DOMINO: A Multifaceted Conceptual Framework for Visual Simulation Modeling*

E. Joseph Derrick and Osman Balci

TR 92-43

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 17, 1992

**Cross-listed as Systems Research Center report SRC-92-007.*

ABSTRACT

The purpose of this paper is to present a new conceptual framework for visual simulation modeling. The need for computer-aided conceptual assistance for visual simulation modeling of complex systems is undeniable. A multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO) has been developed. The DOMINO: (1) provides both design and implementation guidance to furnish a broad range of support during the model development life cycle; (2) enables the modeler to work under the object-oriented paradigm; (3) guides the modeler in graphically structuring a visual simulation model at multiple levels of abstraction; (4) enables the extraction of sufficient information from the modeler so that the model execution can be visualized; (5) embodies a WYSIWYR (What You See Is What You Represent) philosophy and enables the modeler to represent a system as it is naturally perceived; (6) adheres to the principles of the Conical Methodology; (7) contains a rich and expressive terminology applicable for any discrete-event simulation problem domain; (8) differentiates a model component having a representation in the system from the one that does not have a representation in the system; (9) enables the extraction of sufficient information from the modeler so that the model specification can be automatically translated into executable code following the automation-based software paradigm; and (10) enables the creation of a model specification which lends itself for formal diagnostic testing.

CR Categories and Subject Descriptors: I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event, Visual;* D.2.2 [**Software Engineering**]: Tools and Techniques—*Computer-aided software engineering*

Additional Key Words and Phrases: Animation, conceptual frameworks for visual simulation modeling, visual simulation, visual simulation model development.

TABLE OF CONTENTS

Page

ABSTRACT ii

1. INTRODUCTION 1

2. THE DOMINO DESIGN OBJECTIVES 2

3. MODEL COMPOSITION AND GENERAL STRUCTURE 4

    3.1 Static Structure 4

    3.2 Dynamic Structure 7

    3.3 Movement Among Model Structures 10

4. FEATURES FOR VISUAL SIMULATION MODELING 11

    4.1 Graphical Orientation 11

        4.1.1 Layouts, Paths, and Connectors 13
        4.1.2 Creation of Layout and Component Images 13
        4.1.3 Class Layout Creation and Layout Definition 14
        4.1.4 Requirements for Layout Configurations 20
        4.1.5 Top-Down Definition and Hierarchical Traversal 20

    4.2 Object Orientation 21

    4.3 Multi-view Specification of Model Component Logic 22

        4.3.1 Types of Logic Specification 22
        4.3.2 Implementation Views 23
        4.3.3 Implications of Compositional Equivalence 24

5. EVALUATION 24

6. CONCLUSIONS 30

ACKNOWLEDGEMENTS 31

REFERENCES 32

1. INTRODUCTION

The fundamental human limitation, the *Hrair Limit,* indicates that a human being cannot handle more than 7 ± 2 entities *simultaneously* [Miller 1956]. Consequently, the need for computer-aided conceptual assistance for modeling and simulating a complex stochastic system containing hundreds or thousands of simultaneous and random entities is undeniable. Although the Hrair Limit has been known since 1956, its implications for model development have not been fully recognized.

The commercial simulation programming languages (SPLs) widely available today focus on providing assistance in the programming process which is only one of the 10 processes of the life cycle of a simulation study [Balci 1990]. The conceptual frameworks (CFs) underlying these SPLs (e.g., event scheduling, process interaction, transaction processing) [Balci 1988; Derrick et al. 1989; Nance 1981a] were developed in the 1960s for simulation *programming* purposes too. The commercial object-oriented SPLs also support only the programming process and do not provide the much needed high-level conceptual guidance for designing a visual simulation model.

Under the current practice, an SPL is chosen and the simulation model is *designed* under the CF of that SPL. Regrettably, such an approach does *not* recognize model design as a distinct phase. Thus, the modeler is forced to work under an implementation-level CF *during the design phase.* This practice contributes to an error-prone model design, unstructured and difficult to understand especially for large and complex systems. Errors induced within the model design either surface in later phases, resulting in a higher cost of correction, or do never become visible resulting in the type II error—the error of accepting invalid model results [Balci and Sargent 1981].

In the 1990s, simulation programming should not be a major concern for the modeler, i.e., the executable model should be automatically generated from a model specification. The modeler should specify a simulation model using high-level concepts under a CF. The much needed shift in emphasis from simulation programming to simulation modeling is long overdue [Nance 1984].

We have conducted a detailed study, a survey, and a comparison of the CFs for simulation modeling [Balci 1988; Derrick 1988, 1992; Derrick et al. 1989]. We have developed a Visual Simulation Support Environment (VSSE) [Derrick 1992; Derrick and Balci 1992a] under the multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO). We have also created a Visual Simulation Model Specification Language (VSMSL) [Derrick 1992; Derrick and Balci 1992b] under the DOMINO. This related work is not described herein in order not to prolong the length of this paper. The reader is recommended to study the above references to obtain the related information.

The design objectives of the DOMINO are introduced in Section 2. The DOMINO model composition and general structure are described in Section 3. The graphical orientation, object

orientation, and multi-view model component logic specification features of the DOMINO are all presented in Section 4. Section 5 contains the evaluation of the DOMINO. Conclusions are given in Section 6.

2. THE DOMINO DESIGN OBJECTIVES

This section delineates the design objectives of the new CF in no particular order.

