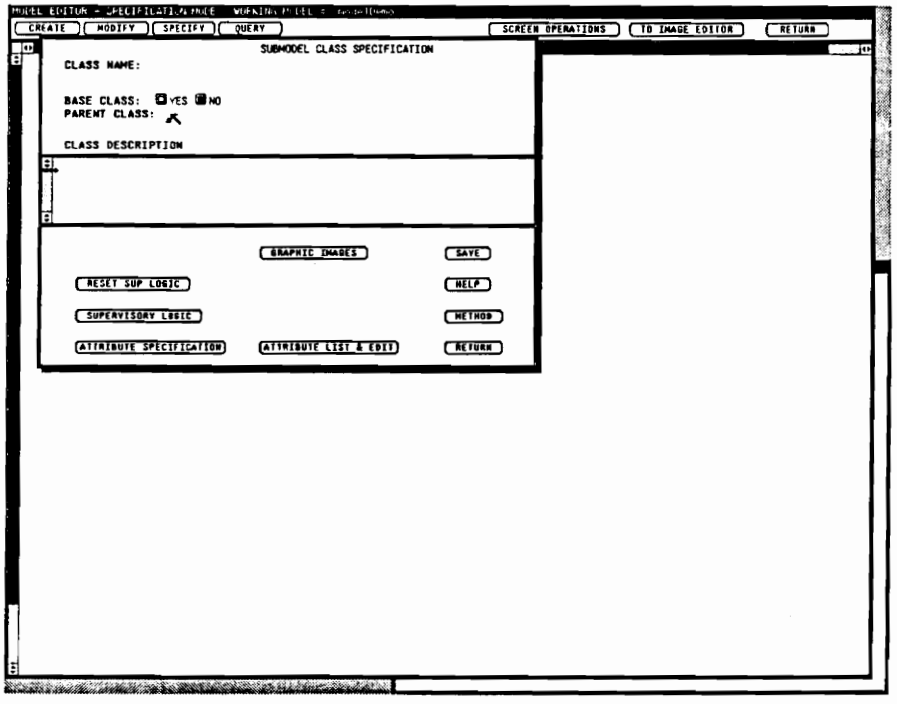

Figure 4.11 Class Specification Window

specified, a modeler may go ahead and retrieve the existing class information for editing; if not previously specified, the class name is zeroed for reentry of a new class name.) The base class toggle (default of "yes") is used to identify if the class is a base class or not. By clicking on "no", the parent class text item is displayed for typing in the parent class name. (If desired, the right mouse button displays a popup of available class names which can be selected.) The base/parent class information is required for inheritance purposes. Also, if not a base class, a **GRAPHIC IMAGES** button appears for indicating the owning class for which objects of the class (under specification) will inherit set or default graphics images. The class description area is a text subwindow in which documentation can be typed. Additionally, buttons are provided for listing and editing attributes (**ATTRIBUTE LIST & EDIT**), attribute (new) specification (**ATTRIBUTE SPECIFICATION**), and logic specification (**SUPERVISORY LOGIC, SELF LOGIC,** or **METHOD,** as applicable). The **RESET SUP LOGIC** button turns off any supervisory logic specification during execution as described in Chapter 3.

Attribute specification is done via the button of that name. Before attribute specification (and other actions), a class name must be first entered at the proper location in the class specification window. The attribute specification window (Figure 4.12) is similar to the class specification window. The name must be typed and is error checked (as demonstrated in Figure 4.12). Documentation can be entered. The attribute can be designated as an array; single dimension arrays are workable within VSSE. The attribute is data typed (using a popup on the right mouse button; Figure 4.13) as integer, long integer, floating point, double precision, character, time (system type) or system constant. Initial values are specified. The attribute information is saved to database using the **SAVE** button. Previously specified attributes can be edited (to include deletion) from the **ATTRIBUTE LIST & EDIT** feature on the class specification window. Figure 4.14 shows how these are selected for editing. Note also from this figure that for dynamic object classes, the **SELF LOGIC** feature is made available.

Model component logic specification is entered using the applicable button (**SUPERVISORY LOGIC, SELF LOGIC,** or **METHOD**). Like class and attribute names, VSSE checks for the existence of a previous logic specification for the class being worked. The logic specification window of Figure 4.15

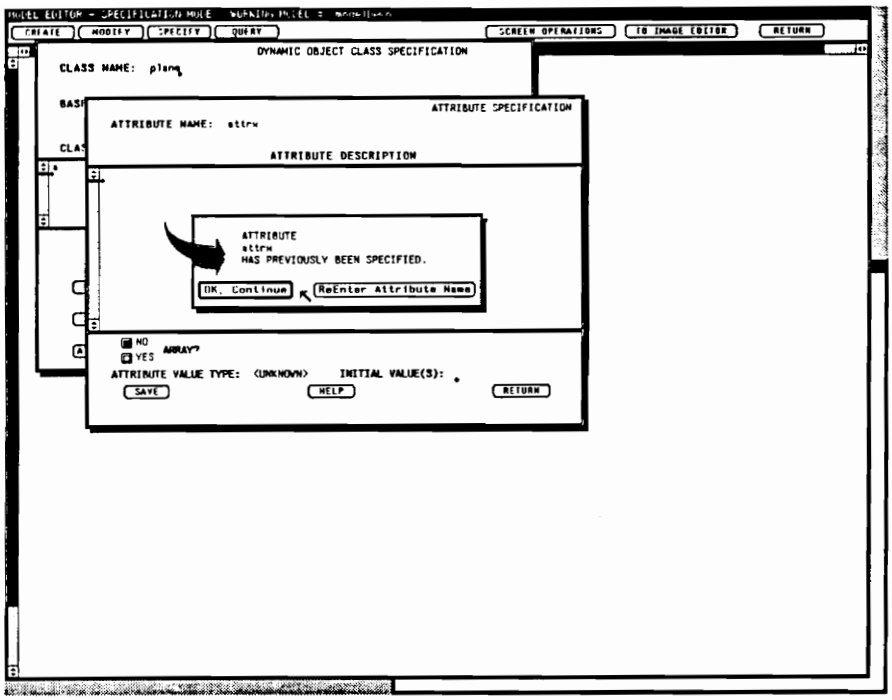

Figure 4.12 Attribute Specification Window

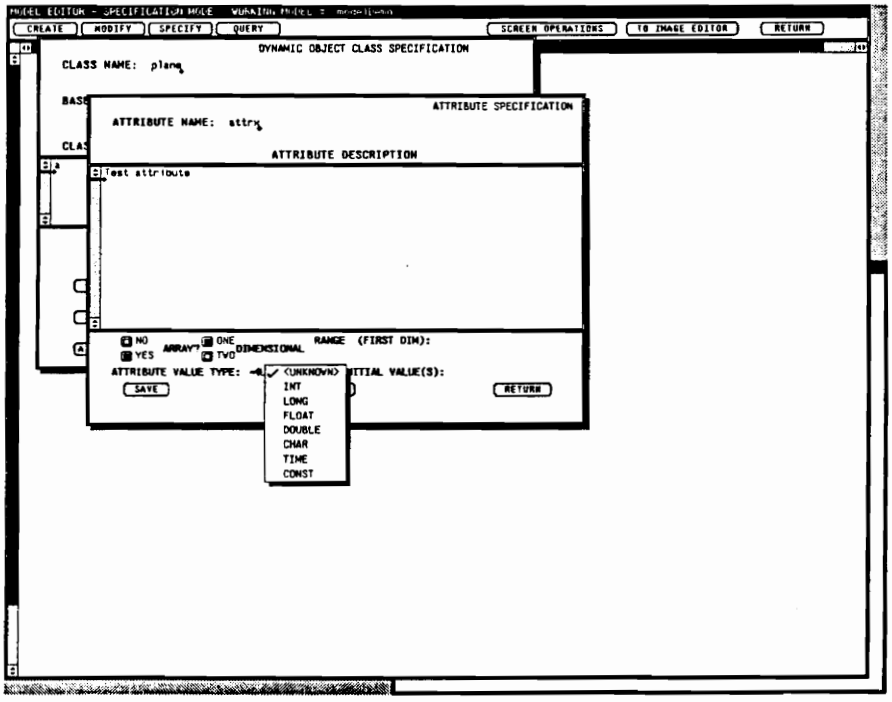

Figure 4.13 Typing Attribute Data (Popup)

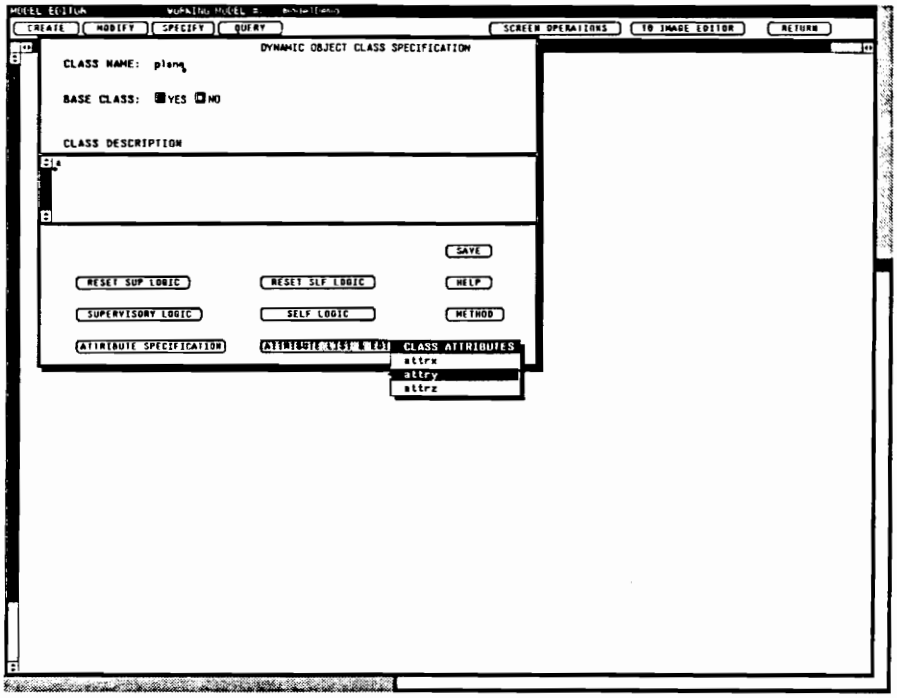

Figure 4.14 Attribute List and Edit

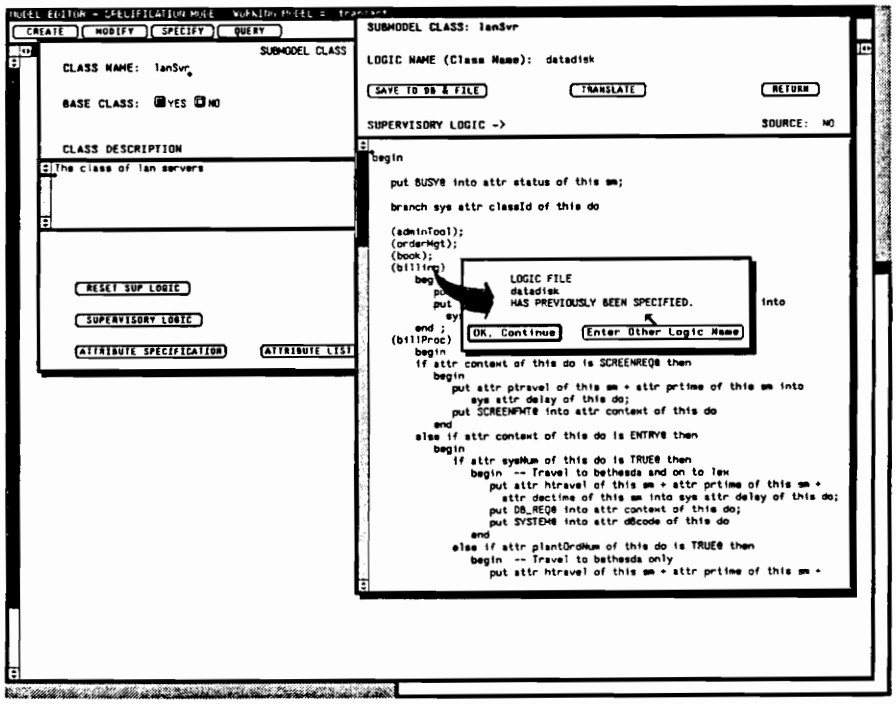

Figure 4.15 Model Component Logic Specification Window

is a text subwindow in which the modeler enters the model component logic using VSMSL (Visual Simulation Model Specification Language), described in Chapter 5. The **TRANSLATE** button activates a translator (built using LEX/YACC). Successful translation automatically produces the source code of the logic in the target language (C). During the translation process, symbol tables for the current class can be produced and the translator modified accordingly (Figures 4.16 and 4.17). With the symbol tables, the translator performs static analysis of the logic specification. Appropriate error messages are displayed and syntax errors identified. Figure 4.18 displays a successful translation. The popup in the top right of this figure provides a means for viewing the resulting source code. This was invaluable during system and translator development but is not required for the general modeler. Once translated, the modeler returns to the logic specification (reloading the text subwindow) by hitting the <return> or <enter> key. Of interest, any logic specification of the same component type can be loaded from the logic specification window. The logic name is typed and <return> is keyed to accomplish the load.

Supervisory, self, and method component logic specifications are all handled similarly. One exception is that method specification includes parameter specification. Parameters can be listed and edited like attributes or specified anew (Figure 4.19). In addition, methods require that a return type be indicated (methods often return data to the sender of the message which activates the method). Figure 4.19 shows the various method return types which are available. Parameter specification (Figure 4.20) includes naming, documentation, typing, and saving.

4.1.2.2 Instantiating Class Layouts: Components and Paths

Prior to performing the **CREATE** operation, the particular class layout image is first loaded from the Image Editor; the creation process from within the Model Editor can then proceed. The goal is to identify the class instances in each class layout image in a general way. This section describes Actions 2 and 3 as discussed earlier in Section 4.1.2.

In Figure 4.21, notice the choices of **REAL** or **VIRTUAL**. In the real case, class layouts can be static or dynamic (associated with the model static structure or dynamic structures). On top of the class

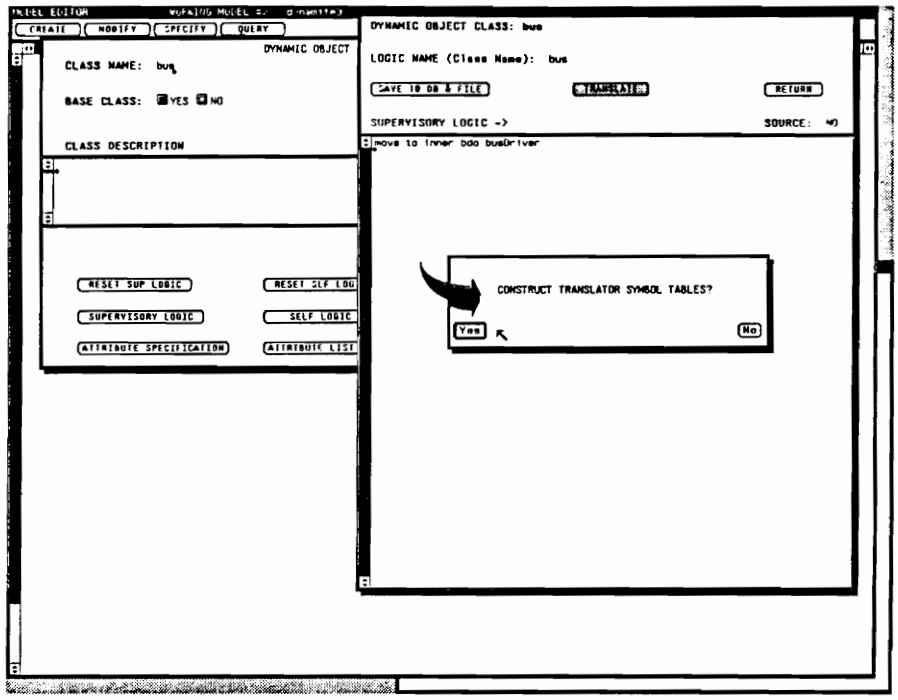

Figure 4.16 Symbol Table Construction

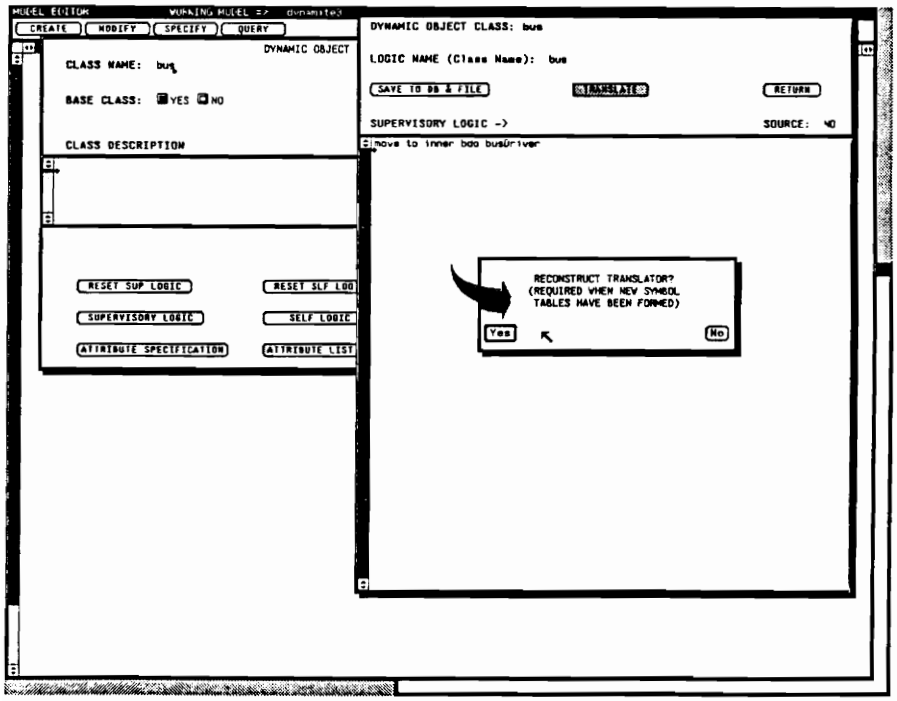

Figure 4.17 Translator Reconstruction

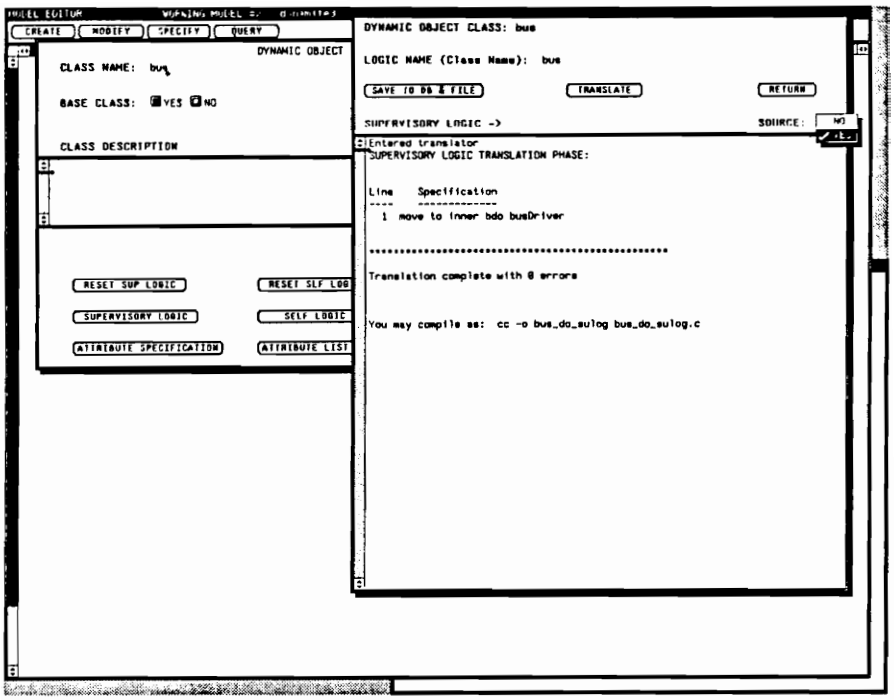

Figure 4.18 Translating Model Component Logic

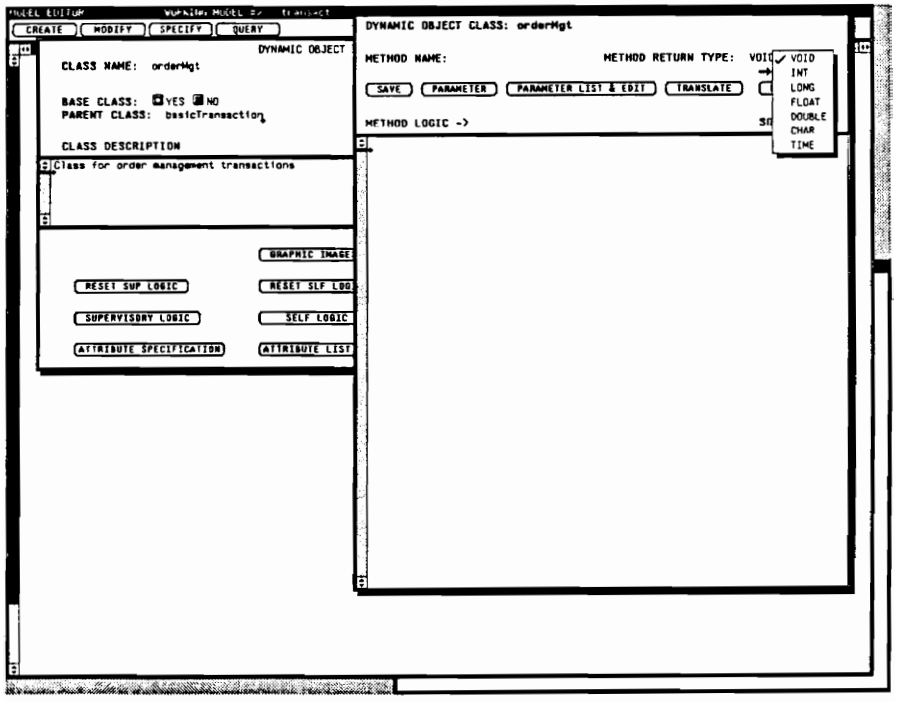

Figure 4.19 Method Logic Specification Window

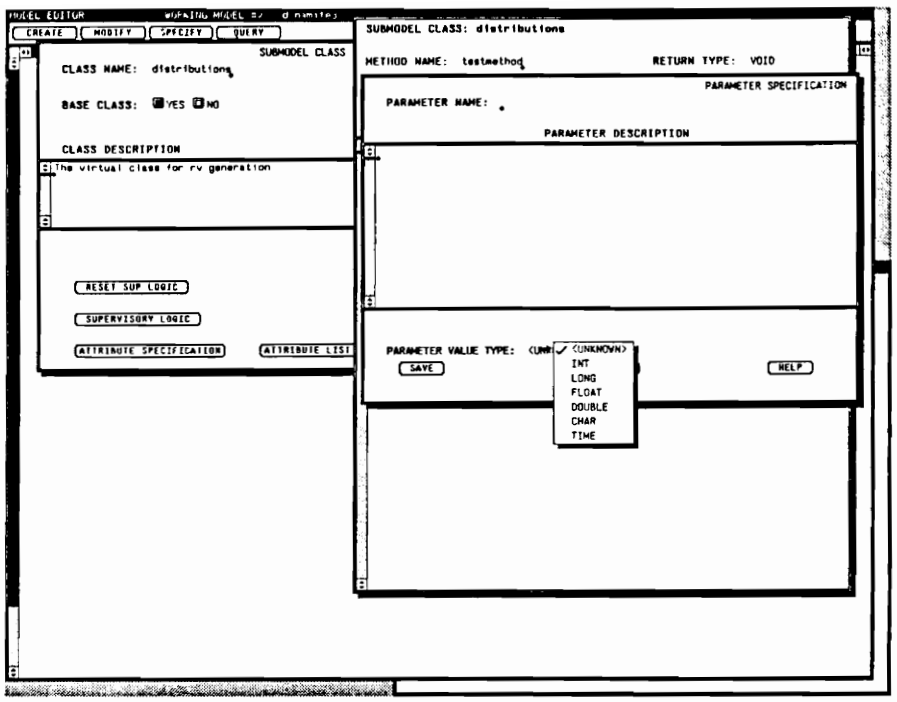

Figure 4.20 Parameter Specification Window

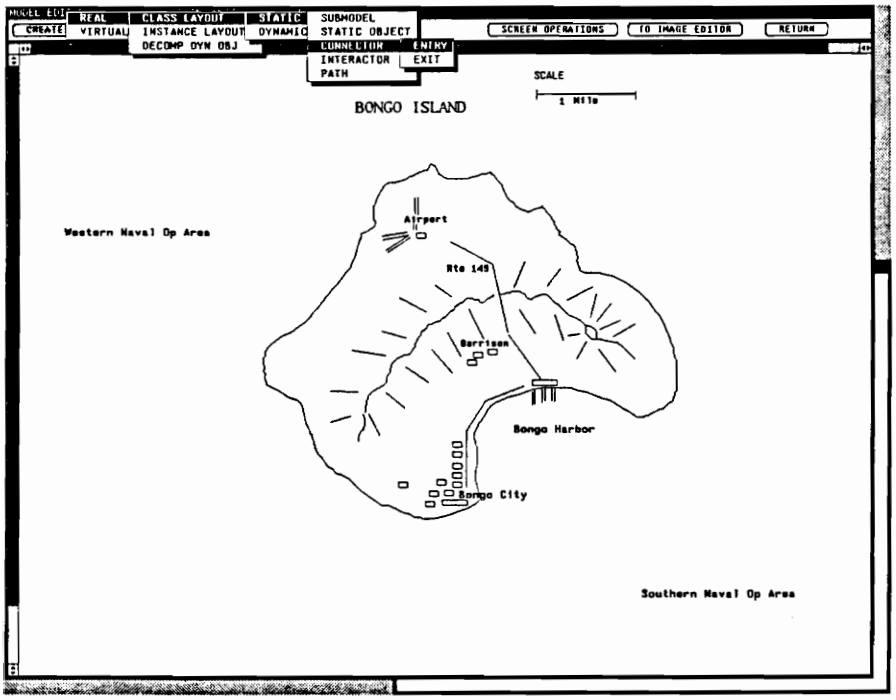

Figure 4.21 Create Operation (Real, Static Class Layouts) of Model Editor

layout image, we identify component locations: submodels and static objects for the static structure or subdynamic objects and base dynamic objects for dynamic structures (Figure 4.22). The locations are set using the mouse (left button) to define a rectangular area in the desired component position (e.g., defining airport submodel component location in Figures 4.23 and 4.24). A modeler must give the component a variable name; this allows the class layouts to be reusable as described in Chapter 3 and as demonstrated in Chapter 7. The status of progress in a class layout can be checked using the **COMPONENT STATUS** button (Figure 4.25).

Paths for dynamic object movements, connectors, and interactors are included as applicable. Connectors, interactors, and paths are automatically named by the system. Paths are drawn (Figures 4.26 and 4.27) using the left mouse button to set the path line from a component to another. Intermediate path points (between components) are set by additional clicks on the left mouse button. Once the path is in the destination component, the middle button terminates path drawing. Connector and interactor positions are set in the same way as component locations (Figure 4.24) with a defining rectangle. Interactors must be associated with a static or base dynamic object (Figures 4.28 and 4.29).

Figures 4.30 and 4.31 show completed class layouts (static and dynamic; general instantiation) with component locations (in rectangles) and paths (lines) displayed. During animation, these instantiation lines are not shown. The completed class layout in this form is called the layout definition. With the **SCREEN OPERATIONS** button, the layout definition can be saved to file or database or loaded from file or database. During the save of a layout definition, the definition must be associated with its particular layout image by name. This is easily done via the interface. Loading a layout definition will draw the location rectangles and path lines only. To view the associated layout image, a modeler can toggle to the Image Editor (using **TO IMAGE EDITOR**) to load the particular layout image. Upon toggling back to the Model Editor, both the layout image and its definition are displayed simultaneously.

4.1.2.3 Forming Model Static and Dynamic Structures

Once the class layouts have been generally instantiated, then the model structures are instantiated in

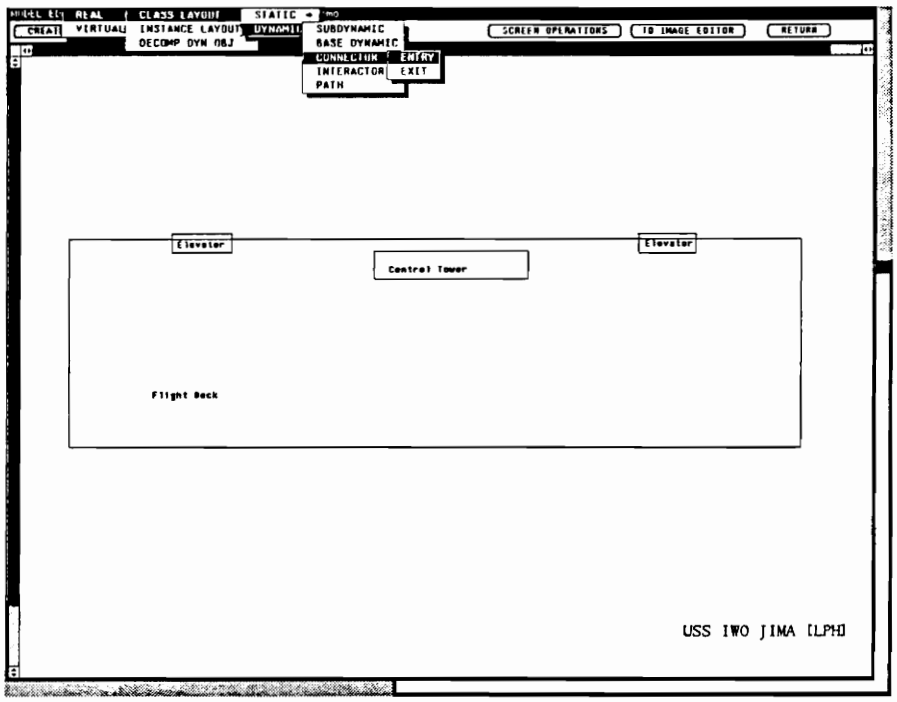

Figure 4.22 Create Operation (Real, Dynamic Class Layouts) of Model Editor

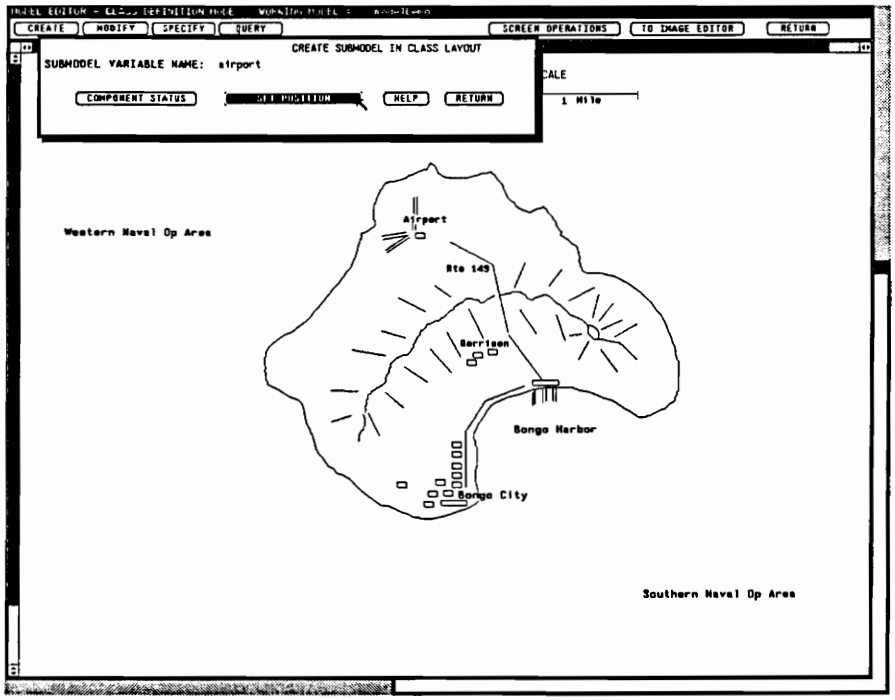

Figure 4.23 Activating Definition of Model Component Location

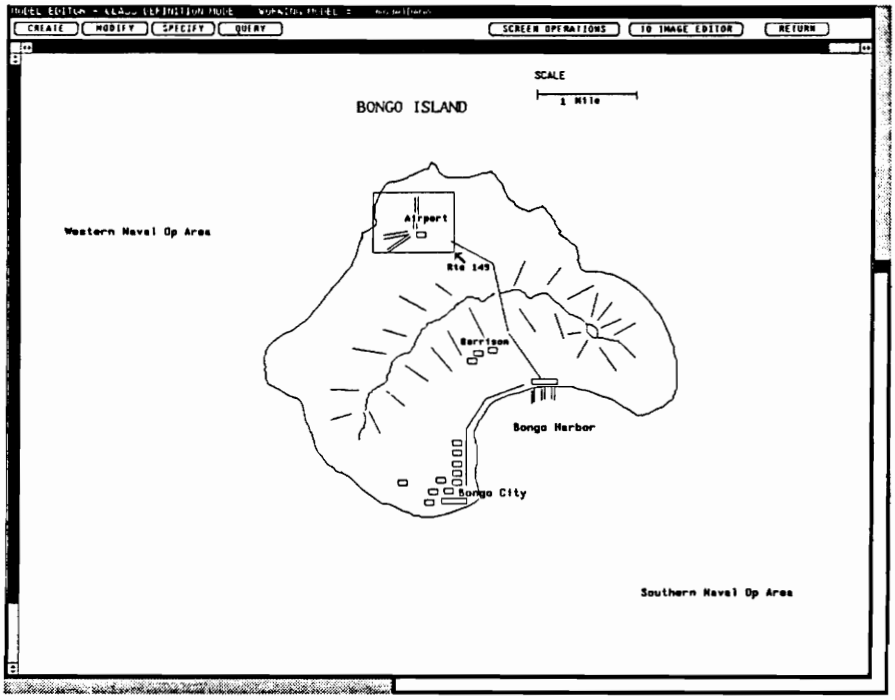

Figure 4.24 Defining Component Location as Rectangular Area

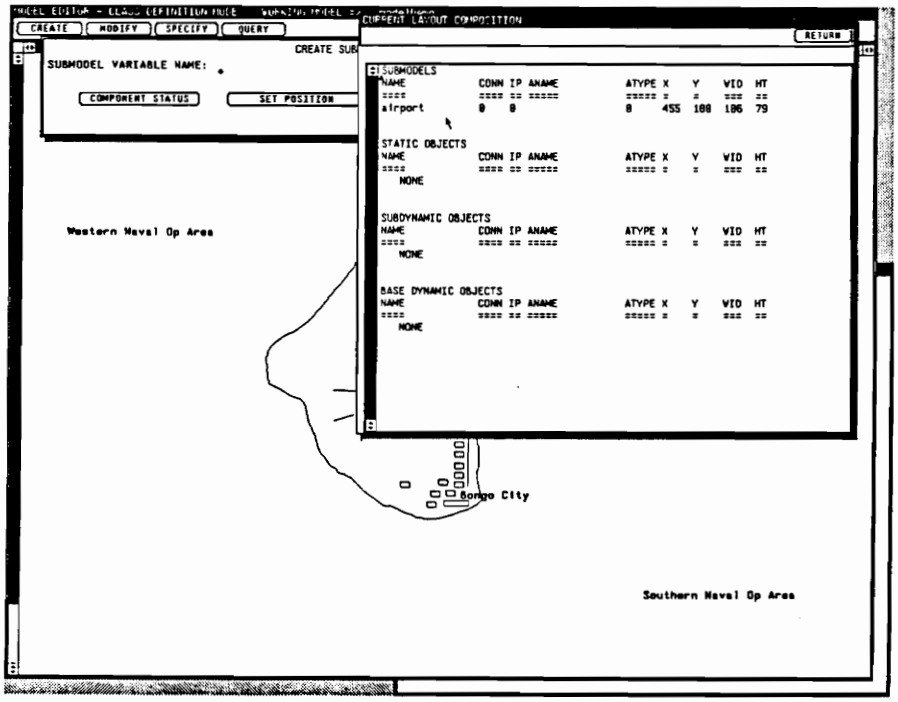

Figure 4.25 Component Status in a Layout Definition

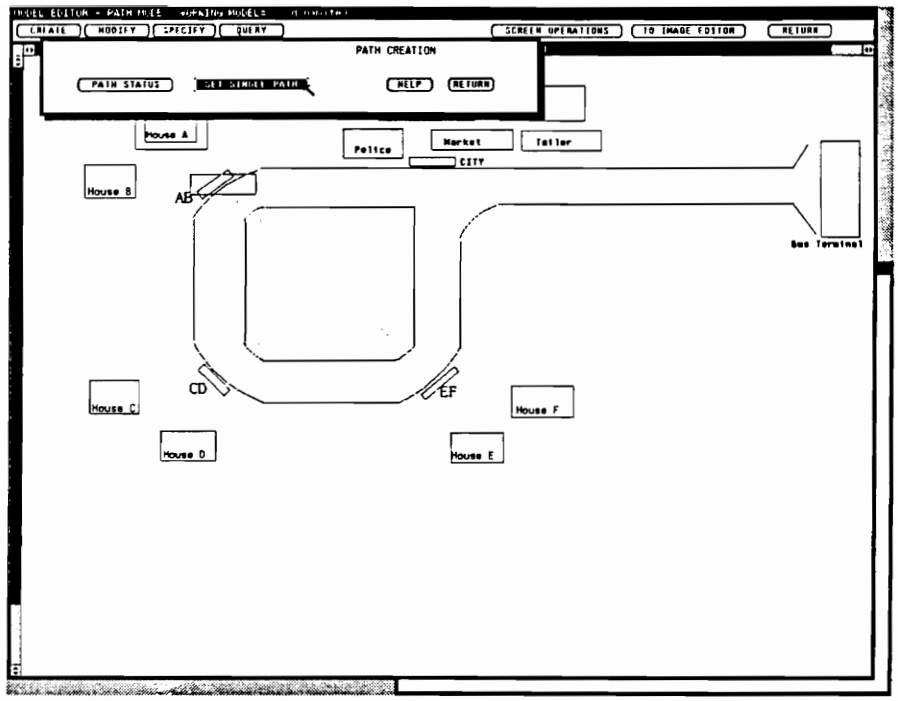

Figure 4.26 Activating Definition of Paths between Model Components

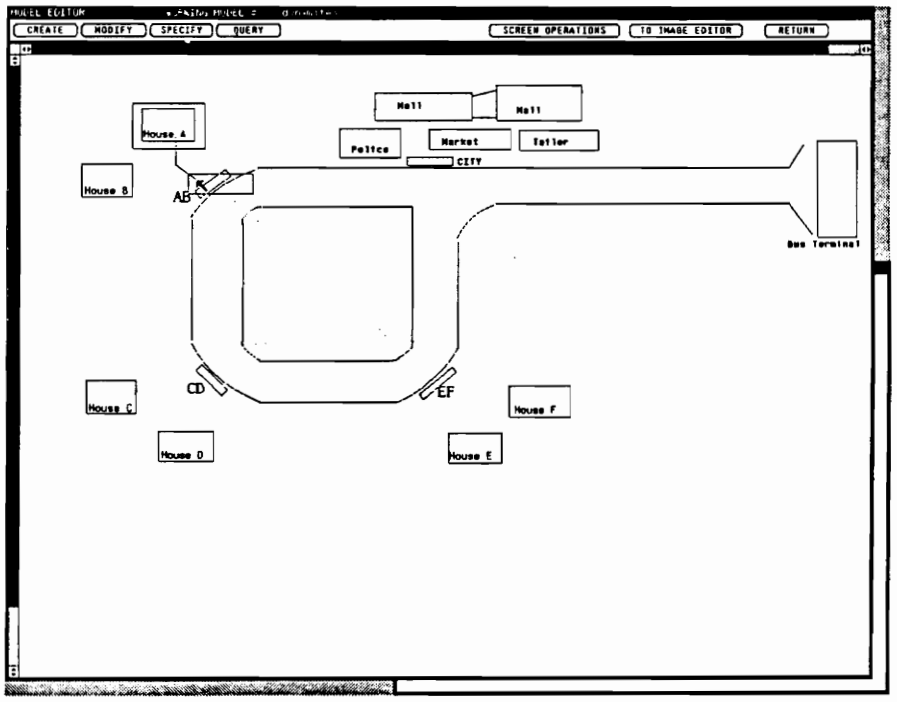

Figure 4.27 Defining Movement Path as Line Segment between Components

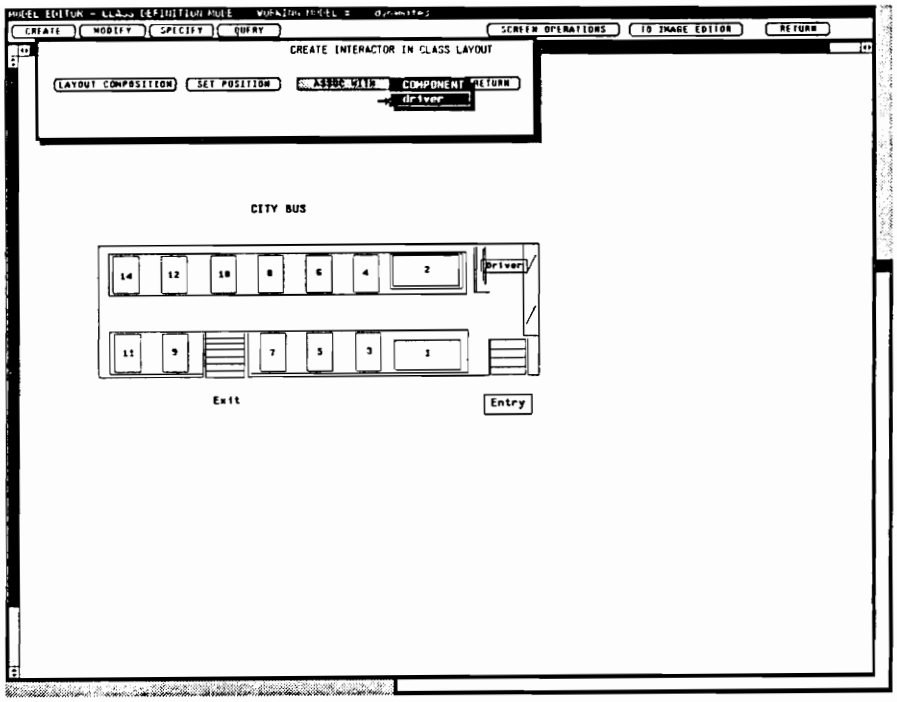

Figure 4.28 Associating Interactor with Static or Base Dynamic Object

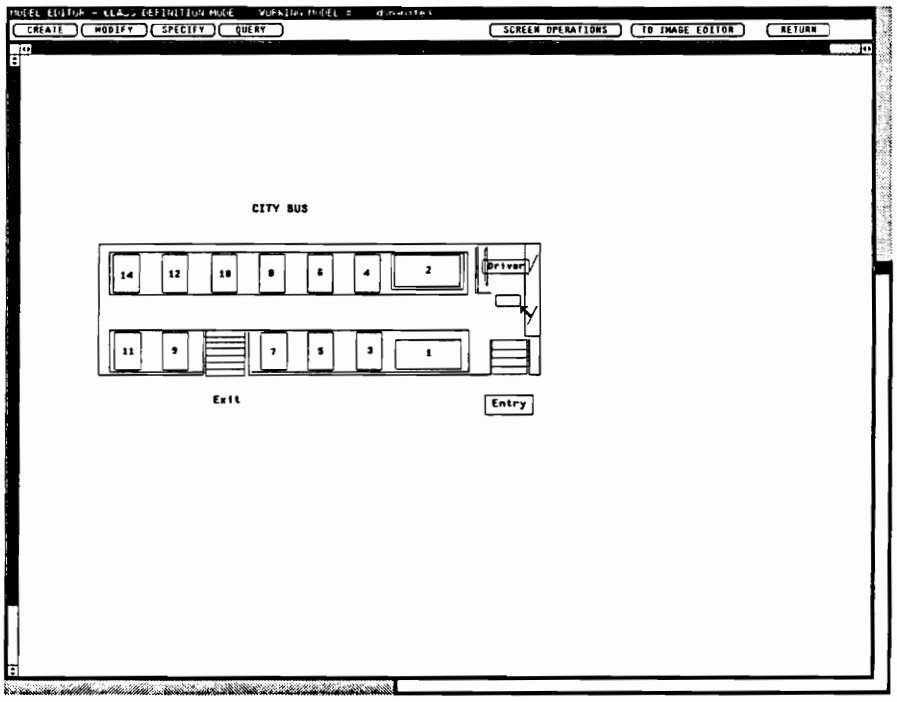

Figure 4.29 Defining Interactor Location as Rectangular Area

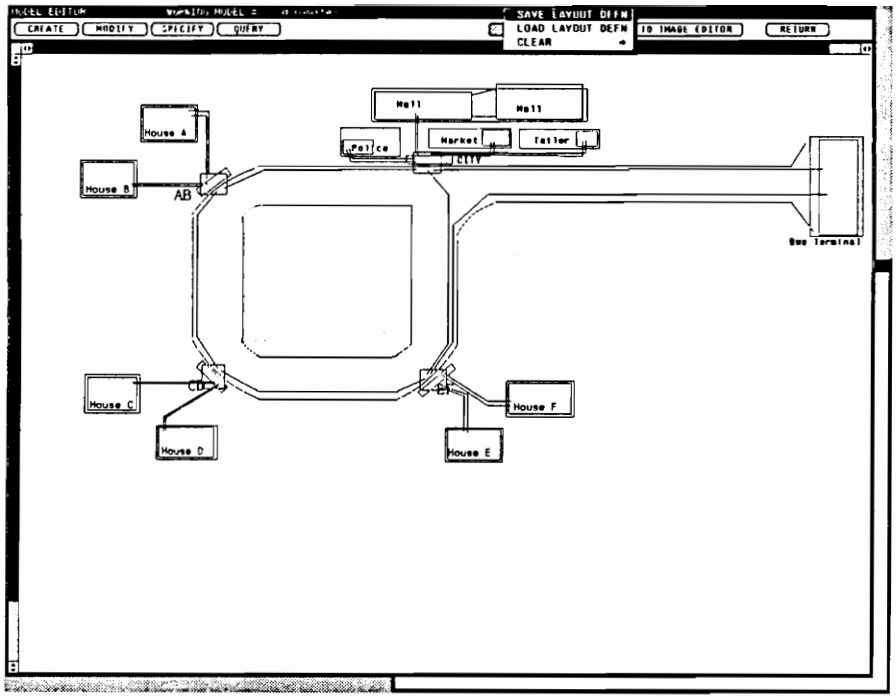

Figure 4.30 Layout Definition of Top Level of Bus Route Example Model Application

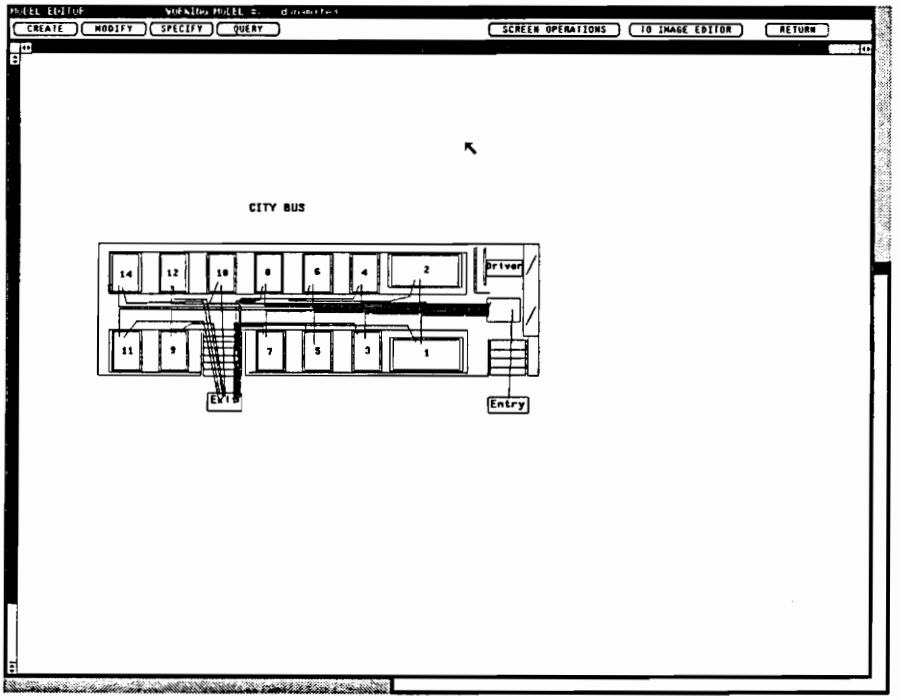

Figure 4.31 Layout Definition of City Bus (Bus Route Example)

a specific way using the **INSTANCE LAYOUT** (Figure 4.32). The following discussion concerns Action 4 and applies to static or dynamic structures. A static structure is used in this explanation but the VSSE provides for the selection of either structure type. Here, we build the hierarchical architecture of the model static and dynamic structures in a top down fashion. Class layouts are instantiated at each decomposition level, giving member components a unique name (literal name). As the instantiation of each decomposition level is completed, we graphically "stack" these instance layouts to form the static or dynamic structure.

We begin at the top level of the structure. The root class layout image and layout definition of the structure (e.g., model static structure in this example) is loaded using the **TOP** button. A display like Figure 4.33 results. The components in the root class layout are first literally instantiated. By clicking on any component in the class layout (using the right mouse button), a popup with three options (**CREATE, DECOMPOSE,** and **DESCEND**) is displayed (Figure 4.34). The blackened option is active. For instantiation, the create/decompose/descend ordering is enforced; creation must come first. In the darkened top portion of the popup, the top component name is the variable name of the component in the class layout.

With the **CREATE** option (on the popup), the modeler assigns the literal (unique) name to the component (Figure 4.35), identifies the class of the component and any documentation, and then saves the component literal instantiation to the database. After component creation, the literal name is displayed in brackets below the variable name (in the popup; Figure 4.36). Once a component is created, the **CREATE** option in its popup is grayed out and the next option (**DECOMPOSE**) is activated and darkened.

During the decomposition (using **DECOMPOSE**), the next decomposition level class layout is indicated (Figure 4.37). After this, the indicated class layout (the decomposition level) is displayed as shown in Figure 4.38 (following Figure 4.37). At each new level in the structure, the create/decompose/descend options are available. A modeler must create/instantiate all the components at every new level but has the flexibility in how deep to take the decomposition hierarchy.

Once decomposed, one can **DESCEND** (from the popup) into that new level (Figure 4.39). Navigation around the structure is easily accomplished. The **ASCEND** button across the top row of

Figure 4.32 Create Operation (Real, Static Instance Layouts) of Model Editor

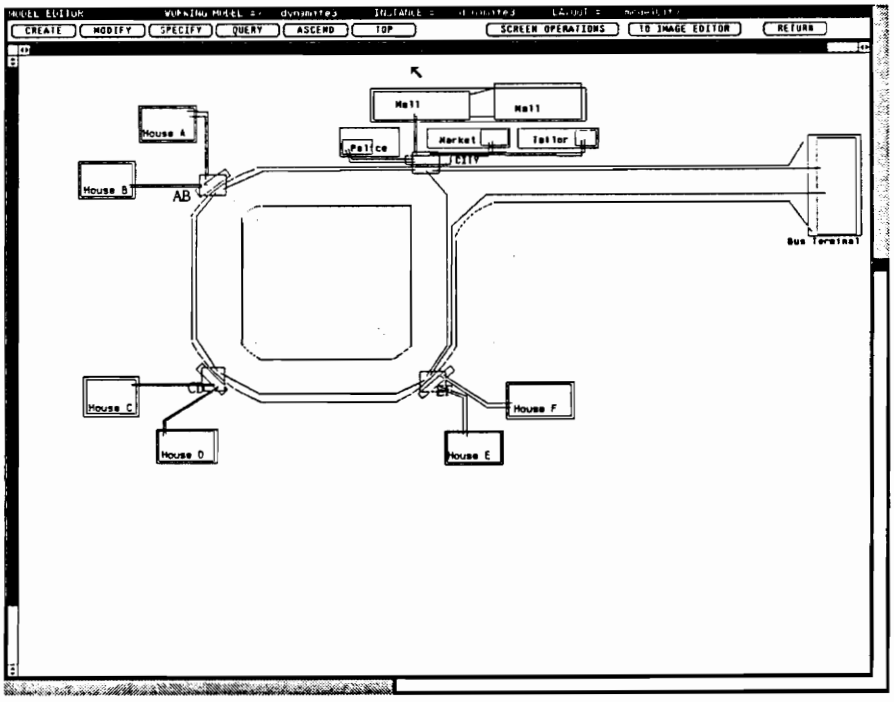

Figure 4.33 Beginning Instantiation of Instance Layout

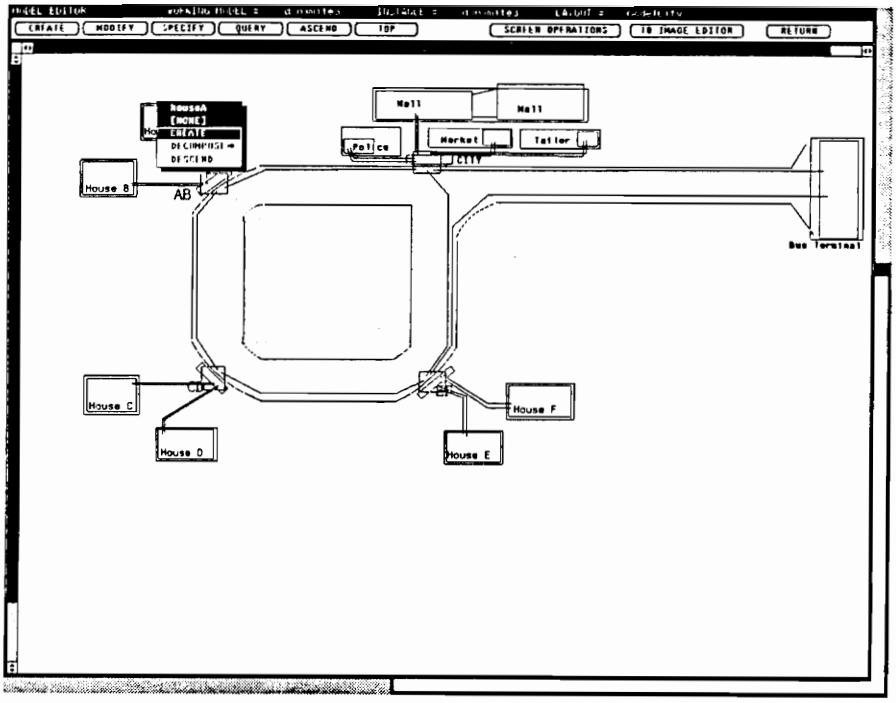

Figure 4.34 Instantiation Popup (Create is Active)

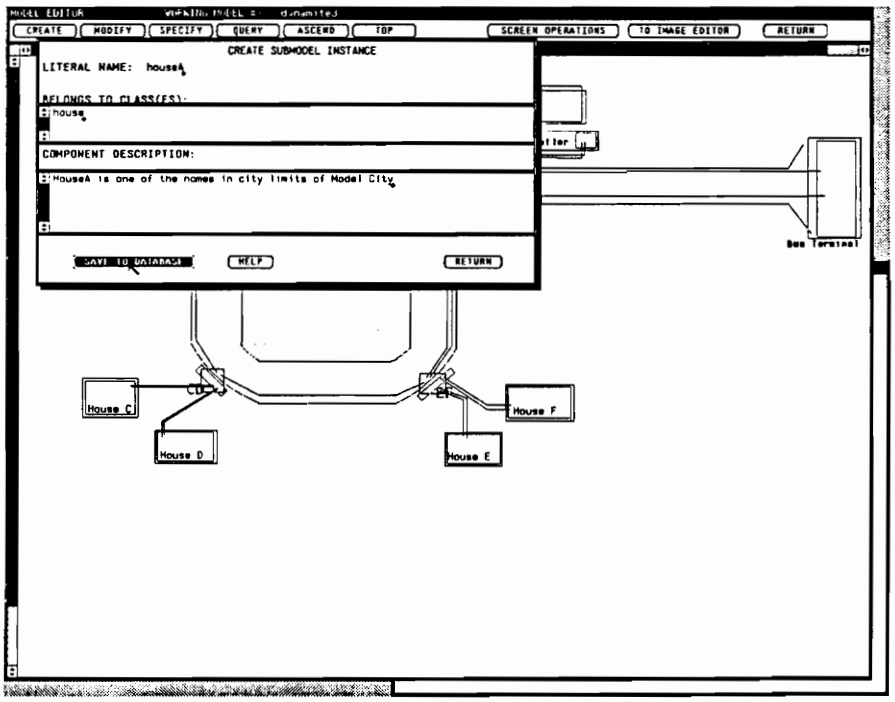

Figure 4.35 Create Window for Component Instances

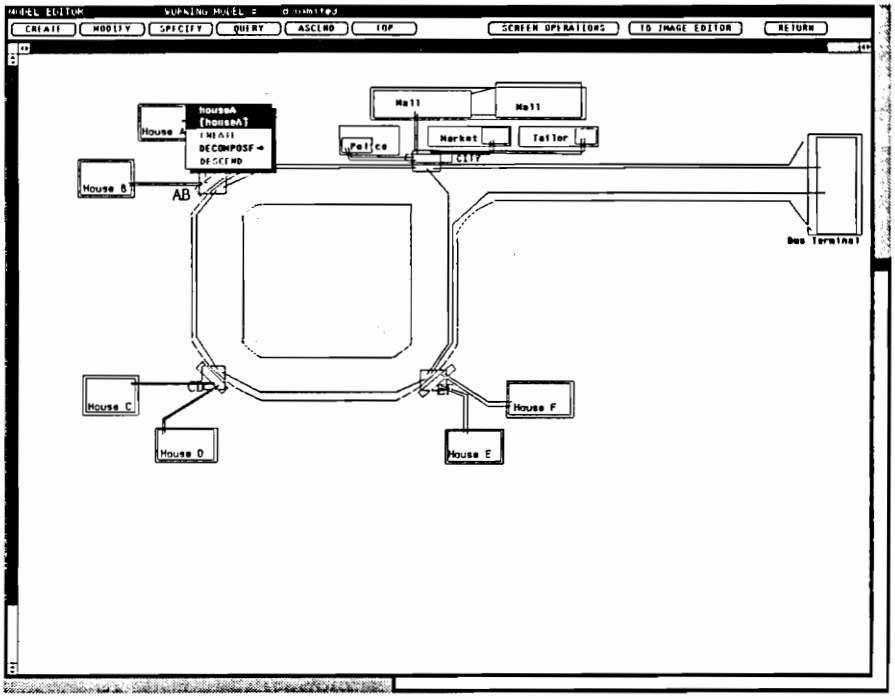

Figure 4.36 Instantiation Popup (Decompose is Active)

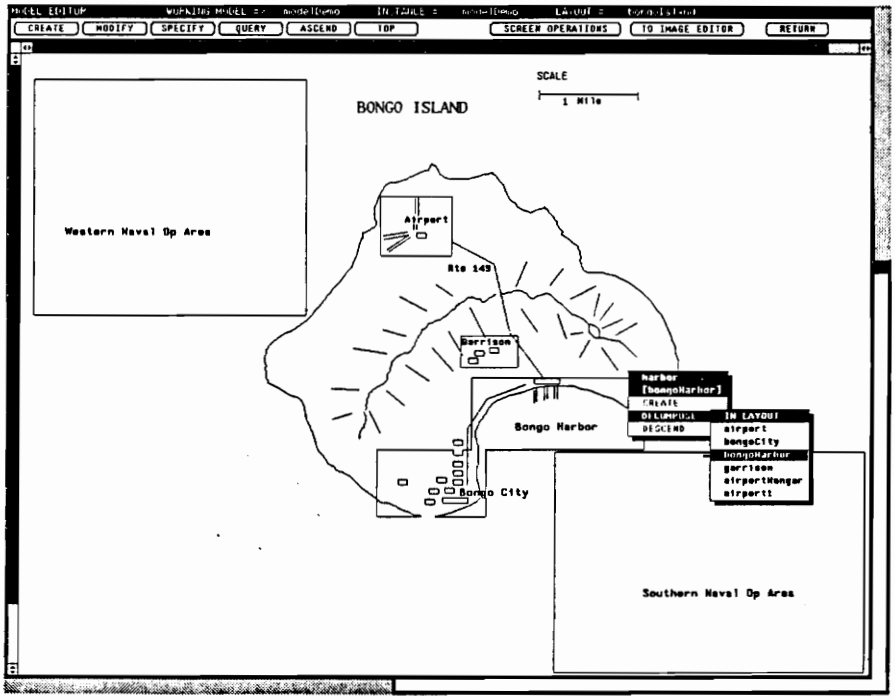

Figure 4.37 Decomposing a Component, Associating a Layout

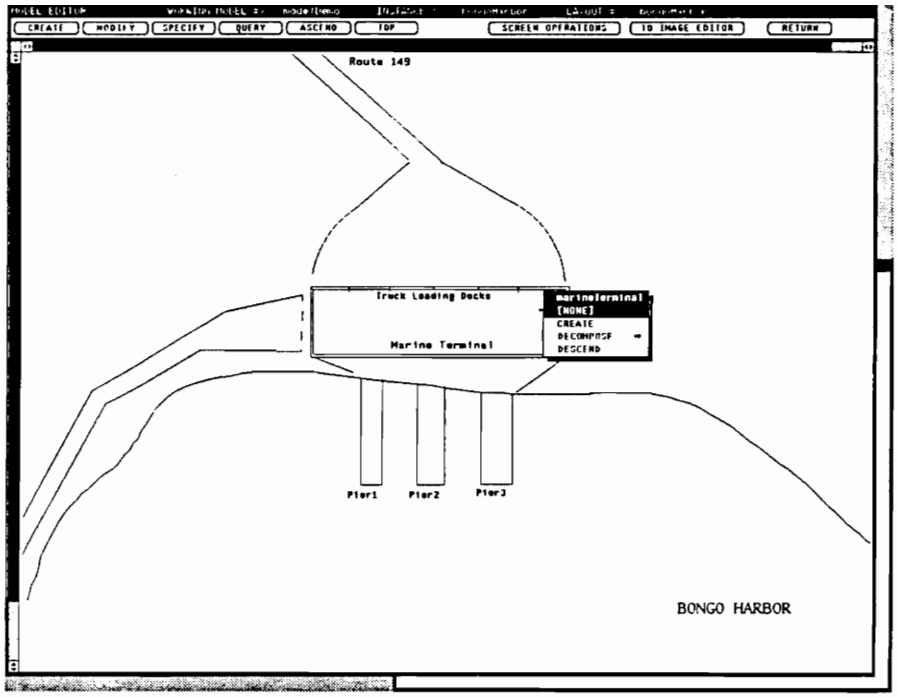

Figure 4.38 Result of Activating Decomposition to Next Level

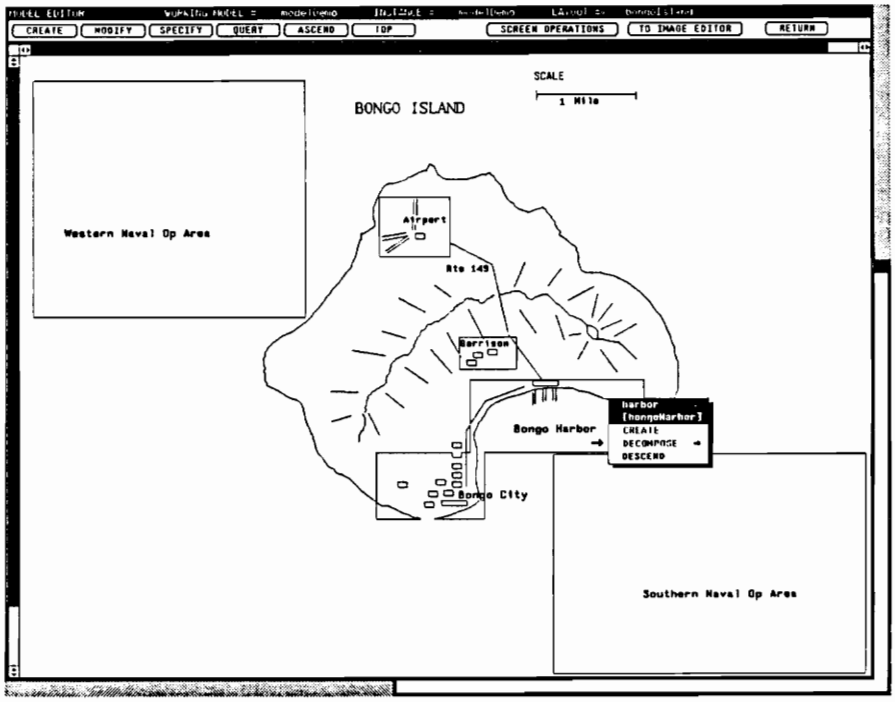

Figure 4.39 Instantiation Popup (Descend Active)

buttons moves the instantiation context up one decomposition level. If at a deep level in the structure, the **TOP** button (on the top row), takes the modeler to the root context of the structure. The combination of descend/ascend/top features gives extreme flexibility and ease of modeler definition. The structure doesn't have to be defined level by level; the points of decomposition can be chosen at the modeler's discretion.