Objective 1: Provide both design and implementation guidance to furnish a broad range of support during the model development life cycle.

Converting a model representation created under one CF (e.g., object oriented) into another under a different CF (e.g., event scheduling) results in an error-prone model development process. Therefore, the same CF should support all phases of the model development life cycle so that the conceptual mapping from one model representation into another is straightforward.

Objective 2: Enable the modeler to work under the object-oriented paradigm.

The object-oriented paradigm with concepts such as encapsulation, classes, inheritence, polymorphism, and dynamic binding supports model development, from high-level design specification to detailed implementation, throughout the entire life cycle. It assists the modeler to achieve the following software quality characteristics: modularity, information hiding, weak coupling, strong cohesion, extensibility, maintainability, reusability, and modifiability.

Objective 3: Guide the modeler in graphically structuring a visual simulation model at multiple levels of abstraction.

The modeler should use graphical specification as much as possible since it is a natural and easy way of describing a system. The graphical model specification should be done: (a) at multiple levels of abstraction, (b) following a hierarchical and functional decomposition, and (c) using the stepwise refinement principle. The graphical model specification prepares the model for visualization.

Objective 4: Enable the extraction of sufficient information from the modeler so that the model execution can be visualized.

Since the CF is intended for visual simulation, extra information needs to be extracted from the modeler for visualization. The information should be extracted in an efficient manner with a well human engineered interface.

Objective 5: Embody a WYSIWYR (What You See Is What You Represent) philosophy and enable the modeler to represent a system as it is naturally perceived.

The modeler should not be coerced to contort his/her own modeling view. For example, consider a machine repairman problem where a repairman services dozens of randomly failing machines that are attached to the production floor and are unmovable. In modeling this problem under the GPSS CF, the repairman would be defined as a facility and the machines would be represented as

transactions that would queue up in front of the "SEIZE RMAN" GPSS block where repairman (RMAN) would be captured whenever available. Although this produces a working and efficient model, the model representation is not conceptually true to the real system. The 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 s/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 contributes to an error-prone model design, unstructured and difficult to understand especially for large and complex systems.

Objective 6: Adhere to the principles of the Conical Methodology [Nance 1987].

The Conical Methodology advocates the following principles: abstraction, concurrent documentation, functional decomposition, hierarchical decomposition, information hiding, iterative refinement, life-cycle verification, progressive elaboration, separation of concerns, and stepwise refinement.

Objective 7: Contain a rich and expressive terminology applicable for any discrete-event simulation problem domain.

Applicable only for a particular problem domain (e.g., manufacturing systems, local area networks, factory planning), commercial software products exist (e.g., Simfactory, Lannet, Comnet, Network), based on an SPL (e.g., Simscript, ModSim), with a graphical front-end for model specification. However, such a product employs a *different* CF which is specially created for the problem domain the product is applicable. The utility of these products is domain dependent and the modelers are required to learn a new software product for each problem domain. Therefore, our objective is to develop a CF that is general-purpose, applicable for any problem domain within the discrete event simulation discipline.

Objective 8: Differentiate a model component having a representation in the system from the one that does not have a representation in the system.

A model is defined as "a representation of an object, system, or idea" [Shannon 1975, p. 4]. However, some essential components of a model such as a random number generator, random variate generators, statistical data collection routines, experimental design routines, visualization routines, and an event scheduling routine do not possess any representation in the system. Thus, a contradiction occurs with respect to the definition of a model. The CF should resolve this contradiction by employing an appropriate terminology.

Objective 9: Enable the extraction of sufficient information from the modeler so that the model specification can be automatically translated into executable code following the automation-based software paradigm [Balci and Nance 1987].

In the 1990s, simulation programming should not be a major concern for the modeler, i.e., the

executable model should be automatically generated from a model specification. The modeler should specify a simulation model using high-level concepts under a CF. The much needed shift in emphasis from simulation programming to simulation modeling is long overdue [Nance 1984].

Objective 10: Enable the creation of a model specification which lends itself for formal diagnostic testing [Nance and Overstreet 1987].

Errors induced within the model design specification either surface in later phases, resulting in a higher cost of correction, or do never become visible resulting in the type II error—the error of accepting invalid model results [Balci and Sargent 1981].

3. 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 static or dynamic.

A *real* model component has a direct correspondence to or a representation of a component in the system being modeled. Real model components are visualized.

A *virtual* model component does not represent anything in the system being modeled. Virtual model components are not visualized. Examples: statistics collection component, random variate generation component, and model startup component.

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 (Section 3.1) are static model components.

Dynamic model components are movable. The movement can be spatial, temporal, or logical as explained in Section 3.3. Dynamic objects, subdynamic objects, and base dynamic objects (Section 3.2) are dynamic 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.

Model static structure is the architecture of the model constructed by its static components.

Model dynamic structure is the architecture of the model constructed by its dynamic components.

A model can have only one static structure while it can have many dynamic structures depending on the number of dynamic components.

3.1 Static Structure

We carefully distinguish between the model static structure and a simple static structure. As mentioned above, only one model static structure exists 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 each of which containing zero or more static objects. The decomposed submodels can be further decomposed into other submodels each of which may or may not contain static objects. Through the hierarchical decomposition of submodels, multiple levels of abstraction can be achieved. A static object cannot be decomposed since it represents the *smallest* element of interest. The model static structure is, therefore, a hierarchy of potentially many simple static structures.

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 $SS_1,...,SS_m$, (each of these sets in turn is a simple static structure) and $n \geq 0$ static object nodes, $so_1,...,so_n$. (Figure 1)

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.

The decomposable/non-decomposable characteristic 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.

A *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 as a submodel or static object is based primarily on the expected need for a decomposition point in the model's static hierarchy. The greatest flexibility in development is retained by modeling the static system components as submodels. However, other considerations (see Section 4.1) related to visualization/animation requirements could dictate otherwise. The hierarchical decomposition, such as that associated with submodels, enables a modeler to break a large modeling problem into smaller, more manageable parts at multiple levels of abstraction. This enriches the framework and produces important implications for visualization.

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

Real submodels: air traffic control tower, combat zone, computing center, database query facility, electrical circuit, hospital, local area network, maintenance center, Navy base, parking lot, production control department, traffic intersection, warehouse, etc.

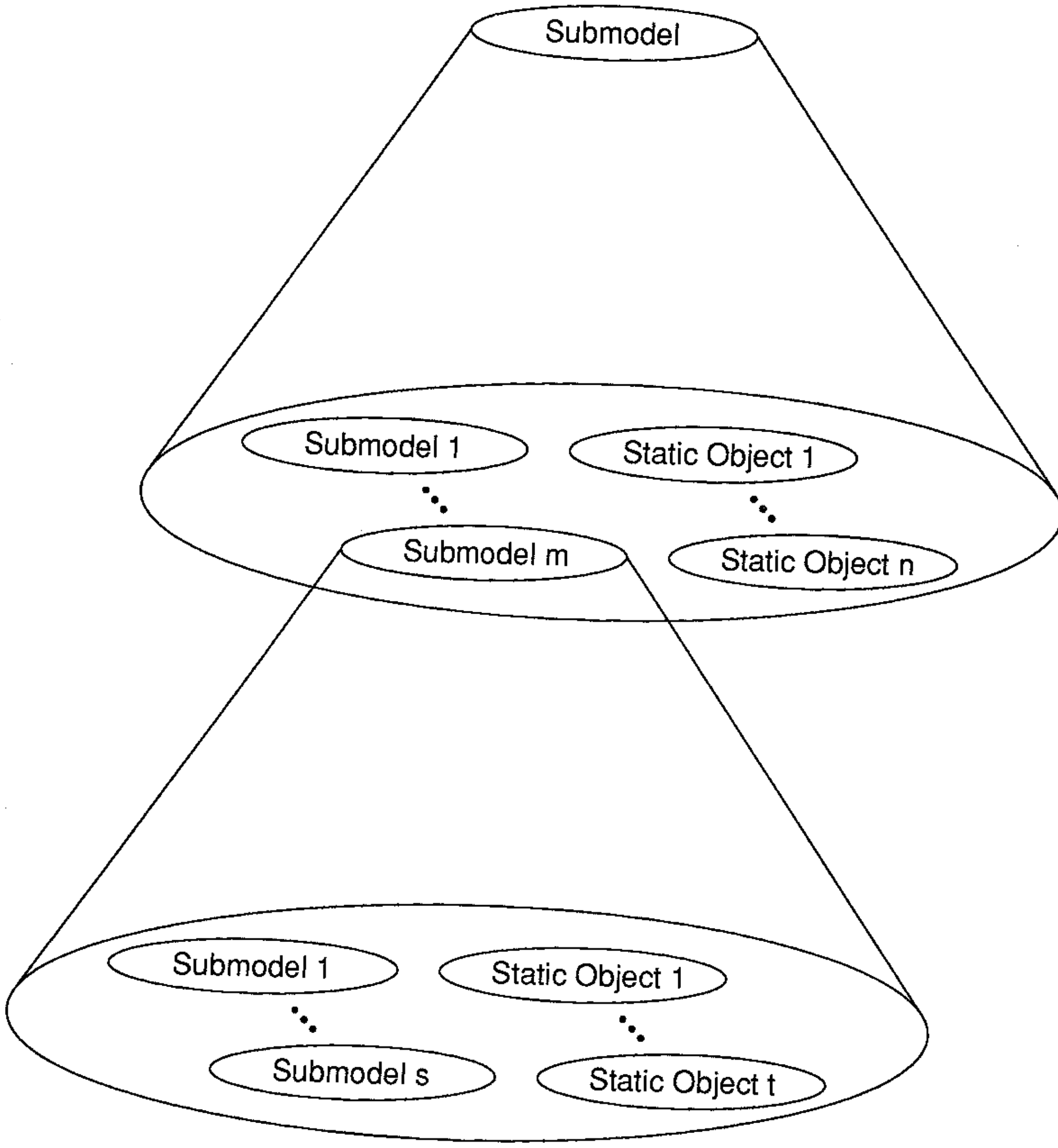

Figure 1. Simple Static Structure

Virtual submodels: database interface module, event scheduling module, experimental design module, graphical display module, model start-up module, output module, random number generator, random variate generator, statistical data collection module, etc.

Real static objects: central processing unit, desk, input/output unit, machine, printer, traffic light, etc.

Virtual static objects: compound logical condition, data file, data structure, logical record, memory buffer, etc.