4.1.2.4 Creating Decomposed Dynamic Objects and Virtual Components

Decomposed dynamic objects (i.e., the root components of dynamic structures) are created via the **DECOMP DYN OBJ** button as shown on Figure 4.32. Selecting that button results in the window of Figure 4.40. A modeler indicates the object's literal name, the decomposition class layout that it owns (right mouse button provides a popup listing possible layouts for selection), class information, documentation, and saves the literal instantiation to the database. In order to form the dynamic structure around this decomposed dynamic object, these steps must be completed prior to those of Section 4.1.2.3.

Virtual components (submodels, only) are created via the **CREATE/VIRTUAL/SUBMODEL** sequence of pullright menus of Figure 4.41. The creation window for virtual submodels (Figure 4.42) contains entries which must be entered or performed by the modeler: name, class information, documentation, saving.

4.1.2.5 Querying the Model Database

At any point in definition and specification, the **QUERY** operation (from the Model Editor top level operations) is available. **QUERY** (Figure 4.43) allows you to inspect the specification progress, check results in the database, or query any aspect of the model which has already been specified. Aspects available for query are information regarding all database, instances, classes, attributes, methods, parameters, graphics, hierarchies, and files. The queries for the instance through parameter queries can be limited by component type (submodel, static object, dynamic object, etc.). Figure 4.44 is an example query of submodel class information (performed during class specification). A graphics query can be oriented to component image, layout image and definition, layout member, or layout path information

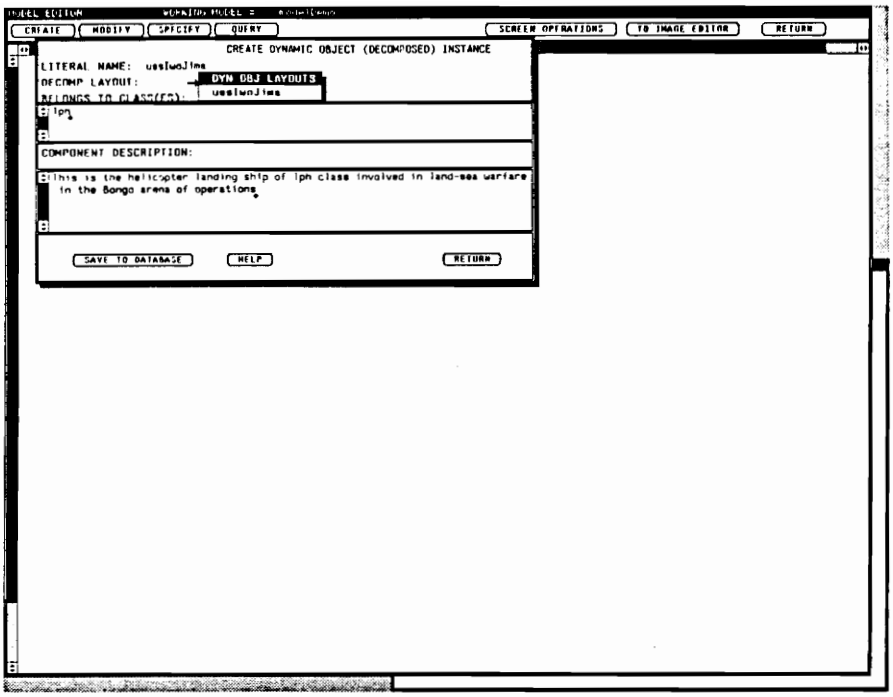

Figure 4.40 Create Window for Decomposed Dynamic Object

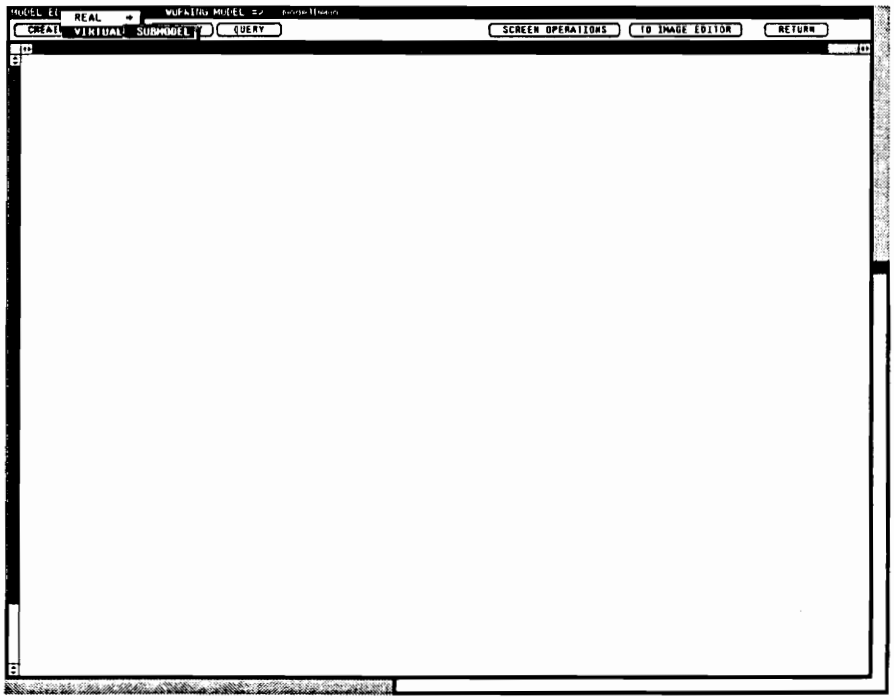

Figure 4.41 Activating Creation of Virtual Submodels

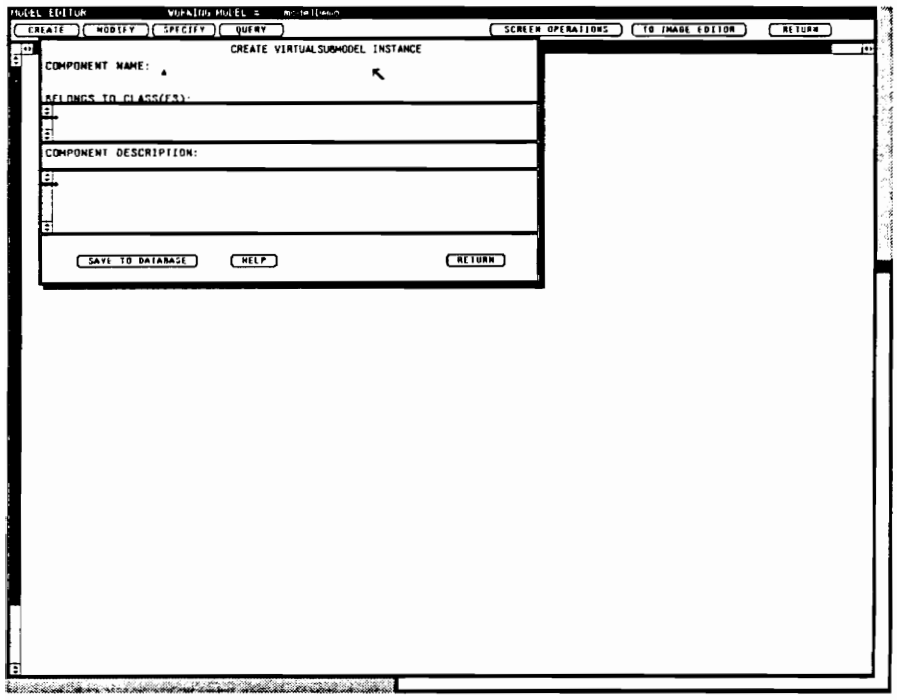

Figure 4.42 Create Window for Virtual Submodels

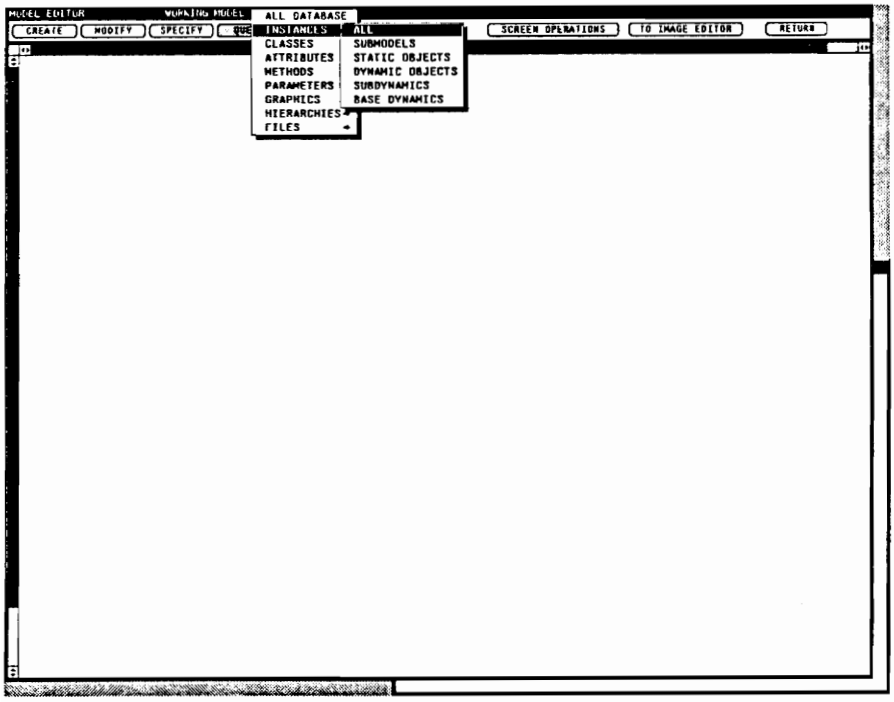

Figure 4.43 Query Operation of Model Editor

(Figure 4.45). Under hierarchy queries, a modeler can view class inheritance hierarchies or model static and dynamic structure hierarchies. Figures 4.46 and 4.47 demonstrate the class hierarchy query. A model dynamic structure query is shown in the sequence of Figures 4.48 and 4.49. Additional examples of these hierarchy queries are displayed in Chapter 7. Finally, a files query (Figure 4.50 and 4.51) can produce a report of model code files or files of graphics images (Figure 4.52).

4.1.2.6 Modifying the Model Database

The **MODIFY** operation is also available at any time during model definition and specification. With it, a modeler can selectively delete information which has been previously stored to the database or to system files. **MODIFY** (Figure 4.53) can be performed on instance, class, attribute, method, parameter, graphics, and files information. Some modifications (instance through parameter) are limited by component type (just like queries). Figure 4.54 gives an example modification window for submodel class. A popup using the right mouse button provides easy and quick selection of class type. Selecting the **DELETE** option causes a confirmation message to be displayed. Confirmations are the "norm" for all deletions. Graphics modifications are limited to component image, layout image and definition, layout member, and layout path information (Figure 4.55). Graphics modification windows are similar to Figure 4.54. Modification of files takes several forms. Figures 4.56 through 4.58 show the sequence for selection of the model code (submodel) file **branch_sm_sulog** (a supervisory logic file), its display, and attempted deletion. Files can be edited here as well and saved in their modified form. Similarly, graphics files (definition - Figure 4.59 and layout or component images - Figure 4.60) can be deleted. Definition files can be viewed and deleted; image files can only be deleted.

4.2 The Model Analyzer

The Model Analyzer provides a modeler with the means to perform consistency and completeness checks on various aspects of the model specification by scanning the database. Activated by the Model Analyzer icon on the top level VSSE display (Figure 4.61) and the **Analyze** button, the Analyzer window

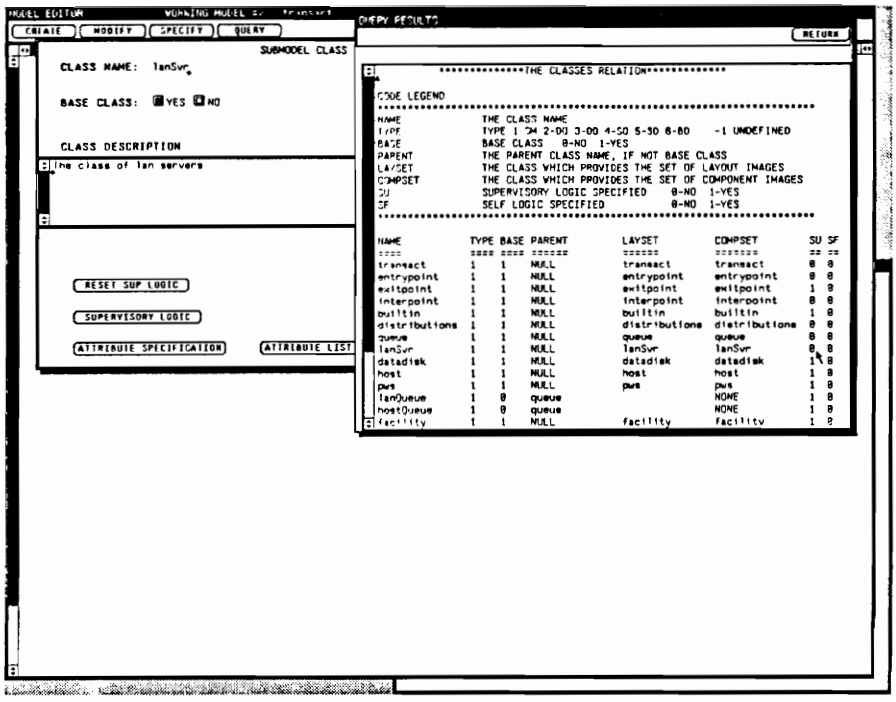

Figure 4.44 Result of Class Query

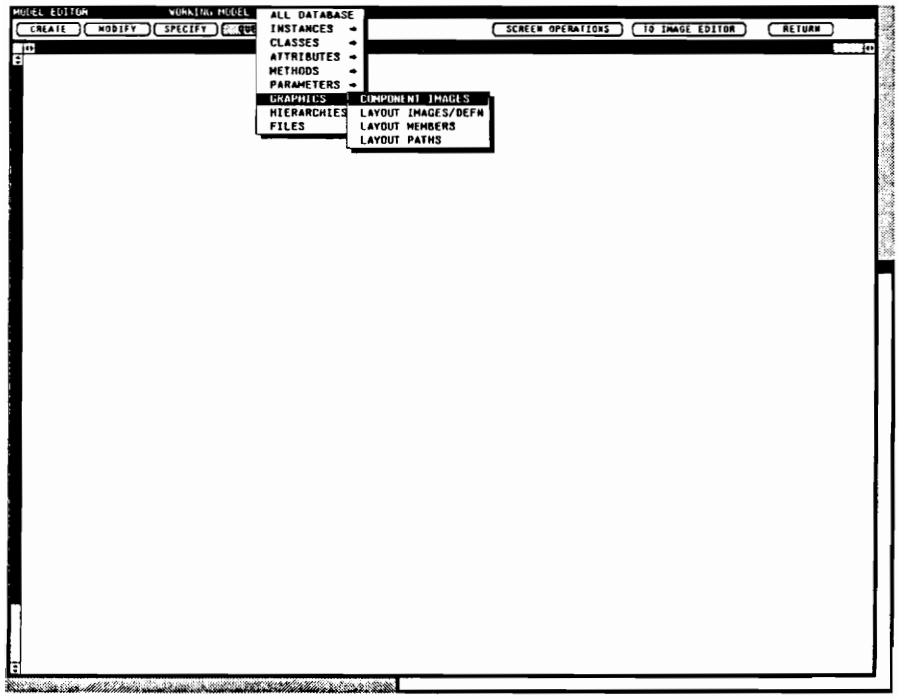

Figure 4.45 Activating Graphics Query

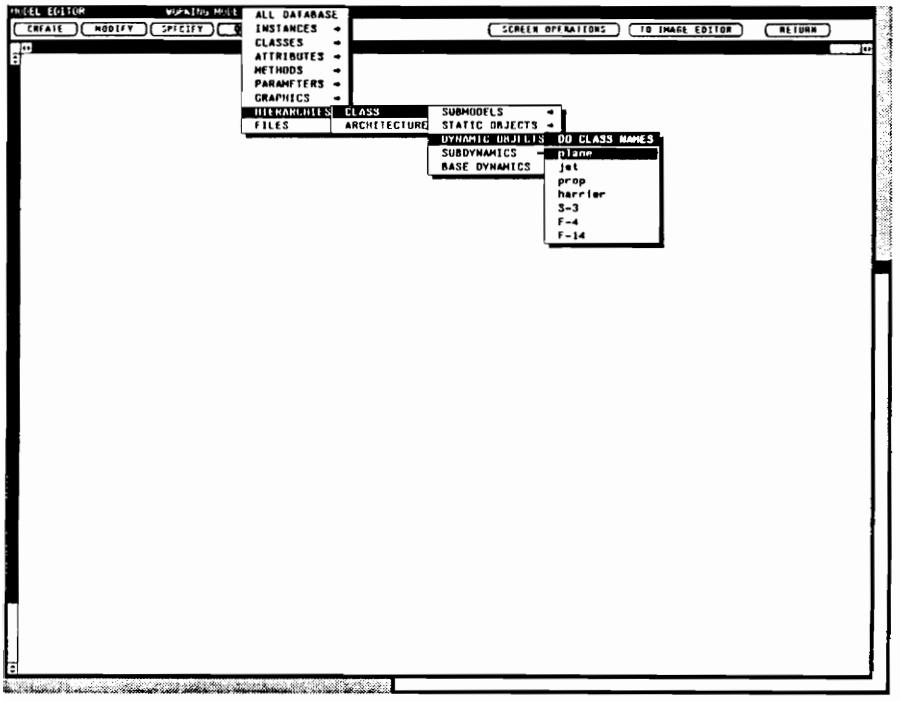

Figure 4.46 Activating Class Hierarchy Query

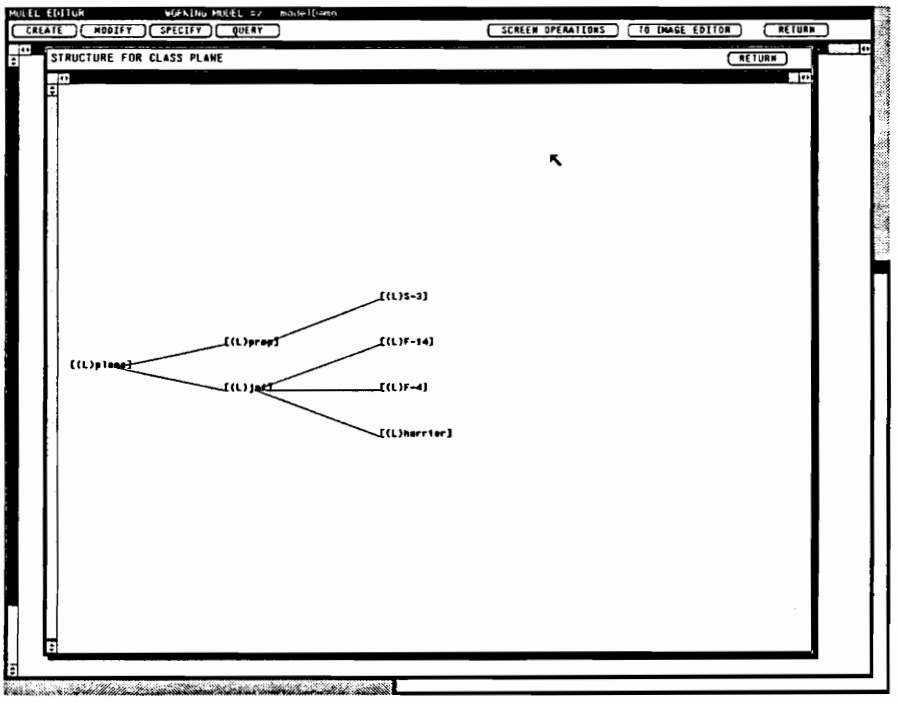

Figure 4.47 Result of Class Hierarchy Query

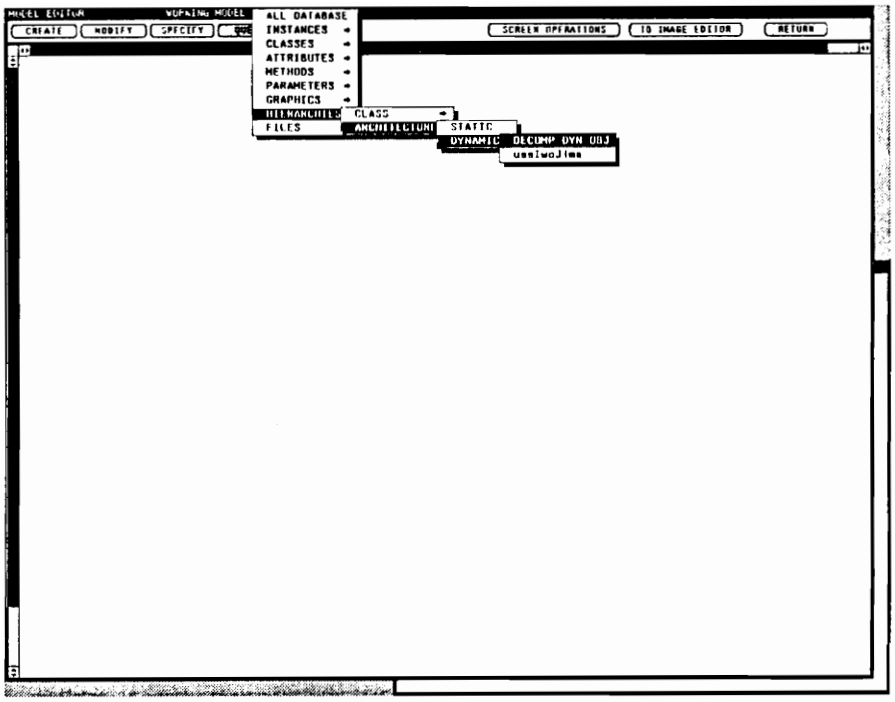

Figure 4.48 Activating Architecture Hierarchy Query

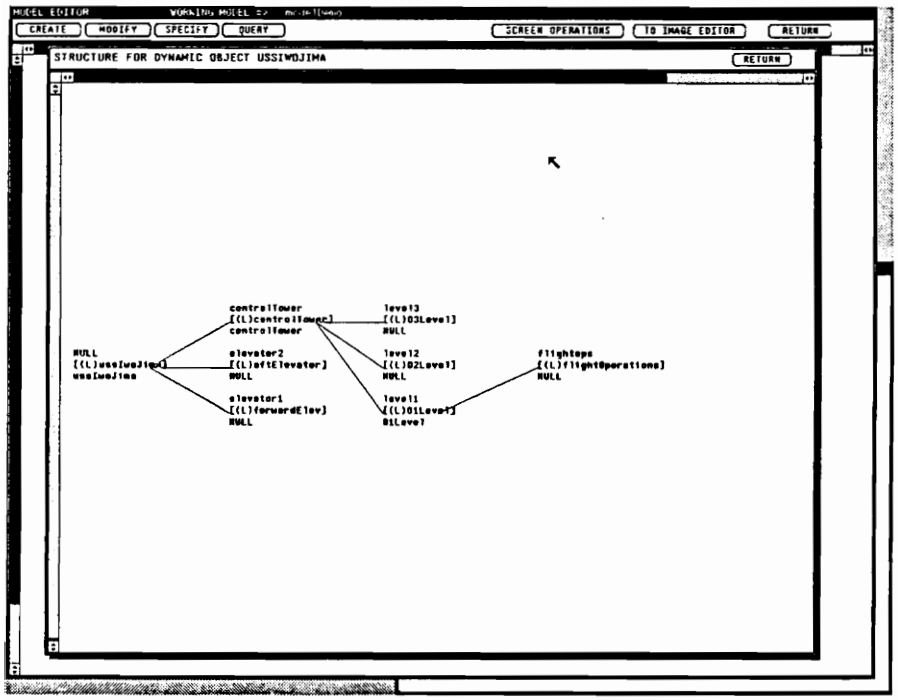

Figure 4.49 Result of Architecture Hierarchy Query

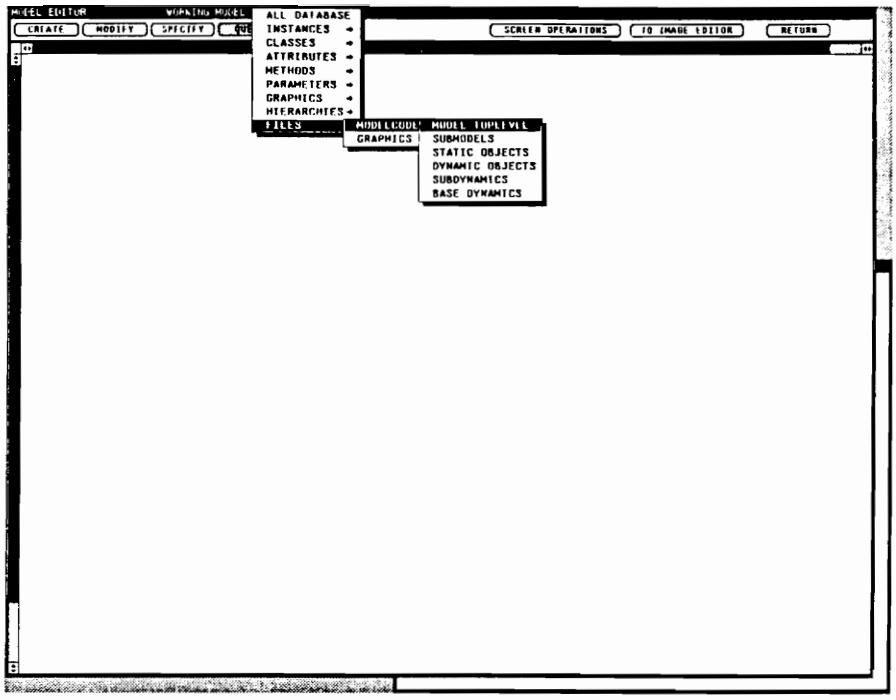

Figure 4.50 Activating Model Code File Query

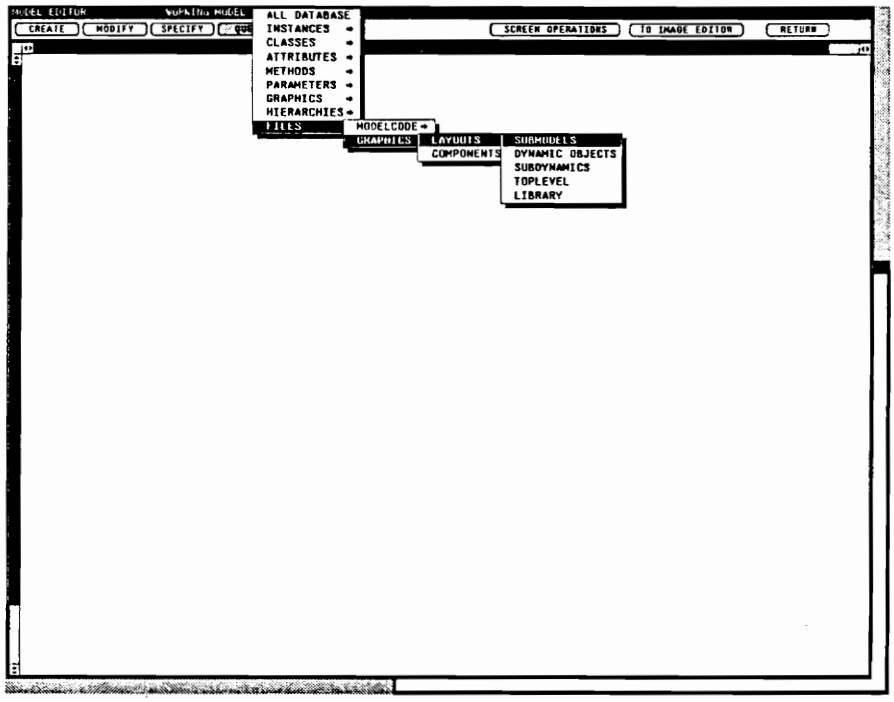

Figure 4.51 Activating Graphics File Query

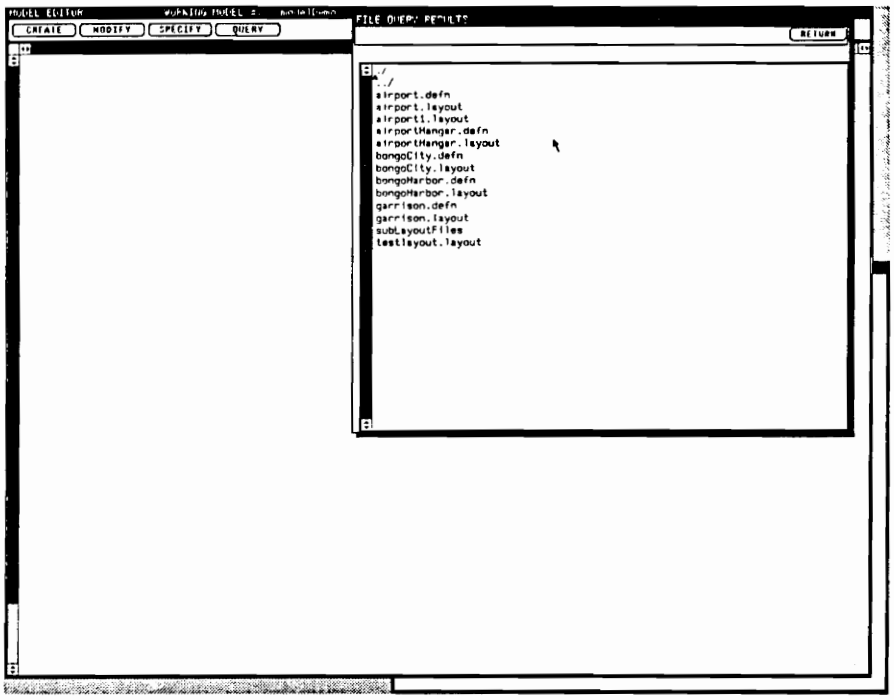

Figure 4.52 Result of Graphics Query

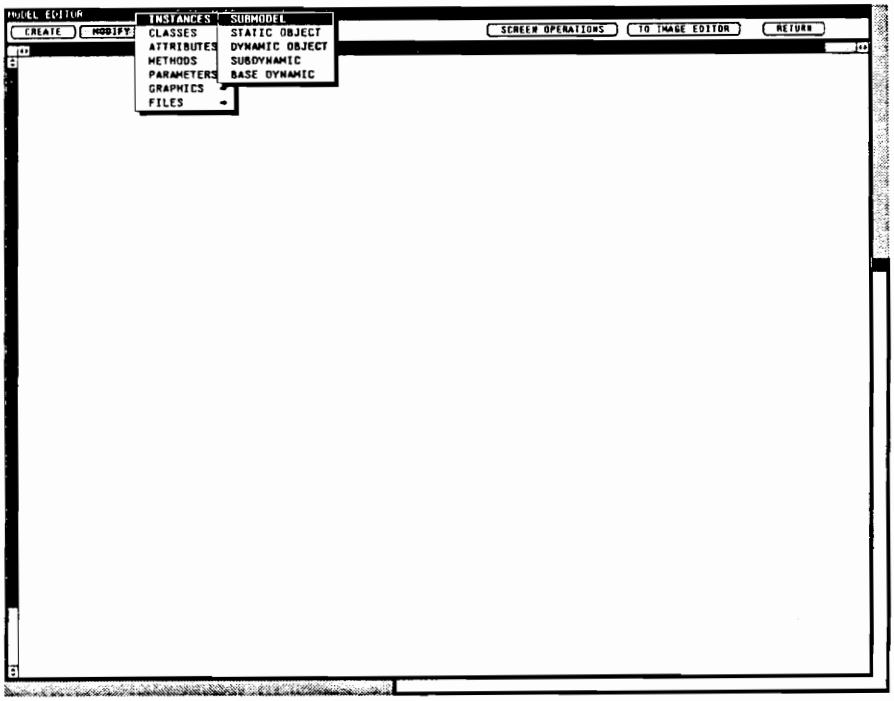

Figure 4.53 Modify Operation of Model Editor

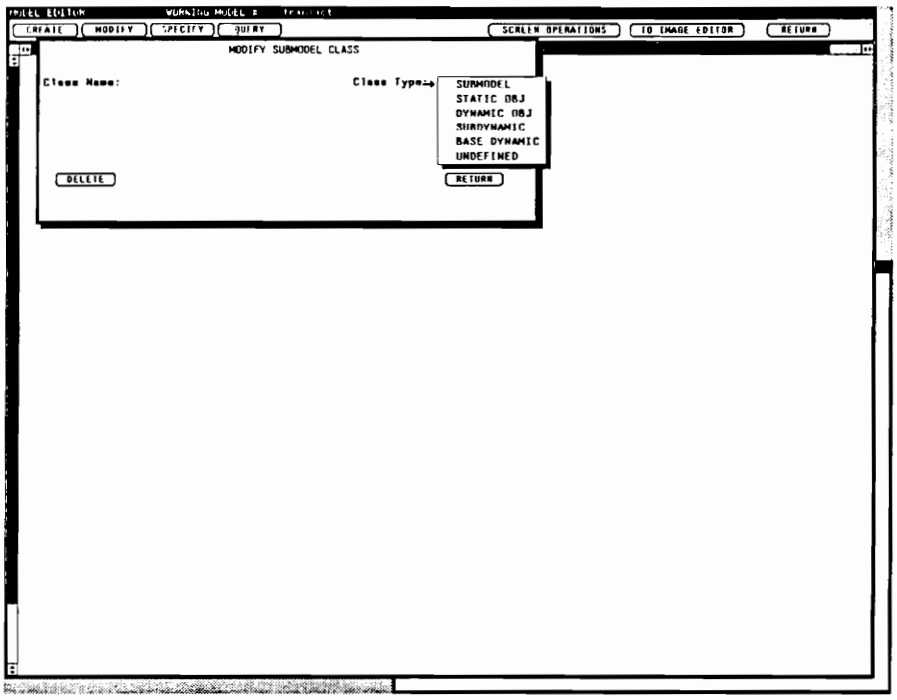

Figure 4.54 Modify Window for Component Class

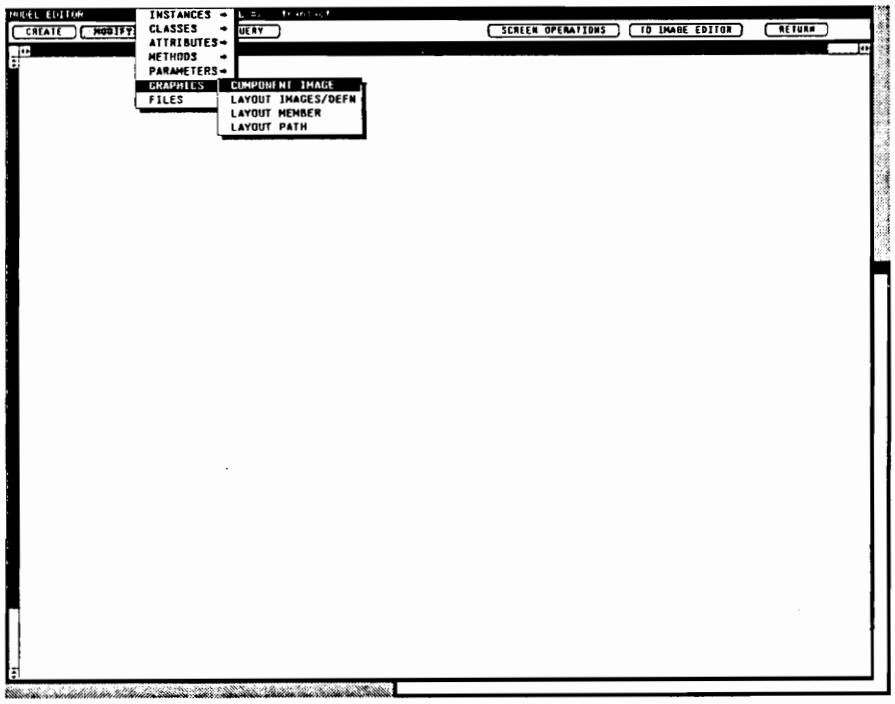

Figure 4.55 Activating Graphics (Data Base) Modification

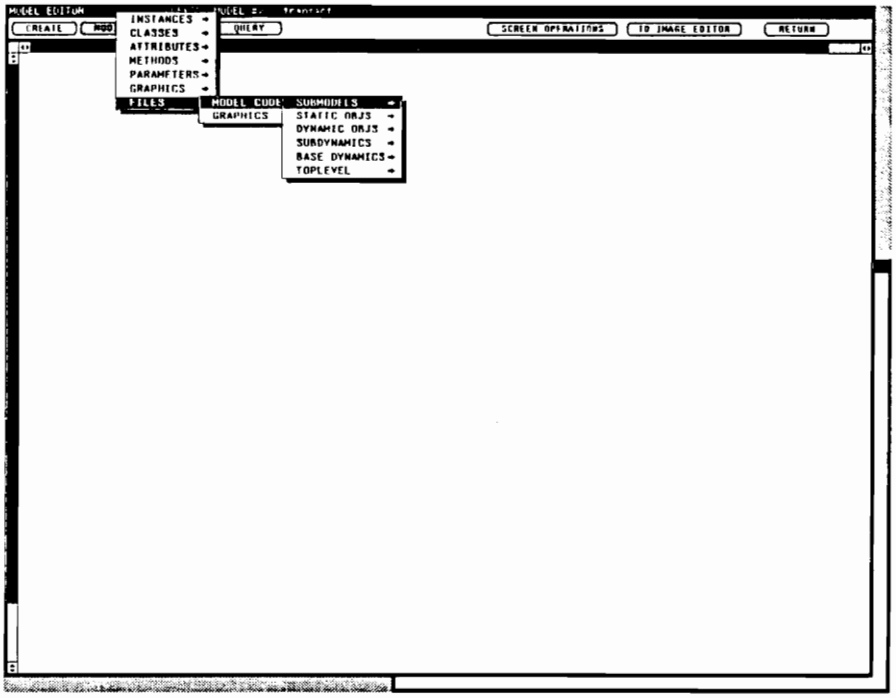

Figure 4.56 Activating Model Code Files Modification

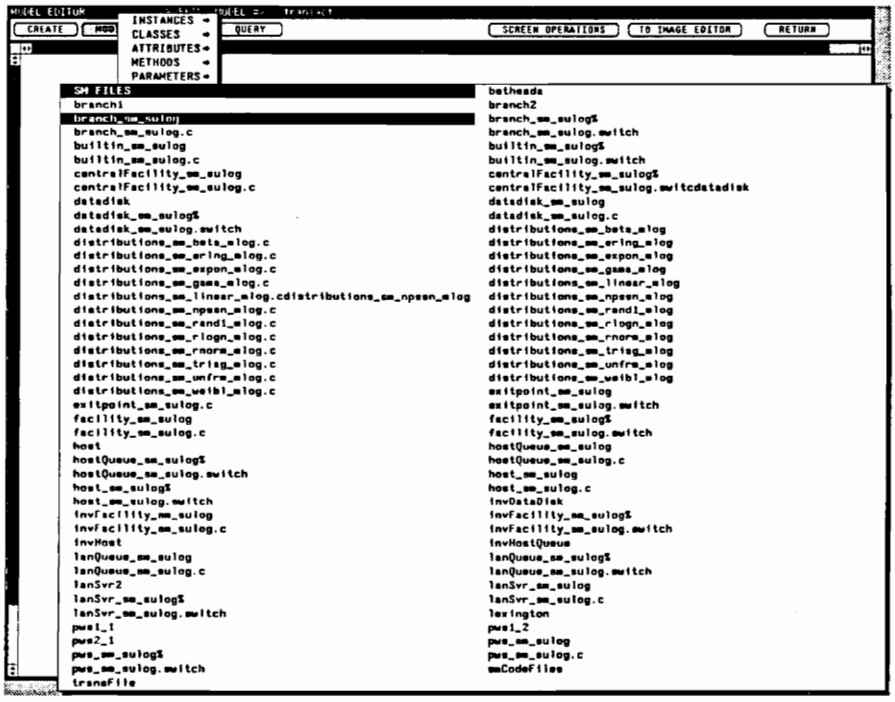

Figure 4.57 Pullright Menu of Model Code Files for Selection and Modification

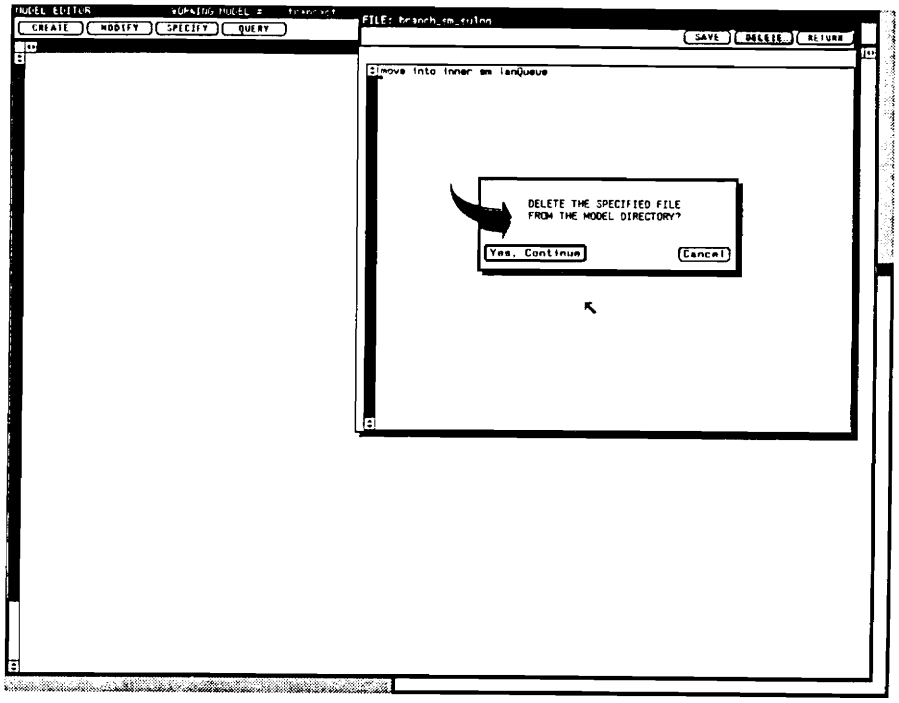

Figure 4.58 Result of File Modification (Deletion)

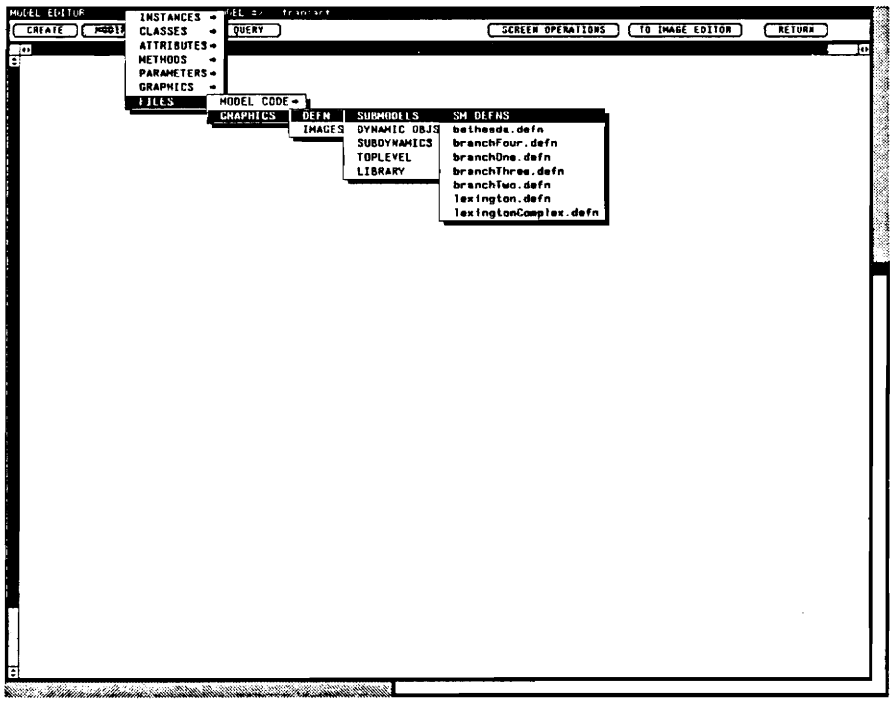

Figure 4.59 Activating Modification of Graphics Layout Definition Files

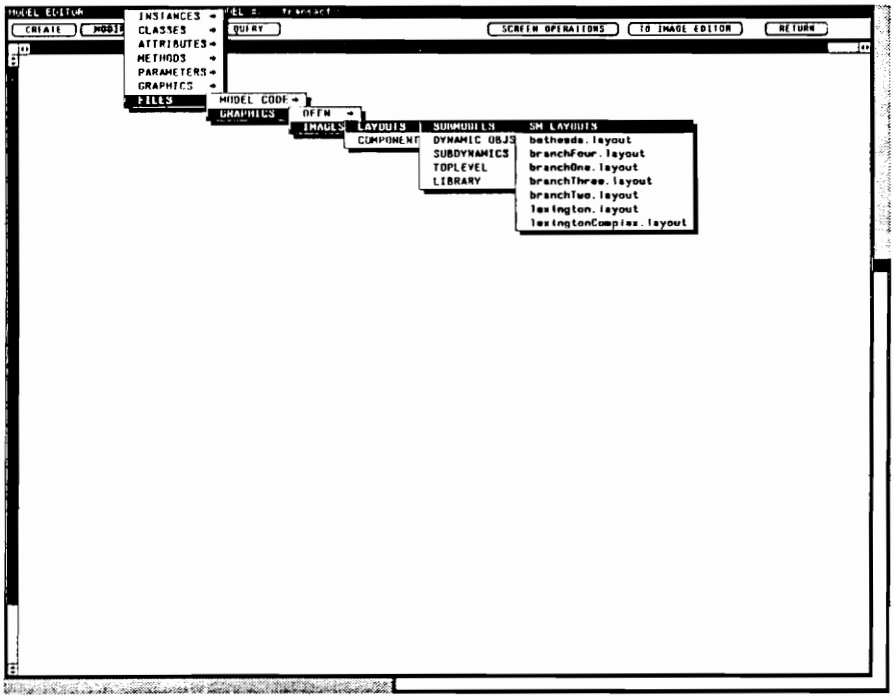

Figure 4.60 Activating Modification of Graphics Image Files

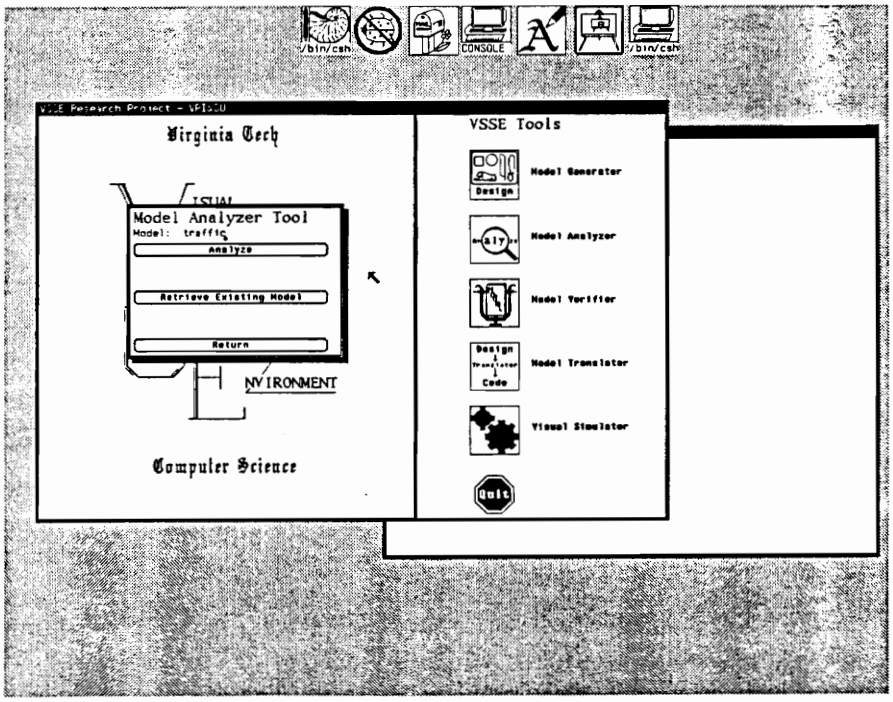

Figure 4.61 Model Analyzer Menu

(Figure 4.62) is displayed. Analysis is possible on the class specification, logic specification, image specification, definition and instantiation information, and even the documentation. Generally speaking, an aspect can be selected for analysis by selecting the associated **Analyze** button (Figure 4.62). If a problem is found during analysis, the "NO" item toggles to a "YES". At that time, the modeler can view the error report by selecting **View Report**. Figure 4.63 represents an example report. The individual aspects of analysis (class, logic, image, definition and instantiation, and documentation) are now discussed.

4.2.1 Analyzing Class Specifications

Three areas relating to completeness and consistency of class specifications can be analyzed: component images, layout images, and object instances. Component images are created, named, and stored as the default image or one of a set of images for some particular class. Decomposition layout images are linked to a class in the same manner. Also, the object instantiation process requires that a class be identified for the object. In each case, there is an association to some class. The VSSE permits these class associations to be made (possibly prior to the class being specified) providing additional flexibility during definition and specification. Therefore, a "YES" report on analysis for any of these three areas indicates that the associated class has not been formally specified, and the class specification is incomplete. The Model Analyzer overcomes the deficiencies of not having automatic checks by the VSSE on class names for their specification. Completeness is not sacrificed for flexibility.

4.2.2 Analyzing Logic Specifications

For all specified classes, analysis can determine a list of classes which do not contain supervisory or self logic specifications. Human knowledge must be applied to ensure that classes requiring some model component logic specification (supervisory or self) are not included on the list (see, again, Figure 4.63). The analysis reports provide notes to assist in this determination.

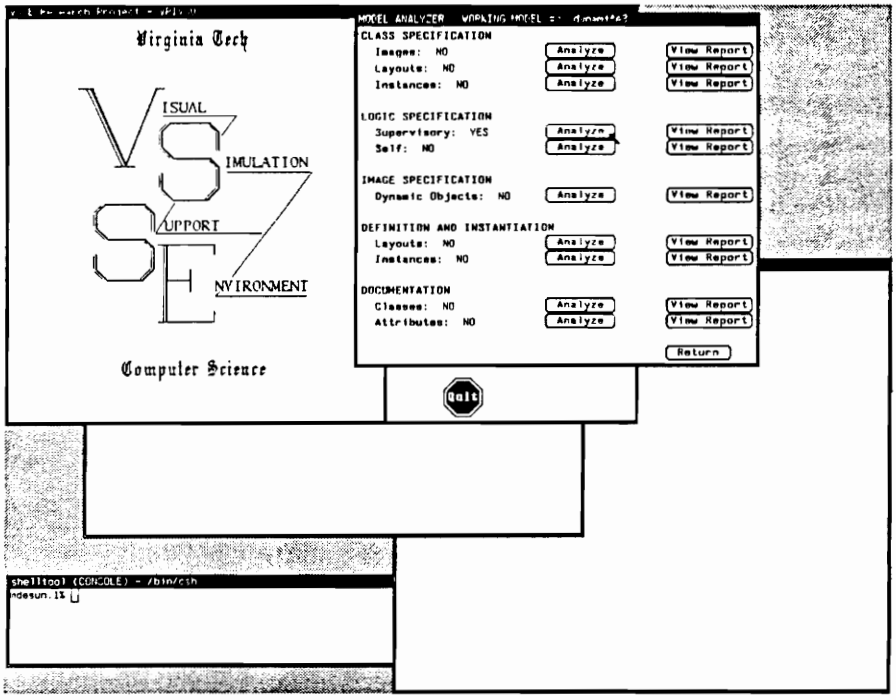

Figure 4.62 Analyzer Window

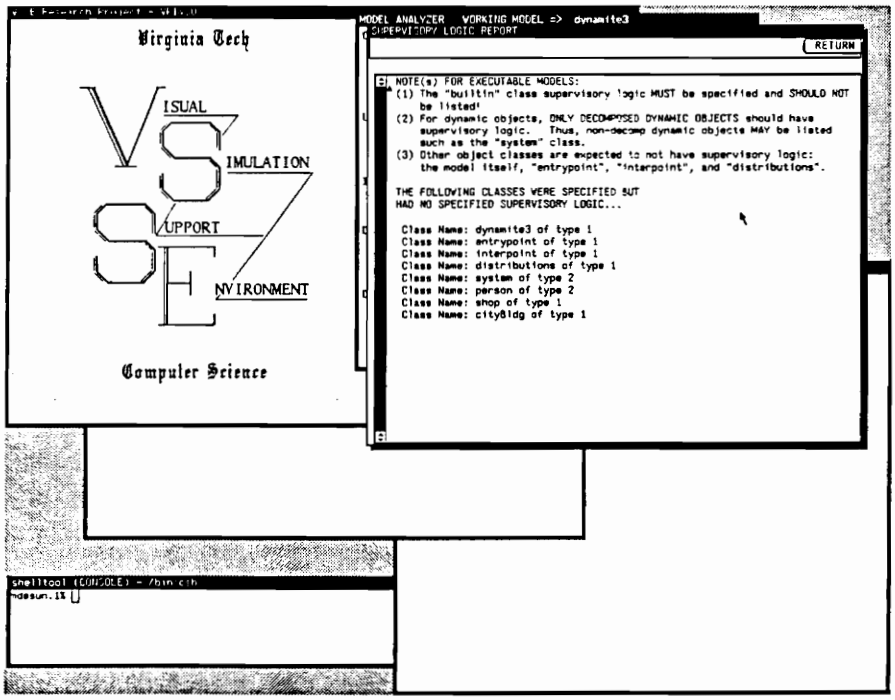

Figure 4.63 Example Analysis Report

4.2.3 Analyzing Image Specifications

Every real dynamic object class must have an image that has been drawn in the Image Editor and saved for association with that class. This type of analysis reports the names of specified real dynamic object classes for which no dynamic object image has been created and stored.

4.2.4 Analyzing Layout Definition and Object Instantiation

In the first type of analysis for this category, the analyzer reports any class layout image name which does not have an associated layout definition. In the second type, the analyzer scans model static and dynamic structures. Each decomposition level must be fully instantiated. The analyzer reports model component object instantiation problems of incompleteness. This report does not include incompleteness of path, connector, or interactor specifications.

4.2.5 Analyzing Documentation

The analyzer identifies incomplete documentation for both classes and class attributes. This analysis helps provide guidance and a check on the modeling effort, supporting one of Nance and Overstreet's [1986] identified purposes for diagnosis. In addition, this analysis capability (in concert with the VSSE design for including documentation features) encourages the accomplishment of model specification and documentation within the same activity, one of Nance's [1987] three rationales (by hypothesis) for the Conical Methodology.

4.3 The Model Verifier

Figure 4.64 indicates the verification features that are available within the VSSE. Toggles ("ON" and "OFF") enable assertion checking, runtime trace, or execution profile. Only one of these toggles can be turned on at any time. Working in-hand with the Model Translator (Section 4.4), setting a toggle to "ON" determines the compilation scheme used by the Translator. For example, with assertion checking "ON", the Model Translator produces an executable version in which assertion statements (within model

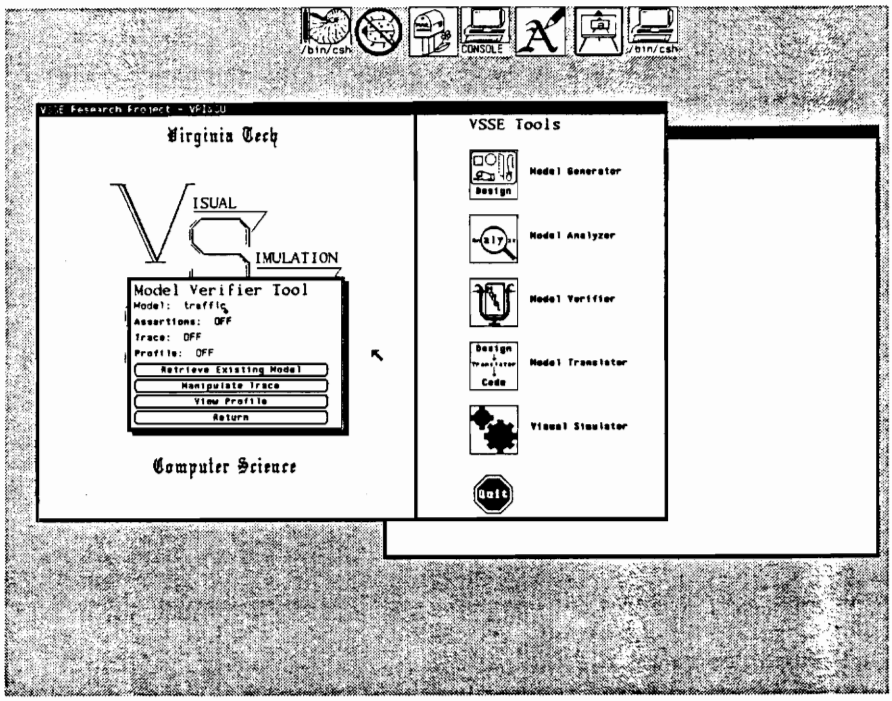

Figure 4.64 Model Verifier Menu

component logic) are executed during runtime. With assertion checking "OFF", the assertion statements are bypassed. Similarly, if the appropriate toggle is turned on, compilation by the Translator generates an executable version that can produce runtime trace data or an execution profile. Once a runtime trace or execution profile has been created, then the **Manipulate Trace** or **View Profile** buttons (Figure 4.64) can be used to start up their respective verification features, each of which is discussed below.

4.3.1 The Trace Manager

Execution tracing, a dynamic analysis technique, provides valuable assistance in the verification of a model. During the execution of an executable version (which has been specifically created for building a runtime trace) of a model, trace data is stored in the relational database. The database structure accomodates the trace information in a useful manner and can be selectively queried by the modeler to locate the source of the error. With the Trace Manager (Figure 4.65), a modeler can manipulate the trace data in order to relate runtime errors to the specification. In discussing the Trace Manager, the trace data is first described so that Trace Manager facilities are not misunderstood.

4.3.1.1 The Trace Data

Each record in the trace data relation contains three fields: name, type, and flow. As execution moves through model component logic, records are written to the relational database, supplying this field information relative to the location of the execution at that time. The name refers to the routine name (e.g., supervisory logic name) or statement name (e.g., **move** statement, **repeat** statement, **if** statement, etc.). The type is the category of routine (i.e., whether a modeler-defined routine or system attribute routine) or statement. System attribute routines (See Chapter 6) are those used by the system for storing attribute values or retrieving attribute values. Flow indicates the direction of logic flow within the routine or statement (i.e., entering or exiting). Thus, a record is produced both upon entry to and exit from each (1) modeler-defined routine, (2) system attribute routine and (3) VSMSL statement in the model component logic. The trace data can be voluminous and represents an accurate sequential record of the execution logic

flow. Because of the potentially enormous amounts of trace data that can be produced, only one trace database is maintained (i.e., trace data cannot be saved for each model; only one set of trace data is kept in the database at a time). The **Clear** operation (Figure 4.65, rightmost operation) clears the trace database in preparation for storing new trace data. The Trace Manager has two additional functions for handling the trace data effectively. The **Locate** and **Trace** facilities are now discussed.

4.3.1.2 Locating End of Execution

Using **Locate**, a modeler can pinpoint and characterize the end of the execution trace. Each record in the trace data is numbered. **Locate** (Figure 4.66) "locates" (as determined by modeler selection) the identity and record number of the last trace entry, the last user (modeler-defined) routine, the last system access to an attribute, or the last VSMSL statement. Therefore, a modeler can ascertain in what part of the logic the execution was interrupted, and even a finer detail: which statement or attribute access was affected. Figure 4.67 displays the location of the last entry of the trace data, giving the supervisory logic name, flowing out (with a temporary time delay, e.g., at an **engageIn** statement), at record number 6080.

4.3.1.3 Viewing a Subset of Trace Data Records

Knowing the scope of the trace data from the record number or position derived from the "last" entries, the **Trace** operation becomes an effective tool for manipulating the trace data. In Figure 4.67 notice that the modeler can now get a good contextual look at a reduced subset of the trace data. The modeler can select to view the last "N" entries, user routines, attribute accesses, or statements. The "N" refers to the number of records (of the appropriate type) desired for viewing from the end of the trace data. Furthermore, the modeler can select any block of records using the "Ith-to-Jth" option. After activating the **Trace** feature, the subset of trace data records is shown in the display area of the Trace Manager. Figure 4.68 gives one such display of the "last N entries." The flow information is used to "pretty print" and appropriately indent the display for better readability of the trace.