The definition and specification of model components assume an object orientation and follows the object oriented paradigm. 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 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 constitute the 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 subdynamic objects can be further decomposed into zero or more subdynamic objects and zero or more base dynamic objects. Through hierarchical decomposition of dynamic objects, multiple levels of abstraction can be achieved for the model's dynamic structure as well. Like static objects, a base dynamic object cannot be decomposed since it represents the *smallest* element of interest in the dynamic structure. 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, modifying the definition of a tree [Knuth 1973]:

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, $bdo_1,\ldots,bdo_j$. (Figure 2)

The distinguishing characteristic 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, a simple dynamic structure has a subdynamic object at its root.

The dynamic model components are defined below:

Figure 2. Simple Dynamic Structure

A *dynamic object* is the dynamic model component which is the root of a model dynamic structure and is decomposable.

A *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.

A *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 are 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 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), they are relatively static. In a sense, each model dynamic structure has a structural kinship to the model static structure. The component hierarchies of a model's static structure and its dynamic structures are separate and distinct from the class inheritance hierarchies originating from the object-orientation of the DOMINO (Section 4.2).

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 is based primarily on the expected need for a decomposition point in the model's dynamic hierarchy. The greatest flexibility in development is retained by modeling the dynamic system components as subdynamic objects. However, other considerations (see Section 4.1) related to visualization/animation requirements could dictate otherwise. The hierarchical decomposition, such as that associated with dynamic objects, enables a modeler to break a large dynamic object (e.g., a Navy aircraft carrier) into smaller, more manageable parts at multiple levels of abstraction. This enriches the framework and produces important implications for visualization.

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

Real dynamic objects: aircraft carrier, automobile, bus, bicycle, message, missile, satellite, travelling repairman, transaction, war ship, 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 module to activate printing; a virtual dynamic object used by the system to create the initial model components which are present at simulation time zero, etc.

Real subdynamic objects: baggage compartment aboard an airplane, bus seat, car engine, flight deck of an aircraft carrier, wagon of a cargo train, etc.

Virtual subdynamic objects: submessages of a decomposed message (virtual dynamic object), 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.

Real base dynamic objects: engine of an aircraft, loading crane aboard a ship, nuclear head of a missile, radio of a car, 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.

3.3 Movement Among Model Structures

Only dynamic model components undergo "movement" which can be spatial, temporal, or logical, depending on the situation.

Spatial movement implies change of physical or logical location. Both *virtual* and *real* dynamic components can undergo spatial movement. Only *real* dynamic component spatial movement may be visualized during animation.

Examples: an aircraft moving into a war zone; a bus driver moving into a bus; a customer moving into a grocery store; the execution control point of a computer program moving through the various modules of the program logic; a transaction moving into a processing center; and a vehicle moving through a traffic intersection.

Temporal movement implies movement in time. Both *virtual* and *real* dynamic components can undergo temporal movement. Only *real* dynamic component temporal movement may be visualized during animation.

Examples: an automobile part moving from a warehouse into an assembly line; a bomber aircraft moving from one combat zone into another; a bus travelling from one city to another; and a mail message moving from one computer system into another.

All of the temporal movement examples above are assumed to cause the advancement of the simulation time. The spatial movement examples above may or may not advance the simulation time depending on the purpose of modeling.

Logical movement implies movement in the logical decision path of a dynamic model component. Both *real* and *virtual* dynamic components can undergo logical movements. Only *real* dynamic component logical movement may be visualized during animation.

Examples: a car moving into a car-wash facility if the capacity is not full; a customer moving into service area if one of six servers is idle; and a job moving into processor 1 with a probability of 0.3, into processor 2 with a probability of 0.2, and into processor 3 with a probability of 0.5.

For real dynamic objects, spatial moves are often associated with logical and temporal moves. Change of location of a dynamic object often occurs based on the satisfaction of a logical compound condition and causes the advancement of time. For virtual dynamic objects, spatial moves are often associated with logical moves (e.g., a virtual dynamic object moving into a virtual random variate generator submodel to retrieve a random variate) and sometimes with temporal moves (e.g., a virtual dynamic object, representing a message, consuming time to reach its destination).

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. They 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 (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.

A summary of the model composition and general structure terminology is presented in Table 1.

4. FEATURES FOR VISUAL SIMULATION MODELING

The graphical orientation, object orientation, and multi-view model component logic specification features of the DOMINO are described in this section.

4.1 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

Table 1. Model Composition and General Structure Terminology

Term	Definition
Real Model Component	Has a direct correspondence to or a representation of a component in the system being modeled. Visualized.
Virtual Model Component	Does not represent anything in the system being modeled. Not visualized.
Static Model Component	Model component which is not movable. Permanent.
Dynamic Model Component	Model component which is movable. Temporary or permanent.
Model Static Structure	Architecture of the model constructed by its static components. Only one model static structure exists with the model itself as the root.
Model Dynamic Structure	Architecture of the model constructed by its dynamic components.
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 model 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 dynamic object as its root in which the hierarchy is extended at points where interior subdynamic objects are decomposed.
Dynamic Object	Basis (root) of a model dynamic structure. Decomposable.
Subdynamic Object	The 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.
Spatial Movement	Implies change of physical or logical location. Both virtual and real dynamic components can undergo spatial movement. Only real dynamic component spatial movement may be visualized during animation.
Temporal Movement	Implies movement in time. Both virtual and real dynamic components can undergo temporal movement. Only real dynamic component temporal movement may be visualized during animation.
Logical Movement	Implies movement in the logical decision path of a dynamic model component. Both real and virtual dynamic components can undergo logical movements. Only real dynamic component logical movement may be visualized during animation.

discussed. Finally, the top-down definition of model component hierarchies and real object instantiation throughout the hierarchy is explained.