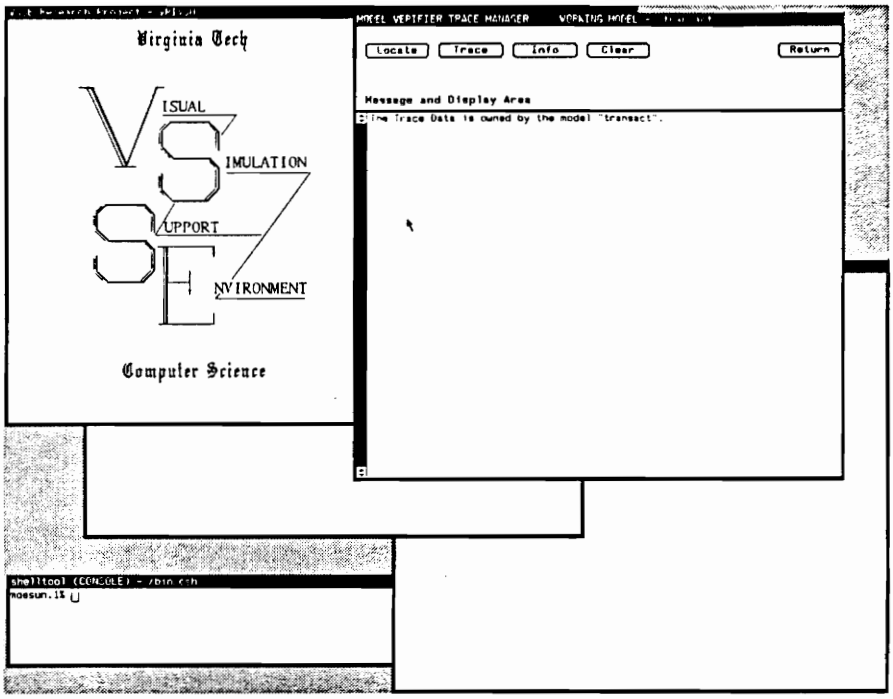

Figure 4.65 Trace Manager Window

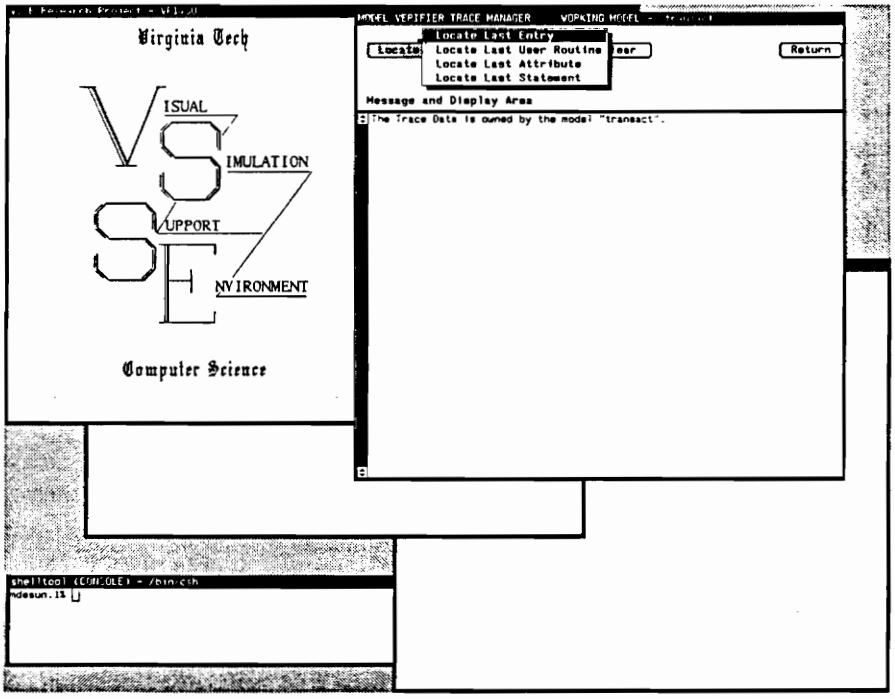

Figure 4.66 Trace Manager Locate Facility

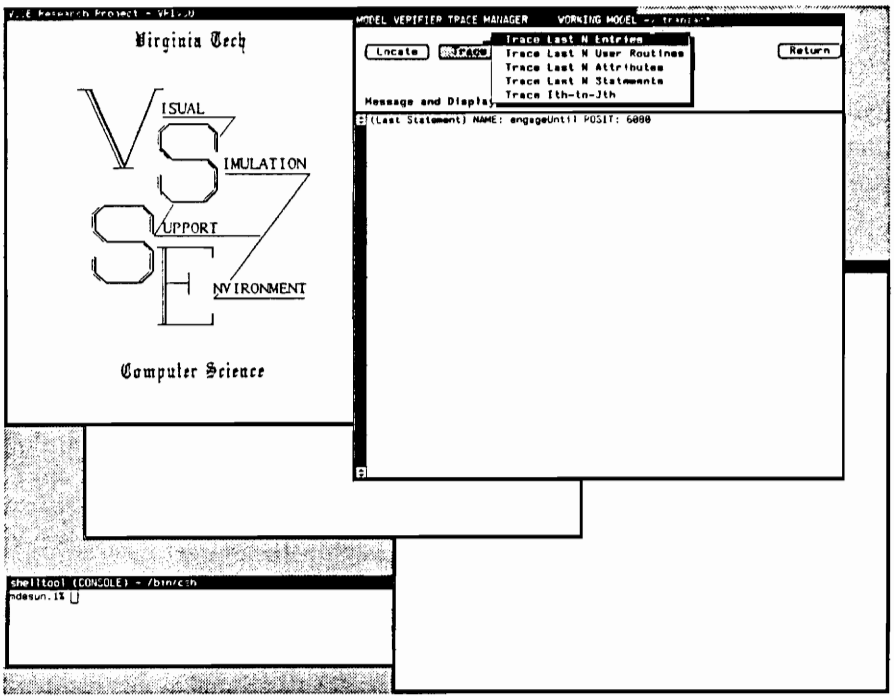

Figure 4.67 Trace Manager Trace Facility

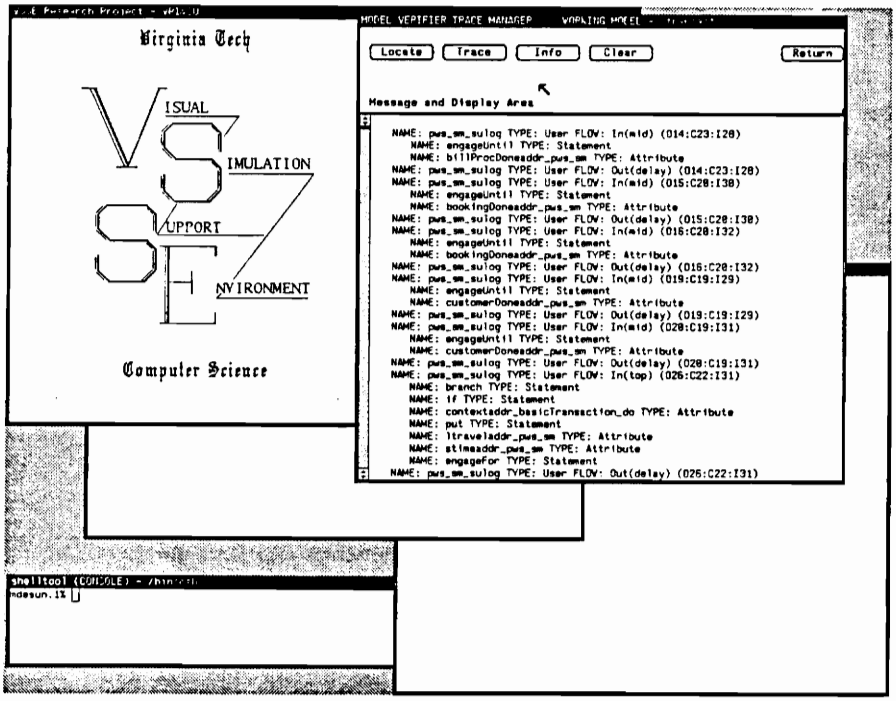

Figure 4.68 Result of "Last N Entries" Trace

4.3.1.4 Detecting Conceptual Errors in Logic

Syntactic errors are effectively caught by the Model Translator; the **Locate** and **Trace** features of the Model Verifier's Trace Manager can detect the position of runtime errors and locate the position of the "crash" within the model component logic. However, detecting conceptual errors in the model component logic is much more difficult. For this reason, the Trace Manager includes a few other features for assistance. A close inspection of Figure 4.68 shows the presence of OCI (Object, Class, and Instance) codes at the end of each record entry for modeler-defined routines (such as supervisory logic). Knowing the location (in which modeler-defined routine) that execution was interrupted is knowing only a part of the puzzle. Since many dynamic objects are causing the various supervisory, self, and method logic to be executed, it is helpful to know the identity of the "causing" dynamic object. The **O** code is the identifier of that dynamic object and the **C** code is the class of that dynamic object. Finally, because model component logic specifications can be inherited by members throughout a class inheritance hierarchy, the **I** code gives the instance code of the specific owner of the logic. The **Info** (Figure 4.65) button provides access to these system-generated codes and enables one to decipher them. For example, Figure 4.69 is a listing of the system codes for instances.

4.3.2 The Execution Profiler

After an execution profile has been set up, and the **View Profile** button (Figure 4.64) activated, the profile results are displayed (Figure 4.70). The execution profile is created using the **prof** library routine under the SUN Operating System. Useful runtime execution information is displayed for routines: percentage of time spent executing between the routine and the next on the list, cumulative execution time, number of calls made to that routine, number of milliseconds per call, and the routine name. For large systems, this information can be used to identify conceptual problems in the model and runtime inefficiencies.

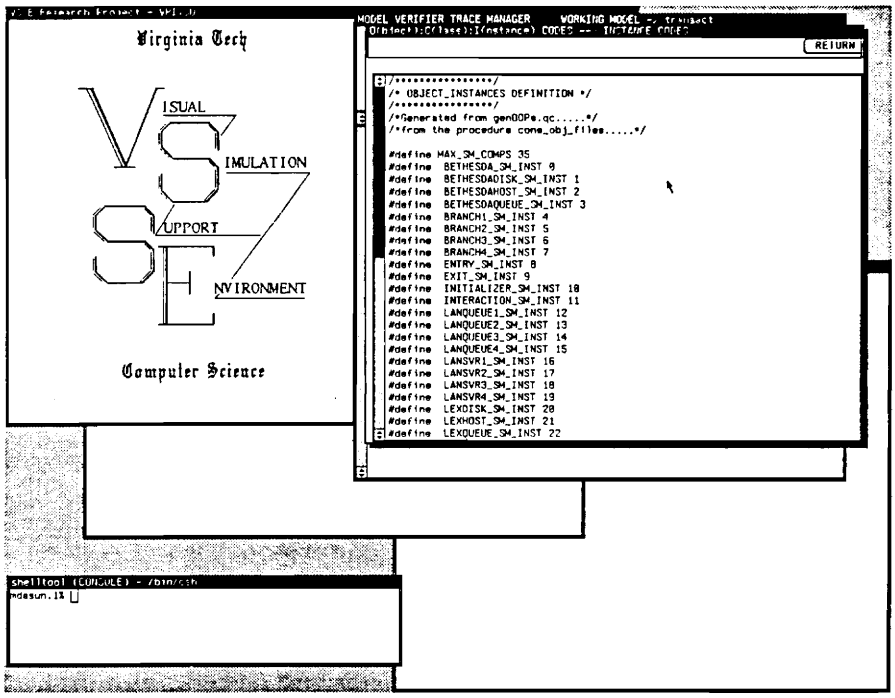

Figure 4.69 Listing of System Instance Codes

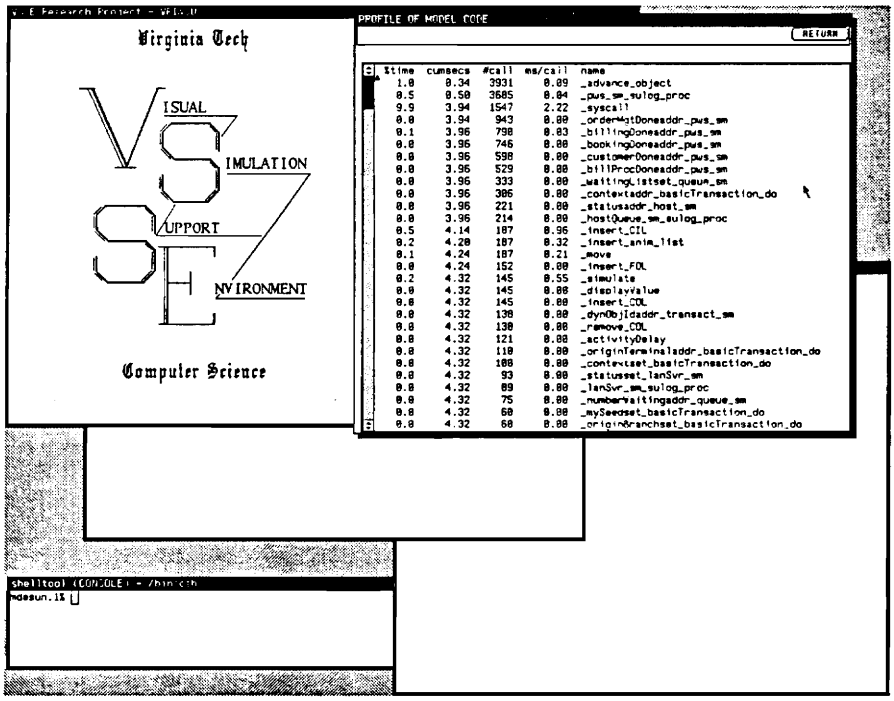

Figure 4.70 Execution Profile Report

4.4 The Model Translator

The Model Translator (Figure 4.71) accomplishes the final automatic generation of code for model execution. This is done in a two-step process: creating source code and creating object code. **Create Source Code** creates the modules and system definitions to accomodate the object-oriented capabilities. The information used in this creation process is retrieved from the specification information stored in the model database. During this phase, the Translator reports on the incremental generation, notifying the modeler at each step (Figure 4.72). **Create Object Code** takes the source modules and definitions created above and combines these with the individual model component logic source files that are created by the modeler during class specification. Compilation of all source files produces a set of object modules; all resulting object modules are linked together to form a single executable model version. During compilation, the Model Translator again reports progress in the compilation effort to the modeler (Figure 4.73). Compilation errors are reported. If desired, compilation warnings can be turned on (default: off) using the designated toggle. As mentioned in Section 4.3, toggles in the Model Verifier cause different affects of this final translation process in producing an executable for different purposes (activation of assertions, production of runtime trace or exection profiles, or normal execution for simulation and animation by the Visual Simulator).

The automatic translation and production of an executable model form supports the Automation-Based Paradigm. Modelers are prevented from having to get involved with the low-level details of C programming. Instead, the modeler deals directly with the specification. Modification and maintenance are performed on the specification which is stored in the model database and in various system files; there is no maintenance or modification on the target code itself. The Model Translator takes the specification and automatically generates the executable code. Guided by the DOMINO during design and implementation while using the integrated VSSE toolset, a modeler produces a specification that can be automatically translated. The specification parts have been organized in an orderly manner. This final translation, in which all parts of the model specification "come together" and are "fused", produces the executable model by what is called the "***DOMINO effect.***"

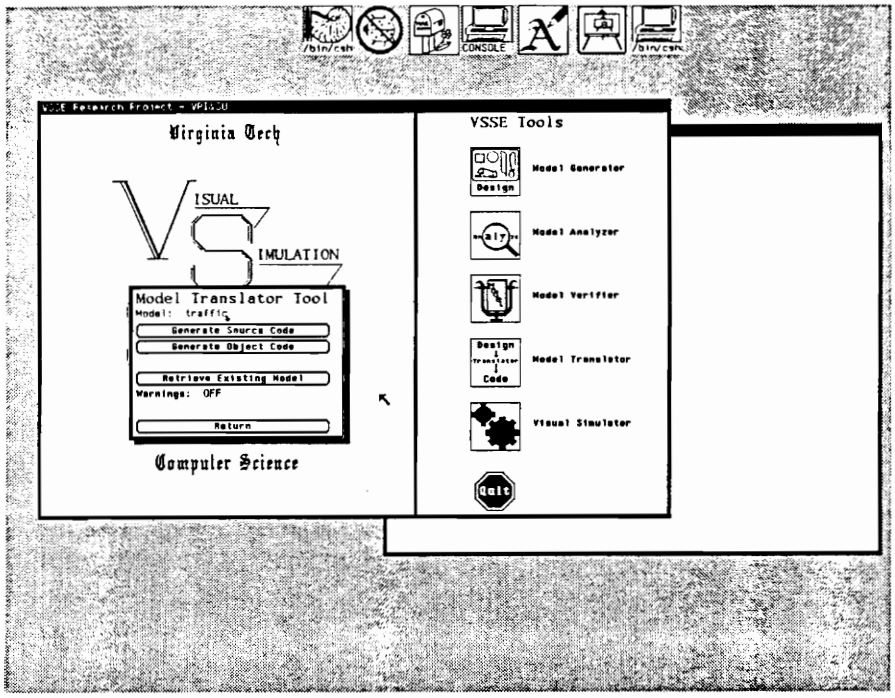

Figure 4.71 Model Translator Menu

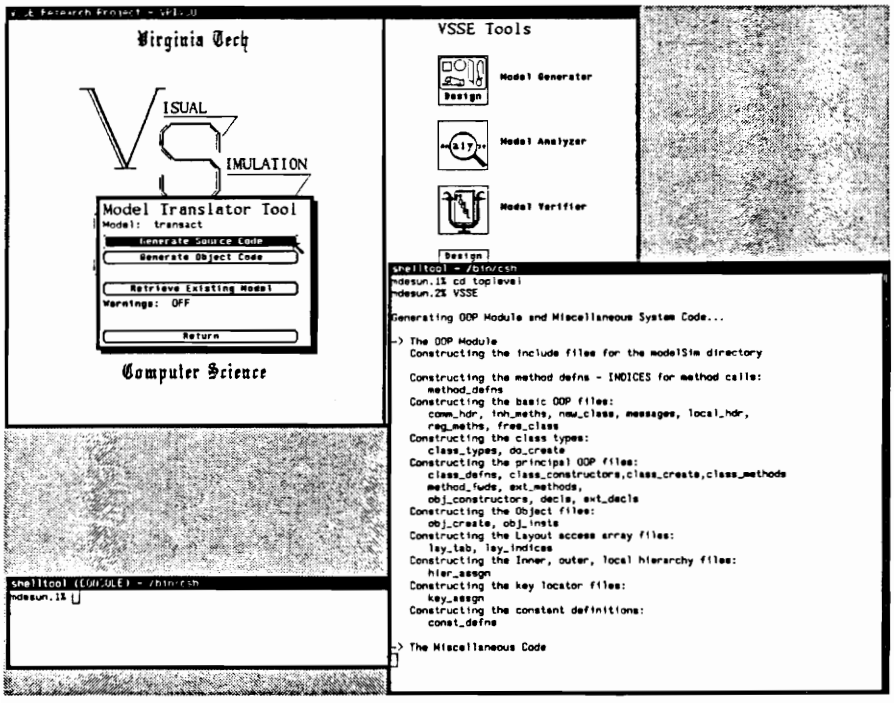

Figure 4.72 Notification of Progress in Source Code Generation

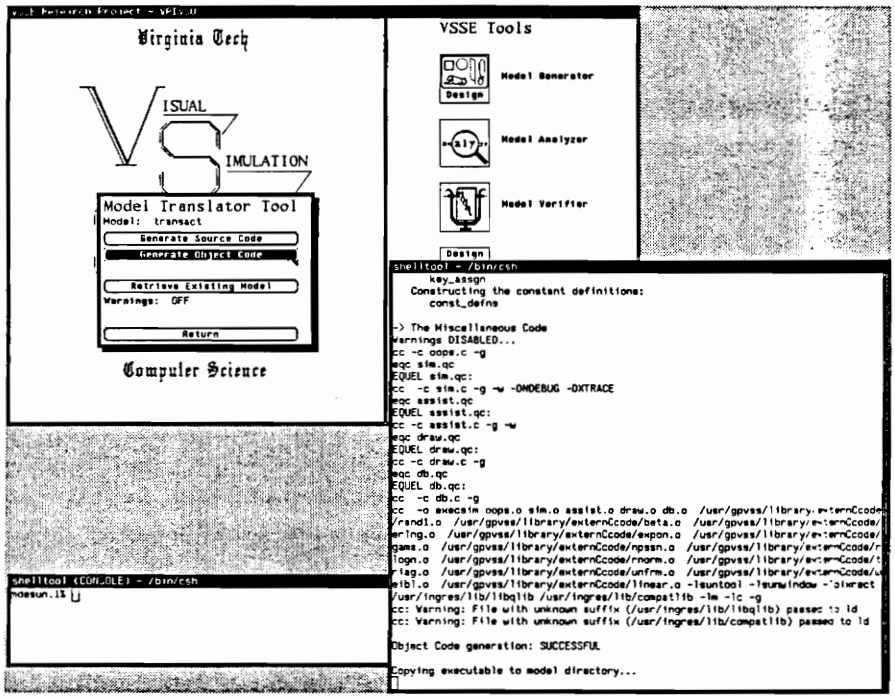

Figure 4.73 Notification of Progress in Object Code Generation

4.5 The Visual Simulator

The diversity of this VSSE tool (Figure 4.74) provides additional yet powerful assistance to a modeler. The visualization of the simulation model supports the dynamic analysis of the model. Two key features are highlighted: (1) the runtime inspection facilities for model component attributes and performance measures, and (2) the contextual visualization of the executing model. The Visual Simulator is described using examples from the Branch Operations example, detailed in Chapter 7. To begin our discussion, note that the Visual Simulator is entered by activating the **Simulate/Animate** button from Figure 4.74. The top level layout associated with the root of the model static structure is loaded and displayed as shown in Figure 4.75. We first introduce the tool.

4.5.1 Introducing the Visual Simulator

With the Visual Simulator, a modeler can execute the model for visualization and animation using the **ANIMATE** button. Previous prototypes demonstrated the ability to run simulations in the background without animation using the **RUN** button. Method of Replications data collection information (e.g., transient period length, steady state length, number of replications, etc.) was entered (Figure 4.76) and the **Run Simulation** button activated. Following execution, statistical summaries (Figures 4.77, 4.78, 4.79) of performance data were available. For this section, we concentrate on the visualization capabilities of the tool via the **ANIMATE** button.

From Figure 4.75, the simulation clock is located in the upper left corner (now reading zero(0)). The runtime inspection facility (next discussed) is toggled on or off using the toggle in the upper right corner. The **ASCEND**, **TOP**, and **INSPECT** buttons are also described later.

4.5.2 The Runtime Inspection Facility

The Runtime Inspection Facilities are designed with three perspectives in mind. First, model component attributes (for non-dynamic object components which are instantiated within model class layouts)

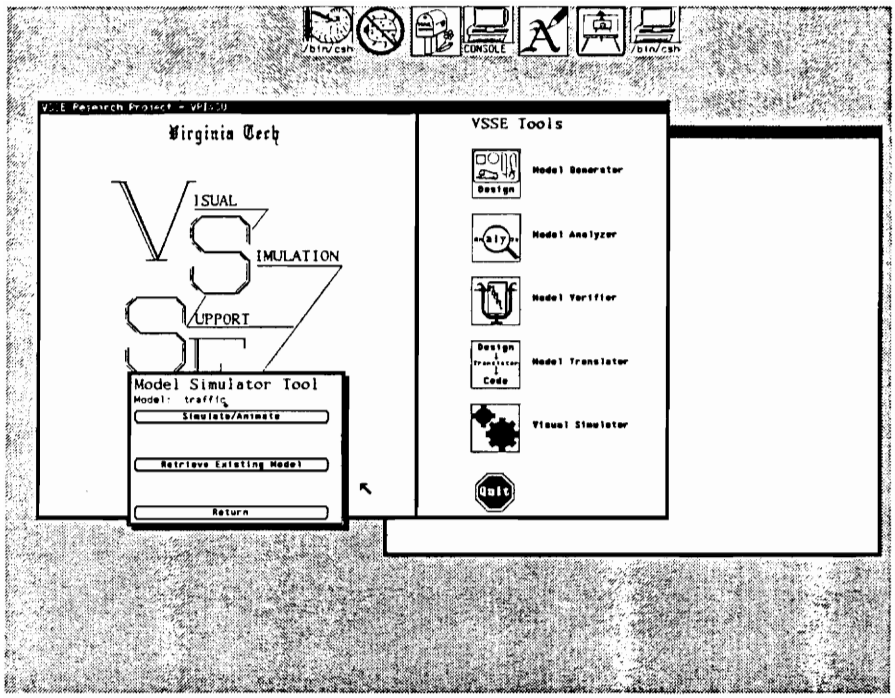

Figure 4.74 Visual Simulator Menu

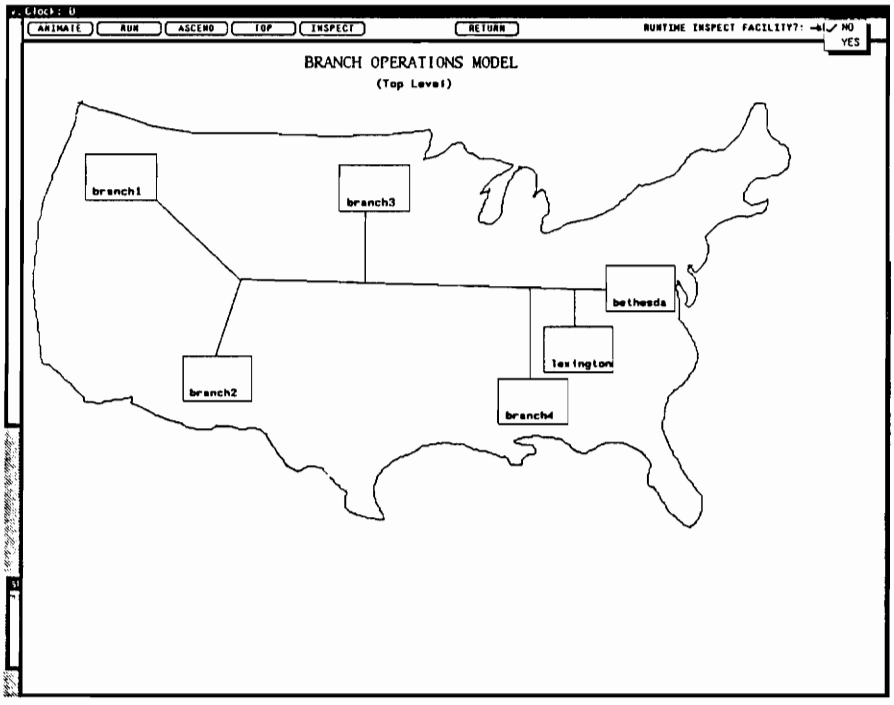

Figure 4.75 Result of Model Load prior to Execution/Animation

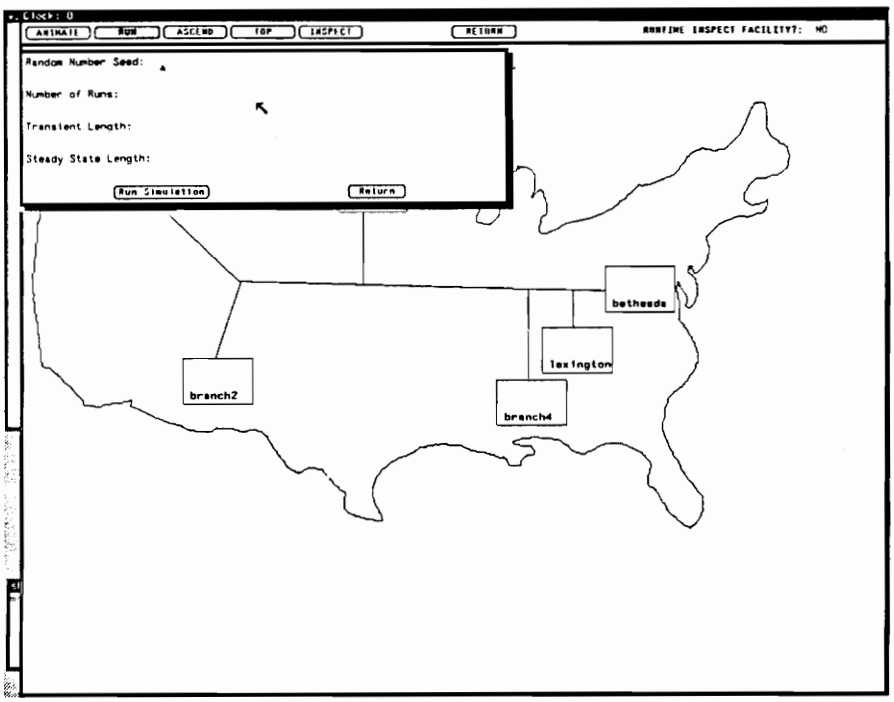

Figure 4.76 Simulation Run (without Animation) Window

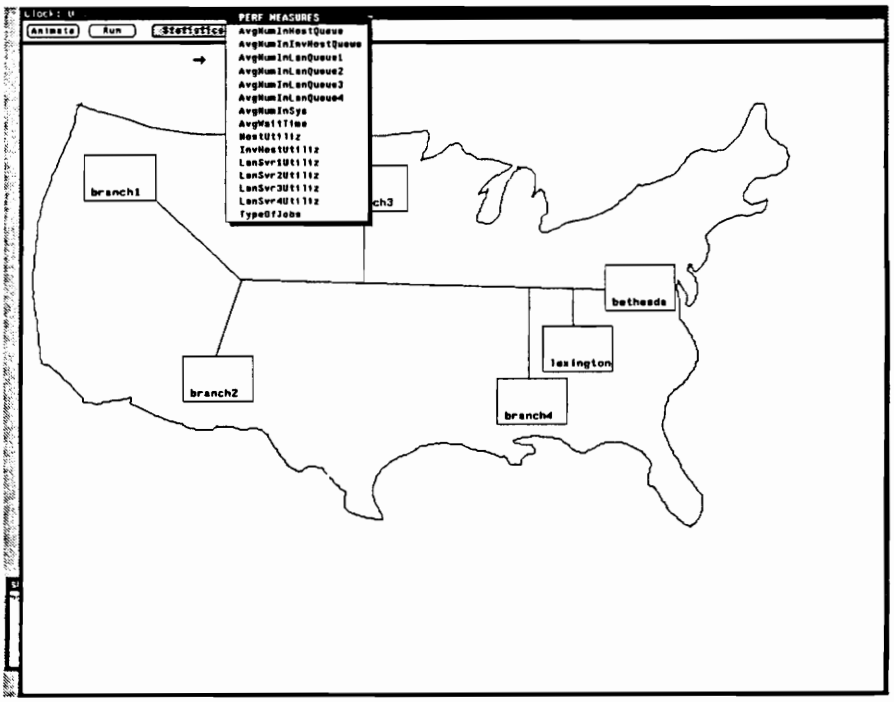

Figure 4.77 Statistical Report Selection

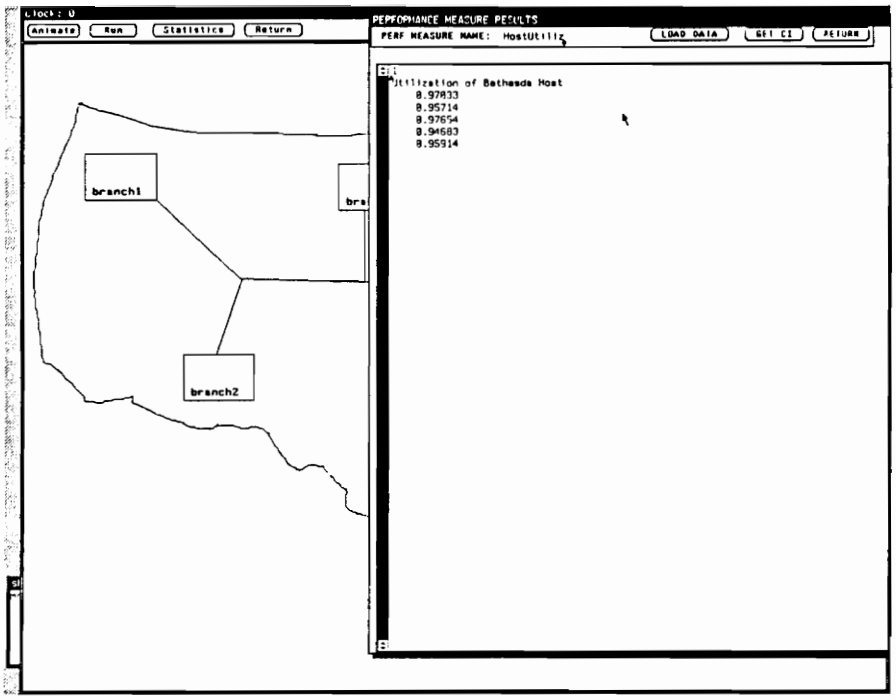

Figure 4.78 Performance Measure Data

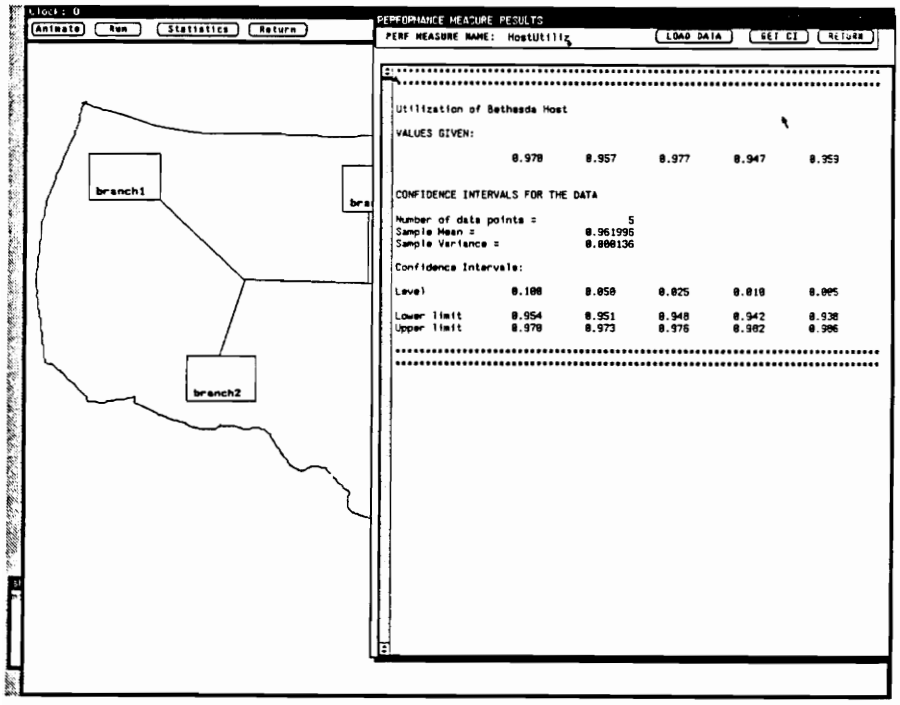

Figure 4.79 Confidence Interval Report

must be accessible for inspection during runtime. Secondly, attributes of a global nature (attached to the model itself; i.e., model attributes) or the attributes of virtual submodels must also be accessible. And finally, attributes of dynamic objects (both decomposed and non-decomposed) must be available to a modeler for viewing. The Visual Simulator satisfies all three perspectives. With these perspectives, a modeler can set up performance or statistical measures of interest and observe their changes during the course of the animation.

4.5.2.1 Inspecting Attributes of Model Components in a Layout

Within any layout, by pointing (using the mouse) at a model component on the layout (and pressing the right mouse button), a popup is displayed which contains an **INSPECT** and a **DESCEND** option. Pulling right and using **INSPECT**, that component's attributes and current values are displayed. Figure 4.80 displays the attributes of **pws1** in Branch Office One. Note that these are initial values, in this case, since the model is not yet executing. Figure 4.81 shows a similar inspection of the LAN Server's attributes in Branch Office Two.

4.5.2.2 Inspecting Model and Virtual Submodel Attributes

The **INSPECT** button in the top horizontal row of Visual Simulator buttons provides the capability for inspection of the model attributes and also virtual submodel attributes. Figure 4.82 contains a display of the attributes of the virtual submodel **statmodule** which has all the attributes for random variate generation.

4.5.2.3 Inspecting Dynamic Object Attributes

Dynamic object attributes of interest and their values are written onto their moving dynamic object images during runtime; this is specified by **display** statements in the VSMSL. Such image modifications are indicated in later figures. The attributes of decomposed dynamic objects are also available using the

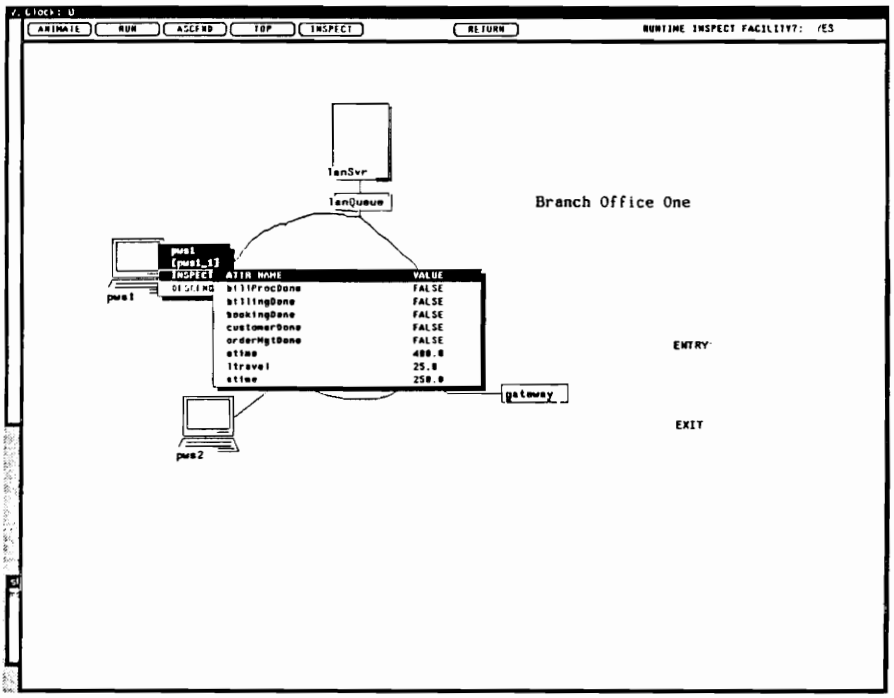

Figure 4.80 Inspection of PWS Attributes

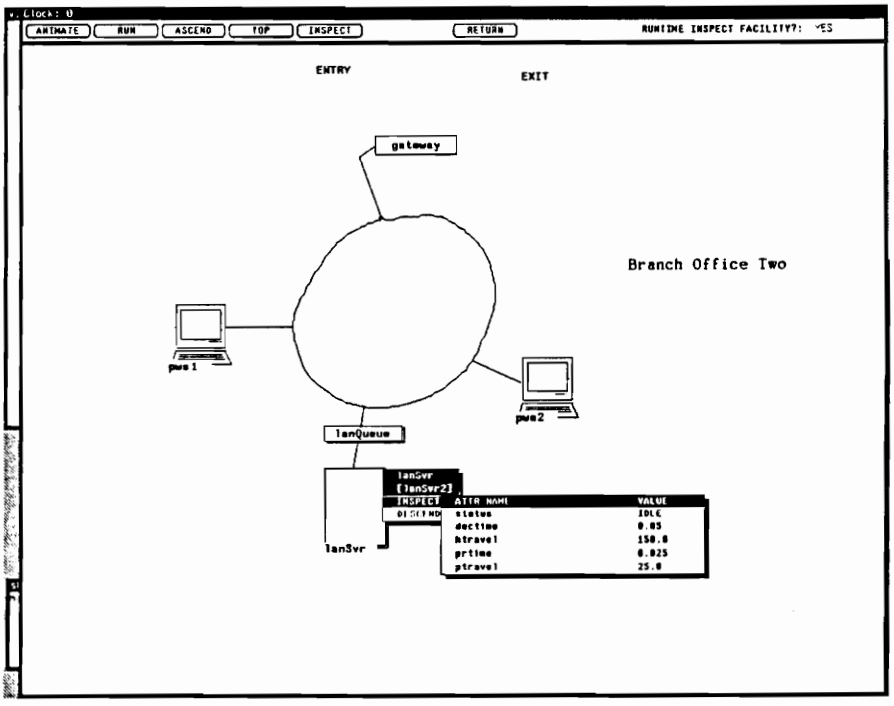

Figure 4.81 Inspection of LANSVR Attributes

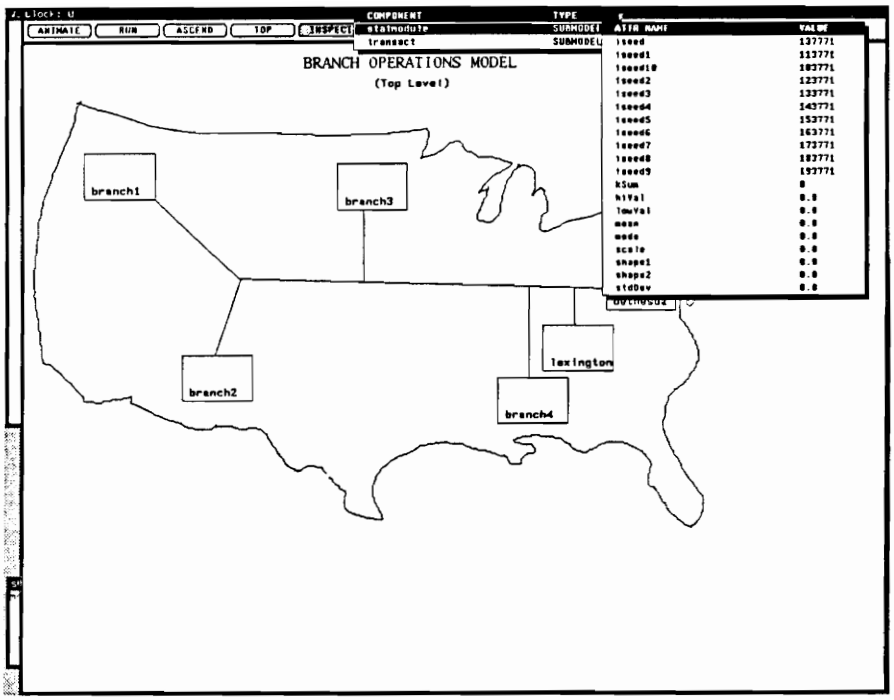

Figure 4.82 Inspection of Model and Virtual Component Attributes

INSPECT button as described in Section 4.5.2.2.

4.5.3 Contextual Visualization

This capability of the Visual Simulator allows a modeler to view the animation from any layout context. Similar to the ability to navigate throughtout the model static and dynamic structures within the Model Generator (for definition and specification), a modeler is given the same flexibility to change the runtime viewing context. The **TOP** button shifts viewing to the top layout of a model static or dynamic structure. **ASCEND** incrementally shifts viewing up one decomposition level. As previously mentioned and shown in Figure 4.80, popups are associated with model components in the layouts. If a component is decomposed, the **DESCEND** option of its popup is activated. This can be used to incrementally descend into a decomposition hierarchy. The Visual Simulator "navigation" facility has an added feature in that by pressing the right mouse button on the top border of the Visual Simulator, a popup is displayed which lists all model layouts. By selecting one (such as the **lexintonComplex**; Figure 4.83), the viewing context can be changed to a deep level of the hierarchy, without having to incrementally get there. Several figures are now discussed which demonstrate the features we have claimed.

4.5.4 Snapshots of a Sample Visualized Execution

Having begun animation and shifted to the Branch One context, Figures 4.84 to 4.86 demonstrate several of the aspects of the runtime inspection facilities. First, note in Figure 4.84 that the context attribute is displayed on the AdminTool dynamic object as it moves toward the LAN queue. Also, inspection of the LAN Server attribute **status** shows that the LAN Server is idle. After the dynamic object arrives at the LAN Server, inspection of its **status** attribute indicates that it is now busy (1).

Figure 4.87 shows the popup on the Visual Simulator top border from which the context can be shifted to any layout. (Any time this popup is displayed, the animation is "frozen"). The top level layout is chosen and Figure 4.88 shows the results of that selection. The BillProc dynamic object exiting Figure 4.87 is seen leaving Branch One on the top level continental United States layout. This dynamic object

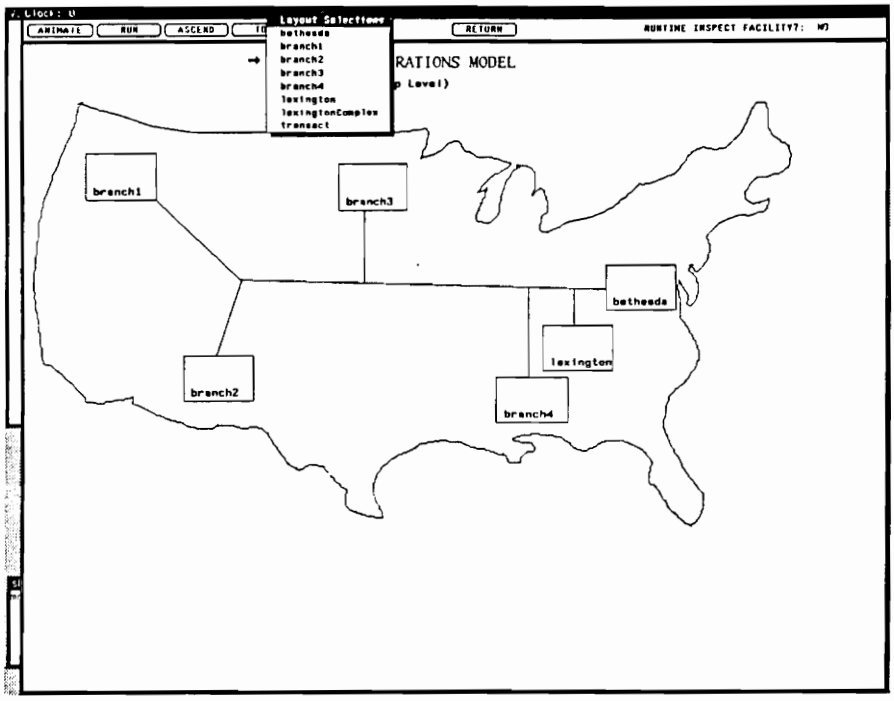

Figure 4.83 Direct Context Selection

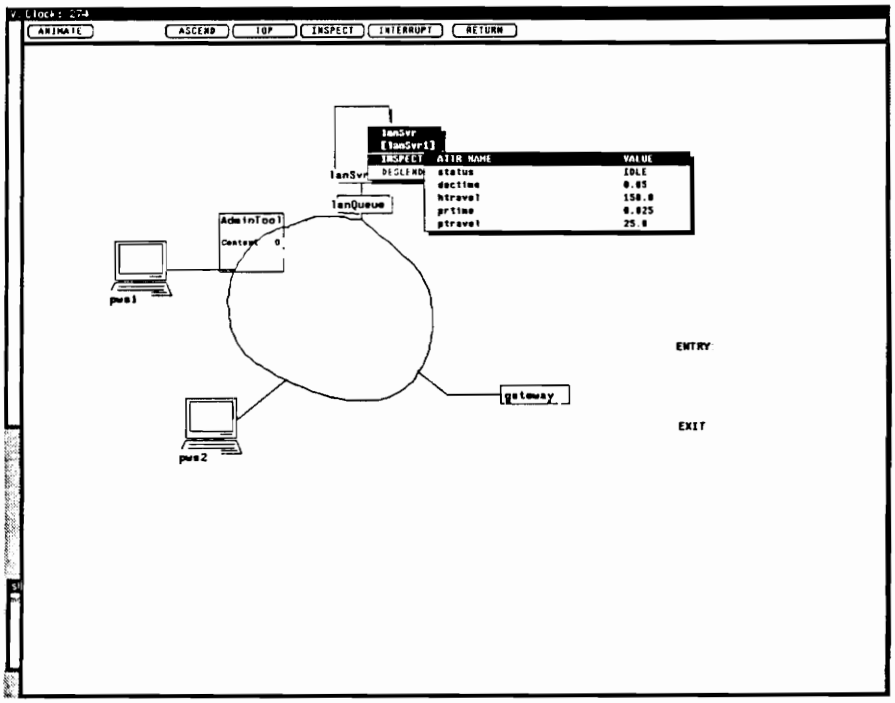

Figure 4.84 Runtime Inspection of LANSVR Attribute (before Dynamic Object Entry)

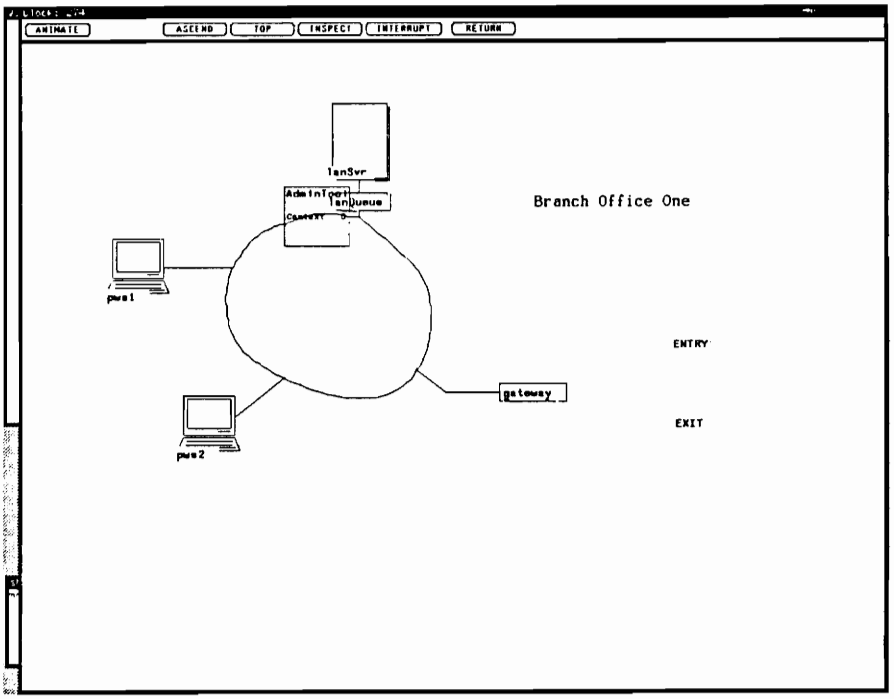

Figure 4.85 Dynamic Object Movement to LANSVR

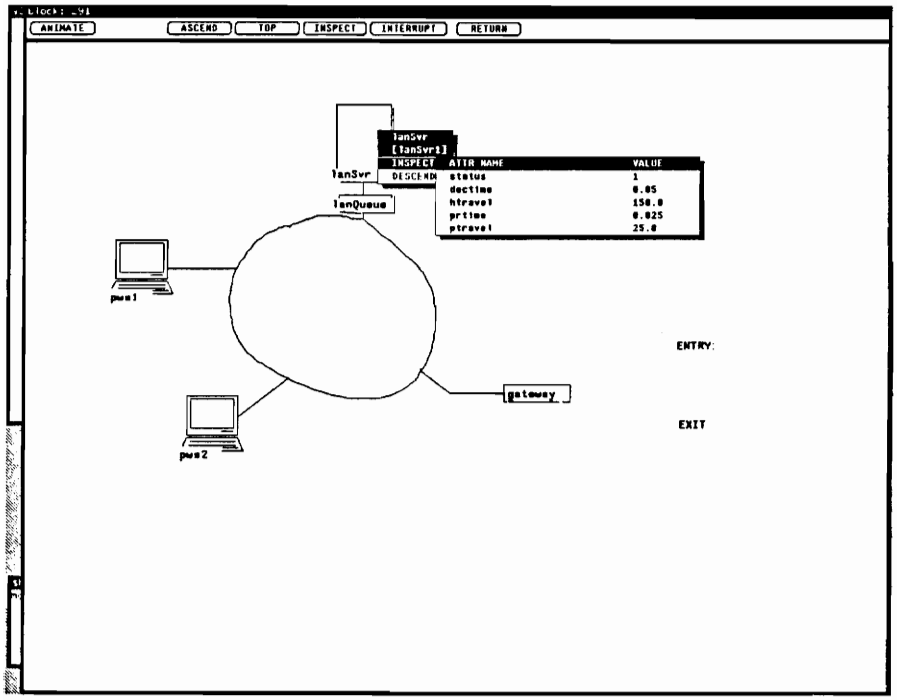

Figure 4.86 Runtime Inspection of LANSVR Attribute (after Dynamic Object Entry)

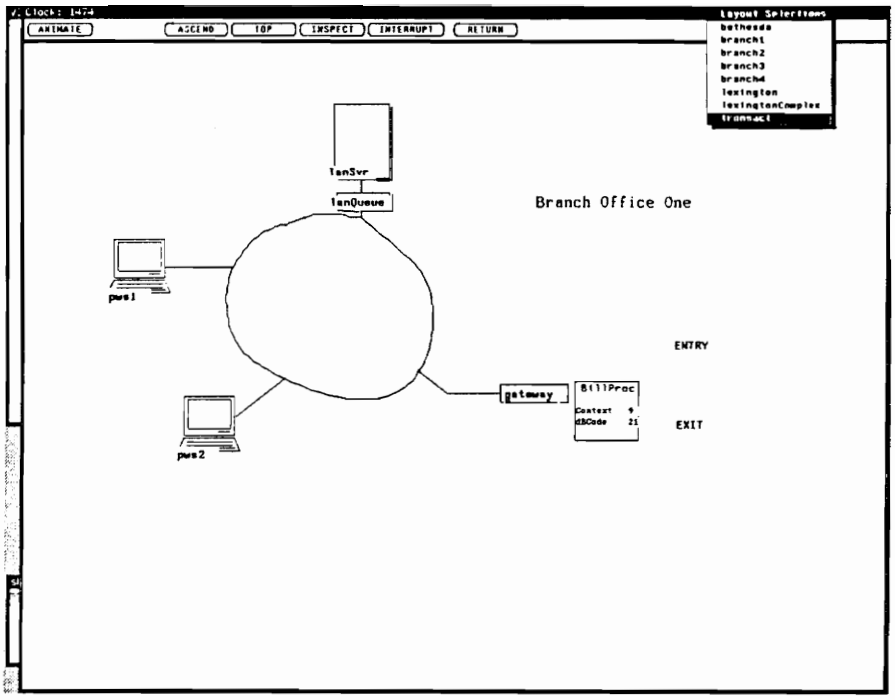

Figure 4.87 Activating Direct Context Shift to Top Level Layout

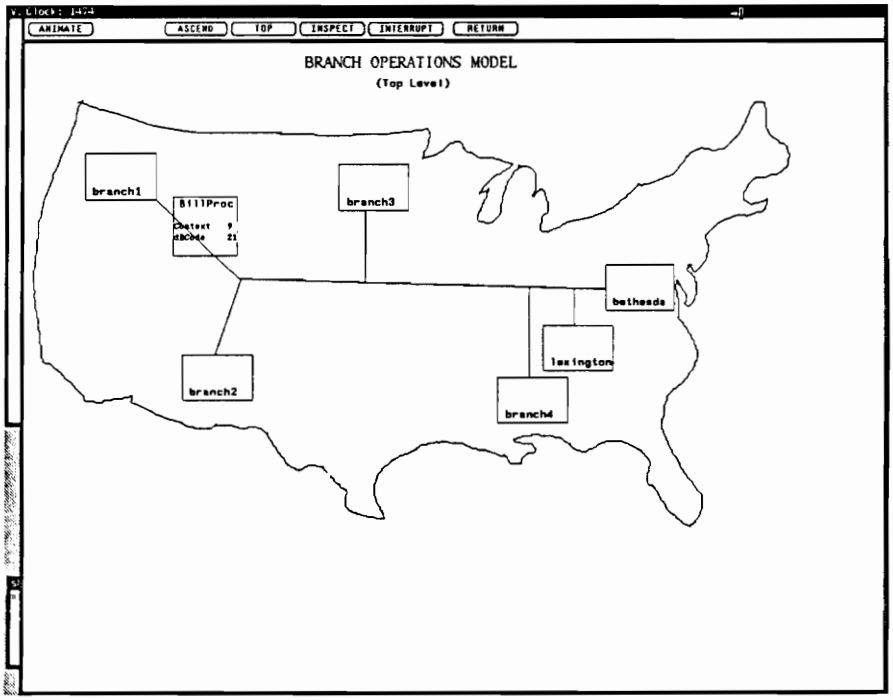

Figure 4.88 Results of Context Shift to Top Level Layout

can be followed to Bethesda, where we descend (Figure 4.89). Figures 4.90 and 4.91 show the object entering the Bethesda Computing Facility and eventually leaving. Another model component attribute inspection is demonstrated (Figure 4.92); this time, the host queue attribute **waitingList** (an array) is featured.

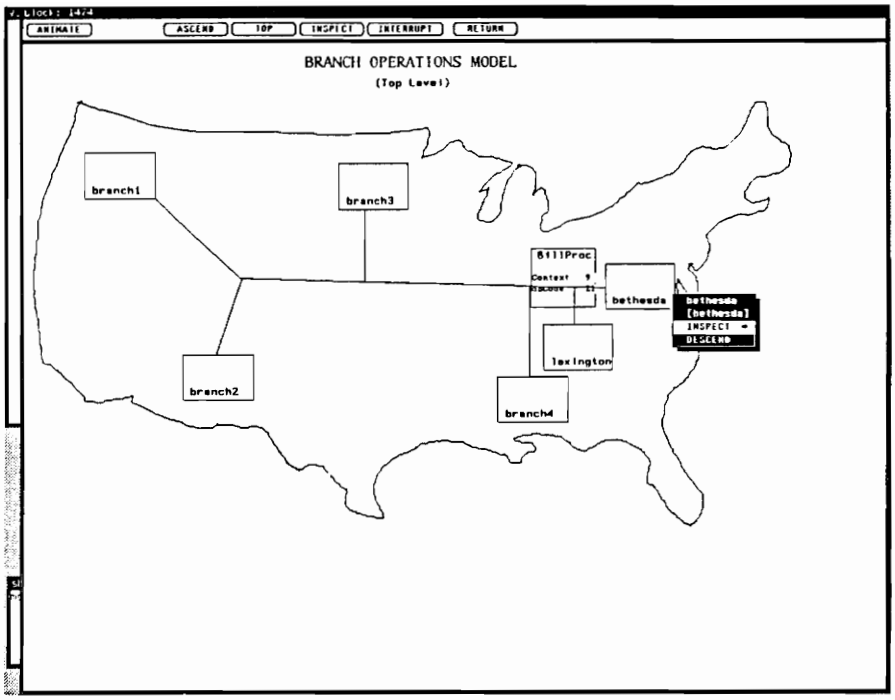

Figure 4.89 Descending into Decomposition

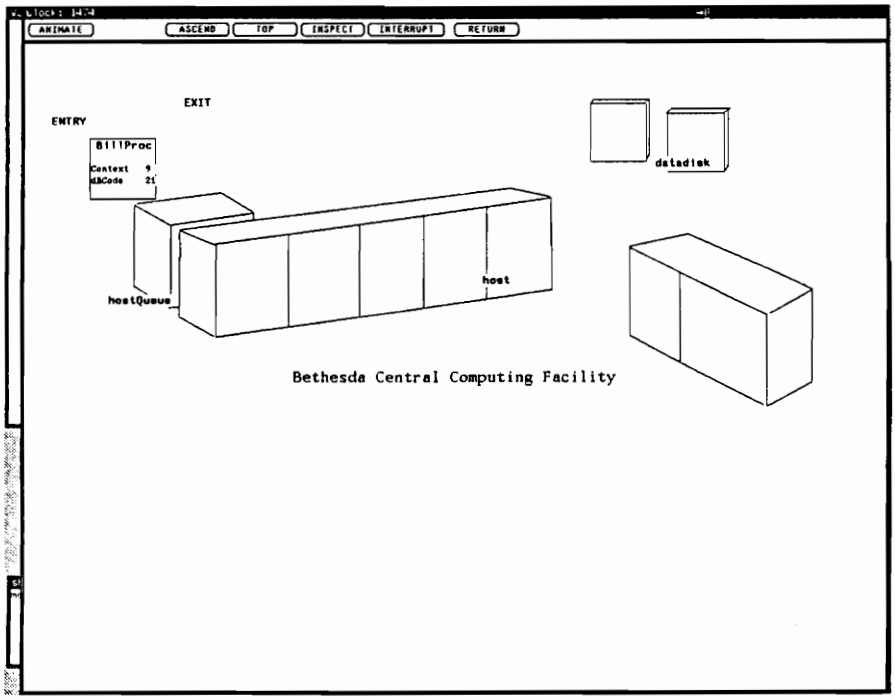

Figure 4.90 Viewing Animation after Descending (Dynamic Object Entry)

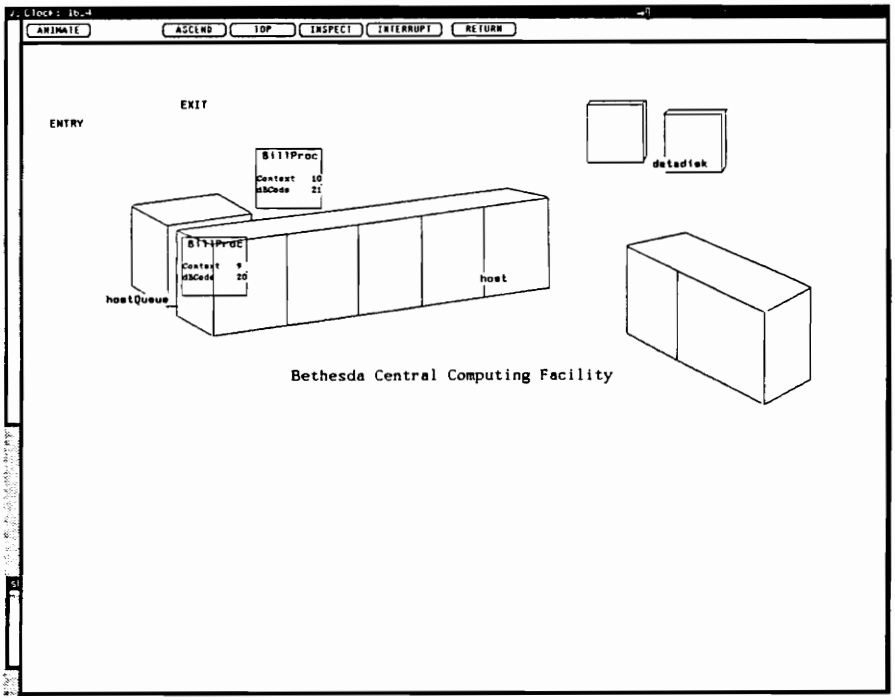

Figure 4.91 Viewing Animation of Dynamic Object Exit

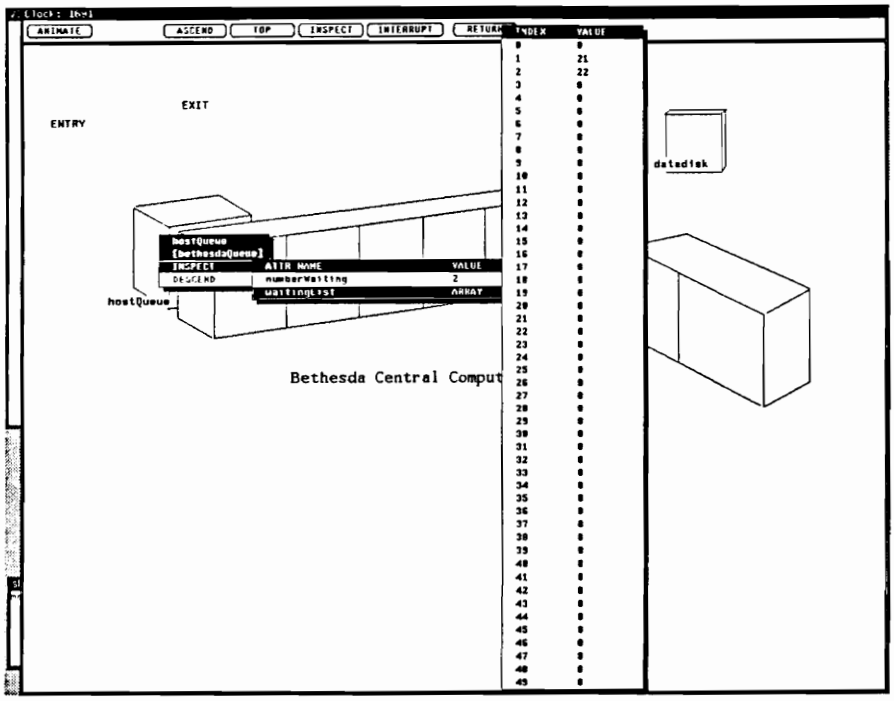

Figure 4.92 Runtime Inspection of HOSTQUEUE Attributes

CHAPTER 5
THE SPECIFICATION LANGUAGE

The specification language, called VSMSL (Visual Simulation Model Specification Language) is an integral portion of the DOMINO. The language enables modelers to specify model dynamics, the rules for component interaction, at a high level. In conjunction with the object-oriented specification of model component classes within the VSSE, modelers develop these rules for the dynamic behavior among model components as supervisory logic, self logic, or methods. This logic, written in the VSMSL, is attached to its particular model component class.

In this chapter we describe the VSMSL and discuss its importance within the DOMINO and its contributions toward a workable implementation of the Automation-Based Paradigm. In addition, this chapter provides detailed coverage of all aspects of the language: name construction, data types, and referencing of classes, components, attributes, dynamic object movement locations, and constants. The constructs of the language are generally introduced and explained. Special sections cover the development of supervisory logic, self logic, and method logic specifications. Other miscellaneous features and the conventions for comments are also included in the discussion.

Throughout this chapter, key terms are displayed in *italics*. **Boldface** (other than section headings or paragraph divisions) indicates the text is taken from the VSMSL in the form of reserved words, constructs, or example code.

5.1 General Description

The VSMSL is similar to the English-like Apple HyperTalk Language [Winkler and Kamins 1990]. During development, a goal was to retain the "feel" of HyperTalk and yet create facilities for the design and implementation for model component logic.

The VSMSL translator is composed of a sophisticated lexical analyzer and companion parser. The lexical analyzer for the VSMSL was developed using LEX (i.e., Lexical) [Lesk and Schmidt 1983], a lexical analyzer generator. Appendix A provides the BNF (Backus-Naur Form) expressions of language

statements which serve as a source for LEX. The LEX compiler produces a tabular representation of a transition diagram used by LEX-generated routines to recognize lexemes. An LALR (Look-Ahead Left-To-Right Rightmost) parser for specification statements was produced using the parser generator YACC (Yet Another Compiler-Compiler) [Johnson 1983].

Using the VSMSL translator, modelers translate the individual supervisory, self, and method logic files from within the class specification facilities of the VSSE (Section 4.1.2.1), automatically producing C source code. In order to provide static analysis capabilities and error checking during this translation process, OOP module components of the VSSE, in conjunction with the special VSSE code generation facilities, create symbol tables. During the translation, the parser is tailored for the specific class and model component logic being worked. That is, the symbol tables are modified for applicability to the current class. This is easily accomplished by a modeler within the class logic translation facility.

The creation of the individual model component logic specifications follows an object-oriented design approach. Prototyping of various model versions is undertaken without complications since each model component logic specification can be incrementally developed, translated, and saved for further use at each stage of model design. Refinements, modifications, or any other necessary maintenance is performed on the specifications. The Model Translator (Section 4.4), when activated, can automatically produce an executable version of the model only after each supervisory, self, and method logic specification is translated successfully.

Throughout the design and implementation of the model specification, modelers are completely separated from the target language and are thus removed from the low level programming details. Significant progress toward the Automation-Based Paradigm is achieved.

5.2 Basic Language Facilities

Ease of logic specification is accomplished by the provision of facilities in the VSMSL for the several categories of operations: common operations, operations specific to simulation model logic, operations supporting OOP capabilities, and operations for animation display. These are outlined as follows:

Common Operations

- attribute assignment and value retrieval,
- logic branching, looping, and decision making, and
- add, subtract, multiply, and divide operations

Operations for Simulation Model Logic

- move statements for directing the movement of dynamic objects, and
- engage statements for the performance of model component activities

Operations Supporting OOP

- create and destroy statements for model dynamic objects,
- message statements for passing messages between objects

Operations for Animation Display

- display statements for the display of data on the images of real dynamic objects

In amplification of portions of the above, the move statements direct the movement of dynamic objects among model components. These statements (depending on context of supervisory or self logic) accomplish the spatial and logical movements of dynamic objects, controlling the resulting animation display of the spatial move, and manipulating the flow of logic execution. Engage statements produce the temporal movement changes within the model; the engage statement information is used as input to the time-flow mechanism of the VSSE system. Engage statements allow time delays which are of set duration (determined) or which continue until certain conditions are met (i.e., a wait until capability).

A description of the various specific requirements of the VSMSL and its facilities follows. As a reminder, Appendix A contains examples of VSMSL statements which demonstrate the succeeding requirements.

5.3 Name Construction Rules

A name can be composed of letters and digits. The first character must be a letter. Underscores are allowed in all except class and method names. Array brackets are allowed (Single dimension arrays only).

Array indices are model attributes or integers. A name can be up to 30 characters long. Names are case sensitive. No automatic conversion to lower or to upper case is made by the translator. As a matter of convention, the user should capitalize the first letter of each word in the name except the first one. Certain words reserved for "system use only" are listed in Appendix E. These reserved words should not be used as modeler-defined names.

5.4 Data Types

Data types of model component attributes are assigned in the class specification facility within the VSSE (specifically, attribute specification). The following C language data types are available: integer ("int", "long"), real ("float", "double"), and character ("char"). The system also allows the typing of "time" for attributes of system time. Methods (return data type) and method parameters can be also typed similarly. The "void" type is also available for methods which have no data return. As a separate feature, constants can be globally defined.

Once the type of attribute, method, or method parameter is set by the modeler, the typing is not directly referred to within logic specifications but is assumed known. Invocation of the logic specification parser performs appropriate but limited type checking to catch typing errors by the modeler.

5.5 Referencing for Data Access and Object Control

Class, component, attribute, and movement location referencing mechanisms are available. In addition, constants use a special referencing mechanism. Each type follows a specific structure and set of guidelines for appropriate use. In many cases, the principles of one referencing mechanism carry over into another type. For example, both attributes and movement locations are based upon component information. Attributes belong to components; movement locations are components. Therefore, the component expressions are part of the attribute and movement location referencing mechanisms. Each type of referencing mechanism is next described. A clear understanding of these referencing principles is necessary for successful application of VSMSL constructs to model component logic specification.

5.5.1 Referencing Classes

Class names are grouped by model component type (dynamic object, submodel, static object, subdynamic object, and base dynamic object class types). Class names are unique within the class type. As mentioned in Section 5.3, underscores are not allowed in class names. Class specification within the VSSE, which includes name and attribute assignment (as described in Chapter 4), builds the foundation for component and attribute referencing under the VSMSL.

5.5.1.1 Automatic Generation of System Classes

The VSSE automatically generates several classes which can be referenced by the modeler:

- **entrypoint-** Submodel class for the virtual model component that handles entry. This class has no supervisory, self, or method logic associated with it.
- **exitpoint-** Submodel class for the virtual model component that handles exits. This class has supervisory logic which is automatically generated.
- **interpoint-** Submodel class for the virtual model component that handles interactions. This class has no model component logic of any kind associated with it.
- **builtin-** Submodel class for the virtual model component that handles model startup. This class has supervisory logic which is specified by the modeler. Within this supervisory logic, the user must move any decomposed dynamic objects (created outside of the VSMSL; Section 4.1.2.4) to their initial starting locations in the model. Additionally, any other needed dynamic objects must be created using the VSMSL.
- **distributions-** Submodel class for the virtual model component that handles random variate generation. This class has no supervisory logic or self logic but does contain methods for random variate generation. These methods are automatically generated for each model. The attributes for this class are also automatically generated. Appendix D lists the methods and attributes of this class.
- **system-** Class for the virtual dynamic startup object. This class has no logic associated with it.
- **"modelname"-** Submodel class for the model object and attributes. This class has no logic associated with it; model attributes are modeler-specified. **"Modelname"** is a generic representation for the model object class name; the appropriate model name should be substituted.

5.5.1.2 Explicit Class Requirements for Modelers

Several requirements from the above paragraph are separately highlighted so as not to get "lost in the

shuffle." First, concerning the **builtin** class, the modeler must specify the supervisory logic for the appropriate model startup. This supervisory logic is executed by the virtual dynamic object of the **system** class. Attribute values can be initialized here. An example of this supervisory logic is given in Section 5.6. Decomposed dynamic objects (such as the city bus in the Bus Route example, Chapter 7) are given their initial starting locations in the model within this logic. Secondly, the modeler must specify any model attributes that are necessary as attributes of the **"modelname"** class. This is done within the attribute specification facility of the VSSE Model Generator tool.

5.5.2 Referencing Model Components

Model component names are either *literal* or *variable*. The literal names are absolute and have meaning relative to the model as a whole, globally identifying a unique model component. Literal names are enclosed within quotation marks and are unique within a component type for a particular parent instance of a particular type. The variable names can take on different component instance associations and generally have meaning relative to a composition of components for a given class layout (described in Section 3.3.3.3 concerning compositional equivalence).

Model component names, whether literal or variable, are used in the component expressions to refer to a specific component. Components can be also referenced without using the component name. Generic references are possible which are dynamically bound to a component instance and depend on the context of the reference (e.g., **this dynamic object**). This generic referencing capability greatly improves the readability of the logic specification and supports the object-oriented character of the specification. This capability is discussed further in the next section. All references to components require that the modeler stipulate the type of model component (dynamic object, static object, submodel, base dynamic object, or subdynamic object). These may be included in the component expressions in full or may be abbreviated as **do**, **so**, **sm**, **bdo**, or **sdo** respectively.

5.5.2.1 Types of Component Expressions

Appendix A gives the details of the makeup of component expressions in the non-terminal (i.e., other expressions combine to produce the non-terminal) **<compexpr>**. Analysis of this element supports the following categorizations. The five principal types of component expressions are shown with examples from the VSMSL:

(1) *This* - **this do, this sm, this so,** etc.

(2) *Current* - **current sm, current so,** etc.

(3) *This supervisor* - **this supervising do**

(4) *Named-by-literal* - **do "ussForestal", sm "lanserver1",** etc.

(5) *Named-by-variable* - **sm lanserver, so branch, do dynobjid**

The first three represent the generic referencing capability described in the last section. The meaning of each of these reference types is carefully explained in Sections 5.6, 5.7, and 5.8 since the meaning is dependent on the context (whether supervisory logic, self logic, or method logic).

The named-by-literal and named-by-variable expression types use component literal and variable names in the expression. For non-do components (sm, so, bdo, sdo), the named-by-variable component expression uses the component's variable name which has been assigned in some generic class layout (before instantiation) as described earlier. However, for dynamic objects, the named-by-variable component expression uses a model attribute variable (defined later) holding an object identification number. For example, in the component expression **do dynobjid** (see named-by-variable examples above), **dynobjid** refers to a model attribute and is not a dynamic object variable name.

5.5.2.2 Automatic Generation of System Components

For some of the classes which the system automatically creates, the system also creates object component instances which a modeler may reference using the given literal names:

- "entry" - Virtual object of class **entrypoint**
- "exit" - Virtual object of class **exitpoint**
- "interaction" - Virtual object of class **interpoint**
- "initializer" - Virtual object of class **builtin**
- "statmodule" - Virtual object of class **distributions**
- "modelname" - Real object of class "**modelname**"

5.5.2.3 Spatial Relationships Among Model Components

Model components can be categorized by their spatial orientation in their static or dynamic structure. Three categories (local, outer, and inner) represent these spatial relationships among components and are defined below:

> *local*: Refers to neighboring components that are resident in the same layout.
>
> *outer*: Within a decomposed component, having its own layout, this refers to components up the hierarchy, components which reside in the layout just above the current layout; in other words, components one level up in the static or dynamic structure.
>
> *inner*: Within a decomposed component, having its own layout, this refers to components down the hierarchy, components which reside in the layout just below the current layout; in other words, components one level down in the static or dynamic structure, children of the decomposed component in question.

The terms local, outer, and inner are more commonly applied to describing the movement destinations of dynamic objects. However, the concepts are important to the referencing of component attributes when applied to attribute expressions which use named-by-variable component expressions. Succeeding paragraphs describe the importance of the local, outer, and inner relationships as applied to attribute referencing and dynamic object movements.

5.5.3 Referencing Attributes

Attribute expressions are generally formed by giving the attribute name and the owning component expression following the rules of component expressions. Two system reserved words augment the

referencing of attributes: **system** (can be abbreviated **sys**) and **attribute** (can be abbreviated **attr**). The requirements for the formation of attribute expressions are given below.

5.5.3.1 Types of Attribute Expressions

Appendix A contains the definition of the non-terminal **<attrexpr>** under Mathematical Statements. Due to the composition of this element, attribute expressions are of three principal types (shown with VSMSL examples):

(1) *System-*	**sys attr x of this do, sys attr x of this sm, sys attr x of sm "lanserver1"**
(2) *Model-*	**attr x, x** (Note the absence of **attr** in the second model attribute example)
(3) *Component-*	**attr x of this do, attr x of sm lanserver, attr x of sdo "motor"**

System attributes are not specified by the user but are predefined. Model attributes are specified by the user for the **"modelname"** class. Component attributes are specified by the user for the appropriate component class. Names of attributes belonging to objects of the same class must be unique. Attributes of a class may override duplicate attribute names of an inherited class. Using only the attribute name (e.g., like the second model attribute example above, **x**) implies that the attribute belongs to the whole model and is global. The user specifies the global model attributes within the "modelname" class.

5.5.3.2 Automatic Generation of Attributes

The system attributes are predefined and automatically generated for every model component instance. There are two broad groups of system attributes. Certain system attributes are available to non-dynamic object components. An entirely different set of system attributes are associated with the dynamic object instances. Appendix C lists each of these attributes and explains their use. These system attributes are available for access by modelers but care must be taken with their use. Some of these attributes are automatically maintained by the system. These system attributes should only be read from and not written

to. Changing one of these attributes could cause undesirable runtime errors. Appendix C includes the identification of those attributes which are automatically updated.

The **distributions** class has its attributes automatically generated for use in the generation of random variates. These attributes contain data for uses such as seed values, boundary values, shape and scale parameters, and mean or standard deviation data. Appendix D lists these attributes in conjunction with the system methods that use them.

5.5.4 Referencing Movement Locations

This section describes the requirements for specifying the movement locations (i.e., how-to dictate the destinations) of dynamic objects. As a matter of review, in supervisory logic, the supervisor directs this movement of the dynamic object to its next supervising logic location. This movement is animated. In self logic, the owning dynamic object directs itself from component to component for animation purposes only. Methods are not allowed to contain directions for the movement of dynamic objects. A clear understanding of the specification of dynamic object movement destinations is essential. Thus, in this separate section, the detailed requirements are given. Like attribute referencing, the principles of referencing components also apply in this situation since destinations for movement are components themselves. There are, however, key differences between attribute and movement location references. The differences are highlighted. Three concerns relate to the specification of movement locations: the type of dynamic object interaction with the destination component during movement (in/into); the spatial relationship which describes the current location of the dynamic object relative to its destination (local, inner, outer); and the location expression itself (named-by-literal or named-by-variable).

5.5.4.1 Describing Interactions at Movement Locations

Moves to non-decomposable components (static objects, base dynamic objects) are moves to the component. Moves to decomposable components (submodels, subdynamic objects, and dynamic objects) are moves into the component. The following statements offer examples of this distinction:

move into sm "lanserver1"

move to so "maincpu"

The first statement shows the designation of movement into a submodel component. The second indicates that the interaction is with the static object **"maincpu"**. The movement is therefore to the static object. Movements into static objects and other non-decomposable components (base dynamic objects) are not allowed. Similarly, movements to decomposable components (submodels, subdynamic objects, dynamic objects) are not allowed.

5.5.4.2 Spatial Relationships which Impact Movement Locations

Moves are allowed only to **local, outer,** and **inner** component destinations. This ensures that movements are across only one boundary of the hierarchy, at most. Moves (or jumps) to distant locations in the hierarchy would require additional complexity of computation as a result of calculating the appropriate path throughout the hierarchy to the destination for animation and for logic execution. Consider these statements:

move into local sm "lanserver1"

move into sm "lanserver1"

move to inner so cpu

move to outer bdo holdingArea

The first indicates that the movement destination is found in the local layout. The second statement is missing the spatial qualifier; in this case, local is assumed when there is no mention of the spatial qualifier. The third and fourth statements direct a dynamic object downward in the component hierarchy to the static object **cpu** or upward to the base dynamic object **holdingArea**. Besides reducing model complexity and giving underlying support to the VSSE system animation and executive routines, the spatial qualifiers for movement locations also improve the readability and understandability of the logic specification.

5.5.4.3 Types of Movement Location Expressions

Borrowing from the types of component expressions, movement component locations are expressed in two (and only two) ways:

(1) *Named-by-literal* - **sm "lanserver1", do "ussForestal"**

(2) *Named-by-variable* - **sm lanserver, do dynObjId** (supervisory logic only)

As before, the variable in a non-dynamic object named-by-variable location (e.g., sm lanserver) refers to the variable name of the component in a particular generic layout. The variable in a dynamic object named-by-variable location (e.g., **do dynObjId**) contains an integer representing the identifier of the dynamic object in question. Moves are not allowed to the locations **this do** or **this supervising do**; they are not named-by-literal nor named-by-variable locations and are not meaningful. Similarly, moves are not valid to the "current" component in self logic.

5.5.4.4 Special Cases

When a move is into a decomposed dynamic object (regardless if named-by-literal or named-by-variable), the context designation should be **local** or should not be included in the destination portion of the move statement. The **inner** or **outer** designations are meaningless due to dynamic nature of both the destination dynamic object and the dynamic object that is executing the move. Their context cannot be fixed. In order to simplify the complexity of computation for animation, the destination dynamic object is assumed to be within a non-decomposed component (most likely, submodel) and co-located (i.e., "local") with the dynamic object that is executing the move. Thus, no animation of the move is required. Otherwise, the component location (moving) of the destination dynamic object would need to be determined and a trajectory calculated for the animation path of the executing dynamic object in order for an animated intercept to be accomplished. Once a move into a decomposed dynamic object is accomplished, the **inner**, **outer**, **local** designations become meaningful. Each decomposed dynamic object has its "inner", "outer", and "local" information updated during moves which change its context. (For now, this has only been

done with the initial move of the decomposed dynamic object just after creation and during its initial positioning. (See the supervisory logic the **builtin** class in Figure 7.8.) Only temporary dynamic objects have been modeled crossing hierarchical barriers. Decomposed dynamic objects have remained within their created/starting context. Movement of decomposed dynamic objects across hierarchical boundaries is a matter of future implementation. In each case of such a move, the "local" and "outer" contexts of the dynamic object would have to be changed.)

5.5.5 Constants

Constants are identified and referenced within the VSMSL by following the constant name with the @ symbol. By convention, modelers should name and initialize constant values while defining other model attributes for the **"modelname"** class. Constants are viewed as global and belonging to the **"modelname"** class. Common constants are predefined by the system and are available for a modeler's use. These are listed in Appendix B.

5.6 Building Model Component Logic Specifications

The principal types of component expressions are realized within supervisory, self, and method model component logic for specific classes. The meaning of the first types of component expressions may vary, depending upon the logic type they are found within. These are now explained. Example specification code (from Chapter 7) is referred to for additional clarification.

5.6.1 Building Supervisory Logic

Within supervisory logic, **this do** refers to the dynamic object that is executing the logic. The supervisor or owner of the logic is referred to as **this sm, this so, this sdo, this bdo,** or as **this supervising do** in the case of supervisory logic owned by a decomposed dynamic object and executed by another dynamic object. In this last case, **this do** remains the dynamic object that is executing the supervising dynamic object's logic. **This supervising do** now refers to the supervising decomposed

dynamic object, necessitating the variation on **this do** for a special type of component expression. Figure 7.8 is the supervisory logic for the **builtin** class of the Bus Route example model. Figure 7.43 is the same type logic but for the Traffic Intersection example. Two other supervisory logic examples from the Traffic Intersection example are shown in Figures 7.44 and 7.45.

5.6.1.1 Attributes within Supervisory Logic

Generally speaking, for any of the model component logic types (supervisory, self, method), the proper referencing of attributes within the logic statements requires that the class of the owning component be known at compile time. The reasons behind the importance of knowing the class are described in Chapter 6 on the OOP implementation. When the class is known in the logic context, there are no problems for referencing attributes for objects of the class because the attributes can be properly typed by compile-time checks of the class symbol tables. Within supervisory logic, the class of the supervising component is always known. For any named-by-literal reference, the class is always known as well, and thus, referencing of these attributes suffers no problems. For system attributes, (whether the class is known or not) data can be stored into or retrieved from them without a problem.

Attribute expressions which utilize name-by-variable components (other than named-by-variable dynamic objects) are only meaningful within the local layout and within the context of supervisory logic. A variable name symbol table (for local names only) is built for the class which owns the supervisory logic through which the class of the attribute's component and the attribute type can be derived. Local variable names can easily be determined from the class layout memberships. Thus, access to attributes of named-by-variable components is only permissible between local components whose class (and thus data type) information is accessible. And, therefore, the dynamic object which is executing the supervisory logic can reference named-by-variable attribute expressions through the local components. Inner and outer component attribute data is not accessible via named-by-variable component expressions.

If the class is not known, data can be stored into model and/or component attributes without a problem. The data type of the attribute does not impact the storage function. For data retrieval, the data

type of the attribute must be known. Thus, there can be problems during retrieval of attribute data when the class is not known. This situation can occur with **this do** references (e.g., **attr x of this do**) and named-by-variable references for specific dynamic objects (e.g., **attr x of do dynObjId**).

In the first case, branching on the classId of this do can remedy the situation. This enables the translator to know the class at compile time. For example, the class of **this do**, the dynamic object executing the supervisory logic, cannot be known at compile time unless a branch is done on the system attribute classId. With no class information in the absence of a branch on the class, the VSSE's underlying implementation of the VSMSL assumes the retrieval is for an attribute of this do with an integer type. In the second case, there is no way to know the dynamic object class, and the integer type is assumed. For both cases when the class is not known, if the attribute type is truly an integer, then the retrieval can still be done properly. If the attribute is not of integer type and the assumption by the compiler is false, expect unpredictable runtime results.

5.6.1.2 Movement Locations within Supervisory Logic

Unlike the component expressions which are named-by-variable in attribute references (only local component data can be accessed), named-by-variable component names can be used to refer to local, inner, or outer movement locations. Named-by-variable movement locations, like named-by-variable attribute references, are usable only within supervisory logic. At compile time, each supervising component has its local, inner, and outer components defined and accessible by variable name. This can be done because the context of each supervising component does not change during model execution.

5.6.1.3 Move Statements within Supervisory Logic

Movement of **this do** (implied) is the prevailing use of the move statement and concerns the movement of a dynamic object already in motion and in the process of executing some supervisory logic. However, when a particular dynamic object is specified for a move, the translator will produce the correct code to accomplish the move. This, in effect, is an interruption to an object's existing logic flow. The simulator,

however, has not yet been modified to handle the effects such a move would or could have upon the current and future objects list (should an object be directed to move that is already on one of these lists). The "moved" object would have to be found and removed from the appropriate list and then reinserted where required.

The translator will generate the code to handle such movement of a specified dynamic object to any location but, for now, the simulator requires the location be one which is decomposed. The typical case where this has been used and tested is with the move of a decomposed dynamic object (created outside of the logic during graphic specification in the model generator). In the initial logic of a simulation (the **builtin** supervisory logic), the dynamic object is moved into its starting location for the simulation. (See last example for move statements in Appendix A). The movement of a specified dynamic object (as opposed to the implied case) to start in a decomposed dynamic object has not been tested. This is a matter for future implementation and would require that the dynamic object, the one being moved, have its context "outer" and "local" set dependent on the context of its starting location. The current implementation easily sets this since non-dynamic object decomposable starting locations have a static context.

5.6.2 Building Self Logic

Within self logic, the owner of the logic is called **this do** which directs it own movement from one component to another. The component that **this do** is currently within is referred to as **this sm**, etc. Alternatively, the component in which the dynamic object resides is called the "current" one (e.g., **current sm, current so**, etc.) defining the meaning of this type of component expression. Note that "current" component expressions have meaning only in the context of self logic and give a very natural means for expressing the immediate component location. The self logic of the Lane 8 Vehicle class for the Traffic Intersection example is displayed in Figure 7.55.

5.6.2.1 Attributes within Self Logic

The previously mentioned effects for supervisory logic (resulting from dependencies upon knowledge

of the class name) upon attribute referencing also apply here. The owner of the self logic is called **this do**. In this context, the class of **this do** is known at compile time. Recall that the acquired component (the component **this do** is currently within) is for example, **this sm**, **current sm**, etc. For attribute referencing, the class of the current component is also known at compile time. Within self logic, the executing dynamic object has access to the currently acquired component's data and to any model attributes. Local, inner, and outer components are relative to the owning component, the dynamic object. But now, the context of the dynamic object is always changing. Current implementation does not provide access to the data of these components due to the dynamic context. Named-by-literal references to attributes are necessary then in most cases. Named-by-variable referencing is not permissible due to the ever-changing local context.

5.6.2.2 Movement Locations within Self Logic

For self-logic, movement destinations are forced to be named-by-literal expressions since the context of the owning dynamic object is continually changing during the course of the execution of the self logic. Named-by-variable locations are therefore illegal.

5.6.2.3 Move Statements within Self Logic

The move statement in this context produces animation of the spatial move. There is no transfer of logic execution since the self logic contains all of the logic for the owning dynamic object. Another type of move statement is unique to self logic, the executing or exec move.

When specifying the self logic of a dynamic object, there is a need to allow the encapsulation of some of the logic that the dynamic object (owning the self logic). Otherwise, it would be necessary to include all of the logic (in excruciating detail) that the object is to execute within the self logic. The executing move (e.g., **move!**) allows the dynamic object to temporarily move into the supervisory logic of another component. Note that if such moves are used, the supervisory logics that are "moved" into must not contain move statements of their own. For now, move locations cannot be "stacked" and execmoves can

only be one move deep. This would be an area for future implementation. The executing move produces a temporary shift in logic source and produces the resulting change in animation display as well. (Remember, this shift is only temporary and is only one call deep.) Subsequent "moves" are not allowed from within the supervisory logic which is executed during this type of move.

5.6.3 Building Method Logic

Before the method can be successfully translated, the method name must be first saved to the database. Once the name is stored, the translator can effectively access the information in the symbol tables which relates to that method. Within each method, the generic component referencing mechanism holds and the owning component of the method is referred to as **this do, this sm, this so,** etc. Move statements are not allowed in methods. Figures 7.50, 7.51, and 7.54 give representative method logic from the Traffic Intersection example.

5.6.3.1 Parameters and the Method Result

Two specialized data holders are utilized within methods: method parameters and the method **result**. These are not attributes of any model component as such but are meaningful only within the method itself. The method **result** is used as the container for information being returned by methods other than void type. The parameters contain information passed to the method for its use. They are passed in to the method via a message statement as attributes of some type. But within the method, they are identified by the parameter name.

The **result** is used as the container which a method returns, if it returns a value. Thus, the method can return a value through the **result**, but the return statement must be explicitly used in this case. You cannot "get" the **result** nor "branch" on it. "Repeat" while incrementing or decrementing **result** is also not allowed. **Result** cannot be used in an expression.

Parameters are allowed in "get", "repeat", and within expressions, unlike result. Parameters are allocated in alphabetical order and should be used in the message statement argument list in that order.

The number and type of arguments to a method is not checked by the compiler.

5.6.3.2 Attributes within Methods

Since the class of the owning component (**this do, this sm,** etc.) is known at compile time, the referencing of its attributes is problem free. The owning component has access to its own data, that of the model (model attributes) and any parameter data which has been passed in. As always, any named-by-literal attribute can be accessed. Within the method logic, names and identifiers are first checked against a parameter list by the translator; any identifier not found is assumed to be a model attribute.

5.7 Comments and Other Miscellaneous Features

The VSMSL provides a modeler with the ability to put comments into the logical specification. The use of "the" and line continuation are also considered.

Comments

Two hyphens "--" are used for in-line and whole line commenting. The compiler ignores anything to the right of "--".

Use of "the"

The use of **the** is optional and ignored by the compiler.

Statement Continuation

Each statement is ended by a semicolon. To continue a statement to the next line, all that is necessary is to hit <return> and continue on the next line. Good specification logic practice dictates that the use of white space will keep the logic readable. The translator ignores the end-of-line at the return and continues processing, looking for the semicolon to end the statement.

CHAPTER 6
THE OOP IMPLEMENTATION

All previous prototypes of the VSSE (Chapter 4) were fully developed in the C programming language. Because of the very narrow and limited Object-Oriented Programming (OOP) capabilities of these earlier prototypes, major improvements were needed, especially to the inheritance mechanism. The use of an object-oriented language (like C++) would have been ideal for this. Incompatibilities between C++ and INGRES/EQUEL C made this course infeasible. Therefore, the OOP facilities were developed directly in C using a combination of approaches adapted primarily from techniques presented by White [1990] with some limited incorporation of ideas from Brumbaugh [1990].

This section describes the implementation of the OOP facilities under C which, without the benefit of C++ or an equivalent, presented interesting challenges and obstacles to the evolutionary development of the DOMINO and the VSSE environment. The implementations of the OOP facilities, VSMSL, and the LEX/YACC VSMSL translator were intertwined; design decisions in one area extended into and affected the others. In the final OOP implementation, facilities include the creation of classes and objects, the use of methods with a limited message passing capability, and a full single inheritance mechanism. Thus, a class can only have a single superclass. Multiple inheritance is not possible. Name conflicts among attributes and methods along the inheritance hierarchy are resolved. Polymorphic referencing among object instances is not possible. Once created, an object remains a member of its class (as created). That is, the static and dynamic references to the object are the same throughout the life of the object.

The VSMSL and its translator have been built in such a way as to allow the referencing name **this object** to be dynamically bound to objects of any class. Whether the class of **this object** is known at compile time has implications on the handling of runtime references to attributes of **this object**, as discussed in Sections 6.1 and 6.3. Thus, in a sense, there is a limited capability for dynamic binding of references to an object and its attributes.

First, we discuss the implementation of classes and objects using the defining data structures. The principal routines and overall organization is shown, and includes class and object creation and the VSSE

implementation for methods and message passing. The underlying inheritance mechanism is described.

6.1 Class and Object Structures

Figure 6.1 gives the two C structures called "class" and "object" which establish the basic contents of every class and instance, respectively.

The class structure has two parts: a standard part which is the same for all classes across all models and a variable part which is tailored to each specific model. A memory data space is allocated for each defined class and is shared among its object instances. The standard part includes:

- **size** - the amount of memory required for each object instance of the class,
- **type** - a numeric identifier for the class,
- **dynobj** - a Boolean variable which is true when the class is a dynamic object class,
- **objImage** - the name of the class owning the object image, and
- **layImage** - the name of the class owning the layout image.

The variable part is dependent upon the superset of defined methods in the model across all classes. This and other variable parts are generated and placed in "include" files by the automatic code generation components of the VSSE. This variable part contains arrays of pointers to methods; the number of arrays depends upon the superset of the return types of the defined methods.

Every component of the object structure is standard, the same for all objects. Objects contain five pointers to various data elements:

- one to the object's user-defined data,
- one to the object's system-defined data,
- one to the object's class data,
- one to an object, next on a system list (e.g., the Current Object List), and
- one to an object, previous on some system list.

The first two pointers are to data and memory locations which are unique for the object. The storage requirements of the user-defined class data (i.e., class attributes) varies among classes. There are two

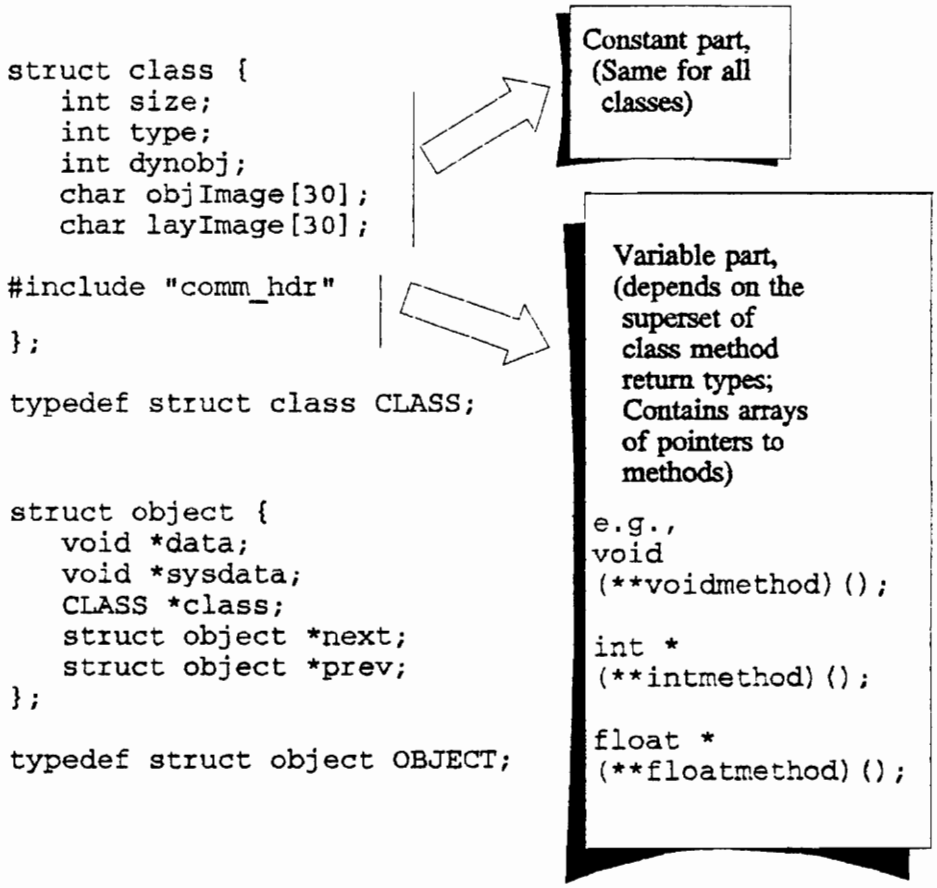

Figure 6.1 CLASS and OBJECT Structures

groups of system-defined data, one for non-dynamic object instances and one for dynamic object instances. Figures 6.2 and 6.3 give the definitions for non-dynamic and dynamic object instances, respectively. All system-defined variables for dynamic or non-dynamic instances are given in a single C definition. Because of this, there can be no duplicate names among the system-defined variables or attributes. As long as this convention is followed, then system-defined attributes can be accessed quickly during runtime for object instances, without the need for type casting of the system-defined data space. Further, there is no need to have knowledge of the object's class in order to reference a system attribute. This is not true for user-defined data. Runtime references to user-defined data require the data space be type cast according to the class of the object instance so that the data can be accessed properly (described in Section 6.3.2, Data Access Scheme for Data). The programming approach for the system attribute definitions was adapted from Brumbaugh [1990]. The system-defined variables are described in the Chapter 5 presentation of the VSMSL. The third pointer points to the class information for the object. This data space is shared by all object instances of that class. The last two pointers are critical to the management of simulation model execution by the VSSE system code. They enable object instances of any class to be linked together on any of the system lists (Current Object List, Future Object List, etc.) for model execution. Previous prototypes of the VSSE required that the data structures linked together be of the same type and having the same storage requirements (which was common across all classes). The latest implementation, however, requires they only be of the same type, **OBJECT**. Now, linked objects may vary in size and storage requirements resulting in a memory savings. Figures 6.4 and 6.5 graphically present the class and object structures. During model initialization, all defined class structures are automatically created. In addition, all objects which are in existence at model initiation (not stochastically generated during a model run) are created at the same time. The next section discusses, in part, the class and object constructors.

6.2 OOP Routines and Organization

The organizational structure and underlying routines follow the approach of White [1990]. Figures 6.6 through 6.11 give the utility routines which are the core of the OOP implementation. These procedures

```
#define sys_data   int noOfDos;      \
                   int entryList[MAX_NUM_DOS]; \
                   int realVirt;     \
                   int xpos;         \
                   int ypos;         \
                   int width;        \
                   int height;       \
                   int compInst;     \
                   int classIdent;   \
                   Pixrect  *comp_image;



typedef struct   {
   sys_dat
}SYS_TYPE;
```

Figure 6.2 Non-Dynamic Object System Data

```
#define dyn_sys_dat int ident; \
        int numDos;  \
        int dosList[MAX_NUM_DOS];\
        int doInst;  \
        int realOrVirt; \
        int classId;   \
        char className[30]; \
        int originComp; \
        int currentComp;  \
        OBJECT *currentCompPtr;\
        int execMove;    \
        int currentSuperLoc; \
        int currentSelfLoc;   \
        int tempComp;    \
        OBJECT *tempCompPtr;    \
        char nextCompName[30];\
        int nextCompType; \
        int prevComp;    \
        char prevCompName[30];\
        int prevCompType; \
        int layoutIndex;  \
        char display[4][20];   \
        Pixrect *DO_image;
        int newImage;    \
        char imageName[30]; \
        time delay; \
        time moveTime;    \
        time compEntry;\
        time modelEntry;

typedef struct {
   dyn_sys_dat
}DYN_SYS_TYPE;

#define SYS_SIZE  sizeof(SYS_TYPE)
#define DYN_SYS_SIZE  sizeof(DYN_SYS_TYPE)
```

Figure 6.3 Dynamic Object System Data

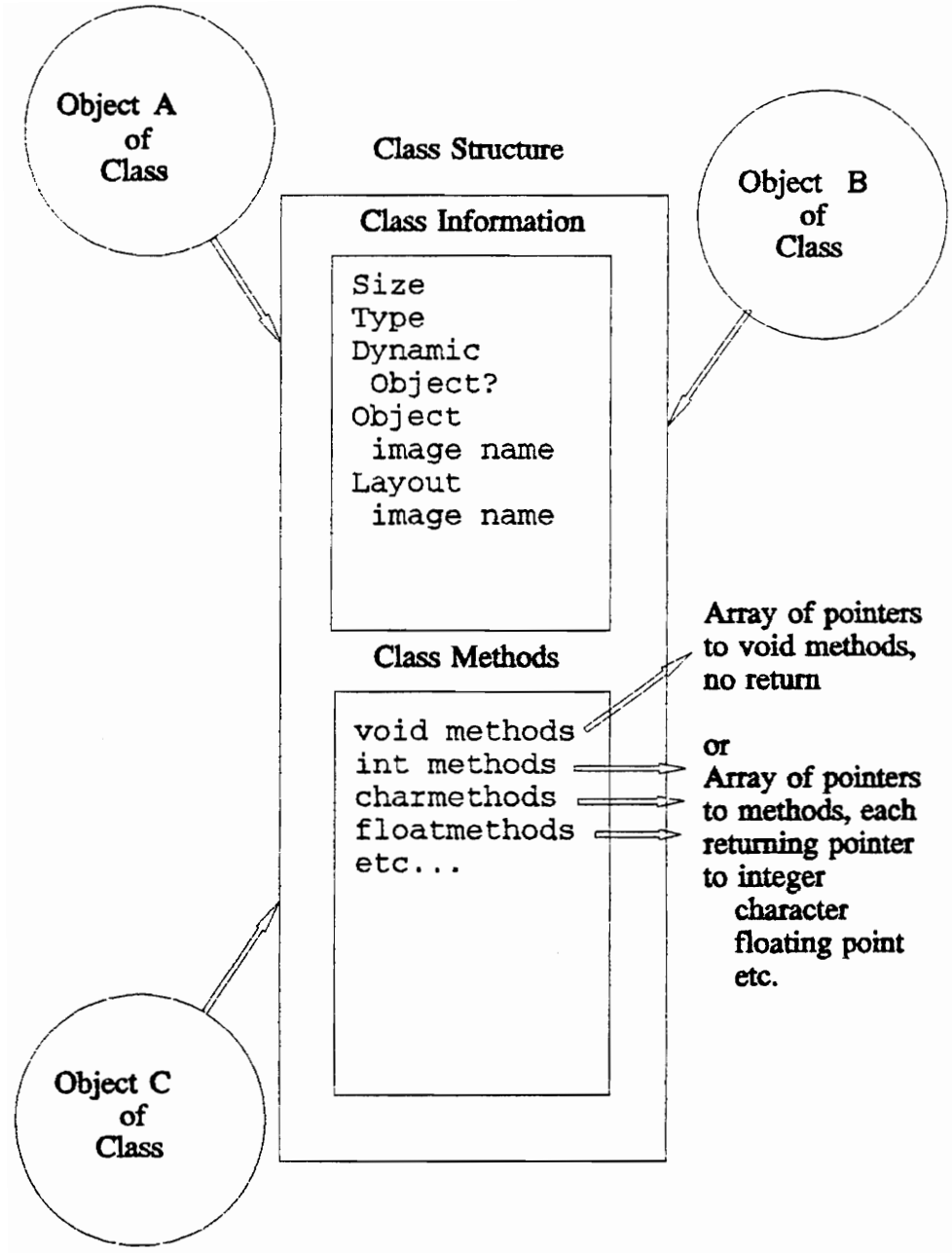

Figure 6.4 Class Structure for Each Defined Class

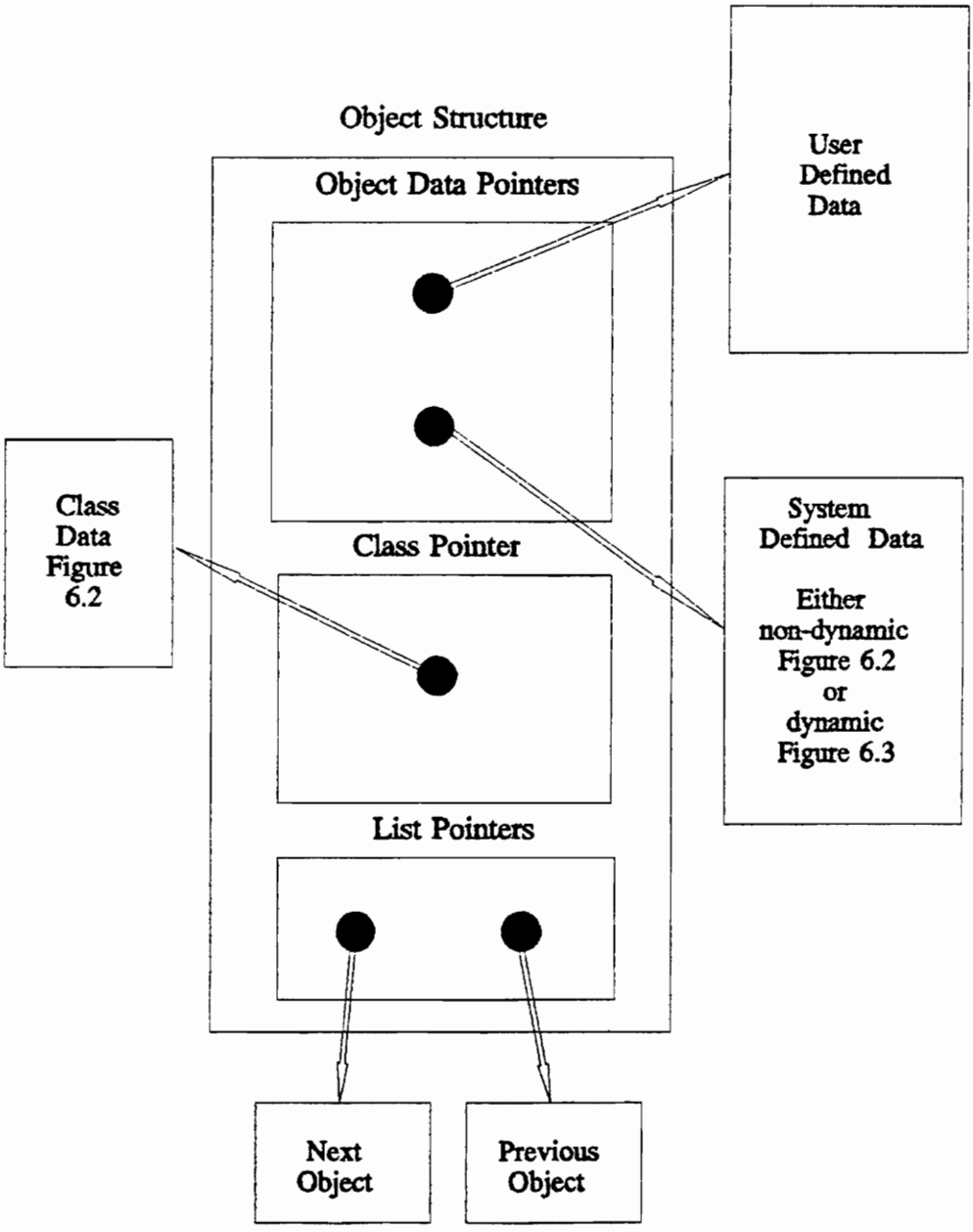

Figure 6.5 Object Structure for Each Object Instance

```
CLASS *new_class(super_class, size, type)
   CLASS *super_class;
   int size;
   int type;
{int i;
 CLASS *class;

 /* Allocate memory */
 class = (CLASS *) malloc (sizeof(CLASS));
 class->size = size;
 class->type = type;

#include "new_class.h"

 /* Inherit methods of superclass */
 inherit_methods(class, super_class);
 return(class);
}
```

```
e.g.,
class->voidmethod =
 (void (**)())malloc
   ((unsigned)(MAX_VOIDS * sizeof(void(*)())));
for (i = 0; i < MAX_VOIDS; ++i)
 class->voidmethod[i] = (void (*)())NULL;
class->intmethod =
 (int * (**)())malloc
   ((unsigned)(MAX_INTS * sizeof(int * (*)())));
for (i = 0; i < MAX_INTS; ++i)
 class->intmethod[i] = (int * (*)())NULL;
```

Figure 6.6 Routine (new_class) for Creating Classes

```
OBJECT*new_object(class)
 CLASS *class;
{
 OBJECT *new_obj;
 void *user;
 void *sys;

 /* Allocate memory for object */
 new_obj = (OBJECT *) malloc (sizeof(OBJECT));

 /* Allocate system data memory */
 if (class->dynobj)
   sys = (void *)malloc((unsigned) DYN_SYS_SIZE);
 else
   sys = (void *)malloc((unsigned) SYS_SIZE);
 if (sys == NULL)
   fatal("System object malloc failed");
 new_obj->sysdata = (void *)
   ((unsigned char *)sys);
 /* Point the object to its class */
 new_obj->class = class;

 /* Allocate space for user defined data */
 user = (void *)malloc((unsigned)class->size);
 if (user == NULL)
   fatal("Object malloc failed");

 /* Assign allocated space to the data */
 new_obj->data = (void *)((unsigned char *)user);

 /* Initialize link pointers to NULL */
 new_obj->next = (OBJECT *)NULL;
 new_obj->prev = (OBJECT *)NULL;
 return(new_obj);
}
```

Figure 6.7 Routine (new_object) for Creating Objects

```
void free_class(class)
 CLASS *class;
{
 /* Free space allocated for class methods */

#include "free_class.h"

}
```

```
free(class->voidmethod);
free(class->intmethod);
free(class->floatmethod);
free(class->doubmethod);
free(class->timemethod);
etc....
```

```
void free_object(obj)
 OBJECT *obj;
{
 if (obj != (OBJECT *)NULL)
   {
   free(obj->data);
   free(obj->sysdata);
   }
}
```

Figure 6.8 Routines (free_class, free_object) for Clearing Memory Space

```
void inherit_methods(class, super_class)
 CLASS *class;
 CLASS *super_class;
{
 int i;

 if (super_class != NULL)
 {
   /* Inherit superclass methods */
   /* Store pointers to all of   */
   /*   superclass methods       */

#include "inh_meths.h"

 }
}
```

```
for (i = 0; i < MAX_VOIDS; ++i)
 class->voidmethod[i] =
super_class->voidmethod[i];

for (i = 0; i < MAX_INTS; ++i)
 class->intmethod[i] =
super_class->intmethod[i];

...

for (i = 0; i < MAX_TIMES; ++i)
 class->timemethod[i] =
super_class->timemethod[i];
```

Figure 6.9 Routine (inherit_methods) for Inheriting Methods among Classes

```
#include "reg_meths.h"

void reg_voidmethod(class, methnum, fnc)
 CLASS *class;
 int methnum;
 void (*fnc) ();
{
 if (methnum < 0 || methnum >= MAX_VOIDS)
   fatal("Attempting to register invalid void
    method");
 class->voidmethod[methnum] = fnc;
}

void reg_intmethod(class, methnum, fnc)
 CLASS *class;
 int methnum;
 int * (*fnc)();
{
 if (methnum < 0 || methnum >= MAX_INTS)
   fatal("Attempting to register invalid int
    method");
 class->intmethod[methnum] = fnc;
}

etc....
```

Figure 6.10 Routines for Registering Methods

```
#include "messages.h"

void voidmessage(va_alist)
 va_dcl
{
 va_list arg_ptr;
 OBJECT *obj;
 int msg;
 va_start(arg_ptr);

 /* Get the object and method ordinal */
 obj = va_arg(arg_ptr, OBJECT *);
 msg = va_arg(arg_ptr, int);

 /* Check validity of msg index */
 if (obj->class->voidmethod[msg] == NULL)
   {
   printf("Problem at method index %d &
     class %d\n",    msg, obj->class->type);
   fatal("No void method of this type and class");
   }

 /* Call the valid method */
 (*obj->class->voidmethod[msg])(obj, arg_ptr);
 va_end(arg_ptr);
}

int *intmessage(va_alist)
 va_dcl
{
 va_list arg_ptr;
 OBJECT *obj;
 int msg;
 va_start(arg_ptr);
 ... as above
 return((int *)(*obj->class->intmethod[msg])
   (obj, arg_ptr));
 va_end(arg_ptr);
}
```

Figure 6.11 Routines which Implement Message Passing

are the baseline routines which are called by others. Variable parts (contained in "include" files and produced by the code generator) are expanded for clarity. Class and object creation, methods, and messages are those facilities which exercise the baseline routines. In the next section, we cover class and object creation via the constructors, methods, and message passing. The baseline routines are discussed in conjunction.

6.2.1 Class and Object Creation

Using class and object information provided by the user (see Chapter 4, VSSE), the code generator produces the class and object constructors, examples of which are given in Figures 6.12 and 6.13. The class constructor of Figure 6.12 is called by model initialization routines. Typical constructor calls from the initialization routine are also given in Figure 6.12. Each class is created once. Within the class constructor, the **new_class** routine is called to create the basic class unit. Figure 6.6, the **new_class** routine, is the baseline procedure for the creation of classes. First, memory is allocated for the class; the size and type (both passed in) are assigned to the class. The variable part of the procedure allocates pointer space for the methods of each type contained in the class and initializes space for each (all possible) method of that type. Finally, once the methods space is set up, the methods of the superclass are inherited. The pointer to the class is returned to the constructor. From the constructor, object and layout image names are assigned. The **dynobj** Boolean is set, and the class methods (of various possible types: void, time, int, etc.) are registered. The class is returned to the system for use during model execution.

The object constructor of Figure 6.13 is also called by model initialization for those objects which are present in the model at the initiation of model execution. Examples of the calls to the constructor are also given in the same figure. Within the object constructor, the baseline call to the routine **new_object** creates the basic object. **New_object** allocates memory space for the object (uninitialized). Memory space is next allocated for system data (as dynamic or non-dynamic). Space is then allocated for user-defined data which is based on the data requirements of the class size. Next and previous pointers are set to NULL and the new initialized (in data space only) object is returned to the constructor. The constructor finishes object

```
CLASS *new_SM_LANSVR_class()
{
 CLASS *class;

 class = new_class(NULL, SM_LANSVR_SIZE,
    SM_LANSVR_TYPE);

 strcpy(class->objImage, "lanSvr");
 strcpy(class->layImage, "lanSvr");
 class->dynobj = FALSE;

 reg_voidmethod(class, STATUS_SET,
    statusset_lanSvr_sm);
 reg_intmethod(class, STATUS_ADDR_INT,
    statusaddr_lanSvr_sm);
 ...
 reg_intmethod(class, SULOG,
    lanSvr_sm_sulog_proc);

 return(class);
}
```

Constructor Calls:

```
sm_lanSvr = new_SM_LANSVR_class();
...
sm_lanQueue = new_SM_LANQUEUE_class(sm_queue);
```

Figure 6.12 Class Constructors and Calls

```
OBJECT *new_SM_LANSVR_object()
{
 int i;
 OBJECT *this;
 OBJECT *super;
 this = (OBJECT *)new_object(sm_lanSvr);
 voidmessage(this, STATUS_SET, IDLE);
 ...
 return(this);
}

OBJECT *new_SM_LANQUEUE_object()
{
 int i;
 OBJECT *this;
 OBJECT *super;
 super = (OBJECT *)new_SM_QUEUE_object();
 this = (OBJECT *)new_object(sm_lanQueue);
 memcpy(this->data, super->data, SM_QUEUE_SIZE);
 free(super);
 return(this);
}
```

Constructor Calls:

```
lanQueue1_sm = new_SM_LANQUEUE_object();
lanQueue2_sm = new_SM_LANQUEUE_object();
...
lanSvr1_sm = new_SM_LANSVR_object();
lanSvr2_sm = new_SM_LANSVR_object();
```

Figure 6.13 Object Constructors and Calls

initialization by making assignments to the object's attributes.

6.2.2 Methods and Message Passing

Methods, as defined for a particular class, are unique to that class. Thus, for all object instances from the class, the methods do not vary. For this reason, access to an object's methods are through its class structure. (On the other hand, an object's data is unique to the object and are accessed via the data pointers within the object.) Figure 6.4 shows this relationship of an object to its methods. Methods may be of various types (void, integer, float, etc.) depending on the return type of the operation. During code generation, the model specification is scanned and the types of methods are retrieved. Each unique method name is logged and a constant identifier for that name is defined. In addition, the total number of each method type is calculated. A constant for each number is defined (e.g., **MAX_VOID**, **MAX_INT**, **MAX_FLOAT**, etc.). This information is used for the method and message passing implementation.

As described earlier, during class creation and the **new_class** routine (Figure 6.6), memory is allocated for arrays which hold pointers to the individual methods (or C functions). An array is defined for each method type that exists in the model. The size for memory allocation for each array is determined from the number of methods of that type (e.g., **MAX_VOID**, **MAX_INT**, etc.). For the method implementation for each class, then, extra space is allocated to hold pointers to methods in these arrays. This is seemingly a waste of space but the allocation is made with a purpose. After space for these arrays is allocated (Figure 6.6), methods of the superclass are inherited with the **inherit_methods** routine (Figure 6.9). **Inherit_methods** assigns the pointers of a class's methods arrays to those of the superclass (if any). Methods for the class itself are registered by the class constructor (Figure 6.12) with a call to the appropriate baseline routine, **reg_voidmethod**, **reg_intmethod**, etc. (See Figure 6.10). The registering routine is passed a pointer to the class, the method identifier, and a pointer to the method (which coincidentally has its code automatically generated by the code generator). The method identifier is validated and the pointer to the method is assigned to the class methods array of the appropriate type. Should the method name of a class be the same as a method in the superclass, the inherited method's

pointer will be overwritten by that of the subclass. Thus, in the class hierarchy, the most recently defined method is used when there are name conflicts. The approach taken for allocation of the method arrays for a class and pointers to its respective methods makes the inheritance of methods an easier task and provides name conflict resolution among methods of the class hierarchy.

The message functions which implement message passing for the OOP implementation (Figure 6.11) decide which method to implement. Variable argument capabilities (Standard arg.h C header) are utilized to make the message calls. For the activation of a method, a message is passed to the object which invokes the method. The message contains a pointer to the object and a pointer to a variable argument list. The first argument is the method identifier. Before the method is activated by the message, the method identifier is validated. If the method is valid, the method identifier is used as an index to the class methods array. The appropriate method of the class is then executed. Upon method invocation, the method is passed a pointer to the object and the remainder of the argument list. White's [1990] approach does not allow for methods which return a value or pointer. The current VSSE implementation does.

6.3 OOP Inheritance Mechanism

The single inheritance mechanism of user-defined data among classes is now described. Using the class specification created by the user (Chapter 4, VSSE), class attributes are defined in C structures similar to those of Figure 6.14. Classes with no attributes are given a dummy attribute. Note that a size definition is made for the size requirements of the class attribute data for later mallocs during object creation. Note also that base class size is given an offset of zero. Subclass sizes (e.g., SM_LANQUEUE_SIZE) are set by taking the size of the subclass attribute data and adding the size of the parent class data. An offset for the class data is set to be the size of the parent class. The size requirements are used for memory allocation during object creation. The offset information is needed for access to object data.

```
typedef struct sm_lanSvr_struc {
 int status;
 time ptravel;
 time htravel;
 time dectime;
 time prtime;
}SM_LANSVR;

#define SM_LANSVR_SIZE sizeof(SM_LANSVR)
#define SM_LANSVR_OFFSET 0

typedef struct sm_lanQueue_struc {
 int dummy;
}SM_LANQUEUE;

#define SM_LANQUEUE_SIZE sizeof(SM_LANQUEUE) + \
  SM_QUEUE_SIZE
#define SM_LANQUEUE_OFFSET SM_QUEUE_SIZE
```

Figure 6.14 Class Definitions

6.3.1 Memory Allocation Scheme

The allocation of memory space for user-defined data for object instances is given in Figure 6.7, the **new_object** routine, as a malloc for user. The amount of space for the allocation is taken from the class size which is set during class creation (Figure 6.6 and 6.12). The class constructor passes in the defined size of the class attribute structure (as described above) to the **new_class** routine. This size is that required for the class and includes space required for attributes of the superclass. Thus, subclass sizes are greater than those of parent or base classes. As objects are created which have a class hierarchy which is more than one level deep (i.e., contain more than one class), additional space is allocated. The data is partitioned in memory to consist of the parent data followed by the subclass data. Within the object constructor for such an object (Figure 6.13, **new_SM_LANQUEUE_object**), a pointer to a temporary object of the superclass is created. The data space from this temporary object is copied to the subclass object with the **memcpy** C library function. The superclass data is at the head of the space allocation. The subclass data is located at the offset distance from the head. Figure 6.15 displays the memory organization for a three-level class hierarchy.

6.3.2 Data Access Scheme

Two system-defined methods are created for every attribute for every class. A "set" method, of void type, for a particular attribute, is sent to an object at any time that an assignment is made to the attribute. An "address" method, of the same return type as the specified attribute, is sent to an object at any time that the attribute value must be retrieved. Figure 6.16 gives representative examples of these two types of methods for an attribute (integer) **status** of class **lanSvr**. Note the pointer **tmp** is type cast (i.e., **(SM_LANSVR *)**) according to the class type for the proper accessing of the attribute in both system methods. The message invoking the assignment of the initial attribute value to **status** is given in Figure 6.13, the object constructor for **lanSvr** class objects. Through the void message, the "set" method for the **status** attribute of the class is invoked with the method identifier **STATUS_SET**. The void message passes the "set" method a pointer to the object, **this**, and the remaining argument list pointer. The remainder of

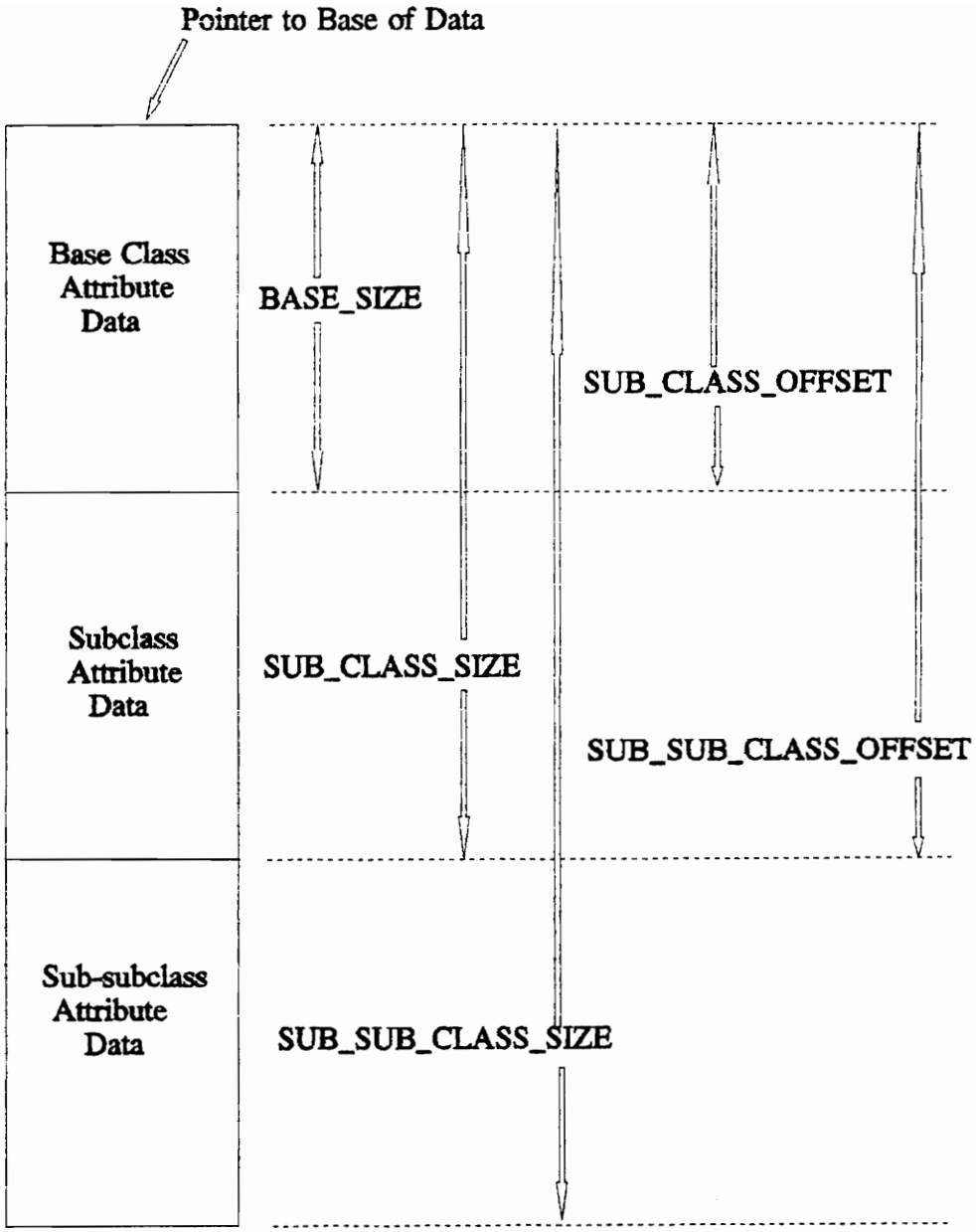

Figure 6.15 Memory Organization and Data Allocation Scheme

```
void statusset_lanSvr_sm(this, arg_ptr)
 OBJECT *this;
 va_list arg_ptr;                          Pointer is type cast
{
 SM_LANSVR *tmp;
 tmp = (SM_LANSVR *)((unsigned char *)this->data +\
    SM_LANSVR_OFFSET);
 tmp->status = va_arg(arg_ptr, int);
}

int *statusaddr_lanSvr_sm(this, arg_ptr)
 OBJECT *this;
 va_list arg_ptr;
{                                          Pointer is type cast
 SM_LANSVR *tmp;
 tmp = (SM_LANSVR *)((unsigned char *)this->data +\
    SM_LANSVR_OFFSET);
 return(&tmp->status);
}
```

Figure 6.16 System Methods for Read and Write Access to Attributes

the argument list contains the value IDLE to be assigned to the attribute status. Attribute assignments and value retrievals in the VSMSL are translated to messages which invoke the appropriate "set" or "address" methods of the attribute.

During translation of a logic specification (written in the VSMSL) where there are references to an object's attributes, the class hierarchy (when the class is known) is traversed to determine if a particular attribute resides at any level of the hierarchy. This is accomplished using the class symbol tables. If an attribute is not found in the current class attribute tables, the parent class attribute tables are scanned, and so forth, up the hierarchy, until the attribute is located. Should there be name conflicts along the class hierarchy among attributes, the most-recently defined attribute with the given name, regardless of type, is the one which is validated. Once located, the code for the appropriate access method call is generated. If an attribute name cannot be validated for a class during translation, an error message is given. Therefore, class attribute hierarchies may be defined which contain name conflicts among attributes. However, should a subclass have an attribute (of the same or different type and) of the same name as a parent class attribute, the most recently defined attribute is the one used for objects of the subclass.

If the reference to an object's attribute is within an assignment statement, the "set" method of the object is used. The appropriate method (like in Figure 6.16) can be properly coded by the translator without knowledge of the class. The method, when called, contains that knowledge and will properly type cast the data space for the assignment. On the other hand, should the attribute reference be one that requires value retrieval, then knowing the class is required in most cases. Note from Figure 6.16 that the "address" method returns a pointer to the attribute's data space; the method, in this example, is defined to return a pointer to an integer. When the class is known, the class hierarchy can then be traversed to determine the attribute typing (integer, floating point, etc.). The appropriately typed message (e.g., intmessage, floatmessage, etc.) implementing the "address" method can then be generated by the translator. The message type and attribute type must coincide. If the class is not known, the attribute is assumed to be integer. The intmessage routine is used to invoke the appropriate "address" method. If the class is not known, a retrieval is required, and the attribute is not of integer type, then a runtime error will occur ...

unless steps are taken to determine the class using the guidelines given in Chapter 5 on the VSMSL.

To summarize, the use of White's [1990] scheme for data access provides the means for name conflict resolution. However, on the negative side, the scheme requires that in order for an object's data to be properly accessed, the class of the object must be known for proper type casting and for attribute typing (during value retrievals). This impacts the ability to reference an object's user-defined attributes when the object's class is not known; the dynamic binding of object references becomes limited. The use of Brumbaugh's [1990] scheme for system-defined attributes eliminates the above problems. Yet, there would be no ability for name conflict resolution if the Brumbaugh scheme was used for both user and system-defined data. For this reason, White's approach was applied to user-defined attributes and Brumbaugh's to system-defined attributes. With this knowledge, one can better understand the capabilities and limitations of the VSMSL (Chapter 5).

CHAPTER 7
EXAMPLE MODEL APPLICATIONS

Three examples are presented in this chapter to illustrate the application of the DOMINO in a variety of forms which highlight the framework's capabilities. The three example models include the Bus Route, the Traffic Intersection, and the Branch Operations. Each model demonstrates a key aspect of the framework as well as its general features. Although the model is relatively simple with few object interactions, the Bus Route contains a decomposed dynamic object, the city bus, which carries passengers among the bus stops along the route. Several different versions of this model showcase the multiple perspectives that the model component logic can take. The Traffic Intersection extends previous work [Derrick 1988] and permits an excellent gauge and measure of effectiveness of the framework's capabilities when compared with the earlier research. In addition, this model is exceptionally complex, based on an actual intersection with many object interactions, and presents the framework's ability to handle such complexity. The DOMINO capabilities for message passing and the use of method logic specifications are additional features which are showcased by the Traffic Intersection. The Branch Operations, a model of four branch computer offices communicating over a network with two host computers, also presents another model with many component interactions. This is another real world example which demonstrates dynamic object movements down into and upward out of a deeply-nested model static structure. Similar to the Traffic Intersection, this model is compared with previous research and displays the importance of compositional equivalence for class layouts.

Each of the following sections covers one of the three example models. First, a general description of the model is given. Model classes with their attributes are described. The images associated with each model, including individual layouts and their composition, and the dynamic object images, are presented. Finally, the details of the model component logic specifications are discussed. This part of the coverage for each example offers supplemental explanation of various features of the specification language.

7.1 Bus Route Example

The Bus Route is a simple model which demonstrates several of the important features of the conceptual framework: (1) the decomposition of dynamic objects, (2) the multiple perspectives of component logic specification, and (3) the runtime change in display of model component images (both non-dynamic object and dynamic object components). The general description of this example is given:

> In some fictitious model city (See ahead to Figure 7.3), there is a bus route on which the bus makes four stops. The four bus stops form a circular bus route. One stop is in the city. Passengers may debark to or embark from the city mall, market, tailor shop or police station. The three other bus stops are in the suburbs of the city. At each of these stops, passengers debark to or embark from their homes. Two homes are located at each of the bus stops in the suburbs. The city bus departs a bus terminal at the beginning of the day and begins the route, picking up or dropping off passengers during transit of the circular route. For demonstration purposes, passengers do not debark at the city stop for shopping. They simply ride the circuit and return to their home bus stop.

This type of model is typically used to study bus loading, capacity, route efficiency, and suitability of service to customers and passengers. This model can be used for any of these purposes. With runtime animation facilities of the VSSE, the modeler can choose to view at the top level, observing the bus as it moves between the bus stops, or the modeler can choose to view the interior of the bus. Interactions between the bus driver and passengers can be observed as well as passenger seating decisions and bus loading.

7.1.1 Classes and Attributes

The model contains the following modeler-defined classes: two dynamic object classes (**bus** and **person**), five submodel classes (**terminal**, **busStop**, **house**, **shop**, and **cityBldg**), one subdynamic object class (**seat**), and one base dynamic object class (**driver**). The class hierarchies are only one level deep. No parent-class-subclass relationships are present. Object instances within the model are formed from these classes. Each class and its attributes are described. Information related to the model class (**dynamite3**) is covered first since the attributes are modeler-defined.