4.1.1 Layouts, Paths, and Connectors

The model 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; therefore, 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 must also 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.

4.1.2 Creation of Layout and Component Images

Before objects are instantiated on the layouts and before paths or connectors are created, the layout images must be drawn or created as shown in Figure 3. 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 layout images can be constructed by using digitized video or photo images, scanned images, paintings, and 2 or 3 dimensional drawings in color or black and white. The model object is allowed only one top level layout (Figure 3). 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. No identifiable model components exist within them.

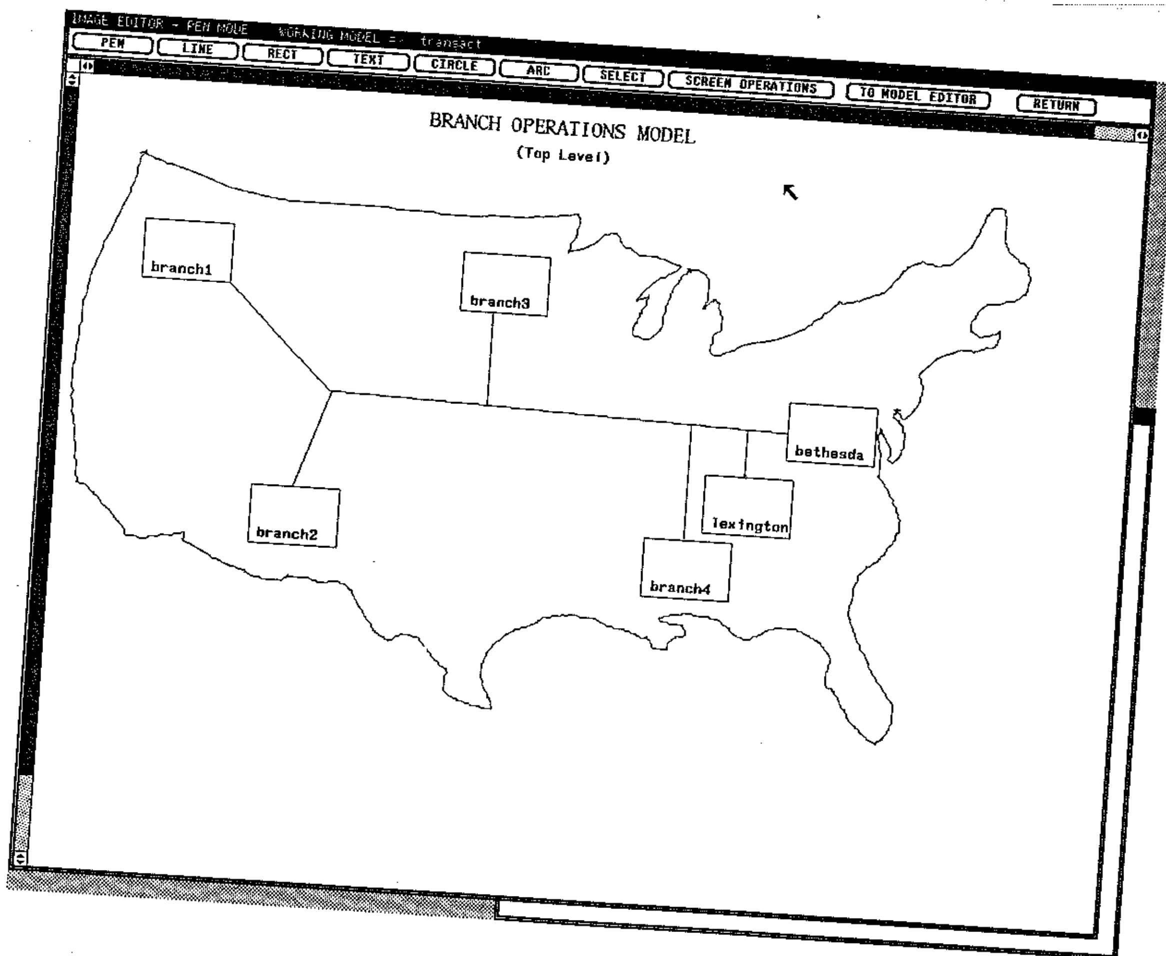

Figure 3. Top Level Layout Image

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.

4.1.3 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. 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 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). 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. 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 4 is the completed class layout definition of the top level for the Branch Operations Model shown in Figure 3. Note that the submodels are bounded by rectangles and are connected by paths. Any portion of the layout image can be defined as a submodel. Five submodels are identified in Figure 4, i.e., the model is decomposed into five submodels at the top level. Each of these submodels has a layout definition as shown in Figures 5-10. The Lexington submodel (Figure 10) is further decomposed into another submodel shown in Figure 11.

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 OOP, 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 5 through 8 depict the set of class layouts and definitions for the submodel class branch for 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 4.3.