Class Name	Attribute	Attribute Description
Model Class		
dynamite3	**dynObjId**	Holder for dynamic object identifiers
	i,j	Counting attributes
	BUSY	Constant
	IDLE	Constant
	MOVING	Constant
	testArray	Array of dynamic object identifiers
Dynamic Object Class(es)		
bus	**location**	Bus location is either MOVING or a bus stop location
	stopStartTime	The time for bus to move out of the bus stop or enter
	exitsAtStopAB	The number of passengers set to get off at bus stop AB
	exitsAtStopCD	The number of passengers set to get off at bus stop CD
	exitsAtStopEF	The number of passengers set to get off at bus stop EF
	numberExiting	Counter, passengers exiting, initialized to exitsAtStop*i* on arrival
	transitTime	Time of transit for bus between stops
person	**busIntention**	Boolean status attribute as passenger, getting ON/OFF
	shoppingCompleted	Boolean status attribute as passenger, TRUE or FALSE
	activityTime	Time of performing an activity
Submodel Class(es)		
terminal	**serviceTime**	Time for servicing buses at the terminal
busStop	**numberWaiting**	Number of persons waiting at the bus stop
house	**houseKeepingTime**	Time for housekeeping chores
shop		No Attributes
cityBldg		No Attributes
Subdynamic Object Class(es)		
seat		No Attributes
Base Dynamic Object Class(es)		
driver	**status**	Boolean status, BUSY or IDLE
	serviceTime	Time for servicing each passenger

7.1.2 Model Structures and Graphical Elements

The model structures are simple (only one level deep). Figure 7.1 gives the model static structure (line drawing created by VSSE) which contains the following model instances as literal names associated with their owning class. These instances are permanent during execution and are not created by the specification language create statement:

Owning Class	Instance(s)
bus	**cityBus**

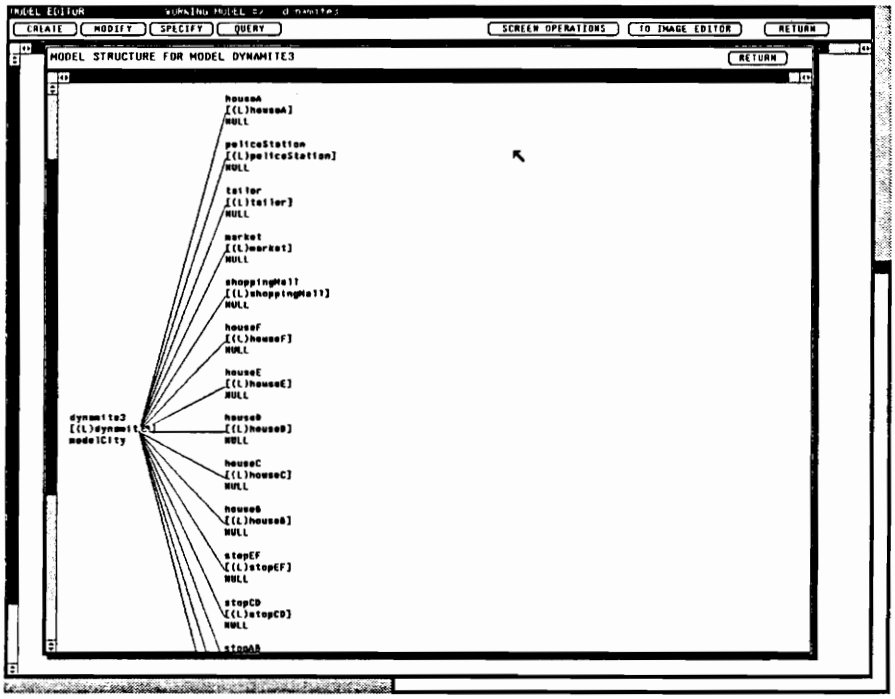

Figure 7.1 Model Static Structure (Bus Route)

person	No permanent instances
terminal	busTerminal
busStop	stopCity, stopAB, stopCD, stopEF
house	houseA...houseE
shop	shoppingMall, market, tailor
cityBldg	policeStation

The VSSE line drawing contains three separate data items at each node in the hierarchical structure. From top to bottom these are (1) the component variable name, (2) the component literal name (in brackets), and (3) the name of its decomposition layout.

The only model dynamic structure of the bus route model is displayed in the VSSE line drawing of Figure 7.2. This structure is associated with the decomposed dynamic object, the cityBus. The literal instances by class for this structure are given:

Owning Class	Literal Instance(s)
seat	seat1...seat14
driver	busDriver

Therefore, the Bus Route contains only two layouts, one for the top level of the model static structure (Figure 7.3) and one for the top level of city bus dynamic structure (Figure 7.4). Figures 7.5 and 7.6 are the completed layout definitions for these two layouts. Figure 7.7 shows the images which the cityBus permanent decomposed dynamic object and the created (by specification language) person instances take during the animation. Notice that there are three bus images which the cityBus dynamic object can assume.

7.1.3 Component Logic Specifications

Of the several features of this model, perhaps the most significant is the demonstration of the three perspectives of the model component logic specification. Each specification has been tested and produces an executable, working model. The three specification perspectives are now explained.

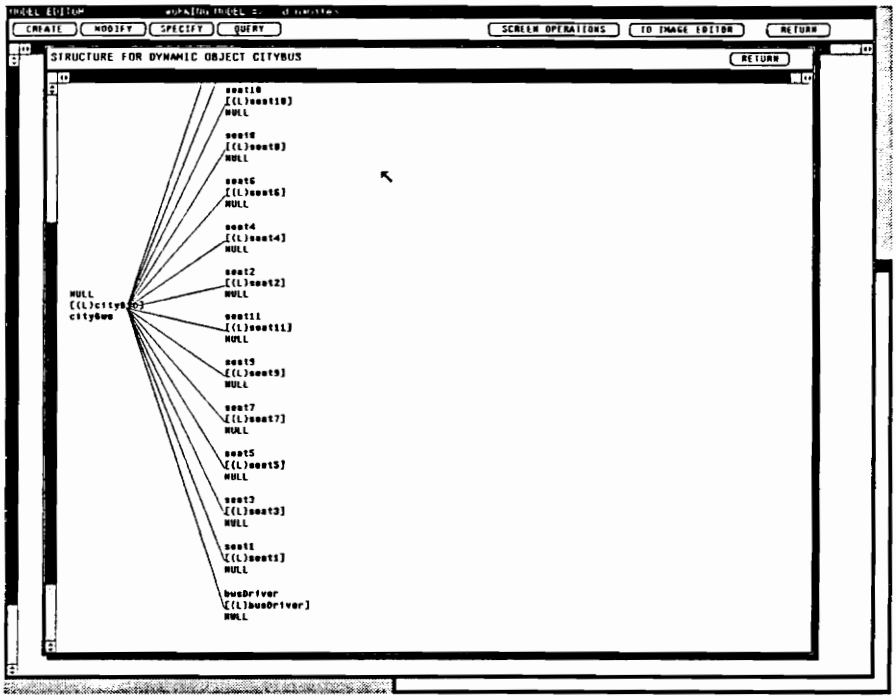

Figure 7.2 Model Dynamic Structure of City Bus (Bus Route)

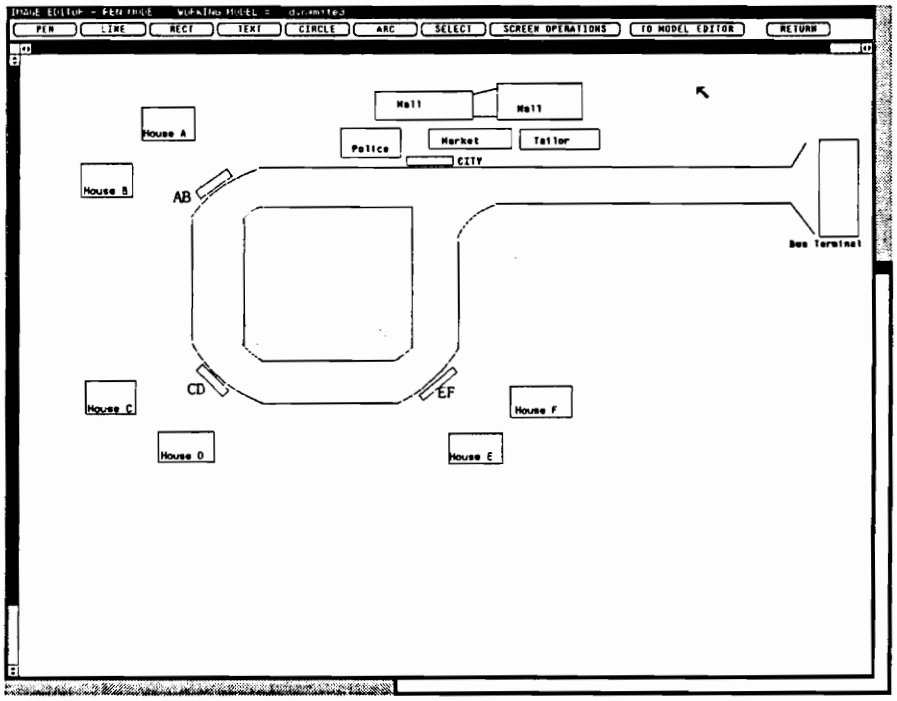

Figure 7.3 Top Level Layout Image (Bus Route)

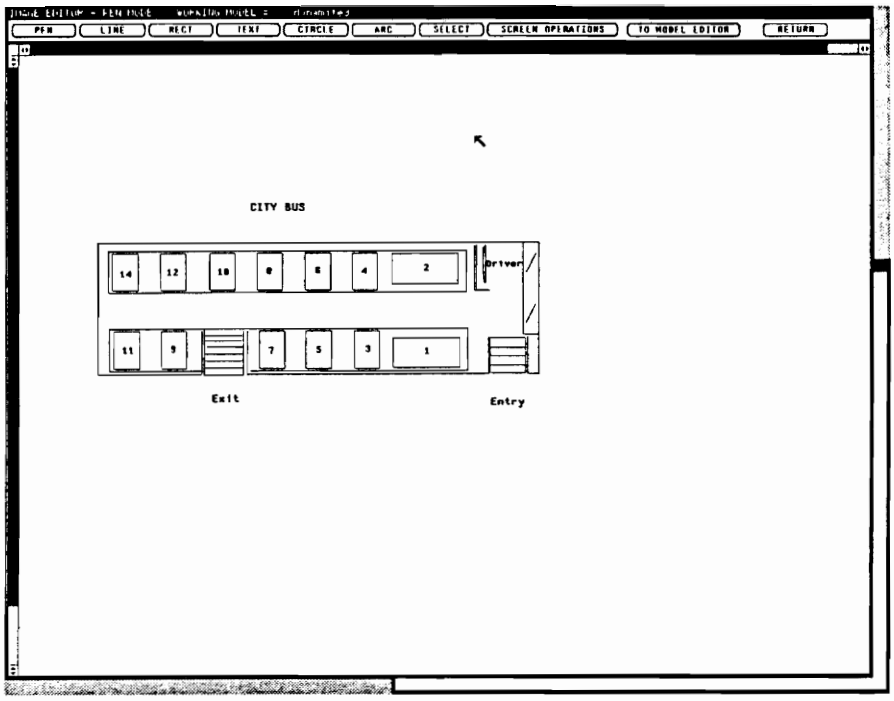

Figure 7.4 Layout Image of City Bus (Bus Route)

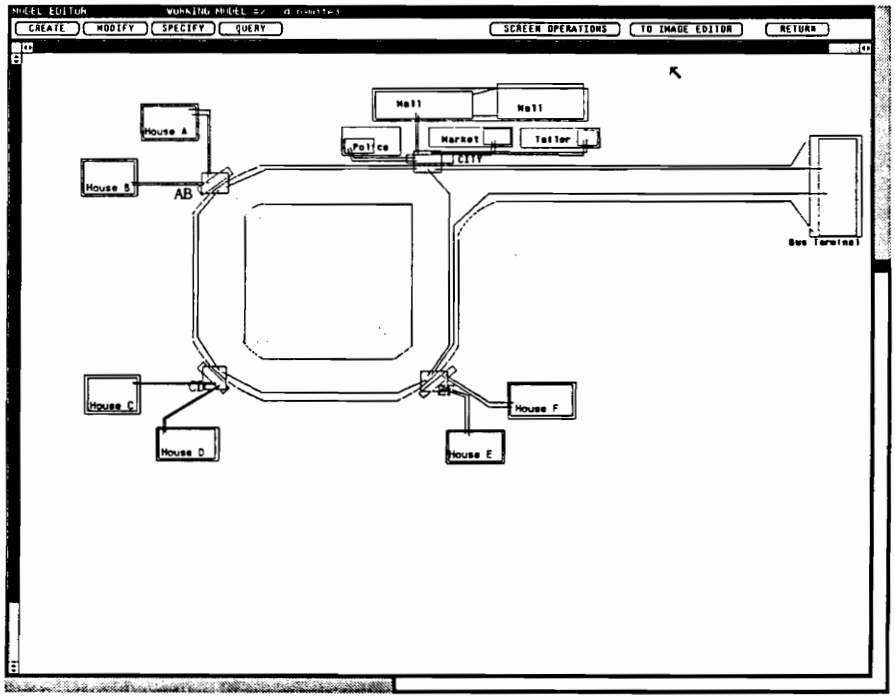

Figure 7.5 Top Level Layout Definition (Bus Route)

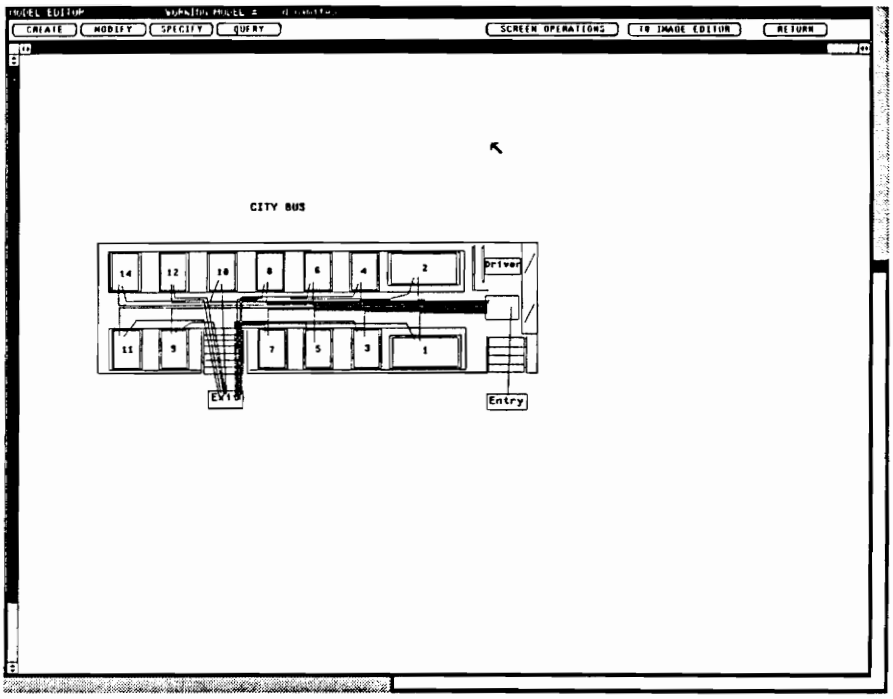

Figure 7.6 Layout Definition of City Bus (Bus Route)

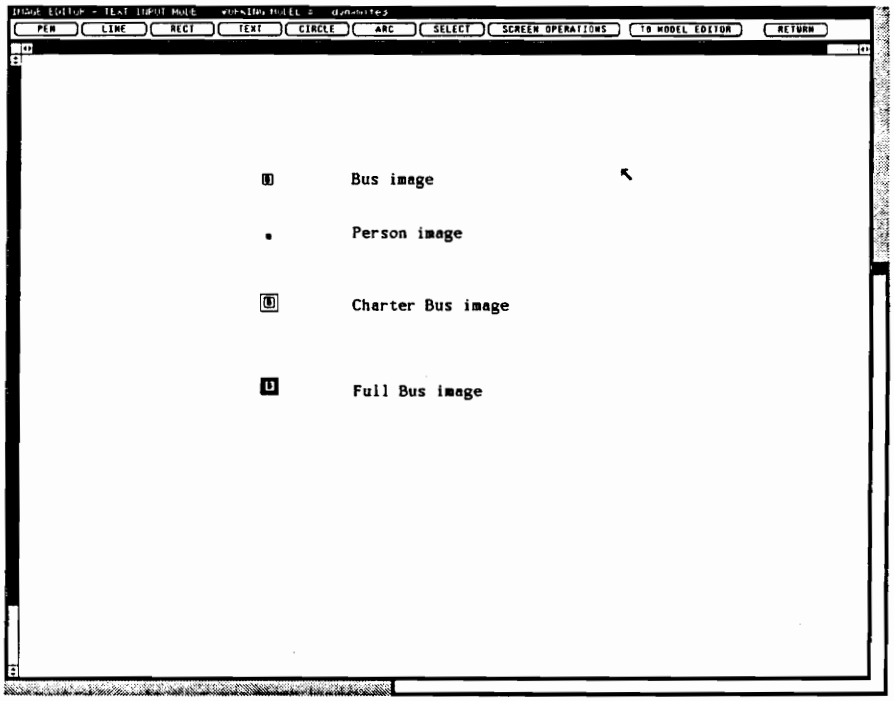

Figure 7.7 Dynamic Object Images (Bus Route)

7.1.3.1 Machine-Oriented Perspective

In the machine-oriented form, supervisory logic (and only supervisory logic) is used to specify model dynamics. Modeler-defined supervisory logic is attached to the **builtin** submodel class, and also to every class listed in Section 7.1.1 except the classes **person**, **shop**, and **cityBldg**. Each logic is described completely and separately in order to give a broad perspective on how model dynamics are accomplished via the supervisory logic specifications.

In the builtin class supervisory logic (Figure 7.8), the decomposed dynamic object **cityBus** is placed to start in the **busTerminal**. Real dynamic objects are created as instances of class **person** to start in the houses. At the model's initiation (clock time of zero), these dynamic objects are positioned in their respective submodel components. Each supervisory logic (the **busTerminal** and the respective houses) is executed to manage the actions of the dynamic objects which have been placed in them. Notice that the virtual system dynamic object that accomplishes the actions of the **builtin** logic is destroyed once its actions are completed.

A message is sent to the statistics module in order to produce an exponential random variate in the terminal class supervisory logic (Figure 7.9). The value of this random variate is assigned to the **serviceTime** attribute of the bus. The bus is delayed by the service for the **serviceTime** attribute of the bus terminal. Once service is completed, the bus is moved to the first bus stop (the one located in the city), and the bus commences the bus route. The **move** statement for this action is a local move that uses a variable component name for its move location.

Each person does some attribute initialization in the house class supervisory logic (Figure 7.10): the house keeping time is set and the person's attributes for bus intention and shopping completed are set to ON and FALSE respectively. The person is then occupied with housekeeping for a time. Then, based on which house the person is in, the person dynamic object is then moved into the nearest local bus stop.

The supervisory logic of the busStop class (Figures 7.11, 7.12, and 7.13) offers the pivotal logic that manages both the movements of the bus and the individual persons. Therefore, the bus stop logic is separated into two sets of actions: those to be accomplished by the bus (Figure 7.11) and by a person

```
-- THE BUILTIN SUBMODEL CLASS SUPERVISORY LOGIC
begin

   -- Move the bus to start in the terminal submodel
   move do "cityBus" into sm "busTerminal";

   -- Create a person for each house
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseA";
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseB";
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseC";
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseD";
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseE";
   create rdo belonging to class person putting its id in dynObjId
      starting in sm "houseF";

   -- destroy the virtual system dynamic object
   destroy

end
```

Figure 7.8 BUILTIN Class Supervisory Logic (Bus Route, Machine-Oriented)

```
-- THE TERMINAL SUBMODEL CLASS SUPERVISORY LOGIC
begin

   -- Set up expon variate draw
   put 60.0 into attr mean of vsm "statmodule";
   send message expon to vsm "statmodule" using
      (attr mean of vsm "statmodule",
      attr iseed of vsm "statmodule")
      putting result in attr serviceTime of this sm;

   -- Delay the bus departure for daily routes
   engageIn servicing for attr serviceTime of this sm minutes;

   -- send the Bus on its route to its first bus stop
   move into local sm stopCity

end
```

Figure 7.9 TERMINAL Class Supervisory Logic (Bus Route, Machine-Oriented)

```
-- THE HOUSE SUBMODEL CLASS SUPERVISORY LOGIC
begin

   -- Set up attribute values
   put 180.0 into attr housekeepingTime of this sm;
   put ON@ into attr busIntention of this do;
   put FALSE@ into attr shoppingCompleted of this do;

   -- Do housekeeping chores
   engageIn housekeeping for attr housekeepingTime of this sm minutes;

   -- Leave house for bus stop
   branch sys attr compInst of this sm
    (HOUSEA_SM_INST@)
      move into local sm stopAB;
    (HOUSEB_SM_INST@)
      move into local sm stopAB;
    (HOUSEC_SM_INST@)
      move into local sm stopCD;
    (HOUSED_SM_INST@)
      move into local sm stopCD;
    (HOUSEE_SM_INST@)
      move into local sm stopEF;
    (HOUSEF_SM_INST@)
      move into local sm stopEF
   end -- branch

end
```

Figure 7.10 HOUSE Class Supervisory Logic (Bus Route, Machine-Oriented)

```
-- BUSSTOP SUBMODEL CLASS SUPERVISORY LOGIC
begin

-- Do Bus Dynamic Object Actions if this do is a bus
branch sys attr classId of this do
(bus)
   begin

--     Stop the bus and set bus location
       engageIn stopping for attr stopStartTime of this do mins;
       put sys attr compInst of this sm into attr location of this do;

--     Set the number exiting, if there are any.
       branch sys attr compInst of this sm
       (STOPAB_SM_INST@)
         begin
         put attr exitsAtStopAB of this do into attr numberExiting of this do;
         put 0 into attr exitsAtStopAB of this do
         end;
         ...
       (STOPCITY_SM_INST@)
       begin
         ...
       -- Set the shopping completed boolean for all passengers on board.
       repeat with i = 1 to sys attr numDos of this do
          begin
             put sys attr dosList[i] of do "cityBus" into dynObjId;
             put TRUE@ into attr shoppingCompleted of do dynObjId
          end
       end -- repeat
       end -- this case
       end; -- compInst branch

--     Bus wait until all passengers have been transferred.
       if attr numberWaiting of this sm is > 0 or
          attr numberExiting of this do is > 0 then
          engageIn passengerTransfer until attr numberWaiting of this sm is 0
            and attr numberExiting of this do is 0;

--     Start the bus back up and set in motion
       engageIn startingUp for attr stopStartTime of this do mins;
       put MOVING@ into attr location of this do;

---    Move to next appropriate bus stop
       if sys attr currentComp of this do is
          sys attr compInst of sm "stopCity" then
          move into sm stopAB
       else
          if sys attr currentComp of this do is
          sys attr compInst of sm "stopAB" then
          move into sm stopCD
       else ...
   end; -- end class bus
```

Figure 7.11 BUSSTOP Class Supervisory Logic - Bus Part (Bus Route, Machine-Oriented)

```
-- Do Person Dynamic Object Actions
(person)
  begin
   -- If person is getting ON the bus
   branch attr busIntention of this do
   (ON@)
   begin

-- If the bus is gone or people are already waiting, put self in line.
   if attr location of do "cityBus" != sys attr compInst of this sm
      or attr numberWaiting of this sm is > 0
      or (attr status of bdo "busDriver" is BUSY@ and attr location of do
          "cityBus" is sys attr compInst of this sm) then
      begin
         add 1 to attr numberWaiting of this sm;
         put attr numberWaiting of this sm into holdingVar;
         put sys attr ident of this do into
            sys attr entryList[holdingVar] of this sm;

         -- Wait until the bus is HERE@ and first in line.
         engageIn waiting until attr location of do "cityBus" is
            sys attr compInst of this sm and
            sys attr entryList[1] of this sm is sys attr ident of
            this do and attr status of bdo "busDriver" is IDLE@;
         put attr numberWaiting of this sm into holdingVar;
         subtract 1 from holdingVar;

         -- Reset everyone's waiting position and get onto bus
         repeat with i = 1 to holdingVar
            begin
              put i+1 into dynObjId;
              put sys attr entryList[dynObjId] of this sm into
                 sys attr entryList[i] of this sm
            end
         end;
         subtract 1 from attr numberWaiting of this sm;
         put OFF@ into attr busIntention of this do;
         add 1 to sys attr numDos of do "cityBus";
         put sys attr numDos of do "cityBus" into holdingVar;
         put sys attr ident of this do into sys attr dosList[holdingVar] of
           do "cityBus";
         move into do "cityBus"
      end

-- Otherwise, the bus is HERE and no one is waiting and driver isnt BUSY
   else
      begin
      put OFF@ into attr busIntention of this do;
      add 1 to sys attr numDos of do "cityBus";
      put sys attr numDos of do "cityBus" into holdingVar;
      put sys attr ident of this do into sys attr dosList[holdingVar] of
        do "cityBus";
      move into do "cityBus"
      end
   end; -- intention ON
```

Figure 7.12 BUSSTOP Class Supervisory Logic - "On" Part (Bus Route, Machine-Oriented)

```
  (OFF@)
   begin
-- Remove the passenger from rider list, First locate him in the list
      repeat with i = 1 to sys attr numDos of do "cityBus"
         begin
            if sys attr ident of this do is sys attr dosList[i] of
              do "cityBus" then
              put i into j
         end
      end;

-- Update the number on the bus
      subtract 1 from sys attr numDos of do "cityBus";
      subtract 1 from attr numberExiting of do "cityBus";

-- Now move everyone up on the list.
      repeat with i = j to sys attr numDos of do "cityBus"
         begin
            put i+1 into dynObjId;
            put sys attr dosList[dynObjId] of do "cityBus" into
                sys attr dosList[i] of do "cityBus"
         end
      end;

-- Move to appropriate house after leaving the bus stop
      branch sys attr originComp of this do
      (HOUSEA_SM_INST@)
         move into sm houseA;
      ...
      (HOUSEF_SM_INST@)
         move into sm houseF
      end -- branch origin
   end -- intention OFF
  end  -- branch intention
  end -- end class person
end  -- branch classId
end  -- begin logic block
```

Figure 7.13 BUSSTOP Class Supervisory Logic - "Off" Part (Bus Route, Machine-Oriented)

(Figures 7.12 and 7.13). At each bus stop, the bus is engaged in slowing down and setting its appropriate bus stop location. The bus initializes the attribute which records the count of passengers that will exit at that stop. At the bus stop in the city, the shopping completed boolean is set to TRUE for all passengers on board. Once there are no passengers waiting to get on the bus or off the bus, the bus starts up and its location is set to MOVING. The bus is then directed to the next stop on the bus route. For persons at the bus stop, their actions depend on their intentions, whether getting on (Figure 7.12) or getting off (Figure 7.13) the bus. For those waiting to get on the bus, a queue is formed. Note that the bus stop supervisory logic handles the queueing of persons. To get on the bus, the following conditions must be satisfied: the bus must be present at the stop, the driver not be busy, and the person must be first in the queue. Once these conditions are met, the person is moved into the bus. The identifier of the person dynamic object is then placed in a list of passengers (single dimension array and system attribute dosList of the bus) that are on the bus. In addition, a counter of the number of passengers on the bus (system attribute numDos of the bus) is updated. If getting off the bus at the stop, the person is removed from the list of bus passengers and the number of passengers on the bus is decremented. The person is directed to his or her house.

When a person is moved into the bus, the bus class supervisory logic (Figure 7.14) of this decomposed dynamic object directs the person to the bus driver. This move of the person dynamic object is designated as an inner move since the bus is decomposed. Once the person is at the bus driver, the driver class supervisory logic (Figure 7.15) takes over supervision of the person/passenger. The bus driver becomes BUSY and services the passenger (checking bus pass, taking transfer slip, etc.). After the driver returns to IDLE status, the person is moved to his or her seat. Once in the seat, the seat class supervisory logic (Figure 7.16) updates the number of passengers which will exit at the new passenger's destination. The person waits until the bus is at the appropriate destination bus stop. When this condition is satisfied, the person is directed off the bus.

In summary, the bus (after leaving the bus terminal) is guided throughout its route by the bus stop supervisory logic. On the other hand, the persons are directed by multiple supervisors. The house logic

```
-- BUS DYNAMIC OBJECT CLASS SUPERVISORY LOGIC
move to inner bdo busDriver
```

Figure 7.14 BUS Class Supervisory Logic (Bus Route, Machine-Oriented)

```
-- DRIVER SUBDYNAMIC OBJECT CLASS SUPERVISORY LOGIC
begin
  -- Set Driver busy
  put BUSY@ into attr status of this bdo;

  -- Determine service time and service passenger
  put 0.10 into attr mean of vsm "statmodule";
  send message expon to vsm "statmodule" using
   (attr mean of vsm "statmodule",
    attr iseed of vsm "statmodule")
    putting result in attr serviceTime of this bdo;

  engageIn servicingPassengers for attr serviceTime of this bdo minutes;

  -- Reset status of driver to IDLE and move to seat
  put IDLE@ into attr status of this bdo;
  branch sys attr originComp of this do
   (HOUSEA_SM_INST@)
     move into sdo seat14;
   ...
   (HOUSEF_SM_INST@)
     move into sdo seat9
  end
end
```

Figure 7.15 DRIVER Class Supervisory Logic (Bus Route, Machine-Oriented)

```
-- SEAT SUBDYNAMIC OBJECT CLASS SUPERVISORY LOGIC
branch sys attr originComp of this do
(HOUSEA_SM_INST@)
   begin
     add 1 to attr exitsAtStopAB of do "cityBus";
     engageIn waiting until attr location of do "cityBus" is STOPAB_SM_INST@
        and attr shoppingCompleted of this do is TRUE@;
     move into outer sm stopAB
   end;
...
(HOUSEF_SM_INST@)
   begin
     add 1 to attr exitsAtStopEF of do "cityBus";
     engageIn waiting until attr location of do "cityBus" is STOPEF_SM_INST@
        and attr shoppingCompleted of this do is TRUE@;
     move into outer sm stopEF
   end
end -- branch on origin
```

Figure 7.16 SEAT Class Supervisory Logic (Bus Route, Machine-Oriented)

gets the person to a bus stop. The bus stop logic gets the person on the bus. Once on the bus, the seat logic activates the conditions which will debark the person from the bus in order to accomplish shopping or to return to the person's local bus stop. Once back at the bus stop, the bus stop directs the person to his or her home.

7.1.3.2 Material-Oriented Perspective

In the material-oriented perspective, all directing logic is self logic. For this model, that means that each person will direct their own movements. Also, the bus will take itself from bus stop to bus stop along its route. Each self logic reads as a single process. In this way, for example, the person takes on the responsibilities of queueing when necessary and all other responsibilities that the individual supervisory logics had assumed in the previous perspective.

The bus class self logic (Figure 7.17) accomplishes the actions of the terminal. The bus is serviced and moves to the first stop. The direction of the bus for its route is then accommodated by a **repeat forever** looping statement in which the bus is repeatedly engaged in slowing down for a particular stop, setting appropriate attributes, waiting for passengers to get on or off, and then starting back up and going on to the next stop.

The person class self logic (Figures 7.18 throught 7.21) similarly performs the management of person objects by the use of a **repeat forever** loop. House actions are performed (Figure 7.18) and are followed by bus stop actions (Figure 7.19) for getting on the bus. Bus supervisory logic actions (Figure 7.20) are also done to move the person (once on the bus) to the driver. After service by the driver, the person self logic performs the logic that had been done by the seat class supervisory logic. Finally, once off the bus, the self logic does bus stop actions for exiting bus passengers (Figure 7.21) and sends the person to home. This process is repeated.

In the self logic of both the bus and persons, notice that all movements are directed using literal movement destinations. Attribute referencing is also done using literal component names. In the person logic, also notice how the meaning of the current component changes depending on the location of the

```
-- BUS DYNAMIC OBJECT CLASS SELF LOGIC
begin
-- Do the actions performed in the terminal submodel
-- The bus is serviced and moves to first stop.
   put 60.0 into attr mean of vsm "statmodule";
   send message expon to vsm "statmodule" using
      (attr mean of vsm "statmodule",
       attr iseed of vsm "statmodule")
      putting result in attr serviceTime of sm "busTerminal";
   engageIn service for attr serviceTime of sm "busTerminal" minutes;
   move into local sm "stopCity";
   put 20.0 into attr transitTime of this do;

-- Now the bus does the logic of the busStops in a loop
   repeat forever
     begin
      -- Stop the bus
      engageIn stopping for attr stopStartTime of this do mins;
      -- Set the bus location
      put sys attr compInst of current sm into attr location of this do;
      set image of current sm to "busIn";
      -- Set the numberexiting from the bus, if any
      branch sys attr compInst of current sm
      (STOPAB_SM_INST@)
      ...
      (STOPCITY_SM_INST@)
         begin
            put attr exitsAtStopCity of this do
              into attr numberExiting of this do;
            put 0 into attr exitsAtStopCity of this do;
            -- Set all passengers shoppingCompleted
            repeat with i = 1 to sys attr numDos of this do
               begin
               put testArray[i] into dynObjId;
               put TRUE@ into attr shoppingCompleted of do dynObjId
               end
            end -- repeat
          end -- case stopCity
       end; -- branch on compInst

       -- Wait until all passengers have been transferred
       if attr numberWaiting of current sm is > 0 or
          attr numberExiting of this do is > 0 then
          engageIn passengerTransfer until attr numberWaiting of
             current sm is 0 and attr numberExiting of this do is 0;

       -- Start the bus back up, set MOVING, and transit to next stop
        engageIn startingUp for attr stopStartTime of this do mins;
        put MOVING@ into attr location of this do;
        engageIn transiting for attr transitTime of this do mins;

        branch sys attr compInst of current sm
        (STOPAB_SM_INST@)
           begin
           set image of this do to "charter";
           set image of current sm to "busAB";
           move into sm "stopCD"
           end;
        ...
        (STOPCITY_SM_INST@)
           ...
           end
         end -- branch on compInst
      end -- repeat forever stmt
   end -- repeat
end -- process logic for bus
```

Figure 7.17 BUS Class Self Logic (Bus Route, Material-Oriented)

```
-- PERSON DYNAMIC OBJECT CLASS SELF LOGIC
repeat forever
   begin

   -- Do the house logic
   -- This is first component DO is in, so compiler must be told the class
   -- of the current sm.

   branch sys attr classIdent of current sm
   (house)
      begin
         put 180.0 into attr housekeepingTime of current sm;
         put ON@ into attr busIntention of this do;

         engageIn housekeeping for attr housekeepingTime of current sm minutes;

         put FALSE@ into attr shoppingCompleted of this do;

         branch sys attr compInst of current sm
         (HOUSEA_SM_INST@)
            move into sm "stopAB";
         ...
         (HOUSEF_SM_INST@)
            move into sm "stopEF"
         end -- branch on compInst
      end -- house case
   end; -- branch on comp classIdent
```

Figure 7.18 PERSON Class Self Logic - House Actions (Bus Route, Material-Oriented)

```
   -- Now do the Bus stop logic for getting on the bus.

-- If the bus is gone or people are already waiting, put self in line.
   if attr location of do "cityBus" != sys attr compInst of current sm
      or attr numberWaiting of current sm is > 0
      or (attr status of bdo "busDriver" is BUSY@ and attr location of do
          "cityBus" is sys attr compInst of current sm) then
      begin
         add 1 to attr numberWaiting of current sm;
         put attr numberWaiting of current sm into holdingVar;
         put sys attr ident of this do into
            sys attr entryList[holdingVar] of current sm;

         -- Wait until the bus is HERE@ and first in line.
         engageIn waiting until attr location of do "cityBus" is
            sys attr compInst of current sm and
            sys attr entryList[1] of current sm is sys attr ident of
            this do and attr status of bdo "busDriver" is IDLE@;
         put attr numberWaiting of current sm into holdingVar;
         subtract 1 from holdingVar;

         -- Reset everyone's waiting position and get onto bus
         repeat with i = 1 to holdingVar
            begin
               put i+1 into dynObjId;
              put sys attr entryList[dynObjId] of current sm into
                 sys attr entryList[i] of current sm
            end
         end;
         subtract 1 from attr numberWaiting of current sm;
         put OFF@ into attr busIntention of this do;
         add 1 to sys attr numDos of do "cityBus";
         put sys attr numDos of do "cityBus" into holdingVar;
         put sys attr ident of this do into sys attr dosList[holdingVar] of
           do "cityBus";
         move into do "cityBus"
      end

-- Otherwise, the bus is HERE and no one is waiting and driver isnt BUSY
   else
      begin
      put OFF@ into attr busIntention of this do;
      add 1 to sys attr numDos of do "cityBus";
      put sys attr numDos of do "cityBus" into holdingVar;
      put sys attr ident of this do into sys attr dosList[holdingVar] of
        do "cityBus";
      move into do "cityBus"
      end;
```

Figure 7.19 PERSON Class Self Logic - BusStop Actions (Bus Route, Material-Oriented)

```
-- Execute the bus logic
      move to inner bdo "busDriver";

-- Execute the logic of the driver
      put BUSY@ into attr status of current bdo;
      put 0.10 into attr mean of vsm "statmodule";
      send message expon to vsm "statmodule" using
        (attr mean of vsm "statmodule",
         attr iseed of vsm "statmodule")
         putting result in attr serviceTime of current bdo;

      engageIn servicingPassengers for attr serviceTime of current bdo minutes;
      put IDLE@ into attr status of current bdo;
      branch sys attr originComp of this do
       (HOUSEA_SM_INST@)
         move into sdo "seat14";
       ...
       (HOUSEF_SM_INST@)
         move into sdo "seat9"
      end;

-- Now do the seat logic

      branch sys attr originComp of this do
      (HOUSEA_SM_INST@)
        begin
         add 1 to attr exitsAtStopAB of do "cityBus";
         engageIn waiting until attr location of do "cityBus" is STOPAB_SM_INST@
            and attr shoppingCompleted of this do is TRUE@;
         move into outer sm "stopAB"
        end;
      ...
      (HOUSEF_SM_INST@)
        begin
         add 1 to attr exitsAtStopEF of do "cityBus";
         engageIn waiting until attr location of do "cityBus" is STOPEF_SM_INST@
            and attr shoppingCompleted of this do is TRUE@;
         move into outer sm "stopEF"
        end
    end; -- branch on origin
```

Figure 7.20 PERSON Class Self Logic - Bus/Other Actions (Bus Route, Material-Oriented)

```
-- Now do the logic for getting off the bus
-- Remove the passenger from the list of riders
-- First locate him in the list
      repeat with i = 1 to sys attr numDos of do "cityBus"
         begin
            if sys attr ident of this do is sys attr dosList[i] of
              do "cityBus" then
              put i into j
         end
      end;

-- Update the number on the bus
      subtract 1 from sys attr numDos of do "cityBus";
      subtract 1 from attr numberExiting of do "cityBus";

-- Now move everyone up on the list.
      repeat with i = j to sys attr numDos of do "cityBus"
         begin
            put i+1 into dynObjId;
            put sys attr dosList[dynObjId] of do "cityBus" into
                sys attr dosList[i] of do "cityBus"
         end
      end;


      branch sys attr originComp of this do
      (HOUSEA_SM_INST@)
         move into sm "houseA";
      ...
      (HOUSEF_SM_INST@)
         move into sm "houseF"
      end -- branch origin
   end -- repeat action
end -- repeat
```

Figure 7.21 PERSON Class Self Logic - BusStop Exiting (Bus Route, Material-Oriented)

person. The assumption of logic responsibilities from the various supervisory logics has lengthened the code.

7.1.3.3 Hybrid Perspective

Several combinations and forms of the above perspectives on logic can be utilized to specify model component logic. One possible form is to use self logic for the bus only, to direct the bus movements. The remainder of model component logic can be handled using supervisory logic to manipulate the persons as in Section 7.1.3.1. "Turning on" or "turning" off self or supervisory logic can easily be done using the "reset" logic button in the class specification window of the Model Generator. For this case, that's all that needs to be done. The model executes smoothly. Using the bus self logic in this way dictates that the terminal supervisory logic be turned off. The bus portions of the bus stop supervisory logic are never executed. Another hybrid form is possible but requires the use of specialized move statements in the specification language. This form, now explained, removes the previously mentioned problems.

A serious drawback to the all-self logic perspective is the very long segments of self logic code. This was particularly true for the person class self logic. During development, the need to shorten the self logic but retain the readability of the code (as a process) was recognized. Clearly, the material-oriented perspective assumes the supervising actions. The *executing move* (move!) statement was derived to provide ways for allowing the use of supervisory logic modules from within self logic. With this statement, the self logic can temporarily pass supervision and control of the dynamic object to a supervisor which uses supervisory logic, but under certain restrictions. The called supervisor must return control to the calling self logic and not to any other. Therefore, no moves can be initiated by these temporary supervisors. This would not properly return logic control.

The bus class self logic is retained in its original form since its length is manageable. The person class self logic is modified (Figures 7.22 and 7.23) in this final form of hybrid logic which we describe. Once the person is on the bus, the executing move statements are utilized. Notice the move! to the bus driver and then to the seat. At each of these executing moves, new versions of the supervisory logic of

```
-- PERSON DYNAMIC OBJECT CLASS SELF LOGIC
repeat forever
   begin

   -- Do the house logic. This is first component DO is in,
   -- so compiler must be told the class of the current sm.

   branch sys attr classIdent of current sm
   (house)
      begin
         put 180.0 into attr mean of vsm "statmodule";
         send message expon to vsm "statmodule" using
          (attr mean of vsm "statmodule",
           attr iseed of vsm "statmodule") putting result
           in attr housekeepingTime of current sm;

         put ON@ into attr busIntention of this do;
         put FALSE@ into attr shoppingCompleted of this do;
         engageIn housekeeping for attr housekeepingTime of current sm minutes;

         branch sys attr compInst of current sm
         (HOUSEA_SM_INST@)
            begin
            display "A" in position 1;
            move into sm "stopAB"
            end;
         ...
         (HOUSEF_SM_INST@)
            ...
         end -- branch on compInst
      end -- house case
   end; -- branch on comp classIdent

   -- Now do the Bus stop logic for getting on the bus.
-- If the bus is gone or people are already waiting, put self in line.
   if attr location of do "cityBus" != sys attr compInst of current sm
      or attr numberWaiting of current sm is > 0
      or (attr status of bdo "busDriver" is BUSY@ and attr location of do
          "cityBus" is sys attr compInst of current sm) then
      begin
         ... -- Same as in Figure 7.19]
         subtract 1 from attr numberWaiting of current sm;
         put OFF@ into attr busIntention of this do;
         add 1 to sys attr numDos of do "cityBus";
         put sys attr numDos of do "cityBus" into holdingVar;
         put sys attr ident of this do into testArray[holdingVar];
         put BUSY@ into attr status of bdo "busDriver";
         move into do "cityBus"
      end

-- Otherwise, the bus is HERE and no one is waiting and driver isnt BUSY
   else
      begin
      put OFF@ into attr busIntention of this do;
      ... [Same as just above]
      move into do "cityBus"
      end;

    put 2.0 into attr activityTime of this do;
```

Figure 7.22 PERSON Class Self Logic - Part One (Bus Route, Hybrid)

```
-- Walk up the bus steps to driver
    engageIn walkingUpSteps for attr activityTime of this do mins;

-- Execute the bus logic
      move! to inner bdo "busDriver";

-- Execute the logic of the driver in his supervisory logic.
-- Here we basically are obtaining service from the driver.
-- Now done with driver, walk to designated seat.

    engageIn walkingToSeat for attr activityTime of this do mins;

      branch sys attr originComp of this do
       (HOUSEA_SM_INST@)
         move! into sdo "seat14";
       ...
      end;

-- Now do the seat logic , but using the seat's supervisory logic.
-- Finished sitting down, let's get off to the busstop.

      branch sys attr originComp of this do
      (HOUSEA_SM_INST@)
         move into outer sm "stopAB";
      ...
      end; -- branch on origin

-- Now do some of the finishing logic for getting off the bus
-- Remove the passenger from the list of riders
-- First locate him in the list
      repeat with i = 1 to sys attr numDos of do "cityBus"
         begin
            if sys attr ident of this do is testArray[i] then
              put i into j
         end
      end;

-- Update the number on the bus
      subtract 1 from sys attr numDos of do "cityBus";
      subtract 1 from attr numberExiting of do "cityBus";

-- Now move everyone up on the list.
      repeat with i = j to sys attr numDos of do "cityBus"
         begin
            put i+1 into dynObjId;
            put testArray[dynObjId] into
                testArray[i]
         end
      end;

      branch sys attr originComp of this do
      (HOUSEA_SM_INST@)
         move into sm "houseA";
      ...
      end -- branch origin
   end -- repeat action
end -- repeat
```

Figure 7.23 PERSON Class Self Logic - Part Two (Bus Route, Hybrid)

the driver and seat (Figures 7.24 and 7.25) are executed. In keeping with the above mentioned restrictions, the move statements from within the supervisory logics of the driver and seat are removed. These moves are now accomplished within the person class self logic.

7.1.4 Changing Display of Images and Context

Another of the features which this model demonstrates is the runtime change in display of model component images. This includes the change in dynamic object images and also the change in non-dynamic object images (such as submodel or static object images). Each type of change is described. The specification language contains special display statements which accomplish these changes. The next two sections liberally refer back to earlier specification logic figures for providing sufficient explanation and description of this feature. In addition, figures are provided which demonstrate the ability to shift animation contexts and display the decomposed dynamic object context.

7.1.4.1 Dynamic Object Images

Dynamic object images are manipulated in two ways. Attribute and text information can be written onto the image of the moving dynamic object. Also, the image itself can be completely modified in form.

First, attribute information can be written to the moving image for runtime display of attribute values and text. The **display** statement is used and has two modes: one for displaying text only and one for displaying attribute names and values. There are four display positions available on each dynamic object image. In each of the display modes, the position must be given. A display statement of the first mode for text display is shown in Figure 7.22. Each person has a letter displayed on its image which indicates from which house the person has originated. The display is set just before the person object moves to the bus stop. The result of this type of display change is shown in Figure 7.26. A person has departed **houseC** and the "C" is annotated alongside the person image. The display of attribute names and values on the dynamic object image is not demonstrated in this model example; see the Branch Operations example, section 7.3. Figure 7.17 includes specification language display statements (at the end of the bus

```
-- DRIVER BASE DYNAMIC OBJECT SUPERVISORY LOGIC
begin

  -- Set driver busy
  put BUSY@ into attr status of this bdo;

  -- Determine service time at driver
  put 5.0 into attr mean of vsm "statmodule";
  send message expon to vsm "statmodule" using
   (attr mean of vsm "statmodule",
    attr iseed of vsm "statmodule")
    putting result in attr serviceTime of this bdo;

  -- Get serviced and return driver to Idle
  engageIn servicingPassengers for attr serviceTime of this bdo minutes;
  put IDLE@ into attr status of this bdo

end
```

Figure 7.24 DRIVER Class Supervisory Logic (Bus Route, Hybrid)

```
-- SEAT SUBDYNAMIC OBJECT SUPERVISORY LOGIC
branch sys attr originComp of this do
(HOUSEA_SM_INST@)
   begin
     add 1 to attr exitsAtStopAB of do "cityBus";
     engageIn waiting until attr location of do "cityBus" is STOPAB_SM_INST@
        and attr shoppingCompleted of this do is TRUE@
   end;
(HOUSEB_SM_INST@)
   begin
     add 1 to attr exitsAtStopAB of do "cityBus";
     engageIn waiting until attr location of do "cityBus" is STOPAB_SM_INST@
        and attr shoppingCompleted of this do is TRUE@
   end;
...
(HOUSEF_SM_INST@)
   begin
     add 1 to attr exitsAtStopEF of do "cityBus";
     engageIn waiting until attr location of do "cityBus" is STOPEF_SM_INST@
        and attr shoppingCompleted of this do is TRUE@
   end
end -- branch on origin
```

Figure 7.25 SEAT Class Supervisory Logic (Bus Route, Hybrid)

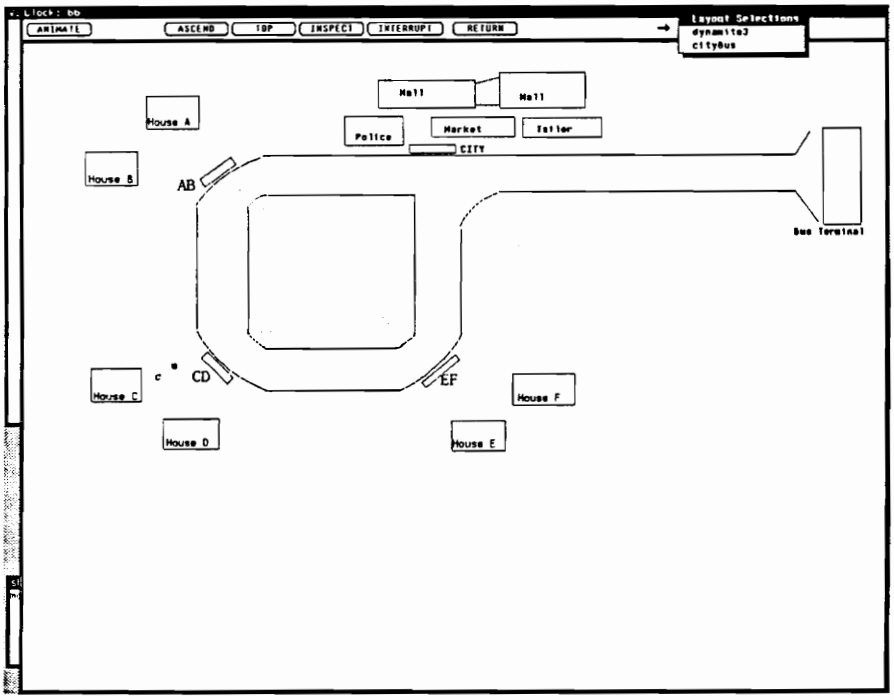

Figure 7.26 Displaying Text on Dynamic Object Images

logic, prior to the move of the bus to the next stop) for changing the entire object image. Within the branch of logic on the current bus stop position, notice that at **stopAB** the bus image is changed to "charter". The set image statement of the specification language is used. From Figure 7.7, the charter image is one of three possible images that the bus dynamic object can assume. Figures 7.27 and 7.28 show this type of change. In Figure 7.27 the bus has just left the terminal with its normal image. At a later point in execution, the image has been changed to the charter image as shown in Figure 7.28.

7.1.4.2 Non-Dynamic Object Images

Images not directly associated with dynamic objects can be manipulated as well. By this, we mean submodel, static object, subdynamic object, and base dynamic object images. This facility is quite useful in order to represent some type of state change for the affected component whose image is changed. For example, consider a server being busy or a queue being full. For these states, we can have the image change to indicate the busy or full condition. In the Bus Route, the bus stop image was changed whenever the bus was at the stop. The image was modified by writing the word "bus" over the bus stop image. Figure 7.17 contains the specification language set image statement to perform this change. At the beginning of the repeat loop and just after the bus slows down to stop, the bus stop image is changed to **"busIn"**. In Figure 7.29 the bus has just arrived at **stopCD**. The figure demonstrates the change of the bus stop submodel image.

7.1.4.3 Changing Context and Viewing Decomposed Dynamic Objects

During runtime, the VSSE animation facilities allow a modeler to change viewing context. The Bus Route offers a unique example of this feature: the ability to view animation and action within a decomposed dynamic object, in this case, the bus. Figures 7.30 through 7.33 give an example sequence of changing the visualization context. In the first figure (Figure 7.30) which shows animation within the bus, person objects from houses C and D are preparing to exit the bus. The runtime inspection facility indicates that there are, in fact, two objects exiting the bus (i.e., the **numberExiting** attribute). With a shift in context

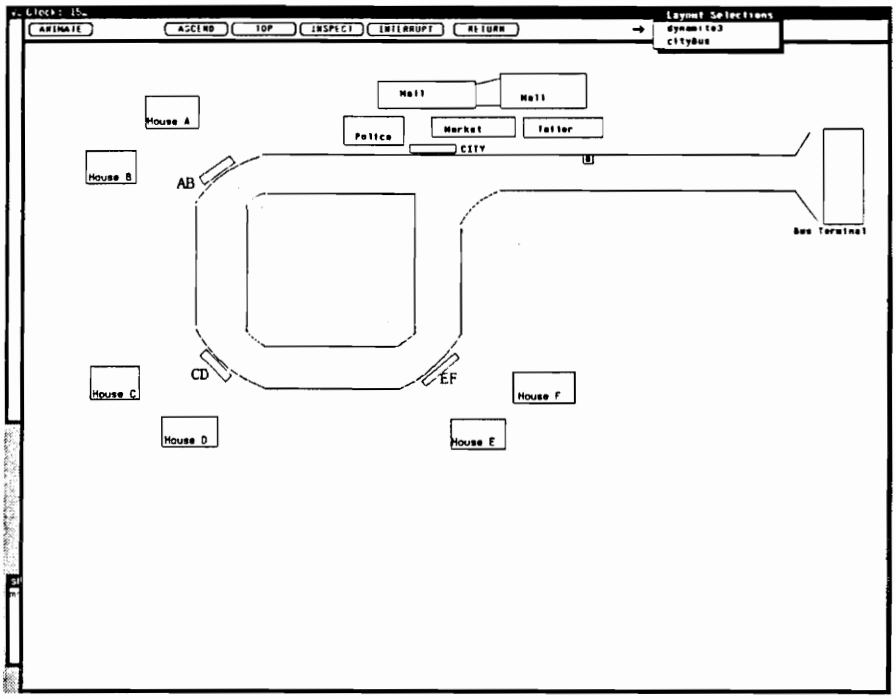

Figure 7.27 Dynamic Object (City Bus) with Normal Bus Image

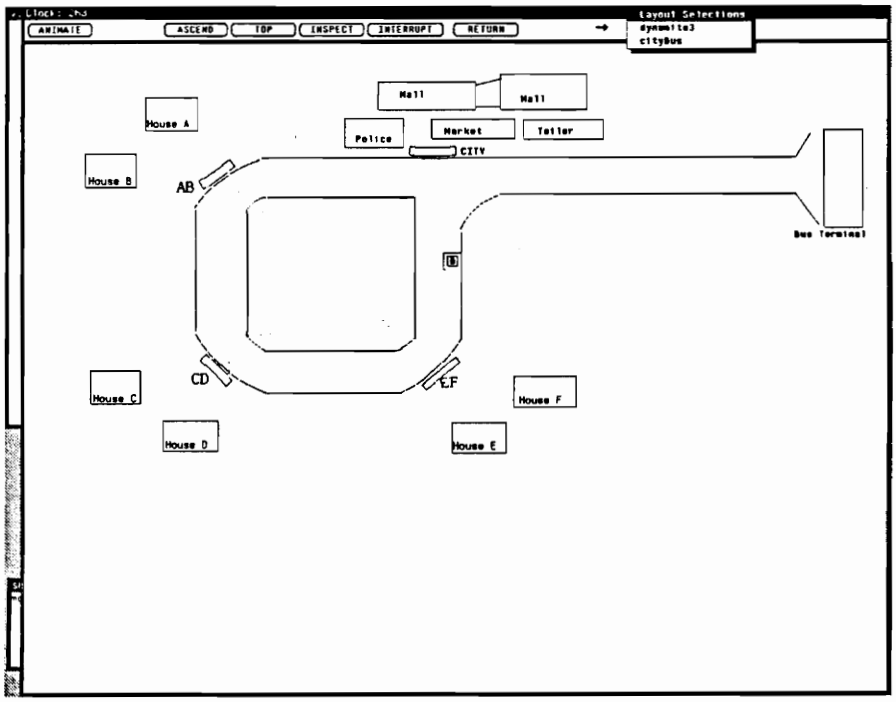

Figure 7.28 Dynamic Object (City Bus) with Charter Bus Image

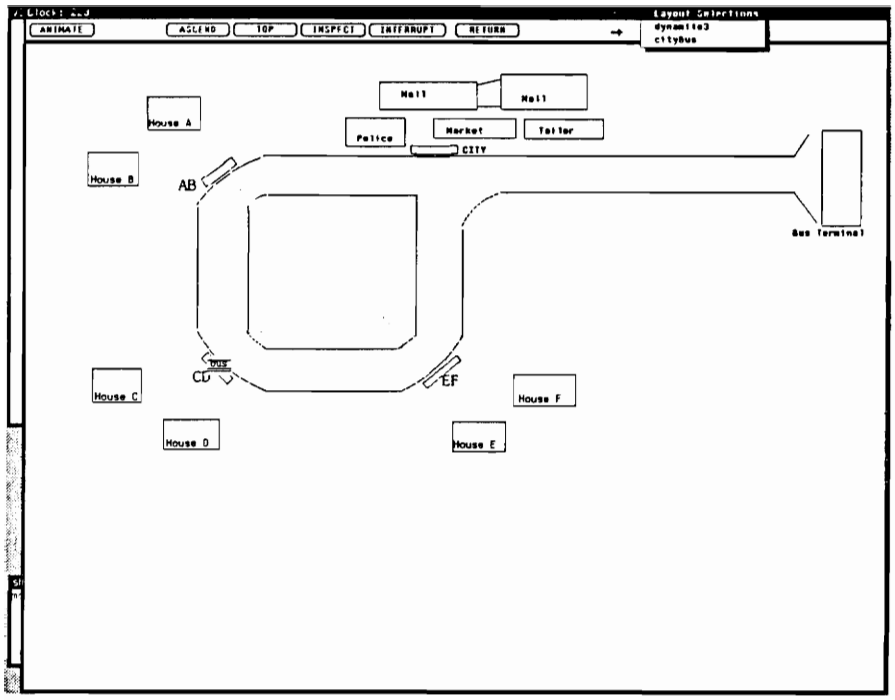

Figure 7.29 Changing Non-Dynamic Component (Bus Stop) Image

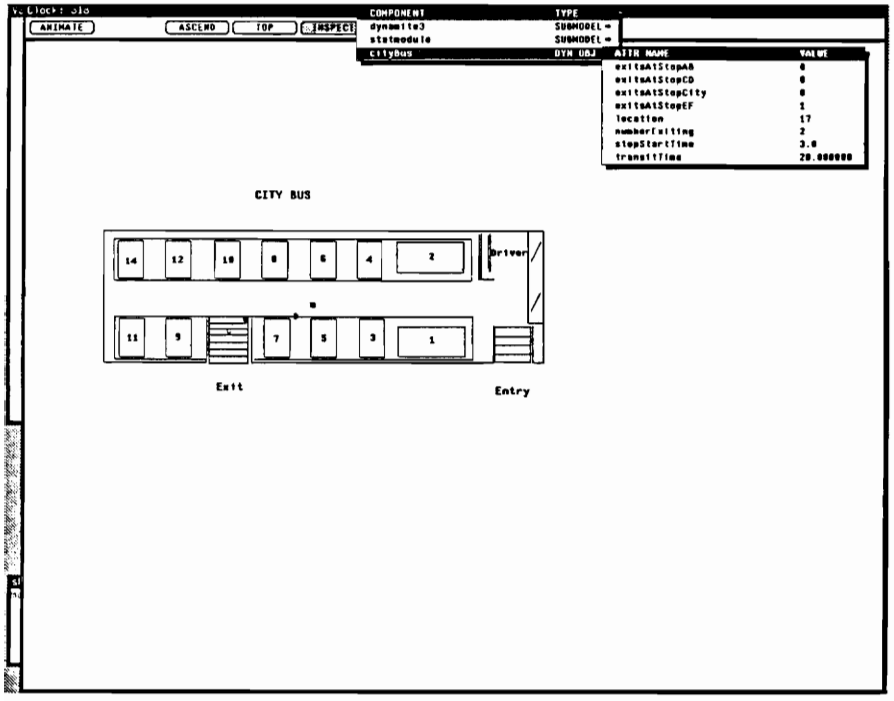

Figure 7.30 Persons (from House C and D) Exiting City Bus

to the top level city layout (Figure 7.31), the two persons are shown leaving the bus stop and returning to their homes. Later, in Figure 7.32, a "B" person is shown making his way to his seat while an "A" person is getting on the bus. Then again, still later in Figure 7.33, these two individuals exit the bus.

Viewing the layout of a decomposed dynamic object brings an interesting perspective to the model. Decomposed dynamic objects are, in a way, models unto themselves and containing their own world system. While viewing such a layout, the location of the dynamic object is not directly visualized. The location of the dynamic object can be only determined by inspection of the its attributes while in this viewing context.

7.2 Traffic Intersection Example

The Traffic Intersection follows previous work [Derrick 1988]. Unlike the Bus Route, the Traffic Intersection is complex with many component interactions. The DOMINO application to this example has proven its ability to handle this complexity. Using the graphical and object-oriented features of the DOMINO, model design and implementation proceeded at a much more rapid pace than the earlier research applications under direct programming using SIMULA, SIMSCRIPT, and GPSS/H. The general description of the Traffic Intersection follows:

> The Traffic Intersection (See ahead to Figure 7.38) is based on the traffic system located at the intersection of Price's Fork and Tom's Creek Roads in Blacksburg, Virginia. A single traffic light with north, south, east, and west directions controls vehicular traffic in each of the intersection's eleven lanes. The central intersection space is conceptually divided into thirty-five blocks through which the vehicles travel. The blocks in a vehicle's path are used as locators for that vehicle as it moves through the intersection and enable the representation of a smoothly flowing traffic pattern. The Traffic Intersection has been used to study the vehicular waiting times at the intersection during rush hour conditions. The analysis of light timing sequences can offer solutions to finding better timing sequences which yield shorter vehicle waiting times.

The discussion of this chapter centers on describing the application of the DOMINO so that the complexity issues are highlighted. With similar format to the last example, we explore the characteristics and form of the Traffic Intersection. One particular aspect, the use of methods and message passing, is substantiated.

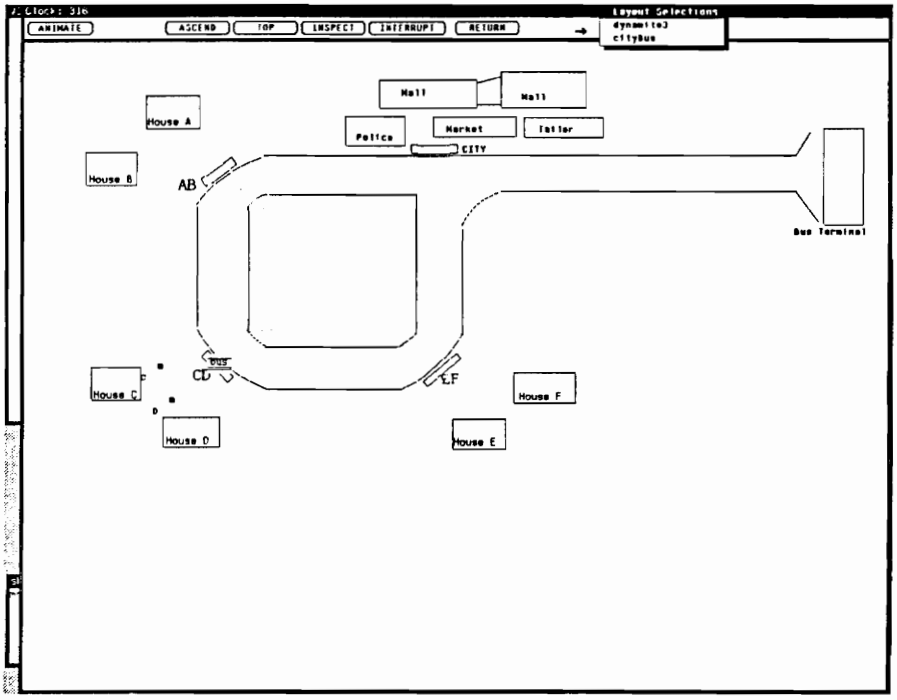

Figure 7.31 Persons (from House C and D) Leaving Bus Stop

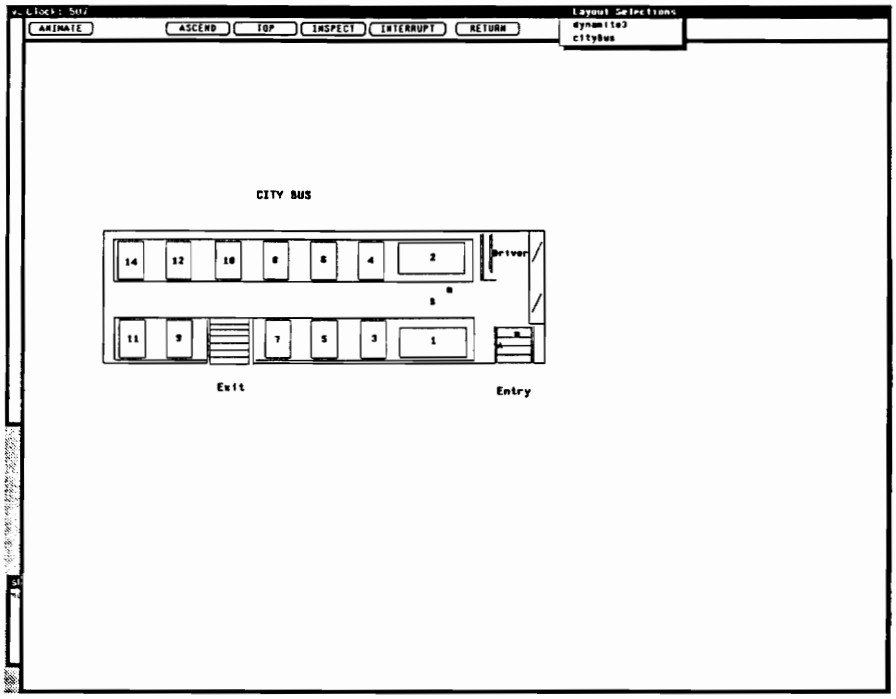

Figure 7.32 Persons (from House A and B) Entering City Bus

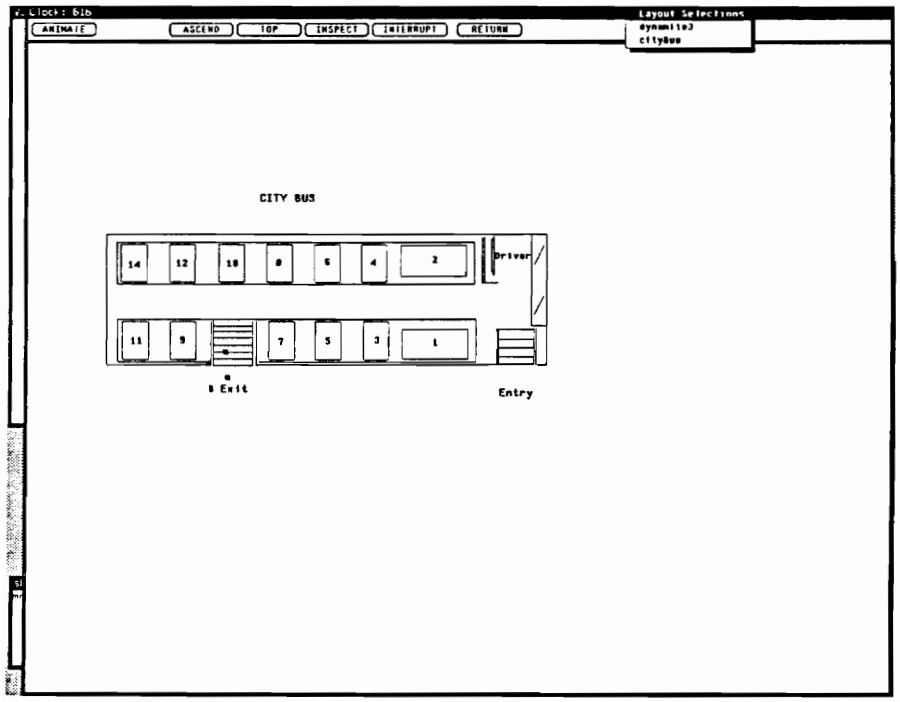

Figure 7.33 Persons (from House A and B) Exiting City Bus

7.2.1 Classes and Attributes

The Bus Route utilized nine basic modeler-defined object classes. By contrast, the Traffic Intersection has twenty-two. Twelve dynamic object classes are involved. The **lightCtrlExec** class contains the information necessary for the control of the light timing sequence and the light controller. A basic **vehicle** class is the parent for ten subclasses, the **laneJointVehicle** class for vehicles originating in lanes 1 and 2, and the **lane3Vehicle** to **lane11Vehicle** classes. Six submodel classes (**lightCtrl, blockQueue, block, laneBlock, vehicleSource,** and **vehicleSink**) and four static object classes (**light, blockSpace, interiorBlockSpace,** and **laneBlockSpace**) complete the class coverage. Class inheritance hierarchies (deeper than one level) are displayed in Figures 7.34, 7.35, and 7.36. The Traffic Intersection has no decomposed dynamic objects and, therefore, no subdynamic or base dynamic object classes. Classes and attributes (some, in a limited way) are described.

Class Name	Attribute	Attribute Description
Model Class		
traffic	**i,j**	Counting attributes
	dynobjId	Holder for dynamic object identifiers
	RED	Constants for light color
	GREEN	
	YELLOW	
	probSelection	Holds random variate selection values
	blockAIdle...	Boolean indicating Idle status of block, TRUE or FALSE
	block9Idle	
	NS_Clear	Boolean indicating intersection is clear, North/South
	WE_CLear	Boolean indicating intersection is clear, West/East
Dynamic Object Class(es)		
vehicle	**lane**	Holds lane identifier of vehicle
	right	Boolean indicating if vehicle is a right turner
	entered	Boolean indicating if vehicle has entered intersection
laneJointVehicle, lane3Vehicle...		
lane11Vehicle		No attributes
lightCtrlExec		No attributes
Submodel Class(es)		
lightCtrl	**colorDuration**	Time duration of each color sequence
blockQueue	**numberWaiting**	Counter for number of vehicles waiting for block in queue
	vehicleList	Single dimension array containing list of vehicle identifiers
block	**status**	Status of block: IDLE or BUSY

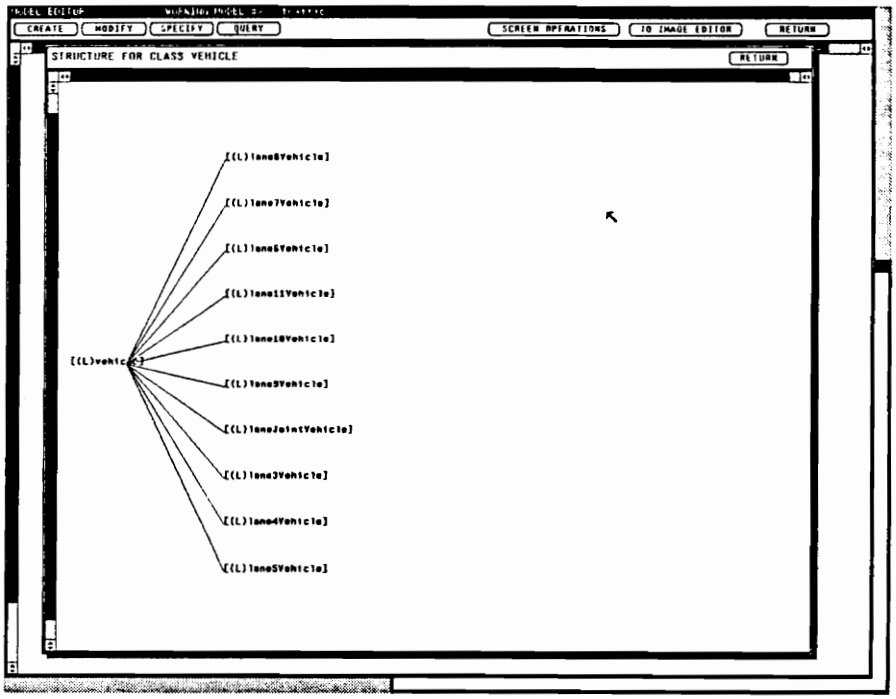

Figure 7.34 VEHICLE Class Hierarchy (Traffic Intersection)

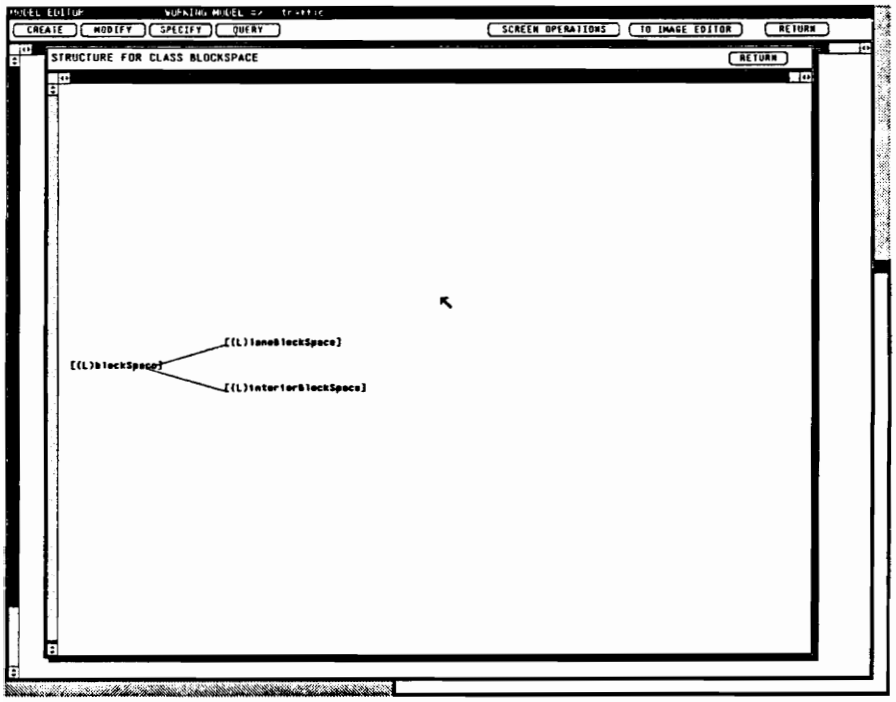

Figure 7.35 BLOCKSPACE Class Hierarchy (Traffic Intersection)

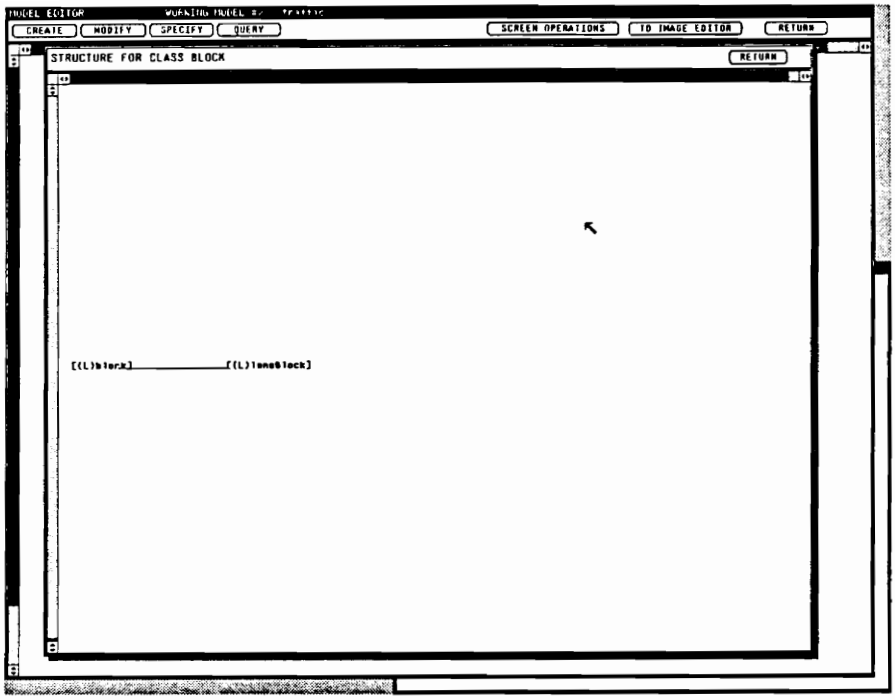

Figure 7.36 BLOCK Class Hierarchy (Traffic Intersection)

	laneUser	Lane origination of vehicle currently using the block
	turner	Boolean indicating if vehicle currently using block is turning
laneBlock	numberOfCarsWaiting	Number waiting for lane entry
vehicleSource		No attributes
vehicleSink		No attributes

Static Object Class(es)

light	color	Light color attribute: RED, YELLOW, GREEN
blockSpace	status	Status attribute: BUSY or IDLE
laneBlockSpace		No attributes
interiorBlockSpace		No attributes

Some of the above classes have no attributes. These classes have none because either none are required or because the parent class attributes are sufficient. Although not shown, the model component logic specifications are also attached to their associated classes. For the classes without attributes, the logic specifications are the important class ingredients. For example, the dynamic object classes for individual lane vehicles all have the same attributes (of the parent vehicle class), but each needs its very own logic specification.

7.2.2 Model Structure and Graphical Elements

The model static structure has is two levels deep. Figure 7.37 is the model static structure with portions hidden outside the VSSE viewing window. A complete listing of model component instances (literal names and owning classes) in the model static structure is given below. This description has some details omitted for brevity (due to the very large number of component instances). This omission does not hinder the explanation, however.

Owning Class	Literal Instance(s)
lightCtrl	lightController
blockQueue	blockAQueue...blockZQueue
blockQueue	block1Queue...block9Queue
blockQueue	laneJointQueue, lane1Queue...lane11Queue
block	blockA...blockZ
block	block1...block9
laneBlock	laneJoint, lane1...lane11
vehicleSource	jointSource, lane3Source...lane11Source
vehicleSink	bypassSink, tomscreekSink, townSink, campusSink
light	weLight, ewLight, nsLight
blockSpace	blockASpace...blockZSpace

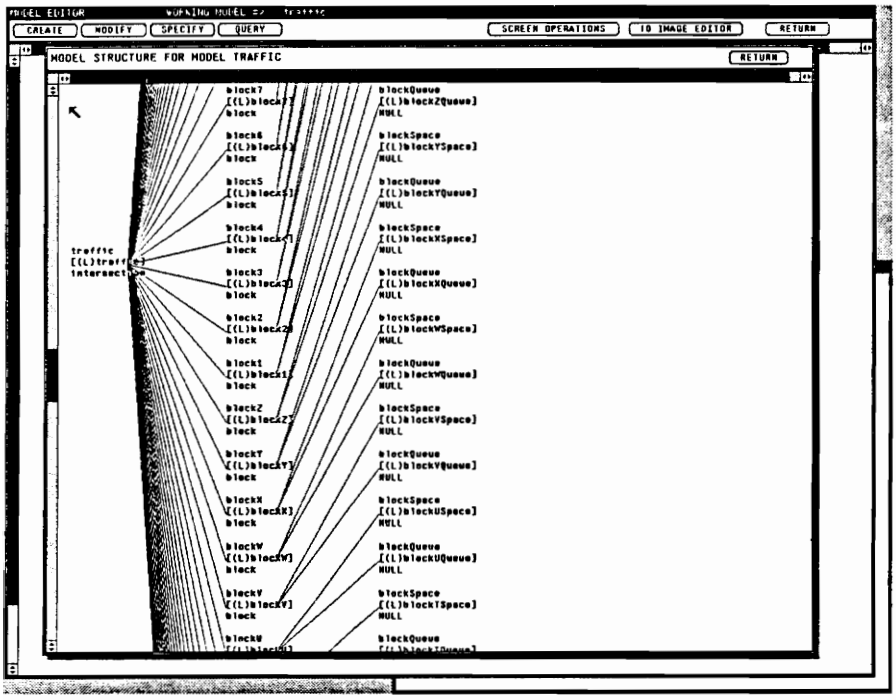

Figure 7.37 Model Static Structure (Traffic Intersection)

laneBlockSpace	laneJointSpace, lane1Space...lane11Space
interiorBlockSpace	block1Space...block9Space

The Traffic Intersection contains only two layouts, one for the top level model static structure (Figure 7.38) and one (Figure 7.39) for each decomposed **block** or **laneBlock** instance. In this case, since the **laneBlock** class is a subclass of the **block** class, then there is no problem with the **laneBlock** instances using the same decomposition layout as the **block** instances. One must carefully notice, however, that this produces **block** class instances (**blockA...blockZ, block1...block9**) which are decomposed into the appropriate **blockQueue** class (**blockAQueue...blockZQueue, block1Queue...block9Queue**) and **blockSpace** class (**blockASpace...blockZSpace**) or **interiorBlockSpace** class (**block1Space...block9Space**) instances. **LaneBlock** class instances (**laneJoint, lane1...lane11**) are decomposed into appropriate **blockQueue** class instances (**laneJointQueue, lane1Queue...lane11Queue**) and **laneBlockSpace** class instances (**laneJointSpace, lane1Space...lane11Space**). The thrust here is that the **blockQueue** class serves double duty for the queues within **block** class instance and **laneBlock** class instance decompositions. The space instances within these decompositions don't have this double requirement. The modeling decisions which produced this unusual trait of the decomposition occurred during the instantiation decisions for the model static structure. Due to the generic nature of the **blockQueue** class supervisory logic, that class was suited for both the intersection and lane block instances. The space logic required different classes within the decompositions because of the varying logic requirements. This is explained further in the next section. The layout definitions for the two layouts are shown in Figures 7.40 and 7.41 respectively.

Finally, Figure 7.42 displays the images which the vehicles, blocks, and the light can assume. Vehicles use only one image. The movement of vehicles in the intersection is from block submodel to block submodel. VSSE animation generally causes the moving dynamic object (in this case the vehicle) to disappear once the dynamic object is in the submodel. To assist the animation, the block images are given new images to indicate the presence of the vehicle. Thus, blocks take on two images, one with a car or vehicle "in" and one with the vehicle "out". The light can have one image from a set of four images (light, red, green, and yellow).

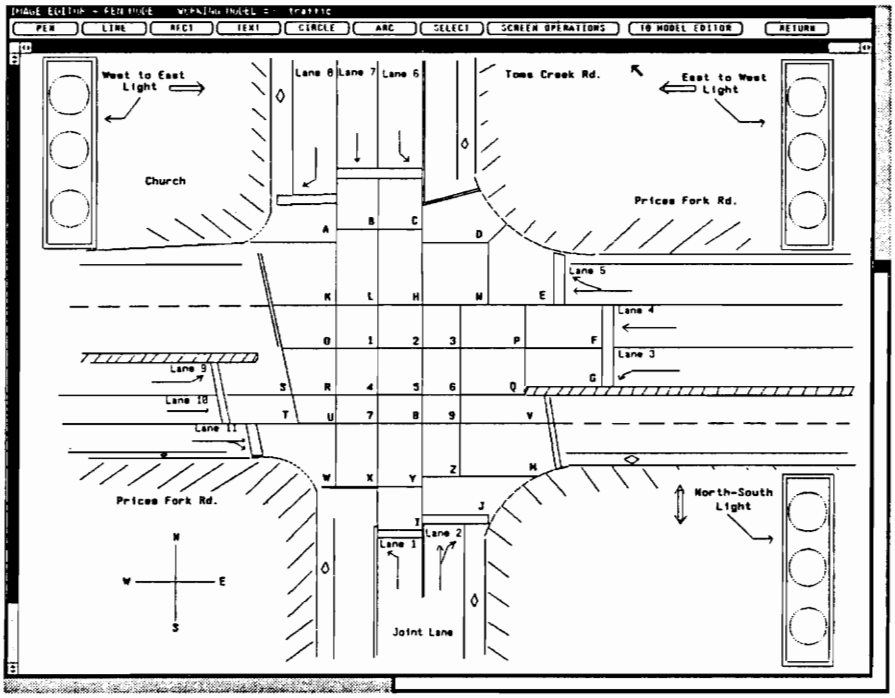

Figure 7.38 Top Level Layout Image (Traffic Intersection)

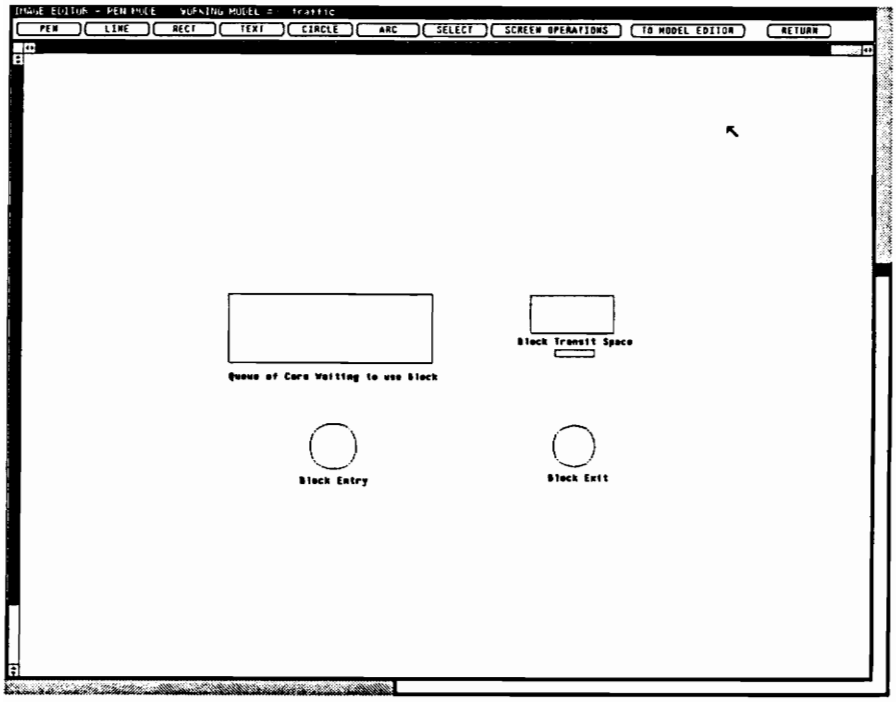

Figure 7.39 Layout Image of BLOCK/LANEBLOCK Class (Traffic Intersection)

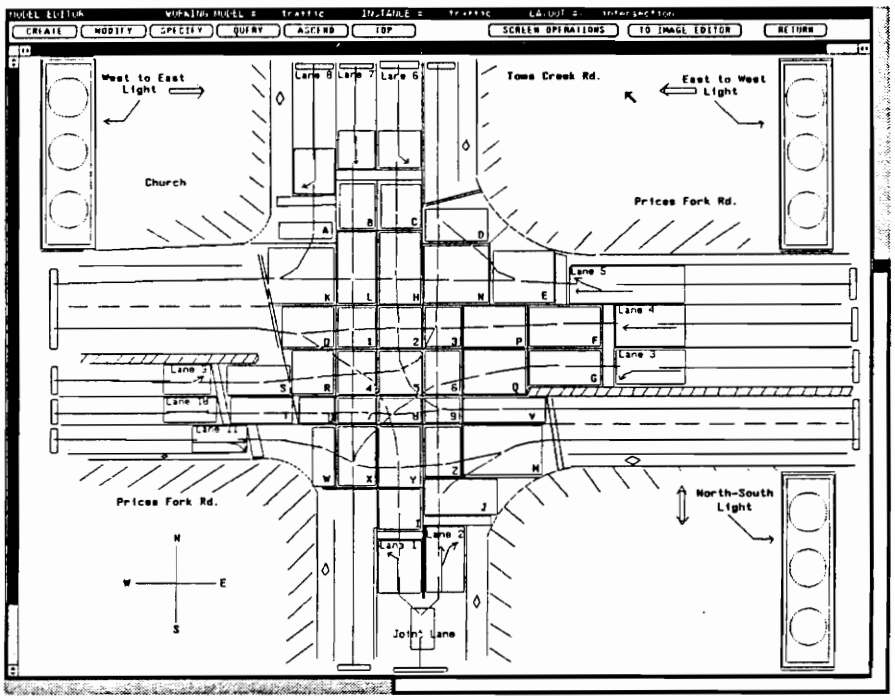

Figure 7.40 Top Level Layout Definition (Traffic Intersection)

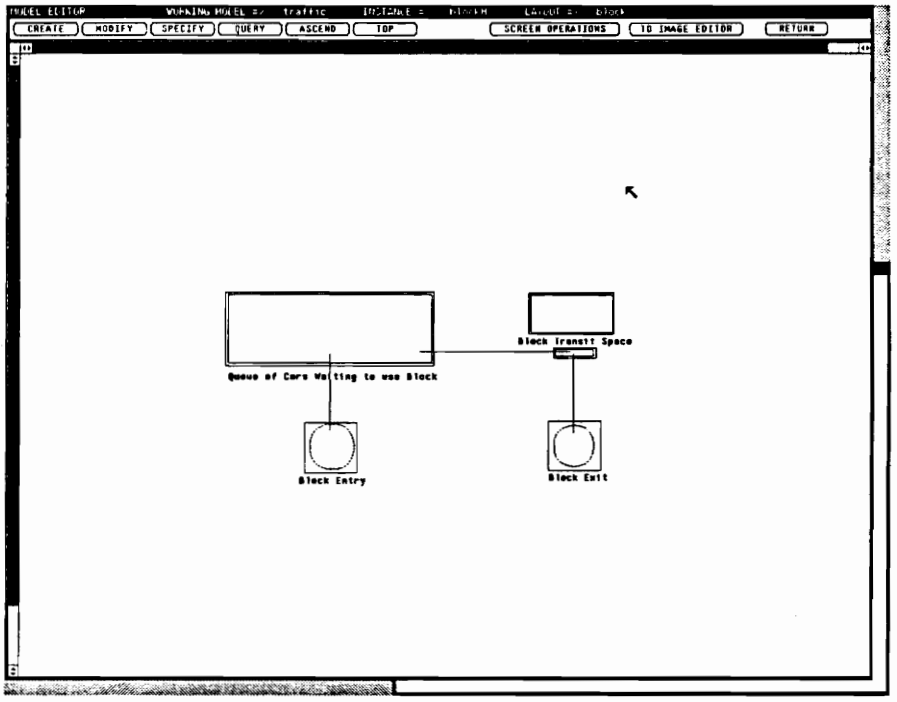

Figure 7.41 Layout Definition of BLOCK/LANEBLOCK Class (Traffic Intersection)

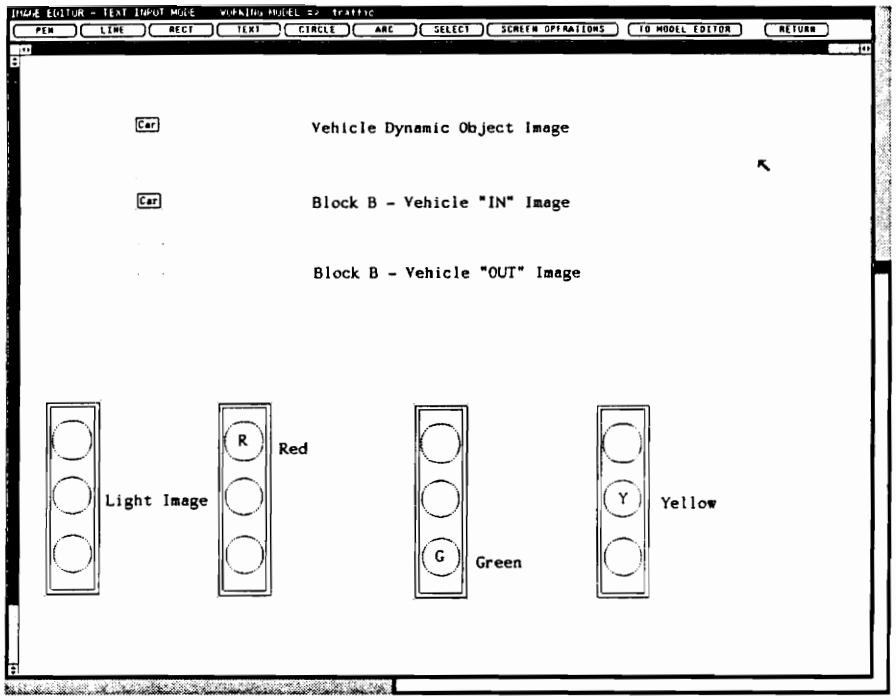

Figure 7.42 Component Images (Traffic Intersection)

7.2.3 Component Logic Specifications

The Traffic Intersection model component logic was developed using the hybrid perspective and, therefore, uses a combination of supervisory logic and self logic. Supervisory logic (modeler-defined) is associated with the **builtin** class, the **lightCtrl** class, the **blockQueue** class, the **blockSpace** class, the **laneBlockSpace** class, and the **interiorBlockSpace** class. The use of methods and message passing, important features of the OOP facilities, are carefully described. The coordination and use of the methods within supervisory and self logic specifications are effective in limiting the conditional complexity (for understanding) among component interactions. The discussion clearly reflects this. Each lane vehicle class (**laneJointVehicle, lane3Vehicle...lane11Vehicle** classes) have self logic for directing vehicle movements.

7.2.3.1 Supervisory Logic Specifications

The builtin class supervisory logic (Figure 7.43) creates the **lightCtrlExec** dynamic object which starts in and initiates the supervisory logic of the light controller. A dynamic object is then created for each lane source. Each of these objects represent the first vehicles to be generated for each lane. The lane source component instances are positioned on the top level layout so that the vehicles are animated as entering the intersection from the boundaries of the layout and from the lanes. The initializing system dynamic object is then destroyed.

The light color sequence and timing is managed by the lightCtrl class supervisory logic (Figure 7.44). Since the sequence is repeated, the actions are contained in a **repeat forever** loop. The initial colors are set for each light and direction. The image of the light components is also assigned to correspond to its color. The intersection clearance boolean is set to FALSE. (One of the conditions that must be met before vehicles can enter the intersection is a "clearance" condition.) Depending on the sequence duration, the light attributes (color and image display) are maintained for that duration. The determined duration is accomplished with the **engageIn** statement. Once a portion of the sequence is completed, the light color and display image are reset. A new duration is determined and attributes are held for that duration. After

```
-- BUILTIN SUBMODEL CLASS SUPERVISORY LOGIC
begin
   -- Set up the dynamic object which will control the Light Controller
   create vdo belonging to class lightCtrlExec putting its id in
      dynObjId starting in vsm "lightController";

   -- Create a dynamic object for each lane source for starting up the model.
   create rdo belonging to class laneJointVehicle putting its id in
     dynObjId starting in sm "jointSource";
   create rdo belonging to class lane3Vehicle putting its id in
     dynObjId starting in sm "lane3Source";
   create rdo belonging to class lane4Vehicle putting its id in
     dynObjId starting in sm "lane4Source";
   create rdo belonging to class lane5Vehicle putting its id in
     dynObjId starting in sm "lane5Source";
   create rdo belonging to class lane6Vehicle putting its id in
     dynObjId starting in sm "lane6Source";
   create rdo belonging to class lane7Vehicle putting its id in
     dynObjId starting in sm "lane7Source";
   create rdo belonging to class lane8Vehicle putting its id in
     dynObjId starting in sm "lane8Source";
   create rdo belonging to class lane9Vehicle putting its id in
      dynObjId starting in sm "lane9Source";
   create rdo belonging to class lane10Vehicle putting its id in
      dynObjId starting in sm "lane10Source";
   create rdo belonging to class lane11Vehicle putting its id in
      dynObjId starting in sm "lane11Source";

   -- Destroy the initializing object
   destroy
end
```

Figure 7.43 BUILTIN Class Supervisory Logic (Traffic Intersection)

```
-- LIGHTCTRL SUBMODEL CLASS SUPERVISORY LOGIC
repeat forever
   begin

      -- Set the colors, NS to green, WE & EW to red
      put GREEN@ into attr color of so "nsLight";
      set image of so "nsLight" to "green";

      put RED@ into attr color of so "weLight";
      set image of so "weLight" to "red";
      put RED@ into attr color of so "ewLight";
      set image of so "ewLight" to "red";

      -- Turn off the traffic intersection clearance for the West-East traffic
      put FALSE@ into WE_Clear;

      -- Perform initial 20 sec delay
      put 200.0 into attr colorDuration of this sm;
      engageIn delay for attr colorDuration of this sm secs;

      -- Change NS to red, and delay for intersection clearance
      put RED@ into attr color of so "nsLight";
      set image of so "nsLight" to "red";

      -- Turn off the traffic intersection clearance for the North-South traffic
      put FALSE@ into NS_Clear;

      put 10.0 into attr colorDuration of this sm;
      engageIn delay for attr colorDuration of this sm secs;

      -- Open up the WE lane by setting WE to green
      put GREEN@ into attr color of so "weLight";
      set image of so "weLight" to "green";

      -- Now delay a little longer, maintaining red for EW
      put 130.0 into attr colorDuration of this sm;
      engageIn delay for attr colorDuration of this sm secs;

      -- New set EW to green and delay for 16 sec
      put GREEN@ into attr color of so "ewLight";
      set image of so "ewLight" to "green";
      put 160.0 into attr colorDuration of this sm;
      engageIn delay for attr colorDuration of this sm secs
   end -- end repeat actions
end -- repeat statement
```

Figure 7.44 LIGHTCTRL Class Supervisory Logic (Traffic Intersection)

the entire sequence has been accomplished, then the sequence is repeated.

In order for vehicles to move into a block (or laneBlock), the block must be available. There is a queue and a transit space associated with each block (Figure 7.41). The blockQueue class supervisory logic (Figure 7.45) handles the entrance of vehicles to the block (or laneBlock). Entering vehicles must wait in the queue if other vehicles are ahead of them (already waiting) or is the space of the block (or laneBlock) is BUSY. Vehicles wait in the queue (see the **engageIn waiting**) statement, until first in the queue and the space of the block (or laneBlock) is IDLE. Since the vehicles are being directed by self logic (Section 7.2.4.2), no movements of the vehicles are generated within the supervisory logics.

Once a vehicle has entered a block (or laneBlock), the appropriate **laneBlockSpace**, **blockSpace**, or **interiorBlockSpace** is given temporary supervision of the vehicle by its self logic. Figures 7.46, 7.47, and 7.48 give the respective supervisory logic specifications. Once in a block (laneBlock), leaving the block and continuing is contingent upon various conditions, depending upon which block (or laneBlock) the vehicle is in. We'll describe which we mean and include a discussion of the use of methods which is also demonstrated.

In Figure 7.46, a vehicle can only be free to continue to lane 1 or lane 2 (from the lane joint space) when the number of vehicles in the lane 1 or 2 queue is less than 5. The multiple departure conditions from lane 7 are those which make it safe for the vehicle to enter the intersection (light green, first block in intersection ("B") IDLE, and intersection clear). The message activating the method **clearedNS** (Figure 7.49) of the model helps check for the many clearance conditions. No parameters are used in this method; it utilizes the **checkIdle** (Figure 7.50) method which is sent to the blocks that must be clear. **CheckIdle** returns TRUE or FALSE.

Also notice the use of methods **setBusy** (Figure 7.51) and **setIdle** (Figure 7.52) of the lane 7 submodel in the laneBlock supervisory logic. These set the laneBlock busy upon vehicle entry and then back to idle upon vehicle departure and entry into the intersection. The **setBusy** method uses parameters **turning** and **userLane** which correspond to the **right** and **lane** attributes of the vehicle (passed to the method at its message call). Notice the parameters (in alphabetical order) associate with the attributes as ordered in the

```
-- BLOCKQUEUE SUBMODEL CLASS SUPERVISORY LOGIC
if attr numberWaiting of this sm is > 0 or
   attr status of so blockSpace is BUSY@ then
--*************************************************
--* If there are others waiting or the blockspace is busy, join the queue and
--*   wait until first in line and blockspace is IDLE
--*************************************************
   begin
      add 1 to attr numberWaiting of this sm ;
      put attr numberWaiting of this sm into holdingVar;
      put sys attr ident of this do into
         attr vehicleList[holdingVar] of this sm;

      engageIn waiting until attr vehicleList[1] of this sm is
        sys attr ident of this do and attr status of so blockSpace is IDLE@;

      -- Reset every vehicle's position in the Queue
      put attr numberWaiting of this sm into holdingVar;
      subtract 1 from holdingVar;
      repeat with i = 1 to holdingVar
         begin
            put i+1 into dynObjId;
            put attr vehicleList[dynObjId] of this sm into
                attr vehicleList[i] of this sm
         end
      end;
      subtract 1 from attr numberWaiting of this sm
   end
--*************************************************
--* Otherwise, proceed immediately through the queue
--*************************************************
```

Figure 7.45 BLOCKQUEUE Class Supervisory Logic (Traffic Intersection)

```
-- LANEBLOCKSPACE STATIC OBJECT CLASS SUPERVISORY LOGIC
  branch sys attr compInst of this so
  --************************************************
  --* At the head of lanes, seek entry into the intersection
  --* or in the case of laneJoint, entry into lanes 1 or 2
  --************************************************
  (LANEJOINTSPACE_SO_INST@)
     begin
        branch attr lane of this do
        (ONE@)
           begin
            put BUSY@ into attr status of this so;
            if (attr numberWaiting of sm "lane1Queue" > 4) then
               engageIn waiting until
                  (attr numberWaiting of sm "lane1Queue" < 5);
            put IDLE@ into attr status of this so
           end;
        (TWO@)
           begin
           ...
           end
        end
     end;
  (LANE1SPACE_SO_INST@)
  ...
  (LANE7SPACE_SO_INST@)
     begin
        put BUSY@ into attr status of this so;
        send message setBusy to sm "lane7" using
           (attr right of this do,
            attr lane of this do);
        if (NS_Clear is FALSE@) then
           begin
              engageIn waiting until
                 ((attr color of so "nsLight" is GREEN@) and
                 (attr status of sm "blockB" is IDLE@) and
                 ((message clearedNS to sm "traffic" is TRUE@) or
                 (NS_Clear is TRUE@)))
           end
        else if (attr status of sm "blockB" is BUSY@) then
           begin
              engageIn waiting until
                 ((attr color of so "nsLight" is GREEN@) and
                 (attr status of sm "blockB" is IDLE@) and
                 ((message clearedNS to sm "traffic" is TRUE@) or
                 (NS_Clear is TRUE@)))
           end;
        put IDLE@ into attr status of this so;
        send message setIdle to sm "lane7"
     end;
  ...
  (LANE11SPACE_SO_INST@)
     begin
     ...
     end
end
```

Figure 7.46 LANEBLOCKSPACE Class Supervisory Logic (Traffic Intersection)

```
-- BLOCKSPACE STATIC OBJECT CLASS SUPERVISORY LOGIC
if ((sys attr compInst of this so is BLOCKASPACE_SO_INST@) or
      (sys attr compInst of this so is BLOCKBSPACE_SO_INST@) or
      (sys attr compInst of this so is BLOCKCSPACE_SO_INST@) or
      (sys attr compInst of this so is BLOCKDSPACE_SO_INST@)) then
   branch sys attr compInst of this so
   (BLOCKASPACE_SO_INST@)
      begin
         put BUSY@ into attr status of this so;
         send message setBusy to sm "blockA" using
            (attr right of this do,
             attr lane of this do);
         send message findTransit to sm "blockA" using
            (attr right of this do,
            attr lane of this do) putting result in
            sys attr delay of this do;
         multiply sys attr delay of this do by 10;
         engageIn transitingA for sys attr delay of this do secs
      end;
   ...
   (BLOCKDSPACE_SO_INST@)
   ...
   end
else
   branch sys attr compInst of this so
   (BLOCKESPACE_SO_INST@)
   ...
   (BLOCKHSPACE_SO_INST@)
      branch attr lane of this do
      (FIVE@)
         begin
            put BUSY@ into attr status of this so;
            send message setBusy to sm "blockH" using
               (attr right of this do,
                attr lane of this do);
            send message findTransit to sm "blockH" using
              (attr right of this do,
               attr lane of this do) putting result in
               sys attr delay of this do;
            multiply sys attr delay of this do by 10;
            engageIn transitingH for sys attr delay of this do secs
         end;
      (SIX@)
         begin
            put BUSY@ into attr status of this so;
            send message setBusy to sm "blockH" using
               (attr right of this do,
                attr lane of this do);
            put IDLE@ into attr status of so "blockCSpace";
            send message setIdle to sm "blockC";
            send message findTransit to sm "blockH" using
              (attr right of this do,
               attr lane of this do) putting result in
               sys attr delay of this do;
            multiply sys attr delay of this do by 10;
            engageIn transitingH for sys attr delay of this do secs
         end
      end;
   ...
   (BLOCKZSPACE_SO_INST@)
   ...
      end
end
```

Figure 7.47 BLOCKSPACE Class Supervisory Logic (Traffic Intersection)

```
-- INTERIORBLOCKSPACE STATIC OBJECT SUPERVISORY LOGIC
-- Handle vehicles in the interior block spaces
branch sys attr compInst of this so
(BLOCK1SPACE_SO_INST@)
   branch attr lane of this do
   (FOUR@)
      begin
      put BUSY@ into attr status of this so;
      send message setBusy to sm "block1" using
         (attr right of this do,
          attr lane of this do);
      put IDLE@ into attr status of so "block3Space";
      send message setIdle to sm "block3";
      send message findTransit to sm "block1" using
         (attr right of this do,
         attr lane of this do) putting result in
         sys attr delay of this do;
      multiply sys attr delay of this do by 10;
      engageIn transiting1 for sys attr delay of this do secs
      end;
   (SEVEN@)
      begin
      ...
      end
   end;
(BLOCK2SPACE_SO_INST@)
...
(BLOCK5SPACE_SO_INST@)
   branch attr lane of this do
   (SIX@)
      begin
         put BUSY@ into attr status of this so;
         send message setBusy to sm "block5" using
            (attr right of this do,
            attr lane of this do);
         put IDLE@ into attr status of so "blockHSpace";
         send message setIdle to sm "blockH";
         send message findTransit to sm "block5" using
            (attr right of this do,
            attr lane of this do) putting result in
            sys attr delay of this do;
         multiply sys attr delay of this do by 10;
         engageIn transiting5 for sys attr delay of this do secs;
         engageIn waiting until
           (message left6OK to sm "traffic" is TRUE@)
      end;
   (NINE@)
      begin
         put BUSY@ into attr status of this so;
         send message setBusy to sm "block5" using
            (attr right of this do,
            attr lane of this do);
         put IDLE@ into attr status of so "blockRSpace";
         send message setIdle to sm "blockR";
         send message findTransit to sm "block5" using
            (attr right of this do,
            attr lane of this do) putting result in
            sys attr delay of this do;
         multiply sys attr delay of this do by 10;
         engageIn transiting5 for sys attr delay of this do secs;
         engageIn waiting until
           (message left9OK to sm "traffic" is TRUE@)
      end
   end;
...
end
```

Figure 7.48 INTERIORBLOCKSPACE Class Supervisory Logic (Traffic Intersection)

```
-- CLEAREDNS METHOD LOGIC
--**************************************
--* Checks clearance of intersection in the
--* North or South directions
--**************************************
begin

   send message checkIdle to sm "blockL" putting result in blockLIdle;
   send message checkIdle to sm "blockH" putting result in blockHIdle;
   send message checkIdle to sm "blockN" putting result in blockNIdle;
   send message checkIdle to sm "block1" putting result in block1Idle;
   send message checkIdle to sm "block2" putting result in block2Idle;
   send message checkIdle to sm "block3" putting result in block3Idle;
   send message checkIdle to sm "blockP" putting result in blockPIdle;
   send message checkIdle to sm "blockF" putting result in blockFIdle;
   send message checkIdle to sm "blockS" putting result in blockSIdle;
   send message checkIdle to sm "blockR" putting result in blockRIdle;
   send message checkIdle to sm "block4" putting result in block4Idle;
   send message checkIdle to sm "block5" putting result in block5Idle;
   send message checkIdle to sm "block6" putting result in block6Idle;
   send message checkIdle to sm "blockQ" putting result in blockQIdle;
   send message checkIdle to sm "blockG" putting result in blockGIdle;
   send message checkIdle to sm "blockT" putting result in blockTIdle;
   send message checkIdle to sm "blockU" putting result in blockUIdle;
   send message checkIdle to sm "block7" putting result in block7Idle;
   send message checkIdle to sm "block8" putting result in block8Idle;
   send message checkIdle to sm "block9" putting result in block9Idle;
   send message checkIdle to sm "blockY" putting result in blockYIdle;
   send message checkIdle to sm "blockZ" putting result in blockZIdle;
   send message checkIdle to sm "blockE" putting result in blockEIdle;
   send message checkIdle to sm "blockW" putting result in blockWIdle;
   send message checkIdle to sm "blockX" putting result in blockXIdle;

   if (blockLIdle and blockHIdle and blockNIdle and block1Idle and
       block2Idle and block3Idle and blockPIdle and blockFIdle and
       blockSIdle and blockRIdle and block4Idle and block5Idle)
       then
          begin
             if (block6Idle and blockQIdle and blockGIdle and blockTIdle and
                 blockUIdle and block7Idle and block8Idle and block9Idle and
                 blockYIdle and blockZIdle) then
                 begin
                   if ((blockEIdle or attr laneUser of sm "blockE" is != 5 or
                        attr turner of sm "blockE" is TRUE@) and
                       (blockWIdle or attr laneUser of sm "blockW" is != 11 or
                        attr turner of sm "blockW" is TRUE@) and
                       (blockXIdle or attr laneUser of sm "blockX" is != 11 or
                        attr turner of sm "blockX" is TRUE@)) then
                       begin
                          put TRUE@ into result;
                          put TRUE@ into NS_Clear
                       end
                   else
                      put FALSE@ into result
                 end
              else
                 put FALSE@ into result
          end
       else
          put FALSE@ into result;

   return result
end
```

Figure 7.49 CLEAREDNS Method Logic Specification

```
-- CHECKIDLE METHOD LOGIC SPECIFICATION
--*****************************************
--* Check value of attr status of block
--*****************************************
begin
   if attr status of this sm is IDLE@ then
      put TRUE@ into result
   else
      put FALSE@ into result;
   return result
end
```

Figure 7.50 CHECKIDLE Method Logic Specification

```
-- SETBUSY METHOD LOGIC SPECIFICATION
--*************************************************
--* Make assignments and change object attribute values
--*************************************************
begin
  put userLane into attr laneUser of this sm;
  put turning into attr turner of this sm;
  put BUSY@ into attr status of this sm
end
```

Figure 7.51 SETBUSY Method Logic Specification

```
-- SETIDLE METHOD LOGIC SPECIFICATION
--*****************************************
--* Release the block to IDLE and reset other values
--*****************************************
begin
   put 0 into attr laneUser of this sm;
   put FALSE@ into attr turner of this sm;
   put IDLE@ into attr status of this sm
end
```

Figure 7.52 SETIDLE Method Logic Specification

message call.

The **blockSpace** class supervisory logic (Figure 7.47) uses the **findTransit** method of Figure 7.53 to set the determined time of delay for transit across the block. Once the vehicle has delayed this transit time, the vehicle is free to continue.

Figure 7.48, the **interiorBlockSpace** class supervisory logic, controls vehicle continuance for blocks in the interior or central regions of the intersection. This logic is unique in that certain of these interior blocks must guide vehicles in turning left across oncoming traffic. In this logic for vehicles from lane 6 which are in the block 5 space, there must be clearance for turning left. Figure 7.54 contains the method logic specification for the **left6OK** method of the model. For this turn, certain blocks must be unoccupied or occupied by vehicles with certain characteristics (e.g., blocks 9 and Z must be idle and block J must be idle or occupied with a vehicle not from lane 2 or occupied with a vehicle that is turning right).

7.2.3.2 Self Logic Specifications

Every vehicle dynamic object is created from within the specification language and is subsequently directed by self logic from its subclass (**laneJointVehicle, lane3Vehicle...lane11Vehicle**). Figure 7.55 is one example of the self logic specifications from the self logic of a lane 8 vehicle. The self logic sets the arrival time of the vehicle to the lane with a "draw" from the weibull distribution. Bootstrapping of the next arrival is performed. Review of the self logic provides a clear description of the path that lane 8 vehicles must take: from **blockA** (and inner queue and space) to **blockK** (and inner queue and space) to the **bypassSink** for system departure. The vehicle is destroyed at departure.

7.2.4 Snapshots of Model Execution

Several views of the animation during model execution are displayed in Figures 7.56 through 7.61. These figures are in chronological sequence (see clock times in upper left corner). As time progresses over the sequence, the light images change in accordance with the previously described **lightCtrl** supervisory logic. (In Figure 7.56, the north-south light is green and becomes red in Figure 7.61. These changes to

```
-- FINDTRANSIT METHOD LOGIC SPECIFICATION
begin
   branch sys attr compInst of this sm
   (BLOCKA_SM_INST@)
       -- Lane 8 User
       put 2.153 into result;
   (BLOCKB_SM_INST@)
       -- Lane 7 User
       put 1.071 into result;
   (BLOCKC_SM_INST@)
   ...
   (BLOCKZ_SM_INST@)
       if userLane is 2 then
          put 0.935 into result
       else if userLane is 11 then
          put 0.529 into result;
   (BLOCK1_SM_INST@)
       if userLane is 4 then
          put 0.519 into result
       else if userLane is 7 then
          put 0.468 into result;
   ...
   (BLOCK4_SM_INST@)
       if userLane is 1 then
          put 1.132 into result
       else if userLane is 7 then
          put 0.636 into result
       else if userLane is 9 then
          put 0.551 into result;
   ...
   (BLOCK9_SM_INST@)
       if userLane is 2 then
          put 0.467 into result
       else if userLane is 6 then
          put 0.725 into result
       else if userLane is 10 then
          put 0.431 into result
   end ; -- branch
   return result
end -- begin
```

Figure 7.53 FINDTRANSIT Method Logic Specification

```
-- LEFT6OK METHOD LOGIC SPECIFICATION
--****************************************
--* Checks clearance of intersection for
--* a leftTurn in lane 6
--****************************************
begin

   send message checkIdle to sm "block9" putting result in block9Idle;
   send message checkIdle to sm "blockZ" putting result in blockZIdle;
   send message checkIdle to sm "blockJ" putting result in blockJIdle;

   if (block9Idle and blockZIdle and
       (blockJIdle or attr laneUser of sm "blockJ" is != 2 or
        attr turner of sm "blockJ" is TRUE@)) then
         put TRUE@ into result
   else
         put FALSE@ into result;

   return result
end
```

Figure 7.54 LEFT6OK Method Logic Specification

```
-- LANE8VEHICLE DYNAMIC OBJECT CLASS SELF LOGIC
   begin
   --*************************************************
   --* Set the delay for this vehicle's arrival at Lane 8
   --*************************************************
      put 56.0592 into attr scale of vsm "statmodule";
      put 0.63923 into attr shape1 of vsm "statmodule";
      send message weibl to vsm "statmodule" using
         (attr scale of vsm "statmodule",
          attr iseed8 of vsm "statmodule",
          attr shape1 of vsm "statmodule";
          putting result in sys attr delay of this do;
      multiply sys attr delay of this do by 10;
      engageIn delaying for sys attr delay of this do msecs;

   --*************************************************
   --* Generate the next arrival to lane 8
   --*************************************************
      create rdo belonging to class lane8Vehicle putting its id in
         dynObjId starting in sm "lane8Source";

   --*************************************************
   --* Setup Initial Vehicle Attributes: Lane, Turning
   --*************************************************
     display "8" in position 1;
     put 8 into attr lane of this do;
     put TRUE@ into attr right of this do;

   --*************************************************
   --* Work Lane 8 Arrival and Entry to intersection
   --*************************************************
     move into sm "lane8";
     add 1 to attr numberOfCarsWaiting of sm "lane8";
     move! into inner sm "lane8Queue";
     move! to so "lane8Space";
     subtract 1 from attr numberOfCarsWaiting of sm "lane8";

   --*************************************************
   --* Now the vehicle is in the intersection...
   --* Transit the lane
   --*************************************************
     move into outer sm "blockA";
     set image of sm "blockA" to "inA";
     move! into inner sm "blockAQueue";
     move! to so "blockASpace";
     set image of sm "blockA" to "outA";
     move into outer sm "blockK";
     set image of sm "blockK" to "inK";
     move! into inner sm "blockKQueue";
     move! to so "blockKSpace";
     set image of sm "blockK" to "outK";
     move into outer sm "bypassSink";

   --*************************************************
   --* Depart the system
   --*************************************************
      destroy
   end
```

Figure 7.55 LANE8VEHICLE Class Self Logic (Traffic Intersection)

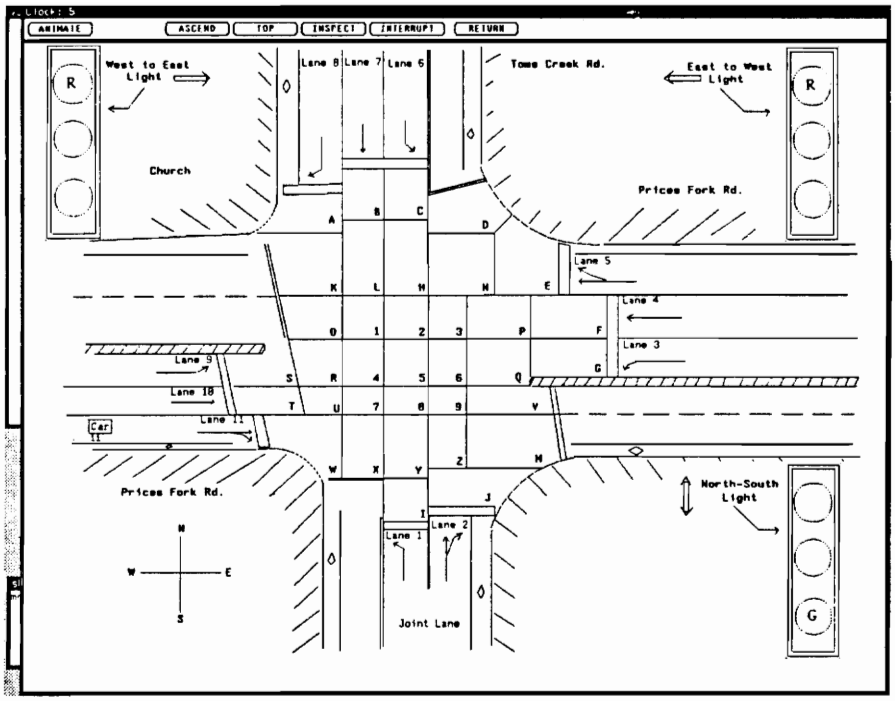

Figure 7.56 Vehicles Arriving to Lane 11

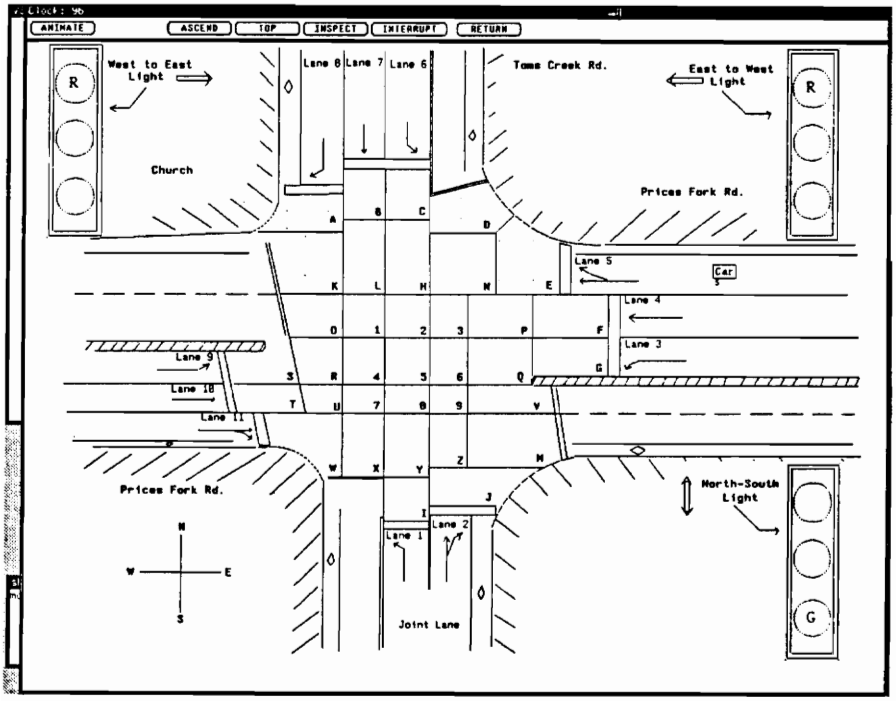

Figure 7.57 Vehicles Arriving to Lane 5

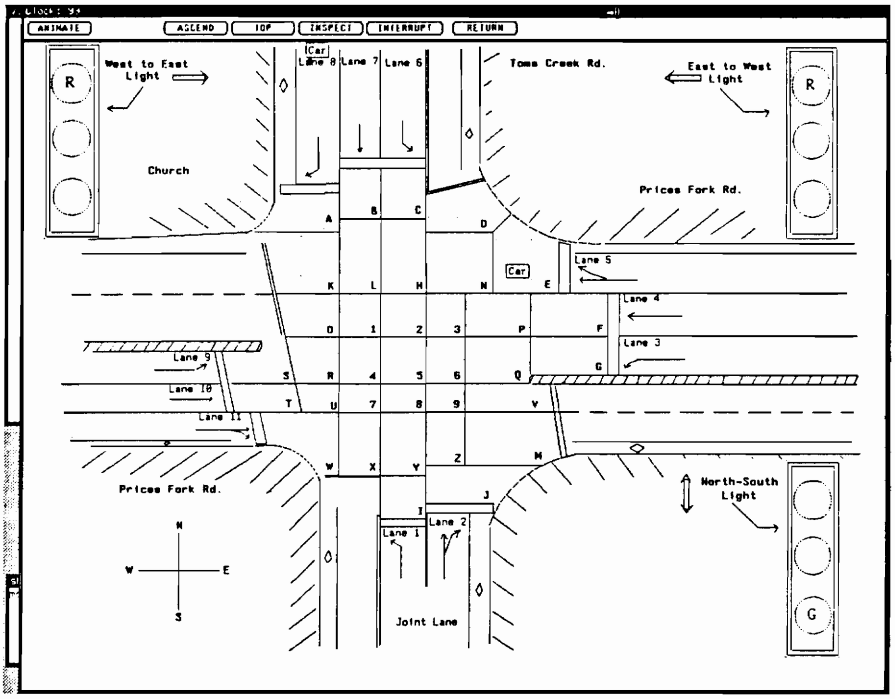

Figure 7.58 Vehicle Entering Intersection from Lane 5

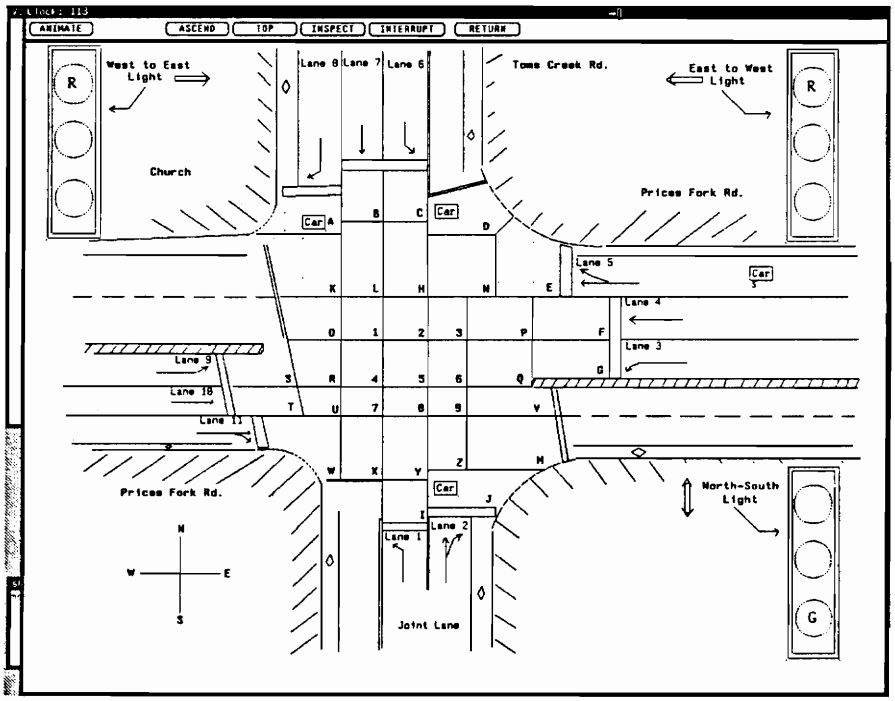

Figure 7.59 Vehicle Turning Right from Lane 5

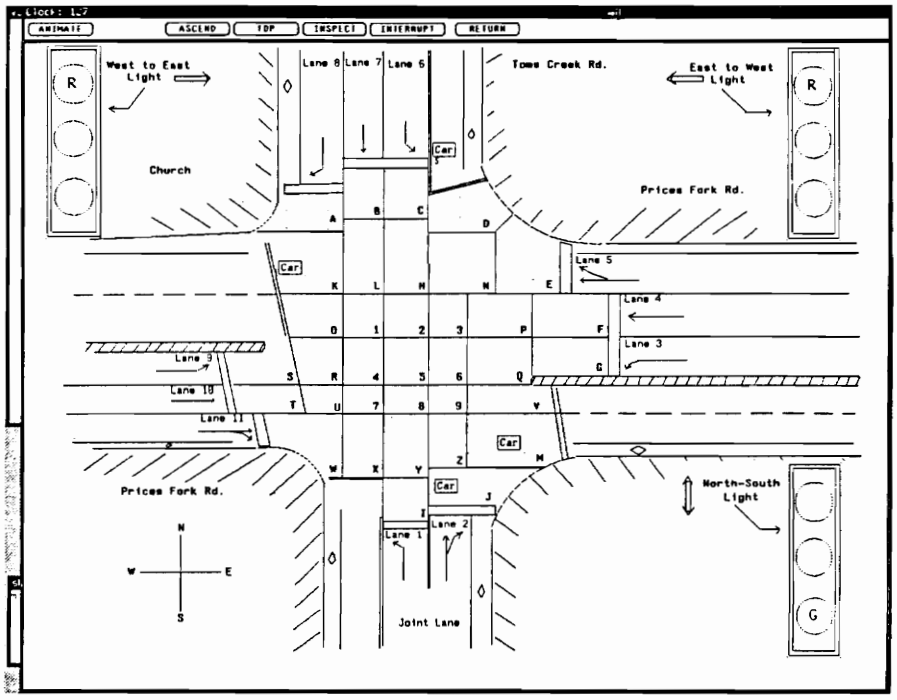

Figure 7.60 Vehicle (Right Turner from Lane 5) Exiting Intersection

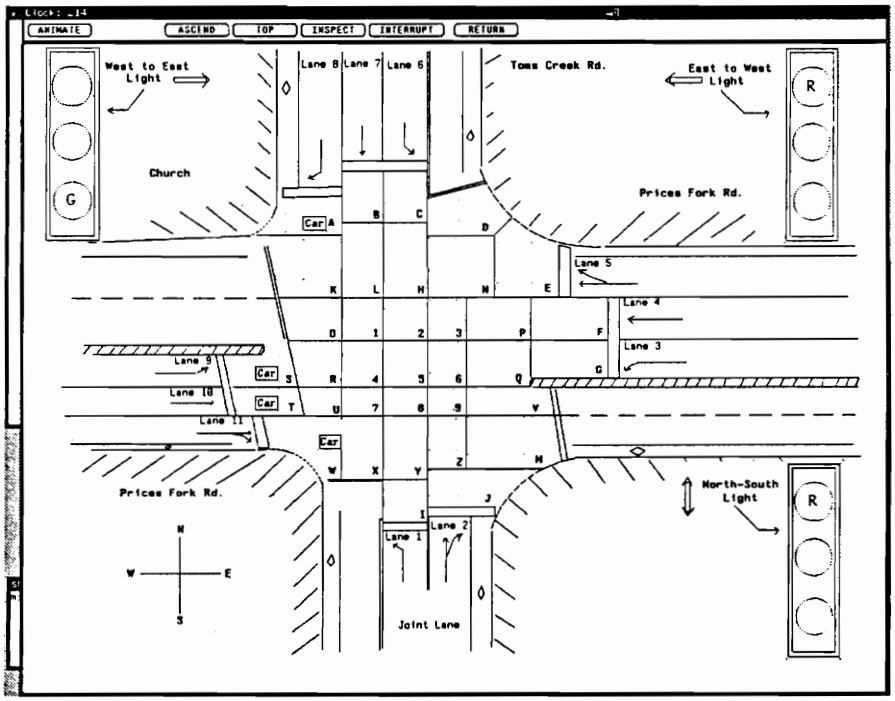

Figure 7.61 Vehicles Entering Intersection from Lanes 9, 10, and 11

the light static object image again demonstrate the utility and use of changing a component's image. Cars are arriving to lane 11 in Figure 7.56 and must wait for the red light. Between Figures 7.57 and 7.60, a vehicle in lane 5 makes a right turn on red. In Figure 7.61, the west-east light has just turned green and vehicles from lanes 9, 10, and 11 are released into the intersection.

7.3 Branch Operations Example

The Branch Operations is another real world example with previous research for comparison. The Branch Operations models a networking system being studied by a large multinational computer company. The example contains many points which are presented in order to provide further coverage of DOMINO applications (to include the application of the specification language constructs). This section, in addition, explains the benefits provided by compositional equivalence to design and implementation. These benefits are more pronounced when making comparisons with the development of this example model under earlier prototypes of the VSSE. The snapshots of execution, contained in the descriptions of the previous two examples, are included in Chapter 4 for this example where they are used to explain the runtime features of the VSSE. The Branch Operations can be outlined as follows:

> In this system, there are four branch offices over the continental United States (See ahead to Figure 7.66). These branch offices communicate with two host computers over a network. The hosts are located in Bethesda, Maryland, and Lexington, Kentucky. Each branch office contains two programmable workstations (PWSs), a LAN (Local Area Network) Server, and a LAN queue. Software for branch office applications resides on the LAN Servers in each branch with the data residing on the host computers. Each host (at the lowest decomposition) contains a host queue, the host computer itself, and a data disk. Messages from users (which activate different application functions) at the branch offices are represented as dynamic objects which travel over the network. The classes of dynamic objects are based on the different types of application functions.

The Branch Operations can be used to study network characteristics. A modeler can also choose to move the applications software to various locations in the system configuration to study effects. Regardless of the study objectives, this example includes several unique features which we discuss: the application of the same component logic specification to different layouts (using compositional equivalence) and a deeper nesting than previous examples.

7.3.1 Classes and Attributes

The Branch Operations contains the following modeler-defined classes: seven dynamic object classes (**basicTransaction, adminTool, book, orderMgt, customer, billing,** and **billProc**), and eleven submodel classes (**queue, lanQueue, hostQueue, facility, invFacility, centralFacility, branch, pws, lanSvr, host,** and **datadisk**). The class inheritance hierarchy of the dynamic object classes is shown in Figure 7.62. Figures 7.63 and 7.64 give the submodel class hierarchies for the **queue** and **facility** submodel classes, respectively. Similar to the Traffic Intersection, the Branch Operations has no decomposed dynamic objects and, therefore, has no subdynamic or base dynamic classes. A detailed accounting of classes (including the model class) and accompanying attributes follows.