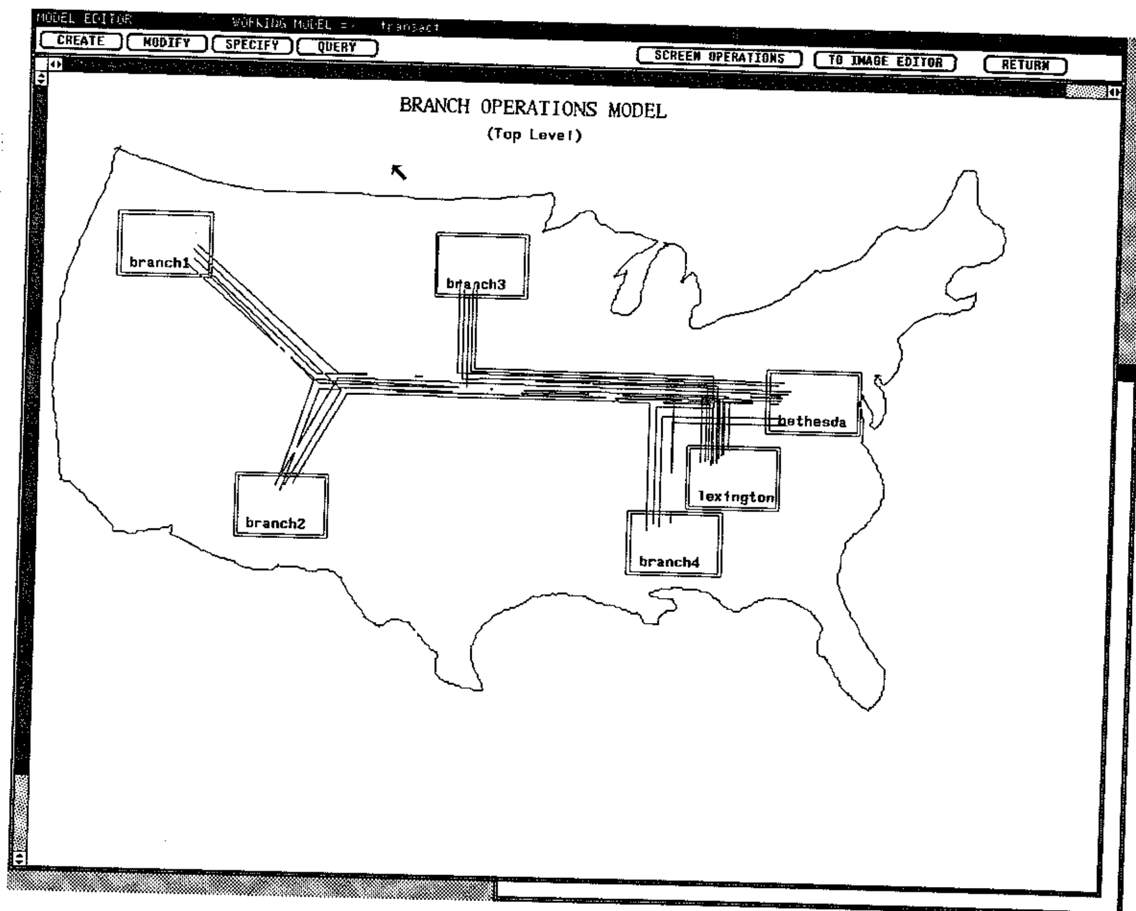

Figure 4. Top Level Layout Definition

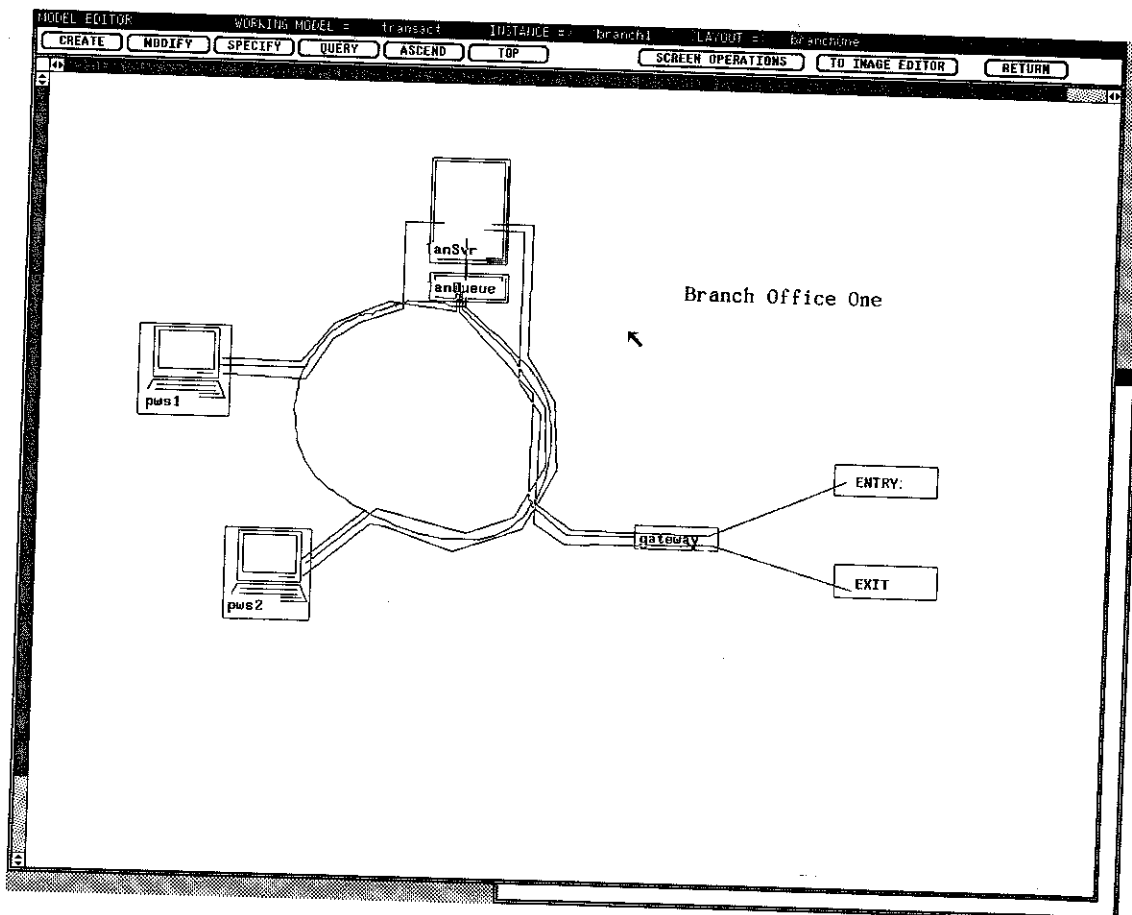