Class Name	Attribute	Attribute Description
Model Class		
transact	**i**	Counting attribute
	dynObjId	Holder for dynamic object identifiers
	holdingVar	Temporary holding variable
	SCREENREQ	Constant for dyn object context: screen request
	SCREENFMT	Constant for dyn object context: screen format
	ENTRY	Constant for dyn object context: user entry
	AWAIT	Constant for dyn object dBcode and dBstatus: initial value
	READ	Constant for dyn object dBstatus: reading data from disk
	WRITE	Constant for dyn object dBstatus: writing data to disk
	CS_SCRNREQ	Constant for dyn object context: customer search screen request
	CS_SCRNFMT	Constant for dyn object context: customer search screen format
	CUSTVALID	Constant for dyn object dBcode: customer search
	CUSTSRCH	Constant for dyn object dBcode: customer validation
	DB_REQ	Constant for dyn object context: database request
	DB_RETN	Constant for dyn object context: database return
	SEARCH_ENTRY	Constant for dyn object context: customer search entry
	FINISH	Constant for dyn object context: last, finishing context
	INV_REQ	Constant for dyn object context: invoice request
	INV_RETN	Constant for dyn object context: invoice return
	SYSTEM	Constant for dyn object dBcode: system
	PONUM	Constant for dyn object dBcode: plant order number
	COMMIT	Constant for dyn object dBcode: commit

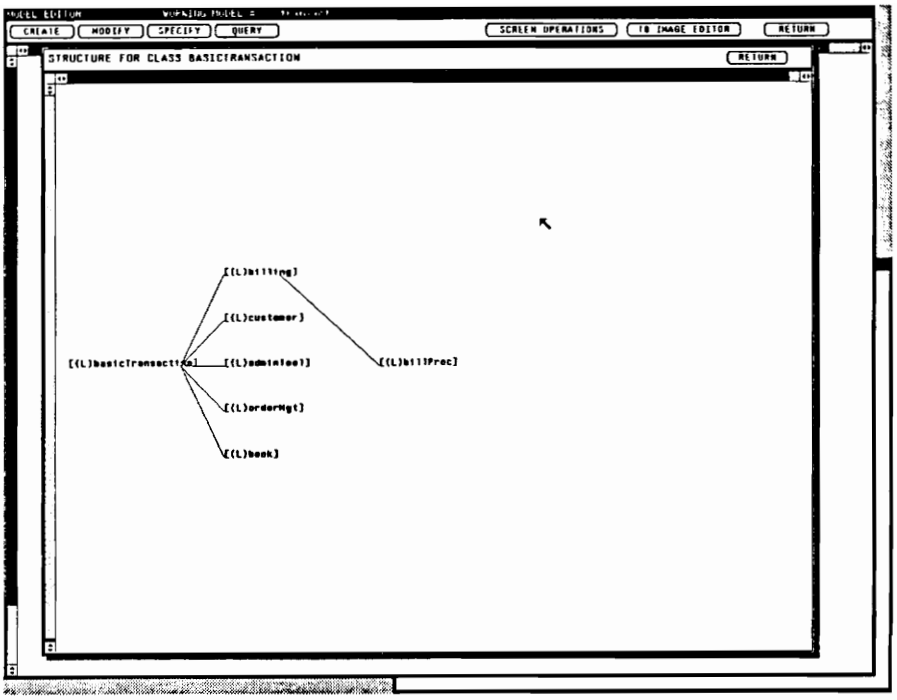

Figure 7.62 BASICTRANSACTION Class Hierarchy (Branch Operations)

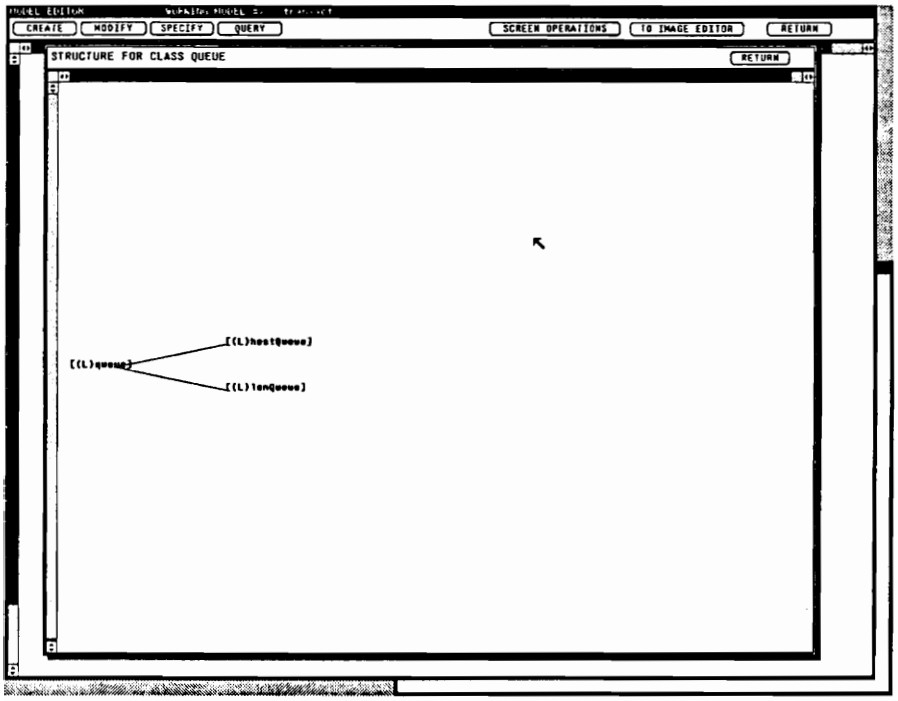

Figure 7.63 QUEUE Class Hierarchy (Branch Operations)

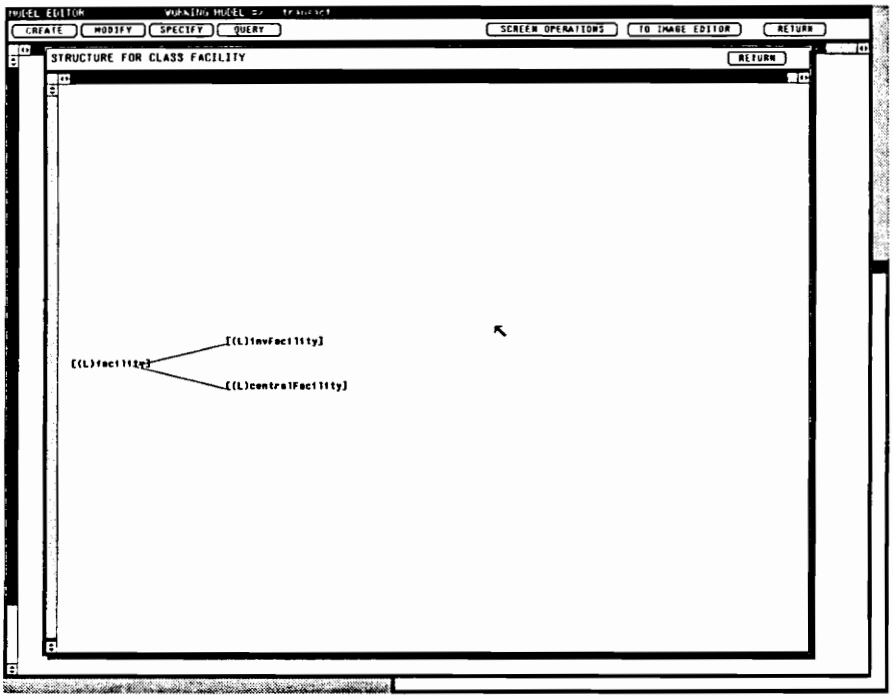

Figure 7.64 FACILITY Class Hierarchy (Branch Operations)

Dynamic Object Class(es)

basicTransaction	**context**	Holder of context codes of dynamic objects (above)
	dBcode	Holder of database codes of dynamic objects (above)
	dBstatus	Holder of database status codes of dyn objects: READ, WRITE
	originBranch	Identification of the origin branch of dyn obj
	originTerminal	Identification of the origin terminal
	mySeed	Seed value carried by dynamic object
adminTool	No attributes	
book	No attributes	
orderMgt	No attributes	
customer	**valid**	validation status: TRUE, FALSE
billing	**probSelect**	Selection probability for decision choices
billProc	**plantOrdNum**	Plant order number
	sysNum	System number
	commitBill	Commitment to billing

Submodel Class(es)

queue	**numberWaiting**	Number of dynamic objects waiting in queue
	waitingList	Single dimension array holding id's objects in queue
lanQueue	No attributes	
hostQueue	No attributes	
facility	No attributes	
invFacility	No attributes	
centralFacility	No attributes	
branch	No attributes	
pws	**ltravel**	Travel time from pws to lanServer
	stime	Time to make selection
	orderMgtDone	Boolean indicating orderMgt processing is done
	bookingDone	Boolean indicating booking processing is done
	customerDone	Boolean indicating customer processing is done
	etime	Time to make entry
	billingDone	Boolean indicating billing is done
	billProcDone	Boolean indicating bill processing is done
lanSvr	**status**	Status of lanServer: BUSY, IDLE
	ptravel	Travel time from lanServer to pws
	htravel	Travel time from lanServer to host
	dectime	Decision time
	prtime	Processing time
host	**status**	Status of host: BUSY, IDLE
	htravel	Travel time between hosts
	ltravel	Travel time from host to lanServer
	dectime	Decision time
datadisk	**wtime**	Time to write to disk
	rtime	Time to read from disk

The discussion of Section 7.3.3 concerning logic specifications amplifies the class and attribute listing and gives a better understanding of the purpose for certain attributes.

7.3.2 Model Structure and Graphical Elements

The model static structure line drawing from VSSE is given in Figure 7.65. Inspection of the figure reveals the initial decomposition to four branches and the host computers. Each branch (e.g., **branch2**) is further decomposed into two **pws** submodel class components, a LAN server, and a LAN queue. One of the host computer decompositions can be seen to the right: a queue for the host, the host itself, and a data disk. To augment this figure, the listing of model component instances (as before, literal names and owning classes) are provided.

Owning Class	Literal Instance(s)
queue	No direct instances
lanQueue	**lanQueue1...lanQueue4**
hostQueue	**bethesdaQueue, lexQueue**
facility	**lexingtonComplex**
invFacility	**lexington**
centralFacility	**bethesda**
branch	**branch1...branch4**
pws	**pws1_1,pws1_2...pws4_1,pws4_2**
lanSvr	**lanSvr1...lanSvr4**
host	**bethesdaHost, lexHost**
datadisk	**bethesdaDisk, lexDisk**

The Branch Operations has a top level layout associated with the decomposition of the root of the model static structure. Figure 7.66 is that layout whose definition is also shown in Figure 7.67. Each branch is decomposed and has a layout (Figures 7.68 through 7.71). Note that these figures represent the definitions and that the component, path, and connector details (lines) are not shown during animation. The **bethesda** central facility is decomposed as shown in the layout definition of Figure 7.72 containing the **bethesdaQueue, bethesdaHost,** and the **bethesdaDisk. Lexington,** the **invFacility** class submodel instance, is first decomposed into the **lexingtonComplex** with the associated layout definition of Figure 7.73. This complex is further decomposed and contains the layout definition of Figure 7.74 (**lexQueue, lexHost, lexDisk**). The nesting through the Lexington host distinguishes this model example from the rest since it is three levels deep.

The images for the various dynamic object images (representing the various user application request

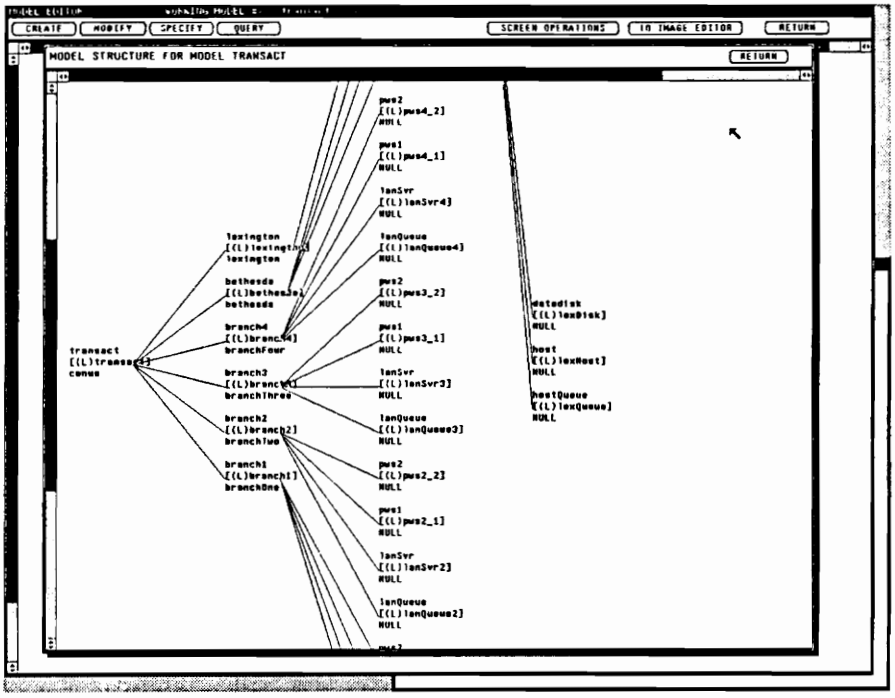

Figure 7.65 Model Static Structure (Branch Operations)

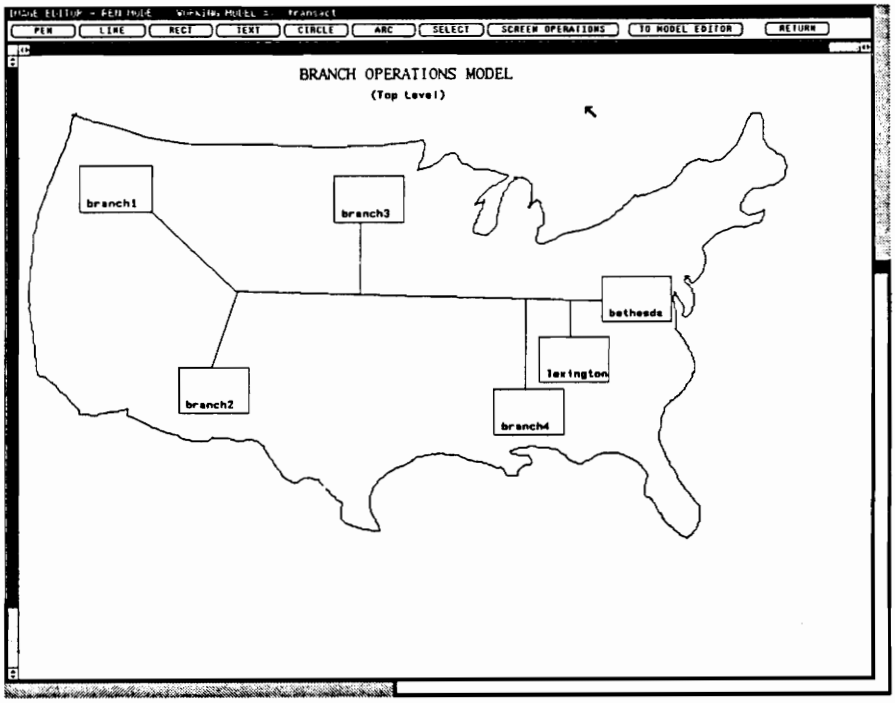

Figure 7.66 Top Level Layout Image (Branch Operations)

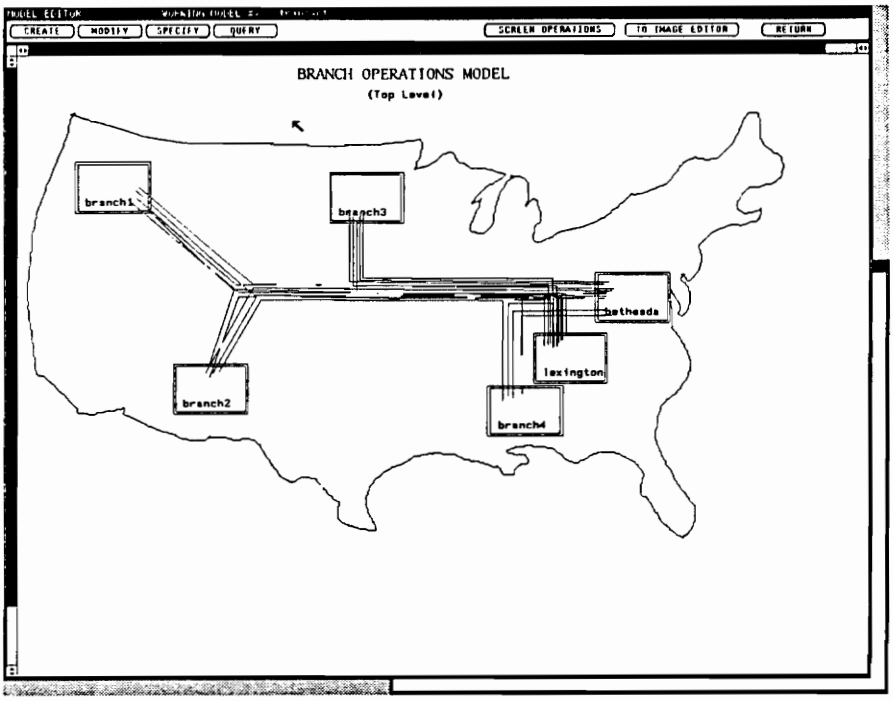

Figure 7.67 Top Level Layout Definition (Branch Operations)

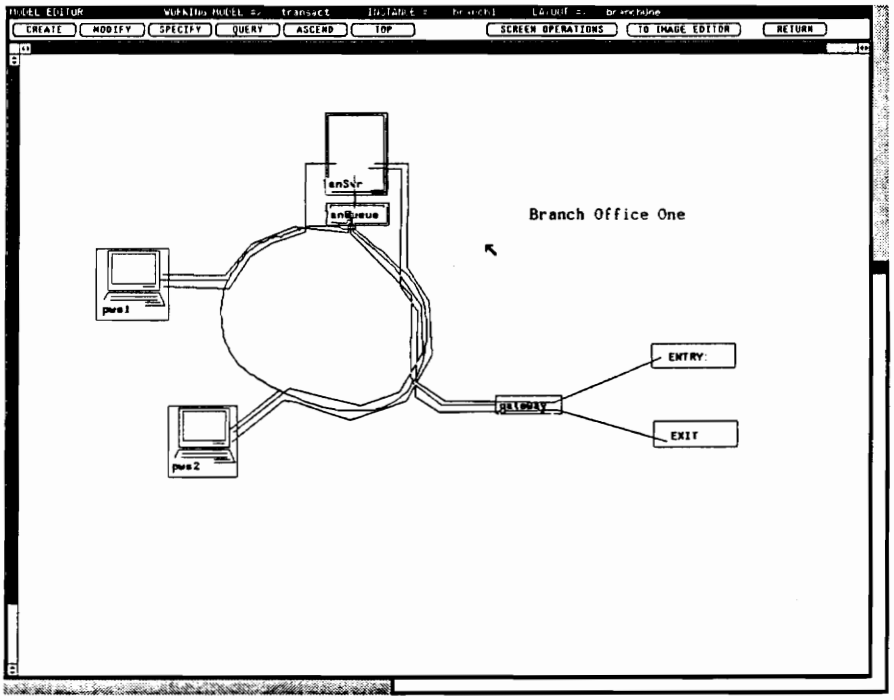

Figure 7.68 Layout Definition of Branch One (Branch Operations)

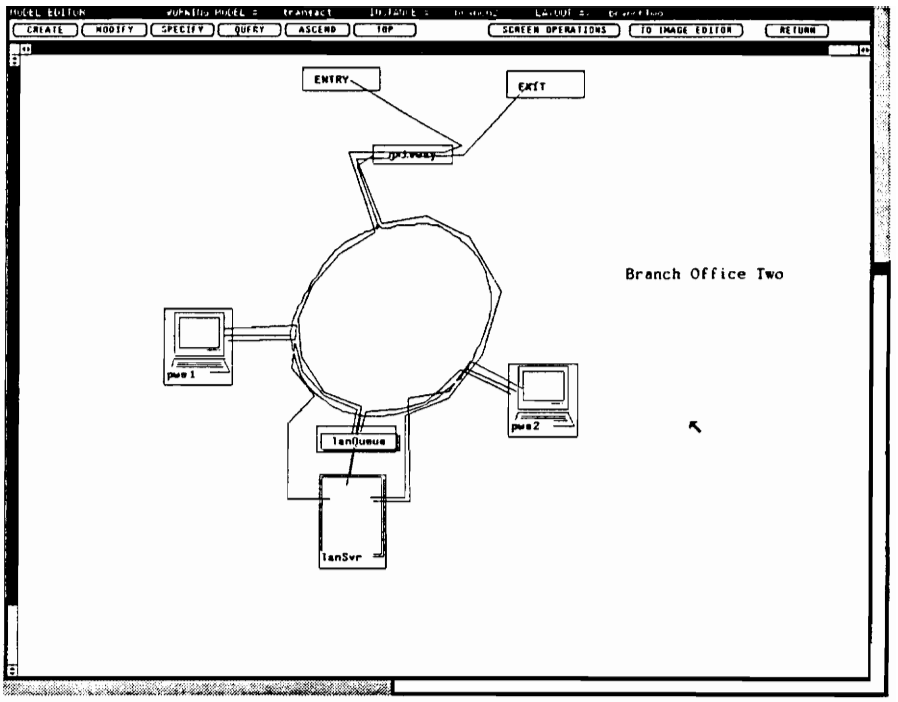

Figure 7.69 Layout Definition of Branch Two (Branch Operations)

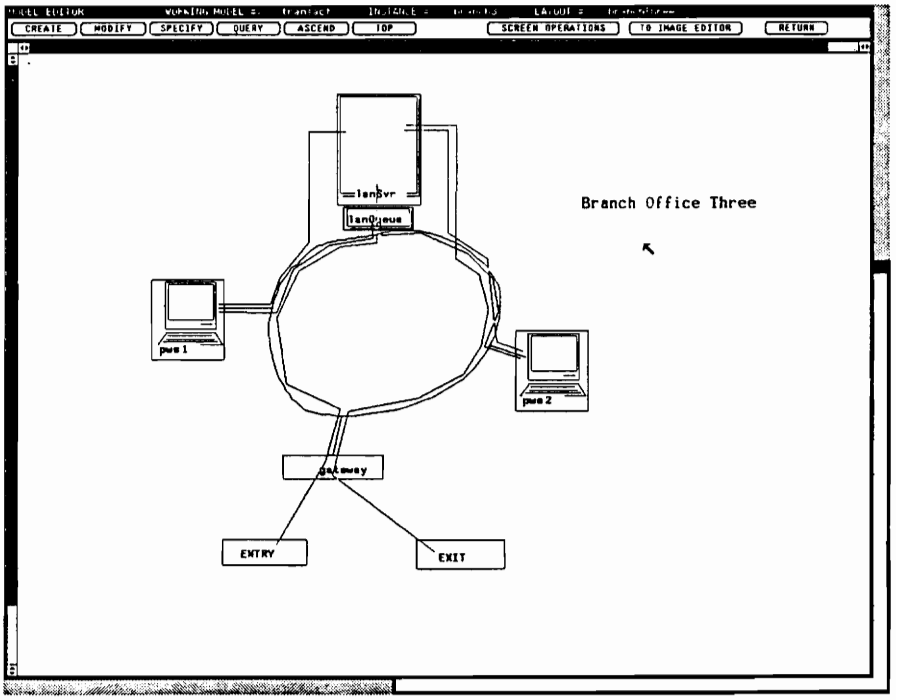

Figure 7.70 Layout Definition of Branch Three (Branch Operations)

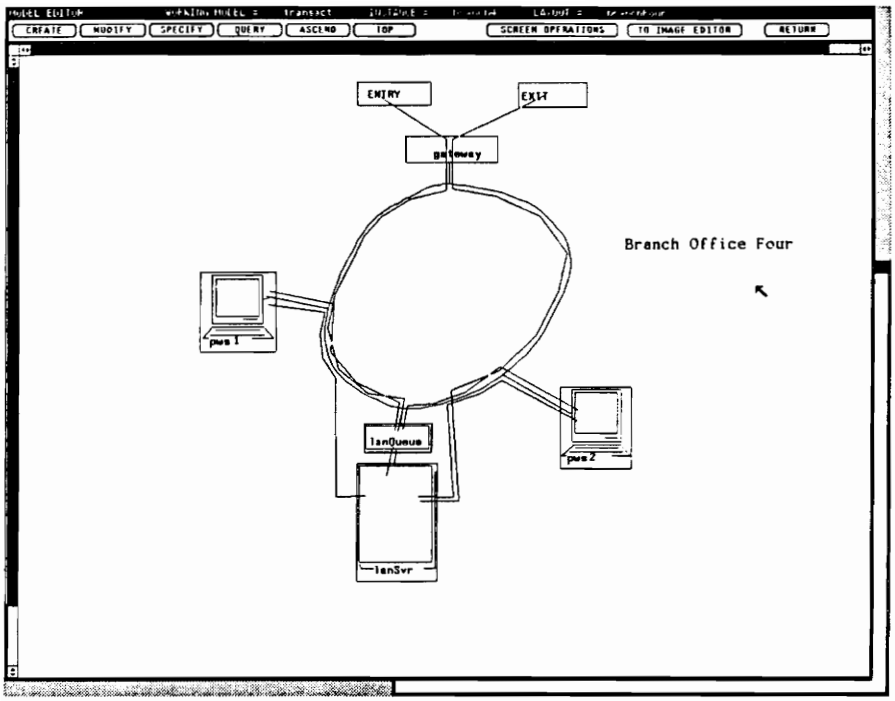

Figure 7.71 Layout Definition of Branch Four (Branch Operations)

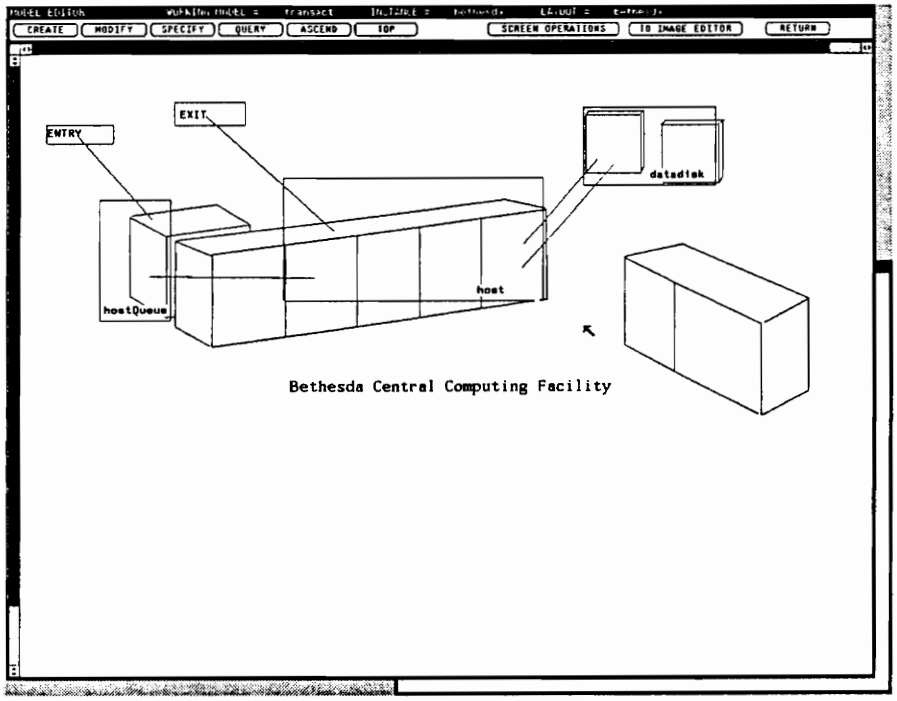

Figure 7.72 Layout Definition of Bethesda (Branch Operations)

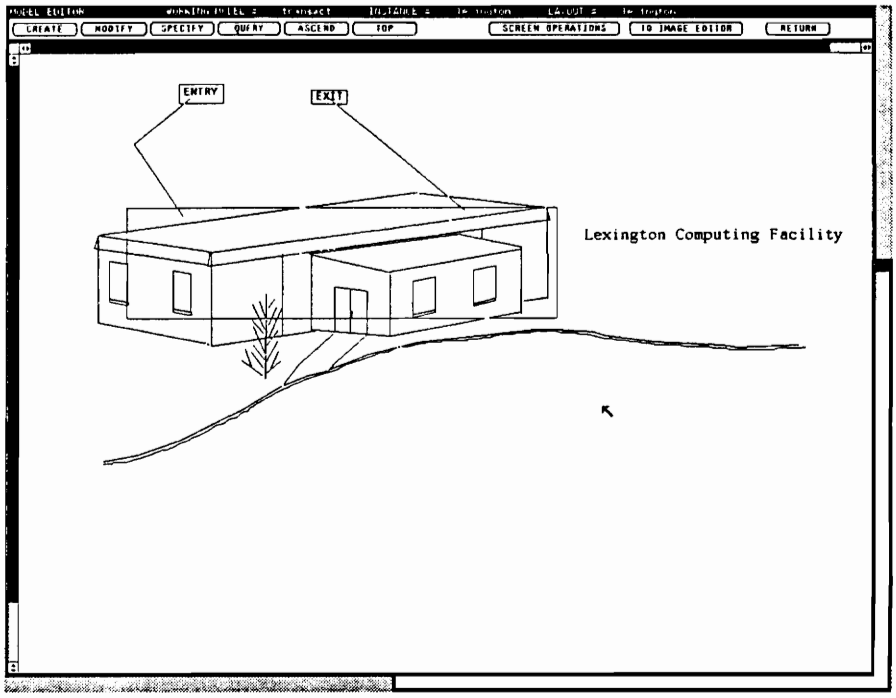

Figure 7.73 Layout Definition of Lexington (Branch Operations)

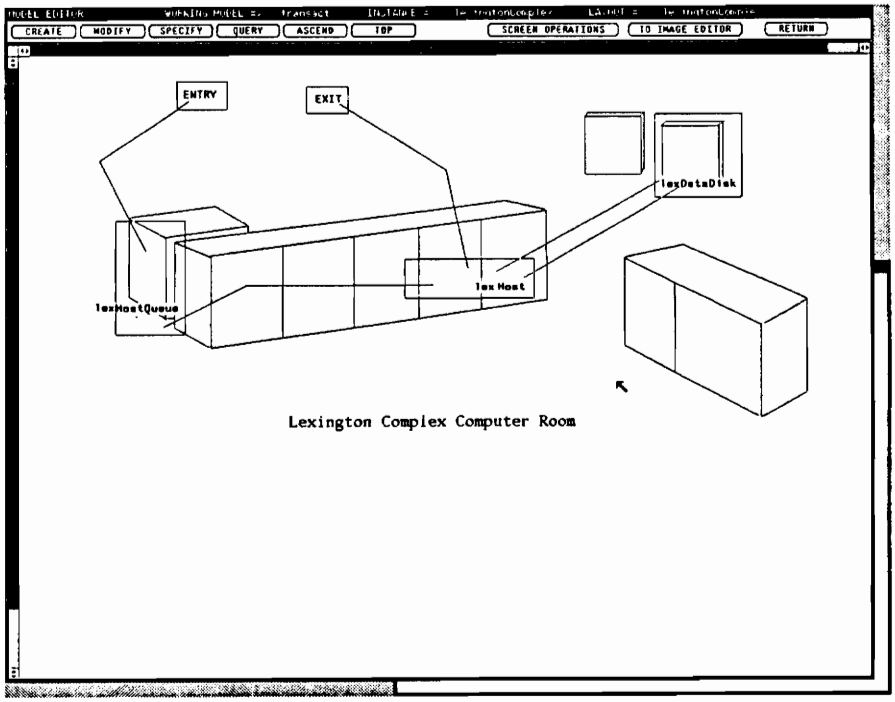

Figure 7.74 Layout Definition of Lexington Computer Room (Branch Operations)

functions from each pws class instance) are displayed in Figure 7.75. The AdminTool is associated with the top level applications menu screen on the pws. OrderMgt is the order management function. Booking, Customer, and Billing are included. BillProc is the bill processing function.

7.3.3 Component Logic Specifications

The Branch Operations uses a machine-oriented perspective for model component logic specifications. Only supervisory logic is used. Thus, no self logic is associated with any dynamic object class. In the next section, a general overview of the logic specification and dynamic object movement is given. Then, each supervisory logic specification is covered briefly.

7.3.3.1 Overview of Logic Specifications

As previously mentioned, the dynamic objects represent a user request for some application function. The administrative tool (**adminTool**) object represents facilities associated with the top level menu of the applications software. From the top level, several icons can be selected which will cause additional dynamic object instances to be created for software application functions: order management (**orderMgt**), booking (**book**), billing (**billing**), bill processing (**billProc**), or customer search and validation (**customer**). The administrative tool function is the first. Each function (dynamic object) context is a basic request for a formatted screen, a screen request. This request is made to the LAN Server and a formatted screen is returned to the user. Depending upon the nature of the request to the LAN Server, the appropriate information is returned to the programmable work station (pws). All requests to the LAN Server are queued. If the requested application function requires data from the host computers, the LAN Server will pass the request (perhaps modified) over the network to the appropriate host computer. All requests to the host computers are queued, like requests to LAN Servers. Data disks are eventually accessed and the request is returned to the LAN Server from which it came, and eventually returned to the originating work station.

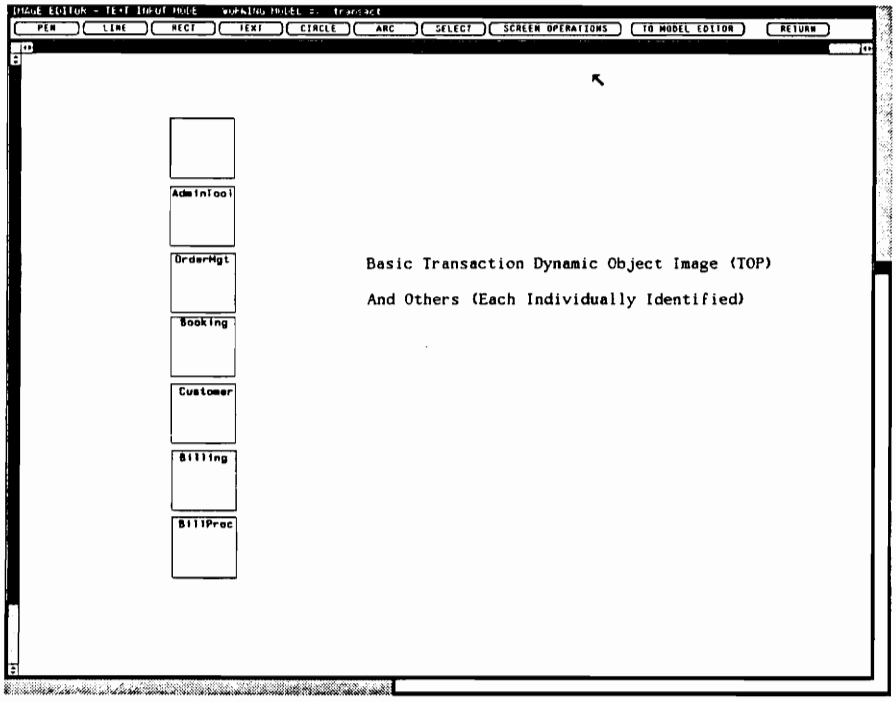

Figure 7.75 Dynamic Object Images (Branch Operations)

7.3.3.2 Supervisory Logic Specifications

The builtin class supervisory logic (Figure 7.76) creates the dynamic object instances which represent the initial user function requests at the programmable work stations. For simplicity, only the initial setup for the branch one and two work stations is shown. The created dynamic objects are directed to start in the appropriate work stations. The origin (branch and terminal) attributes and seeds for the dynamic objects are initialized as well. The images of the dynamic objects are assigned. For the demonstration purposes, this creation and initialization of dynamic objects is staggered for each branch by a constant time interval.

The dynamic object instances get their start in the pws class supervisory logic (Figure 7.77). A different logic path is needed depending on the type of dynamic object. The **branch** statement permits the branching of the classId (or class type) of the dynamic object and accomplishes this division of the logic. Again for simplification of the explanation, only the **adminTool** dynamic object class logic is shown. The initial context of each created dynamic object is for a screen request (i.e., **SCREENREQ**). Therefore, upon entry to the **pws** supervisory logic, the **adminTool** dynamic object instance, is prepared for travel to its local LAN Server. The time the object reaches the queue for the LAN Server is the sum of the time for the user to select (attribute **stime**) and request a formatted screen and the time it takes to travel (attribute **ltravel**) to the LAN queue. The execution is delayed that amount of time and moved to the local LAN queue after the delay. Animated movements are instantaneous; the delays in moving are taken into account before the move is performed. Also, notice that before movement, the dynamic object image is modified to contain the current **context** attribute value with the label "Context".

An important feature is highlighted with the **move** statement for the move to the LAN queue. As a local move, the destination location for the move is specified using the variable name of the destination component in the layout (e.g., **sm lanQueue**). For this reason, although the four branches are different in appearance, due to *compositional equivalence*, the same supervisory logic applies to each **pws** submodel class instance in all of the different branch offices. This is a major advance over previous prototypes which required different logic for each **pws** (and also **lanSvr**) class instances. Due to the eight **pws** class

```
-- BUILTIN SUBMODEL CLASS SUPERVISORY LOGIC
begin
--  Create adminTool dynobj for each terminal and stagger the creation.

   assert FALSE;

--  FOR TERMINAL PWS1_1
   create rdo belonging to class adminTool putting its id in dynObjId
      starting in sm "pws1_1";
   put 1 into attr originBranch of do dynObjId;
   put 1 into attr originTerminal of do dynObjId;
   put attr iseed1 of vsm "statmodule" into attr mySeed of do dynObjId;
   set image of do dynObjId to "adminTool";
-- Delay for 25 time ticks and create do for terminal in other branch
   put 25 into sys attr delay of this do;
   engageIn delaying for sys attr delay of this do secs;

--  FOR TERMINAL PWS2_1
   create rdo belonging to class adminTool putting its id in dynObjId
      starting in sm "pws2_1";
   put 2 into attr originBranch of do dynObjId;
   put 1 into attr originTerminal of do dynObjId;
   put attr iseed2 of vsm "statmodule" into attr mySeed of do dynObjId;
   set image of do dynObjId to "adminTool";
-- Stagger the next creation
   put 25 into sys attr delay of this do;
   engageIn delaying for sys attr delay of this do secs;
...

--  FOR TERMINAL PWS1_2
   create rdo belonging to class adminTool putting its id in dynObjId
      starting in sm "pws1_2";
   put 1 into attr originBranch of do dynObjId;
   put 2 into attr originTerminal of do dynObjId;
   put attr iseed1 of vsm "statmodule" into attr mySeed of do dynObjId;
   set image of do dynObjId to "adminTool";
   put 25 into sys attr delay of this do;
   engageIn delaying for sys attr delay of this do secs;

--  FOR TERMINAL PWS2_2
   create rdo belonging to class adminTool putting its id in dynObjId
      starting in sm "pws2_2";
   put 2 into attr originBranch of do dynObjId;
   put 2 into attr originTerminal of do dynObjId;
   put attr iseed2 of vsm "statmodule" into attr mySeed of do dynObjId;
   set image of do dynObjId to "adminTool";
   put 25 into sys attr delay of this do;
   engageIn delaying for sys attr delay of this do secs;
...

   destroy

end
```

Figure 7.76 BUILTIN Class Supervisory Logic (Branch Operations)

```
-- PWS SUBMODEL CLASS SUPERVISORY LOGIC
begin
   -- branch on class of do
   branch sys attr classId of this do
   (adminTool)
   begin
      if attr context of this do is SCREENREQ@ then
        begin
           put attr stime of this sm + attr ltravel of this sm into
              sys attr delay of this do;
           -- Wait for travel time since objects move instantaneously
           engageIn traveling for sys attr delay of this do secs;
           display attr context of this do with label "Context"
              in position 2;
           move into sm lanQueue
        end
      else if message rand1 to vsm "statmodule" using
        (attr mySeed of this do) <= 0.43 then
        -- 43% of time,i.e. 55 orders/br/da,18 alterations/br/da = 73 order jobs
        begin -- select ORDERMGT icon
          if attr originTerminal of this do is ONE@ then
            create rdo belonging to class orderMgt putting its id in dynObjId
               starting in local sm pws1
          else
            create rdo belonging to class orderMgt putting its id in dynObjId
               starting in local sm pws2;
          put attr mySeed of this do into attr mySeed of do dynObjId;
          put attr originTerminal of this do into attr originTerminal of
             do dynObjId;
          put attr originBranch of this do into attr originBranch of
             do dynObjId;
          set image of do dynObjId to "orderMgt";
          engageIn waiting until attr orderMgtDone of this sm is TRUE;
          -- Start over, at the top of sm logic
          put TOP@ into sys attr currentSuperLoc of this do;
          put SCREENREQ@ into attr context of this do;
          put FALSE@ into attr orderMgtDone of this sm
        end
      else
        begin -- Select Billing Icon, 100 billing jobs/br/day
        -- 15 onDemand, 30 pending, 10 adjustments, 45 misc
          if attr originTerminal of this do is ONE@ then
            create rdo belonging to class billing putting its id in dynObjId
               starting in local sm pws1
          else
            create rdo belonging to class billing putting its id in dynObjId
               starting in local sm pws2;
          put attr mySeed of this do into attr mySeed of do dynObjId;
          put attr originTerminal of this do into attr originTerminal of
             do dynObjId;
          put attr originBranch of this do into attr originBranch of
             do dynObjId;
          set image of do dynObjId to "billing";
          engageIn waiting until attr billingDone of this sm is TRUE;
          -- Start over, at the top of sm logic
          put TOP@ into sys attr currentSuperLoc of this do;
          put SCREENREQ@ into attr context of this do;
          put FALSE@ into attr billingDone of this sm
        end
   end;  -- adminTool
   ...
   end -- branch
end
```

Figure 7.77 PWS Class Supervisory Logic (Branch Operations)

instances alone, this amounts to a logic specification reduction (in length or amount of code) of a factor of eight. Before leaving the discussion of this logic, further clarification of the dynamic object activity is necessary. The **adminTool** dynamic objects return to the pws logic with contexts other than **SCREENREQ**. The **if...else if...else** divisions in the logic support this. Once the requested screen is returned from the LAN Server, the user is able to select the order management application function or the billing application function. This selection is determined by a probability selection using the random variate generator of the virtual statistics submodel **"statmodule"**. Once the user selection is made, the new dynamic object representing the selection application function is spawned (created and initialized) and the **adminTool** dynamic object is essentially suspended until its child process (or application function) is completed.

The next destination of a dynamic object is the appropriate LAN queue. The lanQueue class supervisory logic is given in Figure 7.78 and follows the same general logic of the queue logic in previous examples. Once waiting in the queue is completed (or bypassed), the dynamic object is directed on to the local LAN Server of the branch office.

In the LAN Server (See the **lanSvr** class supervisory logic of Figure 7.79), the application function request is handled by the server. First, the LAN Server is set to the BUSY status. For example, the screen request of the **adminTool** dynamic object is processed and the context of the dynamic object is changed to a formatted screen. The delay time for the returning dynamic object (with formatted screen) from the LAN Server to the local pws is the sum of the travel time to the pws (attribute **ptravel**) and the processing time (attribute **prtime**) at the LAN Server. The travel time delay is accomplished with the **engageIn travelling** statement. Just prior to the move back to the local pws, the dynamic object image context information is updated and then the move is accomplished. Notice the move depends upon the origin terminal location of the dynamic object. The LAN Server status is set to IDLE before departure. In addition, notice the use of the variable names as the movement destination locations. Because there are four LAN Servers, another factor of four reduction in LAN Server supervisory logic code is accomplished over previous prototypes. A customer dynamic object instance can be sent over the network to a host

```
-- LANQUEUE SUBMODEL CLASS SUPERVISORY LOGIC
-- Make next move based on entrance conditions to the lanSvr
-- Wait in the lan queue if the lanSvr is BUSY or others are already waiting
if attr status of sm lanSvr is BUSY@
   or attr numberWaiting of this sm is > 0 then
      begin
         add 1 to attr numberWaiting of this sm;
         put attr numberWaiting of this sm into holdingVar;
         put sys attr ident of this do into attr waitingList[holdingVar] of
           this sm;
         -- The object must wait until the lanSvr is not BUSY and
         -- the do is first in line.
         engageIn waiting until attr status of sm lanSvr is IDLE@ and attr
            waitingList[1] of this sm is sys attr ident of this do;
         -- With waiting over, decrement the number waiting and make the move
         -- Now update waiting identifier list

         put attr numberWaiting of this sm into holdingVar;
         subtract 1 from holdingVar;
         put 0 into attr waitingList[1] of this sm;
  repeat with i = 1 to holdingVar
      begin
       put i+1 into dynObjId;
       put attr waitingList[dynObjId] of this sm into attr waitingList[i]
           of this sm;
       put 0 into attr waitingList[dynObjId] of this sm
     end
  end;
       -- Now update the numberWaiting
       subtract 1 from attr numberWaiting of this sm;
       move into local sm lanSvr
  end
else
   -- The conditions are right for immediately moving into the lanSvr.
   move into local sm lanSvr
```

Figure 7.78 LANQUEUE Class Supervisory Logic (Branch Operations)

```
-- LANSVR SUBMODEL CLASS SUPERVISORY LOGIC
begin

   put BUSY@ into attr status of this sm;

   branch sys attr classId of this do

   (adminTool);
   (orderMgt);
   (book);
   (billing)
       begin
          put SCREENFMT@ into attr context of this do;
          put attr ptravel of this sm + attr prtime of this sm into
            sys attr delay of this do
       end ;
   (billProc)
       begin
       ...
       end;
   (customer)
       ...
   end; -- branch

   engageIn traveling for sys attr delay of this do secs;

   put IDLE@ into attr status of this sm;

   branch sys attr classId of this do

   (adminTool);
   (orderMgt);
   (book);
   (billing)
      begin
      display attr context of this do with label "Context" in position 2;
      if attr originTerminal of this do is 1 then
         move into local sm pws1;
      if attr originTerminal of this do is 2 then
         move into local sm pws2
      end;
   (billProc)
      begin
      ...
      end;
   (customer)
      begin
      display attr context of this do with label "Context" in position 2;
      display attr dBcode of this do with label "dBCode" in position 3;
      if attr context of this do is DB_REQ@ then
         move into outer sm "bethesda"
      else
         begin
            if attr originTerminal of this do is ONE@ then
               move into local sm pws1
            else
               move into local sm pws2
         end
      end
   end
end
```

Figure 7.79 LANSVR Class Supervisory Logic (Branch Operations)

computer to retrieve data instead of being returned to the local and originating pws. This direction is exemplified in the move statement which specifies the move to the outer submodel **"bethesda"**, a **centralFacility** class instance.

Figure 7.80 is the centralFacility class supervisory logic. Once directed to the Bethesda facility, the dynamic object is then directed downward in the decomposition hierarchy to the inner host queue for the host computer at the Bethesda installation. Similarly, direction of movements to the Lexington installation shift logic control to the invFacility class supervisory logic (Figure 7.81) also pass the entering dynamic object into the next decomposition level with a move to the inner component destination.

The hostQueue class supervisory logic (Figure 7.82) handles the queueing of dynamic objects (the application functions) to the host computers. Function requests to the host computers are either requests to read data from the resident disk or to write data. The queue logic is as before. The only difference is that once waiting in the queue is completed or bypassed, the next directed move of the dynamic object is to a local host.

The host class supervisory logic (Figure 7.83) handles the access to disk data. Logic is subdivided depending upon which instance of the host class applies (whether the **bethesdaHost** or the **lexHost**). If the dynamic object context is a database request (e.g., **DB_REQ** - from Bethesda, or **INV_REQ** - from Lexington), the dynamic object is directed to the appropriate disk. The **move** statement at the bottom of the figure indicates this. If the data has been retrieved, the context is a database return (**DB_RETN** or **INV_RETN**). For the Lexington host, note the host is returned to IDLE status and sent upward in the hierarchy with the outer move.

The specific access operations at the disk are accomplished with the datadisk class supervisory logic, an example of which is shown in Figure 7.84. The time spent in access is dependent upon whether the database status code dictates a read or write operation. The context of the dynamic object is changed to the appropriate database return context and the context attribute display is updated. After accomplishing the complete delay, the dynamic object is returned to the associated host.

In returning upward from within a decomposition hierarchy, each directed move of a dynamic object

```
-- CENTRALFACILITY SUBMODEL CLASS SUPERVISORY LOGIC
move into inner sm hostQueue
```

Figure 7.80 CENTRALFACILITY Class Supervisory Logic (Branch Operations)

```
-- INVFACILITY SUBMODEL CLASS SUPERVISORY LOGIC
move into inner sm computerComplex
```

Figure 7.81 INVFACILITY Class Supervisory Logic (Branch Operations)

```
-- HOSTQUEUE SUBMODEL CLASS SUPERVISORY LOGIC
if attr status of sm host is BUSY@ or attr numberWaiting of this sm
   is > 0 then
      begin
         add 1 to attr numberWaiting of this sm;
         put attr numberWaiting of this sm into holdingVar;
         put sys attr ident of this do into attr waitingList[holdingVar] of
           this sm;
         -- The object must wait until the host is not BUSY and
         -- the do is first in line.
         engageIn waiting until attr status of sm host is IDLE@ and attr
            waitingList[1] of sm hostQueue is sys attr ident of this do;
         -- With waiting over, decrement the number waiting and make the move
         -- Now update waiting identifier list
         put attr numberWaiting of this sm into holdingVar;
         subtract 1 from holdingVar;
         put 0 into attr waitingList[1] of this sm;
  repeat with i = 1 to holdingVar
      begin
       put i+1 into dynObjId;
       put attr waitingList[dynObjId] of this sm into attr waitingList[i]
           of this sm;
       put 0 into attr waitingList[dynObjId] of this sm
     end
  end;
       -- Now update the numberWaiting
       subtract 1 from attr numberWaiting of this sm;
       move into local sm host
  end
else
   -- The conditions are right for immediately moving into the host.
   move into local sm host
```

Figure 7.82 HOSTQUEUE Class Supervisory Logic (Branch Operations)

```
-- HOST SUBMODEL CLASS SUPERVISORY LOGIC
begin
   put BUSY@ into attr status of this sm;

   branch sys attr compInst of this sm
   (BETHESDAHOST_SM_INST@)
      branch sys attr classId of this do
      (billProc)
      begin
         if attr context of this do is DB_REQ@ then
            begin
               put attr dectime of this sm into sys attr delay of this do;
               put READ@ into attr dBstatus of this do
            end
          else if attr context of this do is DB_RETN@ then
            begin
               if attr dBcode of this do is SYSTEM@ then
                  begin
                     put attr htravel of this sm into sys attr delay
                        of this do;
                     put INV_REQ@ into attr context of this do
                  end
               else
                  put attr ltravel of this sm into sys attr delay of this do
            end
          else
            put attr ltravel of this sm into sys attr delay of this do
      end;
      (customer)
      begin
      if attr context of this do is DB_REQ@ then
         begin
            put attr dectime of this sm into sys attr delay of this do;
            put READ@ into attr dBstatus of this do
         end
      else
         put attr ltravel of this sm into sys attr delay of this do
      end
      end; -- branch on classId
   (LEXHOST_SM_INST@)
      begin
      ...
      end
   end; -- branch on compInst

   engageIn traveling for sys attr delay of this do secs;

   branch sys attr compInst of this sm
   (BETHESDAHOST_SM_INST@)
      branch sys attr classId of this do
      ...
      end ;-- end branch on classId
   (LEXHOST_SM_INST@)
      begin
         display attr context of this do with label "Context" in position 2;
         if attr context of this do is INV_REQ@ then
             move into local sm datadisk
         else
            begin
               put IDLE@ into attr status of this sm;
               move into outer sm computerComplex
            end
      end
   end -- branch on compInst
end  -- end block
```

Figure 7.83 HOST Class Supervisory Logic (Branch Operations)

```
-- DATADISK SUBMODEL CLASS SUPERVISORY LOGIC
begin
   branch sys attr classId of this do
   (billProc);
   (customer)
      begin
         if attr dBstatus of this do is READ@ then
            put attr rtime of this sm into sys attr delay of this do
         else if attr dBstatus of this do is WRITE@ then
            put attr wtime of this sm into sys attr delay of this do;
         if sys attr compInst of this sm is BETHESDADISK_SM_INST@ then
            put DB_RETN@ into attr context of this do
         else
            put INV_RETN@ into attr context of this do;
         display attr context of this do with label "Context" in position 2
      end
   end;

   engageIn diskOperations for sys attr delay of this do secs;
   move into local sm host
end
```

Figure 7.84 DATADISK Class Supervisory Logic (Branch Operations)

is made one level at a time. Consider the return from the lowest level of this example and from the Lexington host computer. Figure 7.83 gives the first move to the **outer sm computerComplex** (variable destination location). The logic control and supervision of the dynamic object is shifted to the facility class supervisory logic of Figure 7.85. Since the dynamic object is returning (with context **INV_RETN**), the dynamic object is again directed one level upward to the appropriate outer submodel branch. From there, logic control is transferred to the branch class supervisory logic (Figure 7.86). The returning database request, sent to the host earlier from within the LAN Server, is now directed to that LAN Server's queue.

```
-- FACILITY SUBMODEL CLASS SUPERVISORY LOGIC
if attr context of this do is INV_REQ@ then
   move into inner sm hostQueue
else
   begin
      if  attr dBcode of this do is SYSTEM@ then
         move into outer sm bethesda
      else
         begin
            if attr originBranch of this do is 1 then
               move into outer sm branch1
            else if attr originBranch of this do is 2 then
               move into outer sm branch2
            else if attr originBranch of this do is 3 then
               move into outer sm branch3
            else
               move into outer sm branch4
         end
   end
```

Figure 7.85 FACILITY Class Supervisory Logic (Branch Operations)

```
-- BRANCH SUBMODEL CLASS SUPERVISORY LOGIC
move into inner sm lanQueue
```

Figure 7.86 BRANCH Class Supervisory Logic (Branch Operations)

CHAPTER 8
EVALUATION

An evaluation of the thesis work is given in this chapter. The assessment criteria for the evaluation are taken from the developmental objectives of the DOMINO and the design objectives of the VSSE in Chapter 1. Recall that the overall goals were: (1) to develop a new conceptual framework which is versatile and multifaceted, meeting many of the diverse needs of the simulation model life cycle and model development environment support (for visual simulation modeling), and (2) to design and construct a VSSE with an integrated set of tools to provide the desired automated support for model development. In light of these goals, the evaluation is performed in two phases and subjectively considers how well the objectives have been addressed.

First, the DOMINO is evaluated against the criteria, each of which concerns a desired trait of the DOMINO. In the last evaluation phase, the assessment is directed at the VSSE. Because of the close ties between the DOMINO and the VSSE, some of the criteria from both phases are intertwined and the evaluations overlap. Throughout both evaluation phases, the assessment criteria are considered one at a time. That is, each criteria is restated and the evaluation comments for that criteria follow.

8.1 The Conceptual Framework

The demonstrable features of the DOMINO as implemented in the various VSSE capabilities serve as the primary basis for this phase of the evaluation. The lessons learned from the development of the example model applications of Chapter 7 help to substantiate the comments concerning the various criteria.

Objective 1. Provide both design and implementation guidance, thereby enabling a broad range of support during the simulation model life cycle.

Design guidance generally encompasses object and attribute identification, rules of interaction between objects, hierarchical decomposition and relationships, and input and output specification [Derrick 1988]. Under the DOMINO, the model specification (which is the result of the design process) is formed from modeler-defined components and their parent object classes. The classes possess attributes and individual

model component logic describing the interactions among the components. The architectures of the model static and dynamic structures are also present, produced while the modeler performs top-down hierarchical decompositions. The VSSE, which embodies the concepts of the DOMINO, assists the modeler in completing each of these portions of the specification. The DOMINO provides effective design guidance tempered by the following comments.

The VSMSL language, an integral part of the DOMINO, contains operations and constructs (Section 5.2) to sufficiently specify rules for component interaction. Although the DOMINO has no set format or convention for the model component logic, each VSMSL construct is governed by certain rules of usage. Requirements for each specification perspective (multiview) provide additional design aid. The bottom line, however, is that success in creating the model component logic specifications depends upon the modeler's expertise at organizing these constructs.

The hierarchical decomposition guidance of the DOMINO provides excellent abilities for representing *1:1* and *1:many* relationships. However, the DOMINO does not provide comparable features for indicating *many:many* relationships. Compositional equivalence among layout definitions and class image sets (e.g., relating the movements of many objects to many layout configurations) and the expressive power of the VSMSL overcome this difficulty.

Regarding input specification, the modeler is required to establish initial values for all attributes. Modelers are also effectively guided in the object creation of components (submodels, static objects, subdynamic objects, base dynamic objects, and decomposed dynamic objects) which are in place at the start of model execution. The builtin submodel class supervisory logic is the directed location for specifying the creation of dynamic objects (not decomposed) which are used as input to the model. On the other hand, output specification (in the form of modeler-defined performance measures, attributes, etc.) is completely the modeler's responsibility. Although the tools for specifying the output are available, no explicit guidance or outline is provided to the modeler.

Mode of sequencing and method of sequencing [Derrick 1988] are the primary constituents of implementation guidance. The mode of sequencing reflects the form of the logic representation (events,

activities, processes) and the method of sequencing indicates the function of the program executive (explicit scheduling of events, scanning of conditions, concurrent control of component interactions using a combination of scheduling and scanning techniques). The DOMINO explicitly guides the modeler to create the individual model component logic specifications contain the sequencing mode's representation. Although different variations occur, depending upon the view the modeler takes (material-oriented, machine-oriented, or hybrid), the VSMSL statements within each component logic specification collectively represent a series of activities and, therefore, a process. The method of sequencing is determined by the underlying VSSE implementation of the executive which combines scheduling and scanning techniques (process-interaction). The DOMINO gets high marks for implementation guidance based on the modularization of model component logic specifications and the explicit guidance to modelers for multiview logic specification.

We make a final comment concerning the impact of the design and implementation guidance of the DOMINO to the broad range of support during the life cycle. The DOMINO enables (as demonstrated within the VSSE) the automatic translation of the specification, successfully punctuating the implementation process. Analysis and verification are possible. The visualization of the model is useful, not only in verification during early phases, but also during the concluding phase of the life cycle, the integrated decision support phase. Visualization is a valuable presentation technique.

Objective 2. Contain the underlying organization and structure to achieve the definition and specification of visual simulation models.

Comments under Objective 1 relate directly to the demonstrated ability of the DOMINO to achieve the definition and specification of models. The emphasis of the following comments is on the DOMINO capabilities relative to visual simulation models. First, specific facets of the DOMINO are oriented at visual simulation models (creation of layout and component images and also dynamic object path movement). The VSMSL provides constructs for various display techniques regarding component images. The completed specification contains the necessary information for the Visual Simulator to animate the execution. The implementation facilities for animation are hidden to the modeler. Except for the issues

mentioned just above, the resulting visualization of the model does not add significant additional complexity to the modeling task.

Objective 3. Possess object-orientation to include inheritance of class attributes and model component logic.

Chapter 6 describes, at length, the OOP implementation of the DOMINO within the VSSE. Inheritance of attributes and model component logic is demonstrated in Chapter 7. These examples corroborate the success of the implementation. A summary of all of the DOMINO's object-oriented features, which include inheritance of attributes and model component logic, is given in Section 3.3.1.

Objective 4. Possess graphical-orientation to simplify the definition and specification tasks.

The graphical orientation of the DOMINO is detailed in Section 3.3.2. Again, the VSSE implements these concepts. The Model Generator (Image Editor) contains the drawing facilities for constructing all images (component and layout). Although commercial "paint and draw" products are more powerful than the Image Editor, sufficient capability exists for research purposes. With point and click mouse operations, the components, paths, connectors, etc. are graphically indicated in the class layouts. The top-down hierarchical decomposition of the model structures (both static and dynamic) is easily done with the graphical definition (create/decompose) and navigation (top/ascend/descend) facilities under **CREATE** within the Model Editor.

Objective 5. Embody a WYSIWYR philosophy, allowing modelers to represent a system and its components as they are conceptually or naturally perceived, and simplifying the transition between the conceptual model and the communicative model.

Satisfaction of Objective 3 implies, in some respects, the satisfaction of this objective. One of the primary benefits most often cited regarding the Object Oriented Paradigm is its ability to naturally represent systems [Booch 1986]. In addition, the DOMINO supports the WYSIWYR philosophy through its terminology, use of graphic images, and flexibility in specifying model component logic. The DOMINO terminology (dynamic versus static, real versus virtual, etc.) focuses on things as they really are. Unlike

the GPSS example given in Chapter 3, the DOMINO does not suffer from conceptual artificialities. The graphic images for visualization are modeler-defined and are not predefined. The visualization of the model is just as the modeler "sees" it. By providing several different perspectives that the modeler may use to specify the model component logic, the modeler is not constrained to specify the logic in one way. Again, the modeler has several options and can choose the most appropriate view for component interactions.

Objective 6. Adhere to the guiding principles of the Conical Methodology [Nance 1987].

The evaluation for Objective 6, which contends that the DOMINO fully supports the principles of the Conical Methodology (CM), is grouped in accordance with each of the following CM principles:

- **Top-down definition with bottom-up specification**

The hierarchical traversal and definition features (Sections 3.3.2.5 and 4.1.2.3) enforce a top-down hierarchical definition of the model structures while retaining flexibility for depth-first or breadth-first traversal. Once model component classes are identified, the specification (of attributes and model component logic) proceeds bottom-up. As intended under the CM [Nance 1981b], the modeler can freely alternate between definition and specification tasks.

- **Documentation and specification are inseparable.**

The DOMINO prompts modelers for insertion of documentation information while the modeler is involved in the specification task. The various text subwindows in the VSSE interface are convenient for including textual documentation concerning classes, attributes, etc. Using the underlying INGRES relational database, the QUERY facility creates documentation from the specification information that is stored in the database. Reports generated are informative about many aspects of the specification (Section 4.1.2.4). The Model Analyzer includes capabilities for analyzing the completeness of class and attribute documentation which strengthens the tie of documentation to specification. These diagnostic checks can be done during specification.

- **Iterative refinement and progressive elaboration are essential.**

From within the VSSE, development of model specifications can incrementally proceed as desired by a modeler. Incomplete specifications can be saved for work at a later time. Iterative refinement and progressive elaboration applies to all aspects of specifying class information (attributes and model component logic), structure definition, and image construction. The modeler controls the degree of detail for inclusion in a model. For example, detail in model structure can be limited by choosing not to decompose certain submodels (or subdynamic objects). Should greater detail be desired at a later date, then these components can be subsequently decomposed for additional refinement.

Other features support this criteria. The object orientation of the DOMINO facilities information hiding by encapsulating an object's data (attributes and component logic). Access to object attribute data is controlled via both system and modeler-defined methods. The use of methods (attached to model component classes) provides additional capabilities for functional decomposition. The containment of model component logic in supervisory, self, or method logic is another form of modularization.

- **Verification begins with communicative models and continues throughout the development process.**

The VSSE, using the INGRES relational database representation of the specification (complete or not) can perform diagnostic analysis by the Model Analyzer (both automated and semi-automated) on the representation from the very early stages of development and before executable forms are present. The level of diagnostic capabilities are further evaluated under Objective 10. The Model Verifier provides continued verification abilities (assertion checking, trace data, execution profiles) on executable forms at the later stages of the development process.

- **Specification is independent of implementation.**

Here implementation is meant to be the execution [Nance 1987]. The design of the specification under the DOMINO occurs within the Model Generator. Execution is prepared and performed via the Model Translator and Visual Simulator. The separation in tool functionality satisfies this objective.

Objective 7. **Contain a rich and expressive terminology which applies to any modeling domain in the manner of general purpose frameworks.**

Tables 3.1, 3.2, and 3.3 contain the full scope of the DOMINO terminology. All terms are generic and applicable to any problem domain. In additional support to the domain-independence of the DOMINO, the graphical elements of the DOMINO and the VSSE are generic as well. The modeler is not forced to use images from graphics libraries specific to a problem domain (e.g., FMS, Flexible Manufacturing Systems). Instead, the modeler creates images to suit. Of course, the artistic capabilities of the modeler impact the "suitability" of these images.

Objective 8. **Include flexible but powerful representation schemes for the specification of model component logic and the rules for model component interaction.**

The multiview specification of model component logic (Section 3.3.3.2) brings the diversity and flexibility to satisfy this objective. The representative example models in Chapter 7 demonstrate the power of the VSMSL in handling complex models. Furthermore, these examples indicate the versatility of the multiview specification (Sections 7.1.3, 7.2.3, and 7.3.3).

Objective 9. **Support the Automation-Based Paradigm by: (1) enabling modelers to focus attention and energy at the specification level (whether during design or maintenance and modification), and (2) producing specifications which are fully translatable into executable code.**

The DOMINO succeeds on both goals of support to the Automation-Based Paradigm. As described in Objective 6 (last component), specification is separated from implementation in conjunction with the tools involved. Modelers are not required to get involved at the execution level (C programming level). Within the Model Generator, the specification is the key emphasis. Once the specification is completed, the Model Translator automatically creates the executable version. A modeler can, at any time, perform maintenance or modifications to the model specification without ever having to get involved in the target code programming level.

Objective 10. Produce specifications which are analyzable with maximum diagnostic capabilities in terms of analytical, comparative, and informative assistance [Nance and Overstreet 1986, 1987].

After substantial research, a wide variety of diagnostic capabilities (analytical, comparative, and informative) are now available under the Condition Specification. See [Nance and Overstreet 1986] for a full description of these capabilities. The DOMINO, as implemented within the VSSE, retains several of these capabilities but only a few are demonstrated. Current diagnostic capabilities, although limited, are provided via the current Model Analyzer, aspects of the VSSE implementation, and the VSMSL translator. The discussion contains perceptions relative to the potential of the DOMINO to improved levels of diagnostic capability.

The current Analyzer and VSSE demonstrate attribute initialization and attribute consistency from among the analytical measures. Attributes are required to be initialized during class specification; the VSSE automatically performs these initializations at model startup. The VSMSL translator accomplishes attribute consistency (ensures that attribute typing during definition is consistent with attribute usage in the model component specification). Other analytical measures (attribute utilization and revision consistency) and an informative measure (attribute classification) are readily achievable with modifications to the VSMSL translator and the INGRES database relations. Attribute function (control, output, input function) and usage information can be identified during the translation (for classification), compared with symbol table information (for utilization), or stored for later comparisons (consistency). Another informative diagnostic measure, decomposition, is demonstrated with the hierarchical query capabilities of the VSSE (Section 4.1.2.5).

The remaining diagnostic measures, which include all comparative measures, deal with action cluster diagnosis under the Condition Specification representation. In order to "piggyback" on these techniques, a conversion from the DOMINO representation to a Condition Specification representation would be necessary. This would require additional research, applying Artificial Intelligence or other techniques to recognize and define condition action pairs (and action clusters) during translation of the model component logic specification. Once the action clusters are defined, all diagnostic techniques currently available with

the action cluster incidence graphs or attribute graphs are accessible within the DOMINO. Alternatively, rather than convert the DOMINO specification to a Condition Specification, new research could possibly be initiated to develop diagnostic techniques of the same power using the DOMINO representation as the basis for diagnosis rather than the Condition Specification.

The current Model Analyzer diagnostics form a fairly unique set of capabilities (including analysis of documentation completeness) that are not directly addressed by Nance and Overstreet's [1986] analytical, comparative, and informative measures. Consistency and completeness checks are possible on various aspects of the model specification. These are reviewed in detail in Section 4.2. In a one-to-one correspondence, the DOMINO (as implemented) either demonstrates or could easily be modified to perform six of the fourteen diagnostic measures (almost half) currently available under the Condition Specification. However, new and additional techniques are available under the DOMINO.

8.2 The Visual Simulation Support Environment

This phase of the evaluation centers on the VSSE. Like the DOMINO assessment, comments are developed from experience in performing the example model applications of Chapter 7 while using the VSSE. Evaluation comments are grouped by the associated VSSE tool (except for the first set of comments which refer to the complete toolset).

The VSSE Toolset:

Objective 1. Provide a fully functional and highly integrated toolset. Accomplish the integration and communication among the tools using primarily a relational database representation (e.g., INGRES) of the specification.

Having constructed three example model applications, the complete toolset (Model Generator, Model Analyzer, Model Verifier, Model Translator, and Visual Simulator) was rigorously exercised in performing model development. Each prototype tool performs as described in Chapter 4, with all the attendant capabilities listed therein. Two of the three models (Traffic Intersection, Branch Operations) are non-trivial, large, real-life models. Their successful completion supports the claims. The integration among all tools is satisfied by way of the common INGRES relational database. As the central repository for the

representation of the model specification, every tool had access to the specification information. The underlying VSSE implementation prevented simultaneous access of the database relations, preserving data integrity. Furthermore, the access to specification data is well-coordinated among the tools.

During development of the VSSE prototype environment, the relations of the database evolved to their final form. Effective communication among tools depends upon a well-engineered and well-designed set of relations (field composition, relationships, etc.). The query (report generation on status of the specification data) and the modify (ability for manipulating the data) operations in the Model Generator (Sections 4.1.2.5 and 4.2.1.6) give valuable assistance in preserving the specification in a meaningful form for tool communication.

Objective 2. **Provide a user interface that satisfies the nine usability principles for interfaces [Nielsen 1990]: simple and natural dialogue, speak the user's language, minimize the user's memory load, consistency, provide feedback, provide clearly marked exits, provide shortcuts, good error messages, and prevent errors.**

The VSSE is a prototype system and was not expected to have the look and feel of a fully developed commercial system. In several cases, therefore, the evaluation indicates areas for improvement and inclusion in subsequent prototypes of the VSSE. Each usability principle is taken in turn for comment:

- **Simple and natural dialogue**

Nielsen [1990] describes "simple" as not containing irrelevant or rarely needed information. The VSSE, as one in a series of evolutionary prototypes, has been developed in three stages with the DOMINO (Section 3.6). From a broad, developmental perspective, at each step in the evolution, irrelevant and rarely needed displays have been scrapped. Although the interface allows modelers the freedom to produce voluminous or irrelevant information (e.g., querying for all database information or viewing all trace data), there are many examples of how the modeler can limit the information being displayed to only the relevant material. Pullright menus are typical in the interface style (Figure 4.59). In general, the further right that a "pull" is made, the more limited or categorized the data display becomes. Also, Section 4.3.1.3 specifically describes how the trace manager allows the modeler to manipulate and view a small (and relevant) subset of the trace data.

- **Speak the User's Language**

The use of the direct manipulation interface (containing icons, buttons, mouse for point and click operations, etc.) simplifies the communication of actions to users. Buttons are annotated with the domain-independent terminology of the DOMINO of which the user should have some working knowledge. Context-sensitive help would improve communication and direction to users. Context-sensitive help facilities are not fully developed in this VSSE prototype but have been stubbed for later completion.

- **Minimize the User's Memory Load**

Complicated series of keystrokes are not required, thus minimizing the memory load on a user. Consistency (later discussed) across tools also supports the ease of learning the tools and their features in the VSSE. Alert messages are periodically displayed (Figure 4.17) to guide the user. Again, context sensitive help, if available, would be extremely valuable.

- **Consistency**

This principle is effectively supported across all tools. Several aspects of consistency are described. In all cases, the left mouse button activates the actions of buttons which are embedded in the various VSSE windows. The right mouse button displays popup menus. The middle button is clicked for special, unique operations. Figures 4.2, 4.61, 4.64, 4.71, and 4.74 demonstrate the consistency of the top level window display for each of the five VSSE tools. For all major windows, the **RETURN** button is always located in the top right corner of the window.

- **Provide Feedback**

Alert messages (warnings, errors, delays) provide feedback to users after inappropriate or special responses have been made. The query facility of the Model Generator enables a user to essentially tailor his own feedback mechanism to confirm expected results after input or modification to the specification database.

- **Provide Clearly Marked Exits**

From the top level VSSE window (Figure 4.1), the **QUIT** icon is unmistakable in the form of an annotated stop sign. Once any of the tools are entered, **RETURN** buttons are designated (in most cases

in the same window position). As successive windows are activated, a **RETURN** button provides easy return to the previous window. The VSSE application cannot be exited except from the top level VSSE window.

- **Provide Shortcuts**

In order to maximize the benefits of the direct manipulation interface, there has been an overt attempt to minimize use of the keyboard for supplying or typing in model information. For example, from each top level tool menu (e.g., Figure 4.2), the modeler can type in the model name to be worked but the **RETRIEVE EXISTING MODEL** eliminates the need to type. Also, popups using the right mouse button allow easy selection vice typing (e.g., selection of attribute value type in Figure 4.13). Finally, as previously mentioned, the extensive use of pullright menus enables the quick selection of application functions and provides shortcuts over text-based, command-driven systems.

- **Good Error Messages**

Error messages (for inappropriate responses) are prominently displayed with alert popups. During translation of model component logic specifications, error messages are meaningful and direct the modeler to the location of the error within the specification.

- **Prevent Errors**

The VSSE interface alerts modelers to potential sources of error so that problems can be avoided. For example, Figure 4.12 notifies the modeler that an attribute has already been specified for the given name. The modeler can choose to reenter the attribute name or to continue (thus nullifying the previous attribute data).

Figure 4.58 extends a confirmation request to the modeler for file deletion. This ensures that inadvertent deletions are not performed. Confirmation requests for deletions are standard throughout the interface.

Model Generator:
Objective 3. Provide means for storing model component information in a library.

Library storage is available for layout and component images (Figure 4.4; Section 4.1.1.1). Although the current prototype does not provide the ability to store class specifications (attributes and model

component logic) this would not be difficult to implement in future prototypes. Reusability of class specifications is still possible between models by using (1) operating system commands to copy model component logic files to new model file locations and (2) INGRES database copying facilities to transfer data from one model's database relations to the borrowing model.

Objective 4. Provide suitably implemented mechanisms for inheritance of class attributes and model component logic specifications.

In order to satisfy the assessment criteria (Objective 3) of the DOMINO, the VSSE implementation was required to include inheritance mechanisms. The final implementation is described in Chapter 6. A full single inheritance mechanism (to include inheritance of attributes and model component logic specifications between objects) was created. Due to the unavailability of C++, time constraints, and the need to develop this facility in C, multiple inheritance was not pursued. The development of the example model applications (Chapter 7) has demonstrated the complete suitability of full single inheritance for research purposes.

Objective 5. Provide sufficient capabilities for attribute access and communication between model components to include message passing.

Attribute access is fully described in Section 5.5.3 with an explanation of the three types of attribute referencing mechanisms. Class methods and message passing (Section 5.6.3) were included with the object-oriented implementation. Access to an object's attributes is via its own class methods in keeping with the principles of data abstraction and information hiding. Methods are executed by messages between model objects. Attribute access, methods, and message passing are straightforward using the English-like statements of the VSMSL.

Objective 6. Provide detailed querying capabilities for displaying specification status and progress to modelers, and supporting methods of informal analysis verification techniques. Display query results in appropriate tabular or graphical forms.

The query facility of the Model Generator provides for querying 9 categories of model specification

information (Section 4.1.2.5). After considering the various aspects of queries within each category, a total of 59 different types of queries can be made. The ability to query the current status of the modeling task enables a modeler to carefully monitor the specification. Thus, informal analysis is very nicely supported. The displays are informative and helpful. Both graphical and tabular report presentations are used (e.g., Figures 4.44 and 4.49).

Objective 7. **Provide graphically-based means for the flexible hierarchical definition of model static and dynamic structures. This definition should be characterized by ease and simplicity of movement between the levels of the hierarchy.**

The VSSE provides a powerful, graphically-based means for the top-down definition of model static and dynamic structures. Objective 4 and Objective 6 (top-down definition) evaluations of the DOMINO have already elaborated the depth-first versus breadth-first decompositional flexibility of this VSSE feature. The descend/ascend/top combination of "navigating" actions provide for truly fluid movements among the levels of the hierarchies.

Objective 8. **Provide an expressive (able to specify the various model component interactions) specification language which uses English-like expressions like the HyperTalk language [Winkler and Kamins 1990].**

The VSMSL proved extremely capable during the development of the three example model applications (Chapter 7). The Traffic Intersection represents a complex (many component interactions) model. There were no noted deficiencies in expressive power. A close review of Appendix A and the examples of model component logic reveals the English-likeness of the VSMSL statements. With its English-likeness, the VSMSL raises the specification task to a higher level than programming in a target language. However, enough similarities to programming still exist that a modeler with programming experience will have an advantage over a modeler with no such training.

Objective 9. **Provide ability to translate model component logic specifications directly to the target language and eliminate the need for modelers to interact at the target code level. Relate translation errors to the modeler-defined logic specification, not the target language.**

The VSMSL translator converts the statements in the model component logic to the source code in the target language. If such translation is not successful, a display of the model component logic is presented to the modeler with errors and their locations (within the model component logic) identified. Although a modeler can view the resulting source code, NO modeler interaction at the target code level is required.

Objective 10. **Provide extensive use of symbol tables containing specification data from the relational database, supporting enhanced static analysis techniques during the translation process.**

The VSMSL translator contains a creative facility for the dynamic creation of symbol tables for use in the translator. The translator employs a single symbol table (defined for the class and model component logic being worked). If the context of the translation is changed to the model component logic of another class, then a new symbol table is prepared and embedded in the translator. Due to the complexity of the more comprehensive symbol table (with high resource costs/time for construction) and the constantly changing nature of the specification, the simpler symbol tables and ability to dynamically change between them were chosen. The symbol table is effectively used by the VSMSL translator to ensure statements in the model component logic make correct and meaningful references. For example, object attributes must be named and referenced appropriately for that object's class. The symbol tables also permit attribute type checking within mathematical expressions. Besides these static analysis capabilities of the VSMSL translator, the translator also provides effective syntactical analysis.

Model Analyzer:

Objective 11. **Provide completeness and consistency diagnostic checks on the specification. Use the relational database representation as the basis for analysis. Tailor the analysis to the unique features of the framework.**

As noted in the indepth evaluation of Objective 10 of the DOMINO, the Model Analyzer diagnostic checks present new techniques for analysis that are tailored to the DOMINO. Section 4.2 details the unique analyses which are performed by scanning the relational database representation: class specifications, logic specifications, image specifications, layout definitions and object instantiation, and class documentation.

Model Verifier:

Objective 12. Provide execution tracing with appropriate means of effectively managing trace data and relating runtime errors to the modeler-defined specification. Use the relational database for storage of trace data.

Section 4.3 describes the Trace Manager within the Model Verifier. Section 4.1.3 clearly delineates its impressive manipulation capabilities of the trace data, stored within the INGRES database. Additionally, Sections 4.3.1.2 and 4.3.1.4 present the useful approaches that are available for relating runtime errors to the specification.

Objective 13. Provide means for assertion checking as an additional facility for dynamic analysis.

The VSMSL contains an assertion statement. The Model Verifier, through the use of a toggle, enables a modeler to turn on (activating)/off (bypassing) this statement within any of the model component logic specifications. When an assertion fails, the VSSE halts execution, closes any open files, and gives the location of the failed assertion (See Appendix A).

Objective 14. Allow the creation of performance reporting upon runtime execution by providing execution profile reports.

The Execution Profiler of the Model Verifier (Section 4.3.2), when activated, produces an execution profile of the model including percentage of time during routine execution, cumulative execution, number of calls per routine, and the number of milliseconds per call.

Model Translator:

Objective 15. Provide automatic creation of the executable model from the modeler-defined specification, supporting the Automation-Based Paradigm.

The Model Translator, Section 4.4, accomplishes automatic translation of the DOMINO model specification in a two-step process, creating the source code for OOP modules and compiling and linking the resulting source and object modules into the final executable model. As presented in Objective 9 of the DOMINO, the Automation-Based Paradigm is supported.

Visual Simulator:

Objective 16. Provide for the visualization of the model from any desired runtime context. Movement between contexts should be simply executed.

Contextual visualization is well documented throughout (Sections 4.5.3, 4.5.4, 7.1.4.3, and 7.2.4) the thesis. The capabilities for moving between contexts are even more powerful than the ease of movement during structure definition (Objective 7).

Objective 17. Provide the ability to inspect model component attributes or modeler-defined performance measures at any instant during the visualization of the running model.

Section 4.5.2 sketches the important runtime inspection facility which gives instant-by-instant monitoring capability of attributes (which can be modeler-defined performance measures) to modelers. Figures 4.85 to 4.87 demonstrate this facility and substantiate the satisfaction of the objective. Dynamic analysis of the model specification is enhanced with this feature.

Objective 18. Provide the ability to run the simulation in the background without animation, producing statistical analysis reports.

The description of the Visual Simulator (Sections 4.5 and 4.5.1) summarizes the VSSE execution of a model without execution while providing confidence interval (Figures 4.77 to 4.79) and other statistical analyses. The removal of underlying system code for animation is automatically performed. No additional burden is placed on the modeler when the background mode of execution is chosen.

CHAPTER 9
CONCLUSIONS AND FUTURE RESEARCH

The thesis research has produced the DOMINO, a new conceptual framework which is extremely versatile, supporting many diverse needs of the simulation model life cycle. The DOMINO is established as a viable framework, able to serve as the basis for a simulation model development environment. The VSSE represents a massive but extremely productive effort for providing automated support for model development and for demonstrating the utility of the DOMINO. In view of the evaluation in Chapter 8, we draw several important conclusions concerning the DOMINO and the VSSE.

9.1 The Multifaceted DOMINO

The DOMINO is truly multifaceted and represents a step forward in conceptual frameworks for the design and implementation of simulation models.

Providing both design and implementation guidance, the DOMINO guides a modeler over a broader range of tasks in the simulation model life cycle. Previous critical reviews indicated the shortfalls of the common conceptual frameworks that are in use today [Derrick 1988]. The DOMINO contains many desired characteristics (object-oriented, graphical-oriented, and activity-based with a WYSIWYR (What You See Is What You Represent) philosophy) and new approaches which generate a fruitful and nurturing environment within which a modeler is assisted in the creative processes. Incorporating the Object-Oriented Paradigm, the DOMINO contains excellent support for good software-engineered designs [Korson and McGregor 1990] (e.g., modularity, information hiding, strong cohesion and locality of object information, abstraction, extensibility, integrable, etc.). The graphical orientation facilitates and simplifies the modeling task by bringing visual and tactile senses into the highly conceptual definition and specification tasks. The activity basis and modularization of object processes within the model component logic creates new flexibility (supervisory, self, and hybrid logic views) for modeler perceptions surrounding the specification of model component interactions. The WYSIWYR philosophy allows modelers to represent a system and its components as they are conceptually or naturally perceived. The modeler is freed from conceptual artificialities in building the model specification via the strong emphasis on graphics

and visualization, flexibility in specifying model component logic, and the simple, natural terminology (dynamic versus static, real versus virtual, etc.) of the DOMINO. Other multifaceted aspects of the DOMINO (e.g., domain-independence) are discussed below.

The incorporation of visualization as a prominent element of the DOMINO enhances the verification and validation tasks.

From the outset, visualization has been a key focus of the research. The DOMINO effectively embeds visualization and garners the benefits through the graphical approaches of the definition and specification tasks, the unique display statements of the VSMSL, and the addition of dynamic analysis with the animation of the executing model. The lessons learned regarding visualization should have a strong influence in future directions of SMDE research.

The DOMINO is domain-independent in all aspects of terminology and representation supplying significant advantages to project teams with diverse modeling needs.

The DOMINO terminology is generic. The modeler uses modeler-defined graphical representations to suit and is not forced to use "canned" images from a domain-specific library. Project teams can perform model development under a single framework and do not have to acquire knowledge or capability in several model development methodologies.

The DOMINO has achieved marked progress toward achieving the Automation-Based Paradigm, with potential gains in efficiency and productivity in the model development effort, and reductions to development time.

Modelers work only with the specification representation which is automatically translated to the executable model version. This is true during design and implementation and while performing subsequent maintenance and/or modifications. While not having to get involved at the target code programming level (and in light of the DOMINO's domain-independence), the modeler is not required to learn any Simulation Programming Languages. This burden has previously been a severe stumbling block to the development process.

The DOMINO fully supports the guiding principles of the Conical Methodology and is a strong candidate for use as the framework for the SMDE.

The evaluation comments for Objective 6 of the framework give a detailed report on how the DOMINO supports the CM. The DOMINO, although possessing limitations (covered below), provides

comprehensive satisfaction of the Conical Methodology's principles. Use of the DOMINO would not compromise the original goals and objectives of the SMDE.

The most severe weakness of the DOMINO (when compared to the Condition Specification) surrounds its current diagnostic capability. Certainly, the diagnostic measures which are achievable are substantial and effective. However, additional research needs to be conducted to determine how far the boundaries of its diagnostic capabilities can be pushed. Given the range of all that DOMINO offers and the current wealth of knowledge in this area surrounding the Condition Specification, the prospects are promising.

Other weaknesses are evident in the DOMINO's inability to more explicitly represent *many:many* relationships and output specification. Additional work on the VSMSL should easily rectify these deficiencies.

9.2 Automated Modeling Support with VSSE

> **The VSSE with its fully functional and highly integrated toolset demonstrates significant advances in the SMDE prototyping effort and automated support for model development.**

Based on the DOMINO, the VSSE brings new features to the SMDE. First, the VSSE is a complete environment with a minimal set of tools. Several versions of prototyped tools exist in the SMDE, but not in a functional environment. In this context, lessons have been learned regarding the use of INGRES relational database as the basis for communications among the tools, and INGRES has performed well. The use of a library for reusability was introduced. The Object-Oriented Paradigm was more fully implemented than in previous tools with true inheritance, methods, and message passing. Visualization and the extended use of graphical facilities for definition and specification are explored within an SMDE environment for the first time. The first prototype of the Model Verifier is included. Versatile, with facilities for assertion checking, trace management and execution profiling, the Model Verifier provides significant, additional capabilities for automated support.

> **The VSSE provides a wide range of effective verification techniques. This contributes to an area in the life cycle of a simulation study where emphasis is currently lacking.**

The VSMSL translator with its component symbol tables provides static and syntactic analysis capabilities which are realized during the translation of the model component logic. The powerful querying and modification capabilities of the Model Generator are valuable support for informal analysis. The visualization of the executing model and the runtime inspection facility strongly impact and aid a modeler's dynamic analysis capabilities.

The graphical facilities for structure definition and contextual visualization clearly offer help to the modeler in overcoming the complexity problems associated with large modeling efforts. The screen designs demonstrate simplicity and ease of use.

Modelers are able to flexibly define the model structure in depth-first or breadth-first traversals. The navigation of movements between the structure during definition and runtime is as simple as paging through the documents of a word processor with comparable features to the common up, down, page up, page down, top, and bottom. The modeler can view the model structure in the smaller context or effectively "zoom" to a larger context by ascending the hierarchy. Viewed or defined in increments, the complexity of the structure is successfully managed.

The VSSE enables the effective evaluation of the DOMINO and underscores the contributions of the research effort.

The non-trivial example model applications (of Chapter 7) demonstrate the utility of the DOMINO for the design and implementation of simulation models. Guidance and support are given to the modeler through the VSSE and the underlying DOMINO for wide variety of tasks during model development.

Several areas of the VSSE are candidates for future research. Although significant progress toward the Automation-Based Paradigm is achieved, notably lacking within the VSSE is a knowledge-based assistant [Balzer et al. 1983]. Further research is required to assimilate such an assistant into the VSSE. In addition, the VSMSL logic specification facilities need extensions to reduce the "programming-like" requirements while using VSMSL. Appropriate means of facilitating the use of VSMSL need to be identified.

Concerning visualization, modelers must associate an image with all real components under the current VSSE implementation. As a future improvement, modelers should have the option to/not to visualize real components, depending upon his or her view. For example, the light controller of the Traffic Intersection

example (Chapter 7) is a real component which does not need to be visualized.

One inconspicuous aspect of the rich, descriptive DOMINO terminology (as described in Chapter 3) is not implemented under the current VSSE prototype (i.e., decomposition of virtual model components). The current research is not limited by this omission. However, future VSSE prototypes ought to include this capability in order to fully utilize the full range of DOMINO capabilities. This decomposition feature can be readily incorporated into the existing VSSE structure.

Although the DOMINO and VSSE contain limitations, their development and design have made significant contributions, as detailed in the thesis. Gains are expected to the SMDE research effort, and simulation model development has been positively affected.

APPENDIX A
SPECIFICATION LANGUAGE
BNF and Language Examples

TOP LEVEL ELEMENTS AND STATEMENT TYPES

<spec> := <stmt>

<stmt> :=	
<aliastmt>	Aliasing mechanism [NOT IMPLEM]
<locstmt>	Logic move to label location (->) [NOT IMPLEM]
<movestmt>	"Moving" a dynamic object
<branchstmt>	Branching instructions (e.g. CASE stmt)
<addstmt>	Arithmetic add
<substmt>	Arithmetic subtract
<multstmt>	Arithmetic multiply
<divstmt>	Arithmetic divide
<ifstmt>	Decision making "if"
<getstmt>	Retrieval of data into "it"
<putstmt>	Assignment of data
<createstmt>	Dynamic object creation
<destroystmt>	Dynamic object destruction
<displaystmt>	Display of Data on dynamic objects during animation
<erasestmt>	Removal of data display on dynamic objects
<imagestmt>	Changing DO and non-DO images during animation
<assertstmt>	Assertions are allowed
<waitstmt>	Process "wait" and suspend or "passivate" [NOTIMP]
<restartstmt>	Startup from unconditional wait[NOT IMPLEM]
<returnstmt>	Return from method statement
<engagestmt>	Engagement in activity
<msgstmt>	Send message statement
<repeatstmt>	Looping "repeat"
begin <stmtblck> end	Block construction
<error> ; <stmtblck>	Error handling for parser, go to ";" and resume parse

<stmtblck> :=	
<stmt>	Simple statement
<stmtblck> ; <stmt>	Left recursive for YACC preference

SIMPLE REGULAR EXPRESSIONS

{ID} := [A-Za-z][A-Za-z0-9_]*	Letter then 0 or more letters, numbers, underscore
{Comment} := [-][-]	
{Quote} := "	
{Const} := [A-Z]+[A-Z0-9_]*"@"	Caps, numbers, underscore, @sign at end
{Literal} := {Quote}[^\"\n]*{Quote}	Basically a quoted ID

{Integer} := [0-9]+ {Real} := [0-9]+"."[0-9]+ {Char} := '[a-zA-Z]'

Simple Expression Usage and Error Checking Notes:

1. Constant names are not checked during translation.

MOVE STATEMENT

```
<movestmt> :=
        move do <compid> <location>     |  [PROVISIONALLY IMPLEM]
        move <location>                 |  Commonly Used
        move! <location>                   Executing or EXEC move within self logic

<location> :=
        to <blocation>                  |  Move this do to so/bdo (base)
        into <dlocation>                   Move this do into sm/sdo/do (decomp)

<blocation> :=
        local <baselocation>            |  At same level of decomposition
        <baselocation>                  |  (implies "local")
        outer <baselocation>            |  upward in the decomposition, parent
        inner <baselocation>               downward, implying children

<dlocation> :=
        local <declocation>             |  At same level of decomposition
        <declocation>                   |  (implies "local")
        outer <declocation>             |  upward in the decomposition, parent
        inner <declocation>                downward, implying children

<baselocation> :=
        static object <compid>          |  Not decomposable, "base"
        so <compid>                     |
        base dynamic object <compid>    |
        bdo <compid>

<declocation> :=
        submodel <compid>               |  decomposable
        sm <compid>                     |
        virtual submodel <compid>       |
        vsm <compid>                    |
        subdynamic object <compid>      |
        sdo <compid>                    |
        dynamic object <compid>         |
        do <compid>

<compid> :=
        {ID}                            |  named-by-variable
        {Literal}                          named-by-literal
```

Move Statement Usage and Error Checking Notes:

1. All literal names are checked against a literal component instance symbol table to ensure that a literal of the designated component type actually exists. The "local", "inner", "outer" designation is applied to all named-by-literal movement destinations. Without this designation, "local" is assumed. The use of spatial designations improves readability and understandability but there is NO error checking to ensure that a particular literal instance is within the "local", "inner", or "outer" context of the current layout.

2. For non-dynamic component move destinations which are named-by-variable, the variable names are checked only when the move is a "local" one since only a "local" variable component symbol table is available. When such moves are designated as "outer" or "inner", they are, nevertheless, allowed; There is, however, NO error checking in these cases. When the "local", "inner", or "outer" designation is not present in a named-by-variable movement destination for a non-dynamic object, the "local" designation is assumed. NOTE: Remember that named-by-variable moves to "inner", "outer", and "local" components are allowed because of the associated pointers which are initialized for each component at compile time.

3. When a move destination is a named-by-variable dynamic object, there is NO error checking since it is not possible to know the identity of the dynamic object (the id of the dynamic object is contained in the given variable at run-time) at compile time. The variable used to name the dynamic object in such a case is a model attribute. There is limited error checking to determine the validity of this model attribute. (e.g., move into do dynObjId)

4. Moves are allowed only to "local", "outer", and "inner" component destinations. This ensures that movements are across only one boundary of the hierarchy, at most. There is NO error checking to enforce this rule.

Examples:

move into local sm lanserver	Move this do into sm in local layout
move into sm "lanserver1"	Move this do into sm (literal)
move into local sm "lanserver1"	Move this do into sm (literal)
move into sdo flightdeck	Move this do into sdo in local layout
move into outer sdo "cva68deck"	Move this do into sdo (literal)
move into do "cityBus"	Move this do into (literal) dynamic obj
move into do dynObjId	Move this do into dynamic object
move to inner so cpu	Move this do to so in inner layout
move to so "maincpu"	Move this do to so (literal)
move to outer bdo holdingArea	Move this do to bdo in outer layout
move to bdo "cva68hold"	Move this do to bdo (literal)
move do firstDO to ...	[PROVISIONALLY IMPL] Move do whose identifier is in "firstDO"
move do "cityBus" into sm "busTerminal"	Move decomposed do to start in terminal sm
move! into sm "blockQueue"	Move into and execute logic of sm (SelfLogic)

BRANCH STATEMENT

```
<branchstmt> :=
        branch <container> <brblk> end

<brblk> :=
        <brstmt> ; <brblk>          |
        <brstmt>

<brstmt> :=
        <brcase> <stmt>             |
        <brcase>                                (Note empty case allowed; terminate with ";")

<brcase> :=
        ( <brid> )

<brid> :=
        {ID}                        |
        {Const}                     |
        {Char}                      |
        otherwise
```

Examples:

```
branch sys attr compInst of this sm
(STOPAB_SM_INST@) -- Constant with @ sign
        begin
        .....
        end;  -- Each case end has semicolon (except last)
(STOPCD_SM_INST@)
        begin
        .....
        end  -- Last case end has no semicolon
end; -- end Branch

branch sys attr classId of this do
(bus)  -- Class name
        begin
        .....
        end;
(person)
        begin
        .....
        end
end;
```

MATHEMATICAL STATEMENTS

```
<addstmt> :=
        add <expr> to <container>

<substmt> :=
        sub <expr> from <container>

<multstmt> :=
        mult <container> by <expr>

<divstmt> :=
        div <container> by <expr>

<container> :=
        result                                  |
        {ID}                                    |
        {ID} [ {ID} ]                           |
        {ID} [ {Integer} ]                      |
        <attrexpr>

<attrexpr> :=
        attr {ID} of <compexpr>                 |
        sys attr {ID} <compexpr>                |
        sys attr {ID} [{ID}] of <compexpr>      |
        sys attr {ID} [{Integer}] of <compexpr> |
        attr {ID}

<compexpr> :=
        <component> <compid>                    |
        this <component>                        |
        this supervising <component>            |
        current <component>

<component> :=
        submodel | sm | virtual submodel | vsm  |
        static object | so |
        base dynamic object | bdo |
        subdynamic object | sdo |
        dynamic object | do

<expr> :=
        <expval>                                |
        <expr> <op> <expr>                      |
        ( expr )

<expval> :=
        <boolval>                               |
        <litval>                                |
        <container>                             |
        <msgval>
```

```
<op> :=
        is                                  |
        =                                   |
        <addop>                             |
        <multop>                            |
        <relop>                             |
        is <relop>                          |
        <boolop>

<addop> :=
        + | -
<multop> :=
        * | /
<relop> :=
        > | >= | < | <= | !=
<boolop> :=
        and | or

<boolval> :=
        true|TRUE|false|FALSE

<msgval> :=
        message {ID} to <compexpr>      |
        message {ID} to <compexpr> using <args>

<litval> :=
        {integer}|{real}|{Char}|{Literal}|{Const}

<getstmt> :=
        get <container>

<putstmt> :=
        put <expr> into <container>     |
        put it into <container>
```

Examples:

```
add 1 to sys attr numDos of do "cityBus";
subtract 1 from attr numberExiting of do "cityBus";

get attr thumpy of this do;                     -- Retrieves value into "it"
put it into attr y of sm "stationWagon";        -- "It" must be used immediately after get

put sys attr ident of this do into sys attr dosList[i] of do "cityBus"
put OFF@ into attr busIntention of sm busStop;
put i+1 into dynObjId;
put message right5Clear to sm "traffic" into rightTurnClear;
```

CREATE and DESTROY

```
<createstmt> :=
    create <otype> belonging to class {ID} putting its id in <container>
        starting in <dlocation>

<otype> :=
    rdo          |
    vdo

<destroystmt> :=
    destroy      |
    destroy do <compid>
```

Examples:

```
create rdo belonging to class person putting its id in dynObjId
        starting in sm "houseE";
```

Note: For the create statement, there is no error checking on the class name.

```
destroy;  -- this do implied
destroy do "cityBus";
```

ASSERTIONS

```
<assertstmt> :=
        assert <expr>
```

Assertion Statement Usage Notes:

1. If the assertion is ever false, the system will halt execution, close any open files, and give the source code file and source line number where the assertion failed. Also, the affected logic and line number within the logic will be given.

Examples:

```
assert attr numberWaiting of sm "cpuQueue" > 20;
```

WAIT and ENGAGE

```
<waitstmt> :=                                   [NOT IMPLEMENTED]
        wait

<restartstmt> :=                                [NOT IMPLEMENTED]
        restart do <compid>

<engagestmt> :=
        engageIn {ID} for <container> <time> |
        engageIn {ID} until <condition>

<time> :=
    hour          | hr    |
    hours         | hr    |
    minute        | min   |
    minutes       | mins  |
    second        | sec   |
    seconds       | secs  |
    millisecond   | msec  |
    milliseconds  | msecs

<condition> :=
        <container> <crelop> <cval>          |
        ( <condition> )                      |
        <condition> <boolop> <condition>     |
        <msgval> <crelop> <cval>

<crelop> :=
    is            |
    is =          |
    is <relop>    |
    <relop>

<cval> :=
        <container>     |
        <boolval>       |
        {Integer}       |
        {Const}
```

Examples:

```
engageIn passengerTransfer until attr numberWaiting of this sm is 0 and
        attr numberExiting of this do is 0;
engageIn startingUp for attr stopStartTime of this do secs;
engageIn waiting until ((attr color of so "ewLight" is GREEN@) and
        (message right5Clear to sm "traffic" is TRUE@));
```

LOOPING and DECISION MAKING

```
<repeatstmt> :=
        repeat <rtype>

<rtype> :=
        forever <stmt> end                                          |
        with <container> = <rval> <rtoval> <rval> <stmt> end        |
        while <rcondition> <stmt> end                               |
        until <rcondition> <stmt> end

<rval> :=
        {integer}                                                   |
        <container>

<rtoval> :=
        downto | to

<rcondition> :=
        <boolval>                                                   |
        <condition>

<ifstmt> :=
    <ifx>       |
    <ifx> else <stmt>

<ifx> :=
   if <expr> then <stmt>
```

Examples:

```
        repeat with i = 1 to sys attr numDos of this do
                begin
                put sys attr dosList[i] of do "cityBus" into dynObjId;
                put TRUE@ into attr shoppingCompleted of do dynObjId
                end
        end;  -- Repeat end

        if sys attr currentComp of this do is sys attr compInst of sm "stopCity" then
                move into sm stopAB
        else if attr numberWaiting of this sm is > 0 then
                move into sm busQueue;

        if blockAIdle then move into sm blockA;

        if message rand1 to vsm "statmodule" using
                (attr mySeed of this do) < = 0.90 then ...
```

ANIMATION DISPLAY STATEMENTS

<displaystmt> :=
display <container> with label {Literal} in position {Integer}
display {Literal} in position {Integer}

<erasestmt> :=
turn off display in position {Integer}

<imagestmt> :=
set image of <compexpr> to {Literal}

Display Statements Usage and Error Checking Notes

1. The display and erase statements are for display of data (such as attribute values) on the image of real dynamic objects. Implicit in the use of this statement is that the display will be on "this do". There are four display positions available. In the first display statement, the container's value is displayed at the designated position with the given text label to identify the given value. In the second statement, only the given text label is displayed in the designated position. The erase statement removes the display from the designated position.

2. The image statement provides the means for changing the component image during runtime. A dynamic object or non-dynamic object image may be switched to one of its image set. Ideally, there would be a check made to ensure the image being used is from the allowed class image set. However, there is no error checking to force this.

Examples:

```
display attr x of this do with label "X" in position 2;
display "5" in position 3;
turn off display in position 3;
set image of this do to "fullBus";
set image of so "light" to "red";
set image of current sm to "busAB";
```

MESSAGE AND RETURN STATEMENTS

```
<msgstmt> :=
        send message {ID} to <compexpr>                          |
        send message {ID} to <compexpr> putting result in <container>  |
        send message {ID} to <compexpr> using <args>             |
        send message {ID} to <compexpr> using <args> putting result
                in <container>

<args> :=
        ( <paramlist> )

<paramlist> :=
        <param>                  |
        <paramlist> , <param>

<param> :=
        <container>              |
        <litval>

<returnstmt> :=
        return result
```

Message and Return Statement Usage Notes

1. Messages sent to named-by-variable components can currently only be sent to "local" components.

2. Parameters are allocated in alphabetical order and should be used in the message statement argument list in that order. Before the method can be successfully translated, the method name must be first saved to the database. Note that the number and type of arguments to a method is not checked by the compiler.

3. The method can return a value through the "result" container, but the return statement must be explicitly used in this case.

Examples:

```
put 0.10 into attr mean of vsm "statmodule";
send message expon to vsm "statmodule" using
        (attr mean of vsm "statmodule",
        attr seed of vsm "statmodule")
        putting result in attr serviceTime of this bdo;
send message linear to vsm "statmodule" using
        ("data5.out",
        attr iseed5 of vsm "statmodule")
        putting result in sys attr delay of this do;
```

APPENDIX B
SYSTEM CONSTANTS

TRUE@	(Value 1)
FALSE@	(Value 0)
YES@	(Value 1)
NO@	(Value 0)
ON@	(Value 1)
OFF@	(Value 0)
BUSY@	(Value 1)
IDLE@	(Value 0)
ZERO@	(Value 0)
ONE@	(Value 1)
TWO@	(Value 2)
THREE@	(Value 3)
FOUR@	(Value 4)
FIVE@	(Value 5)
SIX@	(Value 6)
SEVEN@	(Value 7)
EIGHT@	(Value 8)
NINE@	(Value 9)
TEN@	(Value 10)
ELEVEN@	(Value 11)
TWELVE@	(Value 12)

APPENDIX C
SYSTEM ATTRIBUTES

Non-Dynamic Objects

noOfDos - Number of Dynamic Objects resident in the component (non-auto)
entryList[] - List of resident Dynamic Objects by order of arrival (non-auto)
compInst - Int identifier of the component instance which has constant (auto)
Constants are formatted as follows:
InstancenameInCaps_ComponentTypeInCaps_INST@
HOUSEA_SM_INST@, LIGHT_SO_INST@, SEAT7_SD_INST@,
DRIVER_BD_INST@, etc.
classIdent - Int identifier of the component class of instance with constant (auto)
Constants are formatted as follows:
ComponentTypeInCaps_ClassnameInCaps_TYPE@
SM_TERMINAL_TYPE@, SD_SEAT_TYPE@, BD_DRIVER_TYPE@,
SO_LIGHT_TYPE@, etc.
realVirt - Boolean, Real is 1, Virtual is 0 (auto)

Dynamic Objects

ident - Int identifier of the Dynamic Object instance (auto)
numDos - Number of Dynamic Objects resident (only for decomp Dyn Objs, non-auto)
dosList - Like entryList above (non-auto)
doInst - Int identifier of the instance, has constant as above (auto)
CITYBUS_DO_INST@ is an example constant.
realOrVirt - Boolean, Real is 1, Virtual is 0 (auto)
classId - Int identifier of the class, with constant (auto)
DO_BUS_TYPE@ is example constant.
originComp - Int identifier of the origin component (auto)
currentComp - Int identifier of the current component the DO is in (auto)
prevComp - Int identifier of the last component the DO was in (auto)
delay - Time for use in activity delay (non-auto)
moveTime - Time the Dynamic Object will be moved off FOL (auto)
compEntry - Time of Dynamic Object entry to component (non-auto)
modelEntry - Time of Dynamic Object entry to model (auto)
currentSuperLoc - Int identifier of Dynamic Object location in supervisory logic (auto)
currentSelfLoc - Int identifier of Dynamic Object location in self logic (auto)

Notes: auto - system attribute automatically updated by system
non-auto - not automatically updated but available for inspection

APPENDIX D
SYSTEM METHODS

methodName - Description
parameters in order: description (attribute of distribution class available)

rand1 - Random variate generator, variates between 0 and 1
seed: integer seed value (iseed, iseedn)

beta - Beta distribution
seed: integer seed value (iseed, iseedn)
shape1: double shape 1 parameter (shape1)
shape2: double shape 2 parameter (shape2)

erlng - Erlang distribution
k: integer, number of exponential rv to be summed (kSum)
mean: double mean (mean)
seed: integer seed value (iseed, iseedn)

expon - Exponential distribution
mean: double mean (mean)
seed: integer seed value (iseed, iseedn)

gama - Gamma distribution
scale: double scale parameter (scale)
seed: integer seed value (iseed, iseedn)
shape: double shape parameter (shape1)

linear - Linear inverse transformation (User defined)
inputFile: pointer to char, input Data file
seed: integer seed value (iseed, iseedn)

npssn - Poisson distribution
mean: double mean (mean)
seed: integer seed value (iseed, iseedn)

rlogn - Lognormal distribution
mean: double mean (mean)
seed: integer seed value (iseed, iseedn)
std: double standard deviation (stdDev)

rnorm - Normal distribution
same parameters as rlogn

triag - Triangular distribution
hiValue: double highest value in interval (hiVal)
lowValue: double lowest value in interval (or location) (lowVal)
mode: double mode (or shape) (mode)
seed: integer seed value (iseed, iseedn)

unfrm - Uniform distribution
hiBound: double highest boundary value (hiVal)
lowBound: double lowest boundary value (lowVal)
seed: integer seed value (iseed, iseedn)

weibl - Weibull distribution
same parameters as gama

APPENDIX E
RESERVED WORDS

and, or
attribute
attr
base dynamic object
bdo
begin
belonging
branch
by
class
create
current
destroy
display
divide
downto
dynamic object
do
else
end
engageIn
false, FALSE
for
forever
from
get
id
if
image
in
inner
into
is
its
it
label
local
message
move
move!
multiply
of
off
otherwise
or
outer
position
put
putting
rdo
repeat
restart
result
return
send
set
static object
so
starting
subdynamic object
sdo
submodel
sm
subtract
supervising
sys
system
then
this
[time words]
to
turn
true,TRUE
until
using
virtual dynamic object
vdo
virtual submodel
vsm
wait
with
while

REFERENCES

Adelsberger, H.H., U.W. Pooch, R.E. Shannon, and G.N. Williams (1986), "Rule Based Object-Oriented Simulation Systems," In *Proceedings of the Conference on Intelligent Simulation Environments*, SCS, San Diego, CA, 107-112.

Au, G. (1990), "A Graphics-Driven Approach to Discrete Event Simulation," Ph.D. Dissertation, Department of Statistical and Mathematical Sciences, London School of Economics, London, England.

Baeker, R.M. (1974), "Genesys Interactive Computer-Mediated Animation," In *Computer Animation*, J. Halas, Ed. Hastings House, New York, NY, 97-115.

Balci, O. (1986a), "Guidelines for Successful Simulation Studies: Part I and II," Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Balci, O. (1986b), "Requirements for Model Development Environments," *Computers & Operations Research 13*, 1, 53-67.

Balci, O., Ed. (1987), *Proceedings of the Conference on Methodology and Validation*, SCS, San Diego, CA.

Balci, O. (1989), "How to Assess the Acceptability and Credibility of Simulation Results," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE. Piscataway, NJ, 62-71.

Balci, O. (1990), "Guidelines for Successful Simulation Studies," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE. Piscataway, NJ, 25-32.

Balci, O. and R.E. Nance (1987a), "Simulation Support: Prototyping the Automation-Based Paradigm," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 495-502.

Balci, O. and R.E. Nance (1987b), "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society 38*, 8, 753-763.

Balci, O. and R.E. Nance (1989), "Simulation Model Development: The Multidimensionality of the Computing Technology Pull," In *Impacts of Recent Computer Advances on Operational Research*, R. Sharda, B.L. Golden, E. Wasil, O. Balci, and W. Stewart, Eds., Elsevier Science Publishing, New York, NY, 385-395.

Balci, O., R.E. Nance, E.J. Derrick, E.H. Page, and J.L. Bishop (1990), "Model Generation Issues in a Simulation Support Environment," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 257-263.

Balmer, D.W. (1987), "Modelling Styles and Their Support in the CASM Environment," In *Proceedings*

of the 1987 Winter Simulation Conference, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 478-485.

Balmer, D.W. and R.J. Paul (1986), "CASM - The Right Environment for Simulation," *Journal of the Operational Research Society 37*, 5, 443-452.

Balmer, D.W. and R.J. Paul (1990), "Integrated Support Environments for Simulation Modeling," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, and R.E. Nance, Eds., IEEE, Piscataway, NJ, 243-249.

Balzer, R., T.E. Cheatham, Jr., and C. Green (1983), "Software Technology in the 1990's," *Computer 16*, 11, 39-45.

Barger, L.F. (1986), "The Model Generator: A Tool for Simulation Model Definition, Specification, and Documentation," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Barger, L.F. and R.E. Nance (1986), "Simulation Model Development: System Specification Techniques," Technical Report SCR-86-005, Systems Research Center, Virginia Tech, Blacksburg, VA.

Basili, V.R. and J.D. Musa (1991), "The Future Engineering of Software: A Management Perspective," *Computer 24*, 9, 90-96.

Beams, J.D. (1991), "A Premodels Manager for the Simulation Model Development Environment," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Beams, J.D. and O. Balci (1992), "Providing Reusability and Learning Support in the Simulation Model Development Environment," Technical Report TR-92-3, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Bell, P.C. and R.M. O'Keefe (1987), "Visual Interactive Simulation - History, Recent Developments, and Major Issues," *Simulation 49*, 3, 109-115.

Bishop, J.L. (1989), "General Purpose Visual Simulation System," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Bishop, J.L. and O. Balci (1990), "General Purpose Visual Simulation System: A Functional Description," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 504-512.

Booch, G. (1986), "Object-Oriented Development," *IEEE Transaction on Software Engineering SE-12*, 2, 211-221.

Box, C.W. (1984), "A Prototype of the Premodels Manager," MDE Project Memorandum, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Brumbaugh, D. (1990), "Object-Oriented Programming In C," *The C Users Journal 8*, 7, 113-122.

Bryan, O.F. (1989), "Productivity Tools in Simulation: SIMSCRIPT II.5 and SIMGRAPHICS," In

Proceedings of the 1989 Winter Simulation Conference, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds., IEEE, Piscataway, NJ, 164-170.

Campbell, R.H. (1986), "SAGA: A Project to Automate the Management of Software Production Systems," In *Software Engineering Environments*, I. Sommerville, Ed. Peter Peregrinus Ltd., London, 182-201.

Chen, P.P. (1976), "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems 1*, 1, 9-36.

Clementson, A.T. (1978), "Extended Control and Simulation Language," In *Proceedings of the 1978 UKSC Conference on Computer Simulation*, IPC Science and Technology Press, Guildford, England, 174-180.

Conway, R. and W. Maxwell (1986), "XCELL: A Cellular Graphical Factory Modelling System," In *Proceedings of the 1986 Winter Simulation Conference*, J.R. Wilson, J.O. Henriksen, and S.D. Roberts, Eds., IEEE, Piscataway, NJ, 160-163.

Corey, P.D. and J.R. Clymer (1991), "Discrete Event Simulation of Object Movement and Interactions," *Simulation 56*, 3, 167-174.

Cox, B.J. (1986), *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA.

Dart, S.A., R.J. Ellison, P.H. Feiler, and A.N. Habermann (1987), "Software Development Environments," *Computer 20*, 11, 18-28.

Derrick, E.J. (1988), "Conceptual Frameworks for Discrete-Event Simulation Modeling," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Derrick, E.J. (1989a), "Conceptual Frameworks for Simulation Model Development Environments," In *Proceedings of the 1989 Virginia Computer User's Conference*, Blacksburg, VA, 9-18.

Derrick, E.J. (1989b), "Discrete-Event Simulation Conceptual Frameworks: Another Look at Event, Activity, and Process-Oriented Approaches," In *Proceedings of the 1989 Beijing International Conference on Systems Simulation and Scientific Computing*, International Academic Publishers, Beijing, 16-20.

Derrick, E.J., O. Balci, and R.E. Nance (1989), "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 711-718.

Doukidis, G.I. (1987), "An Anthology on the Homology of Simulation with Artificial Intelligence," *Journal of the Operational Research Society 38*, 8, 701-712.

Doukidis, G.I. and R.J. Paul (1985), "Research into Expert Systems to Aid Simulation Model Formulation," *Journal of the Operational Research Society 36*, 4, 319-325.

Doukidis, G.I. and R.J. Paul (1986), "Experiences in Automating the Formulation of Discrete-Event Simulation Models," In *Proceedings of the Conference on AI Applied to Simulation*, SCS, San Diego, CA, 79-90.

Dowson, M. (1987a), "ISTAR - An Integrated Project Support Environment," *ACM SIGPLAN Notices 22*, 1, 27-33.

Dowson, M. (1987b), "Integrated Project Support with IStar," *IEEE Software 4*, 6, 6-15.

Fishman, G.S. (1973), *Concepts and Methods in Digital Discrete-Event Simulation*, John Wiley, New York.

Frankel, V.L. (1987), "A Prototype Assistance Manager for the Simulation Model Development Environment," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Frankel, V.L. and O. Balci (1989), "An On-Line Assistance System for the Simulation Model Development Environment," *International Journal of Man-Machine Studies 31*, 6, 699-716.

Franta, W.R. (1977), *The Process View of Simulation*, North Holland, Amsterdam, the Netherlands.

Gilman, R.A. and C. Billingham (1989), "A Tutorial on SEE WHY and WITNESS," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds., IEEE, Piscataway, NJ, 192-200.

Gordon, R.F., E.A. MacNair, K.J. Gordon, and J.F. Kurose (1990), "Hierarchical Modeling in a Graphical Simulation System," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, R.E. Nance, Eds., IEEE, Piscataway, NJ, 499-503.

Grant, E.M. and D.W. Starks (1988), "A Tutorial on TESS: The Extended Simulation Support System," In *Proceedings of the 1988 Winter Simulation Conference*, M.A. Abrams, P.L. Haigh, and J.C. Comfort, Eds., IEEE, Piscataway, NJ, 136-140.

Habermann, A.N. and D. Notkin (1986), "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering SE-12*, 12, 1117-1127.

Hansen, R.H. (1984), "The Model Generator: A Crucial Element of the Model Development Environment," Technical Report SRC-85-004, Systems Research Center, Virginia Tech, Blacksburg, VA.

Harel, D. (1992), "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer 25*, 1, 8-20.

Harmon, P. and D. King (1985), *Expert Systems, Artificial Intelligence in Business*, John Wiley, New York.

Henderson, P.B. and D. Notkin (1987), "Integrated Design and Programming Environments," *Computer 20*, 11, 12-16.

Hillston, J., R. Pooley, and N. Stevenson (1990), "An Experimentation Facility within the Integrated Modelling Support Environment," Preliminary Paper, Department of Computer Science, University of Edinburgh, U.K.

Humphrey, M.C. (1985), "The Command Language Interpreter for the Model Development Environment: Design and Implementation," Technical Report SRC-85-011, Systems Research Center, Virginia Tech, Blacksburg, VA.

Hurrion, R.D. (1986), "Visual Interactive Modeling," *European Journal of Operational Research 23*, 3, 281-287.

Hurrion, R.D. and R.J. Secker (1978), "Visual Interactive Simulation: An Aid to Decision Making," *Omega 6*, 5, 419-426.

Hurrion, R.D. (1989), "Graphics and Interaction," In *Computer Modelling for Discrete Simulation*, M. Pidd, Ed., John Wiley and Sons, Chichester, England, 101-119.

Johnson, S.C. (1983), "YACC: Yet Another Compiler-Compiler," In *The UNIX Programmer's Manual, Volume 2*, Bell Telephone Laboratories, Inc., Murray Hill, NJ.

Kiviat, P.J. (1969), "Digital Computer Simulation: Computer Programming Languages," Memorandum RM-5883-PR, The Rand Corporation, Santa Monica, CA.

Klahr, P. (1986), "Expressibility in ROSS, an Object-Oriented Simulation System," In *Proceedings of the Conference on AI Applied to Simulation*, SCS, San Diego, CA, 136-139.

Korson, T. and J.D. McGregor (1990), "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM 33*, 9, 40-60.

Kreutzer, W. (1986), *System Simulation: Programming Styles and Languages*, Addison-Wesley, Reading, MA.

Knuth, D.E. (1973), *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA.

Kurose, J.F., K.J. Gordon, R.F. Gordon, E.A. MacNair, P.D. Welch (1986), "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," In *Proceedings of the 1986 Fall Joint Computer Conference*, Dallas, 719-728.

Lackner, M.R. (1964), "Digital Simulation and System Theory," Technical Paper SP-1612, System Development Corporation, Santa Monica, CA.

Lehman, A., B. Knodler, E. Kwee, and H. Szczerbicka (1986), "Dialog-Oriented and Knowledge-Based Modeling in a Typical PC Environment," In *Proceedings of the Conference on AI Applied to Simulation*, SCS, San Diego, CA, 91-96.

Lesk, M.E. and E. Schmidt (1983), "Lex: A Lexical Analyzer Generator," In *The UNIX Programmer's Manual, Volume 2*, Bell Telephone Laboratories, Inc., Murray Hill, NJ.

Mathewson, S.C. (1984a), "The Application of Program Generator Software and its Extensions to Discrete Event Simulation Modeling," *IIE Transactions 16*, 1, 3-18.

Mathewson, S.C. (1984b), "Simulation Program Generators," *Simulation 23*, 6, 181-189.

Mathewson, S.C. (1985), "Simulation Program Generators: Code and Animation on a P.C.," *Journal of the Operational Research Society 36*, 7, 583-589.

Mathewson, S.C. (1989), "Simulation Support Environments," In *Computer Modelling for Discrete Simulation*, M. Pidd, Ed., John Wiley and Sons, Chichester, England, 57-100.

McArthur, D., P. Klahr, and S. Narain (1984), "ROSS: An Object-Oriented Language for Constructing Simulations," Technical Report R-3160-AF, The Rand Corporation, Santa Monica, CA.

McFall, M.E. and P. Klahr (1986), "Simulation with Rules and Objects,"In *Proceedings of the 1986 Winter Simulation Conference*, J.R. Wilson, J.O. Henriksen, S.D. Roberts, Eds. IEEE, Piscataway, NJ, 470-473.

Meyer, B. (1987), "Reusability: The Case for Object-Oriented Design," *IEEE Software 4*, 2, 50-64.

Moose, R.L., Jr. (1983), "Proposal for a Model Development Environment Command Language Interpreter," Technical Report SRC-85-012, Systems Research Center, Virginia Tech, Blacksburg, VA.

Moose, R.L., Jr. and R.E. Nance (1987), "Model Analysis in a Model Development Environment," Technical Report SRC-87-010, Systems Research Center, Virginia Tech, Blacksburg, VA.

Murray, K.J. and S.V. Sheppard (1987), "Automatic Model Synthesis: Using Automatic Programming and Expert Systems Techniques Toward Simulation Modeling," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 534-543.

Murray, K.J. and S.V. Sheppard (1988), "Knowledge-based Simulation Model Specification," *Simulation 50*, 3, 112-119.

Nance, R.E. (1981a), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24*, 4, 173-179.

Nance, R.E. (1981b), "Model Representation in Discrete-Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Nance, R.E. (1987), "The Conical Methodology: A Framework for Simulation Model Development," In *Proceedings of the Conference on Methodology and Validation*, O. Balci, Ed. SCS, San Diego, CA, 38-43.

Nance, R.E. and J.D. Arthur (1988), "The Methodology Roles in the Realization of a Model Development Environment," In *Proceedings of the 1988 Winter Simulation Conference*, M.A. Abrams, P.L. Haigh, and J.C. Comfort, Eds. IEEE, Piscataway, NJ, 220-225.

Nance, R.E. and O. Balci (1987), "Simulation Model Management Objectives and Requirements," In *Systems and Control Encyclopedia: Theory, Technology, Applications*, M.G. Singh, Ed., Pergamon Press, Oxford, 4328-4333.

Nance, R.E. and C.M. Overstreet (1986), "Diagnostic Assistance Using Digraph Representations of Discrete-Event Simulation Model Specifications," Technical Report SRC-86-001, Systems Research Center, Virginia Tech, Blacksburg, VA.

Nance, R.E. and C.M. Overstreet (1987), "Diagnostic Assistance Using Digraph Representations of Discrete-Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation 4*, 1, 33-57.

Nielsen, J. (1990), "Traditional Dialogue Design Applied to Modern User Interfaces," *Communications of the ACM 33*, 10, 109-118.

Nielsen, J. (1992), "The Usability Engineering Life Cycle," *Computer 25*, 3, 12-22.

Nielsen, N.R. (1986), "Knowledge-based Simulation Programming," In *Proceedings of the 1986 National Computer Conference*, AFIPS Press, Reston, VA, 125-134.

Nielson, G.M. (1991), "Visualization in Scientific and Engineering Computation," *Computer 24*, 9, 58-66.

O'Keefe, R.M. (1987), "What is Visual Interactive Simulation? (And is There a Methodology for Doing it Right?)," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 461-464.

O'Keefe, R.M. and J.W. Roach (1987), "Artificial Intelligence Approaches to Simulation," *Journal of the Operational Research Society 38*, 8, 713-722.

Ozden, M.H. (1991), "Graphical Programming of Simulation Models in an Object-Oriented Environment," *Simulation 56*, 2, 104-116.

Overstreet, C.M. (1982), "Model Specification and Analysis for Discrete-Event Simulation," Ph.D. Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Overstreet, C.M. and R.E. Nance (1983), "Graph-Based Diagnosis of Discrete-Event Model Specifications," Technical Report SRC-85-003, Systems Research Center, Virginia Tech, Blacksburg, VA.

Overstreet, C.M. and R.E. Nance (1985), "A Specification Language to Assist in Analysis of Discrete-Event Simulation Models," *Communications of the ACM 28*, 2, 190-201.

Overstreet, C.M. and R.E. Nance (1986), "World View Based Discrete-Event Model Simplification," In *Modelling and Simulation Methodology in the Artificial Intelligence Era*, M.S. Elzas, T.I. Oren, and B.P. Zeigler, Eds. North-Holland, Amsterdam, 165-179.

Overstreet, C.M., R.E. Nance, O. Balci, and L.F. Barger (1986), "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report SRC-87-001,

Systems Research Center, Virginia Tech, Blacksburg, VA.

Page, E.H. (1990), "Model Generators: Prototyping Simulation Model Definition, Specification, and Documentation under the Conical Methodology," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Page, E.H. and R.E. Nance (1989), "Model Generators: An Example of Evolutionary Prototyping," In *Proceedings of the 1989 Virginia Computer User's Conference*, Blacksburg, VA, 19-28.

Paul, R.J. (1989), "Visual Simulation: Seeing is Believing?" In *Impacts of Recent Computer Advances on Operations Research*, R. Sharda, B.L. Golden, E. Wasil, O. Balci, and W. Stewart, Eds. Elsevier Science Publishing, New York, NY, 422-432.

Peterson, J.L. (1977), "Petri Nets," *Computing Surveys 9*, 3, 223-252.

Pidd, M. (1984), *Computer Simulation in Management Science*, John Wiley, New York.

Poorte, J.P. and D.A. Davis (1989), "Computer Animation with CINEMA," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds., IEEE, Piscataway, NJ, 147-154.

Puthoff, F.A. (1991), "The Model Analyzer: Prototyping the Diagnosis of Discrete-Event Simulation Model Specifications," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Reddy, R., M.S. Fox, N. Husain, and M. McRoberts (1986), "The Knowledge-Based Simulation System," *IEEE Software 3*, 2, 26-37.

Reps, T. and T. Teitelbaum (1984), "The Synthesizer Generator," *ACM SIGPLAN Notices 19*, 5, 42-48.

Roberts, D.S. and M.A. Flanigan (1989), "Simulation Modeling and Analysis with INSIGHT: A Tutorial," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds., IEEE, Piscataway, NJ, 291-300.

Rohrbough, M.C. (1989), "Introduction to SIMFACTORY II.5," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds., IEEE, Piscataway, NJ, 201-204.

Rothenberg, J. (1988), "Knowledge-Based Simulation at Rand," *Simuletter 19*, 2, 54-59.

Shub, C.M. (1980), "Discrete-Event Simulation Languages," In *Proceedings of the 1980 Winter Simulation Conference 2*, T.I. Oren, C.M. Shub, and P.F. Roth, Eds. IEEE, Piscataway, NJ, 107-124.

Sommerville, I. (1989), *Software Engineering*, Addison-Wesley, Reading, MA.

Standridge, C.R. (1986), "Animating Simulations Using TESS," *Computers and Industrial Engineering 10*, 2, 121-134.

Stenning, V. (1986), "An Introduction to ISTAR," In *Software Engineering Environments*, I. Sommerville, Ed. Peter Peregrinus Ltd., London, 1-22.

Sun Microsystems (1986a), *SunINGRES*, Volumes I, II, and III, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1986b), *SunINGRES/EQUEL/C*, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1988), *SunView Programmer's Guide*, Sun Microsystems, Inc., Mountain View, CA.

Taylor, R.N., F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young (1989), "Foundations for the Arcadia Environment Architecture," *ACM SIGPLAN Notices 24*, 2, 1-13.

Teitelman, W. (1985), "A Tour Through Cedar," *IEEE Transactions on Software Engineering SE-11*, 3, 285-302.

Tocher, K.D. (1965), "Review of Simulation Languages," *Operational Research Quarterly 16*, 2, 189-217.

Tocher, K.D. and D.G. Owen (1961), "The Automatic Programming of Simulations," In *Proceedings of the Second International Conference on Operational Research*, John Wiley, New York, NY, 50-68.

Unger, B., G. Birtwistle, J. Cleary, D. Hill, G. Lomow, R. Neal, M. Peterson, I. Witten, and B. Wyvill (1983), "JADE: A Simulation and Software Prototyping Environment," Research Report 83/133/22, Department of Computer Science, University of Calgary, Alberta, Canada.

Unger, B., A. Dewar, J. Cleary, and G. Birtwistle (1986a), "A Distributed Software Prototyping and Simulation Environment: JADE," In *Proceedings of the Conference on Intelligent Simulation Environments*, SCS, San Diego, CA, 63-71.

Unger, B., A. Dewar, J. Cleary, and G. Birtwistle (1986b), "The JADE Approach to Distributed Software Development," In *Proceedings of the Conference on AI Applied to Simulation*, SCS, San Diego, CA, 178-188.

Vujosevic, R. (1990), "Object Oriented Visual Interactive Simulation," In *Proceedings of the 1990 Winter Simulation Conference*, O. Balci, R.P. Sadowski, and R.E. Nance, Eds., IEEE, Piscataway, NJ, 490-498.

Wallace, J.C. (1985), "The Control and Transformation Metric: A Basis for Measuring Model Complexity," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Wallace, J.C. and R.E. Nance (1985), "The Control and Transformation Metric: A Basis for Measuring Model Complexity," Technical Report SRC-85-007, Systems Research Center, Virginia Tech, Blacksburg, VA.

Wasserman, A.I. and P.A. Pircher (1987), "A Graphical, Extensible Integrated Environment for Software

Development," *ACM SIGPLAN Notices 22*, 1, 131-142.

Webster's Encyclopedic Unabridged Dictionary of the English Language, Portland House, New York, 1989.

White, E. (1990), "Object-Oriented Programming as a Programming Style," *The C Users Journal 8*, 2, 43-57.

Whitner, R.B. (1988), "A Taxonomical Review of Software Verification Techniques: An Illustration using Discrete-Event Simulation," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Whitner, R.B. and O. Balci (1989), "Guidelines for Selecting and Using Simulation Model Verification Techniques," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, P. Heidelberger, Eds. IEEE, Piscataway, NJ, 559-568.

Winkler, D. and S. Kamins (1990), *HyperTalk 2.0 The Book*, Bantam Books, New York, N.Y.

Zeigler, B.P. (1976), *Theory of Modeling and Simulation*, John Wiley and Sons, New York.

Zeigler, B.P. (1984), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, New York.

VITA

Name: Emory Joseph Derrick

Address: 2719 Newton Court
Blacksburg, VA 24060

Phone: (703) 953-2000

Birthdate: 17 January 1952

Marital Status: Married (Ruth), three children (Evan, Collin, Jillian)

Education:

Ph.D.- Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
1992

M.S.- Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
1988

B.S.- Electrical Engineering
United States Naval Academy
Annapolis, MD
1974

Mr. Derrick has been an instructor for the Department of Computer Science for the complete 1991-92 academic session. Except for the current academic session, he has been employed as a teaching and research assistant by the Department of Computer Science and the Systems Research Center since 1985. Before 1985, he served as an officer in the U.S. Navy and as an engineer at the Naval Sea Systems Command. Mr. Derrick has been a student member of the Association for Computing Machinery, The IEEE Computer Society, and The Society for Computer Simulation. He is a member of Upsilon Pi Epsilon, the national Computer Science honor society.