Figure 5. Layout Definition of Branch One

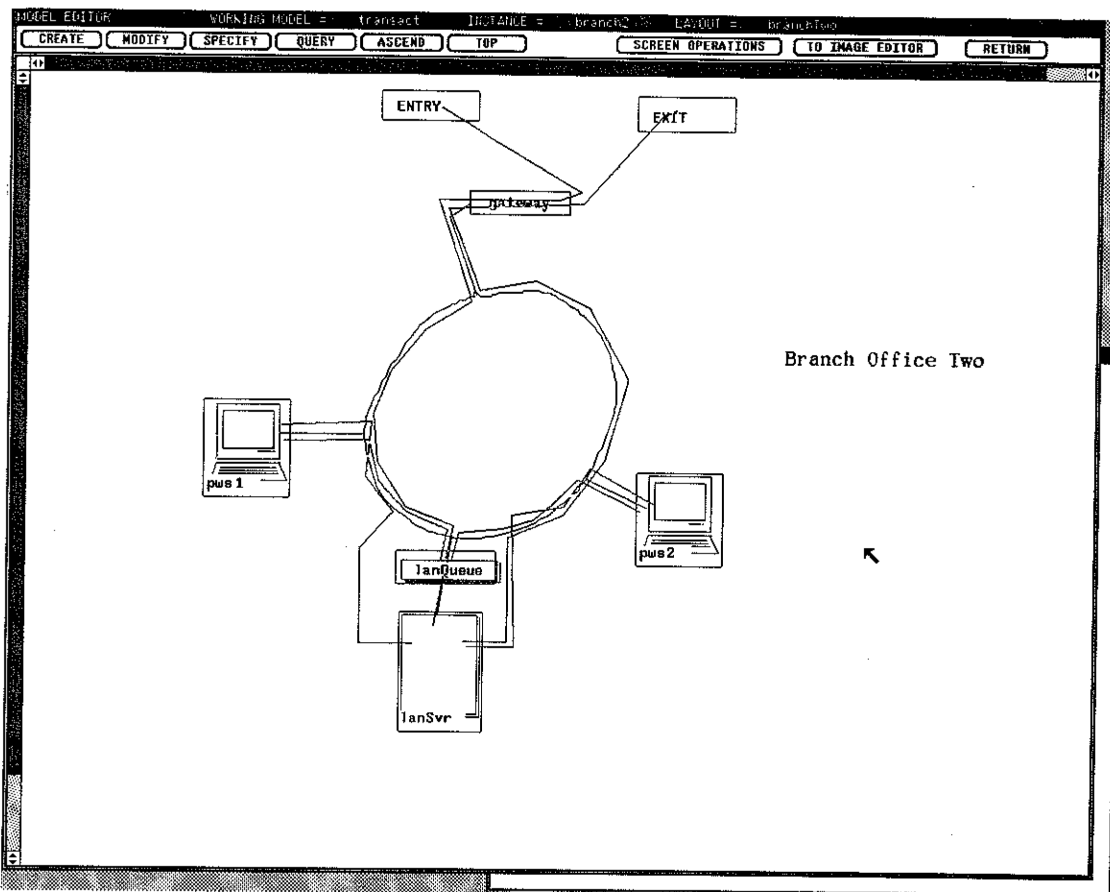

Figure 6. Layout Definition of Branch Two

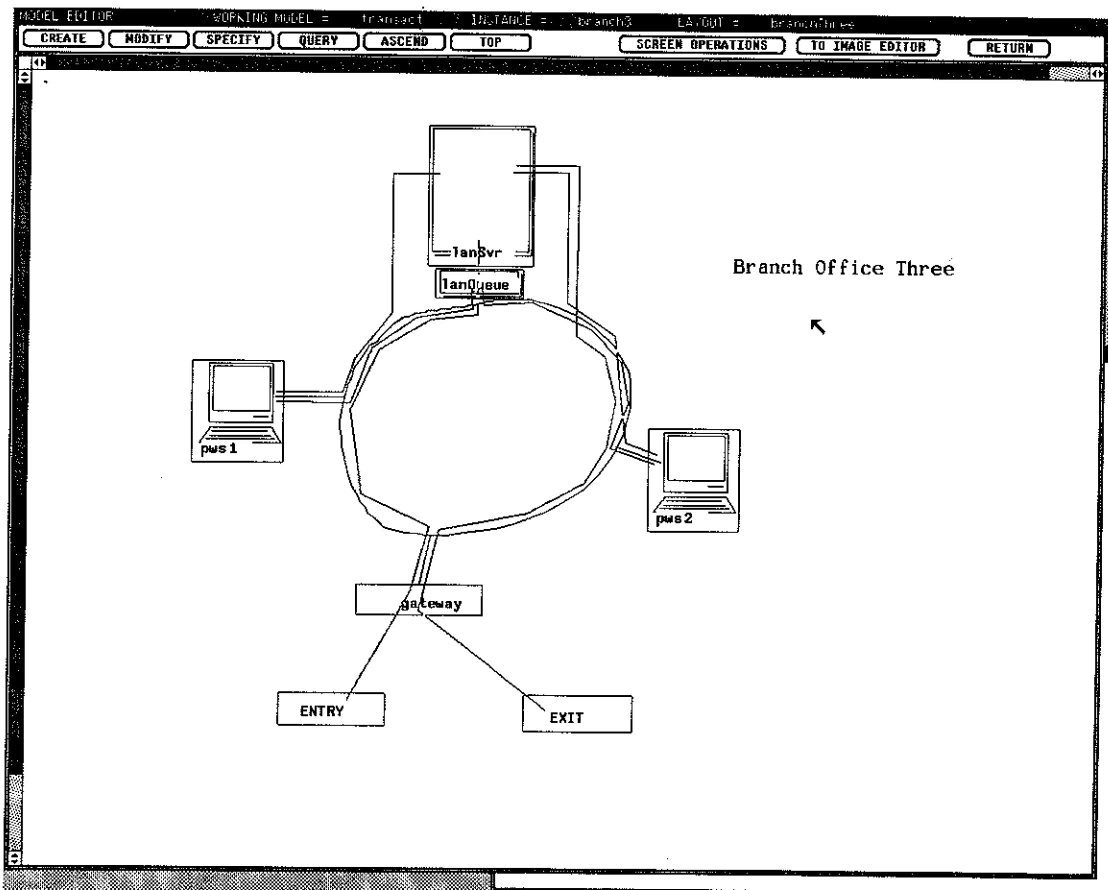

Figure 7. Layout Definition of Branch Three

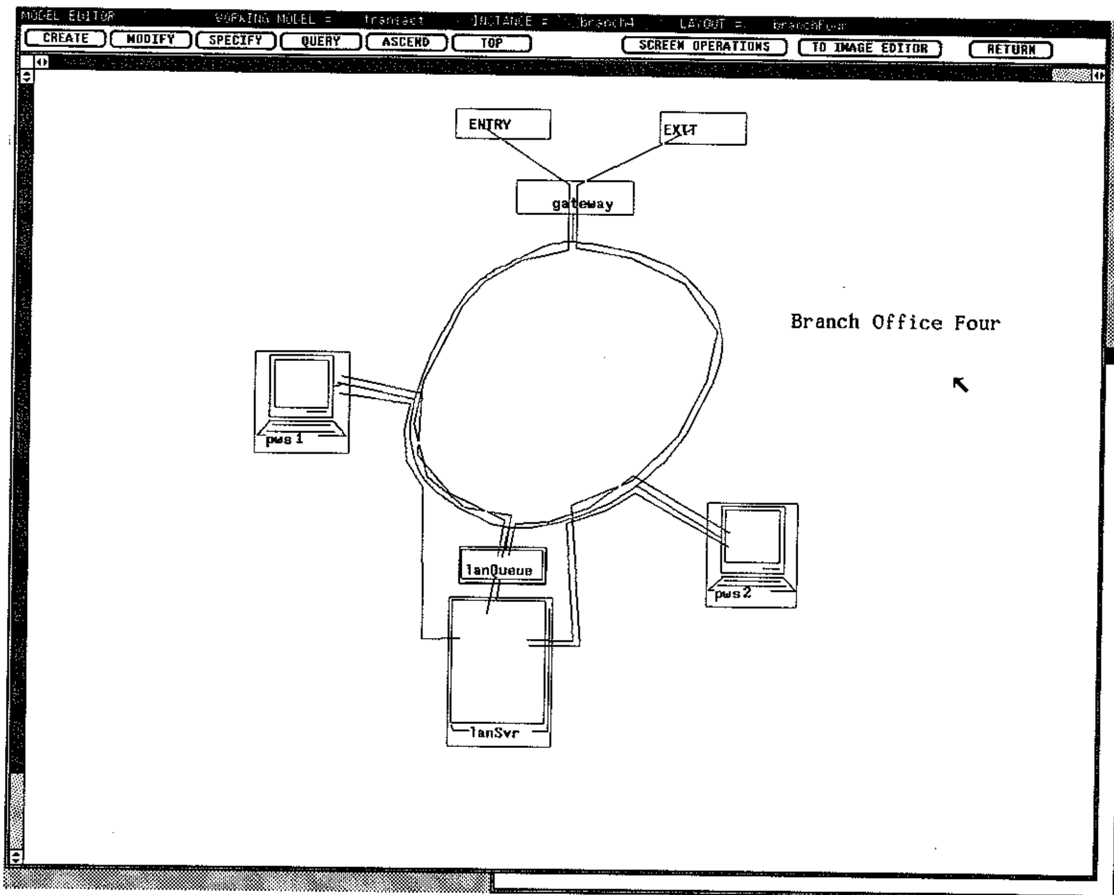

Figure 8. Layout Definition of Branch Four

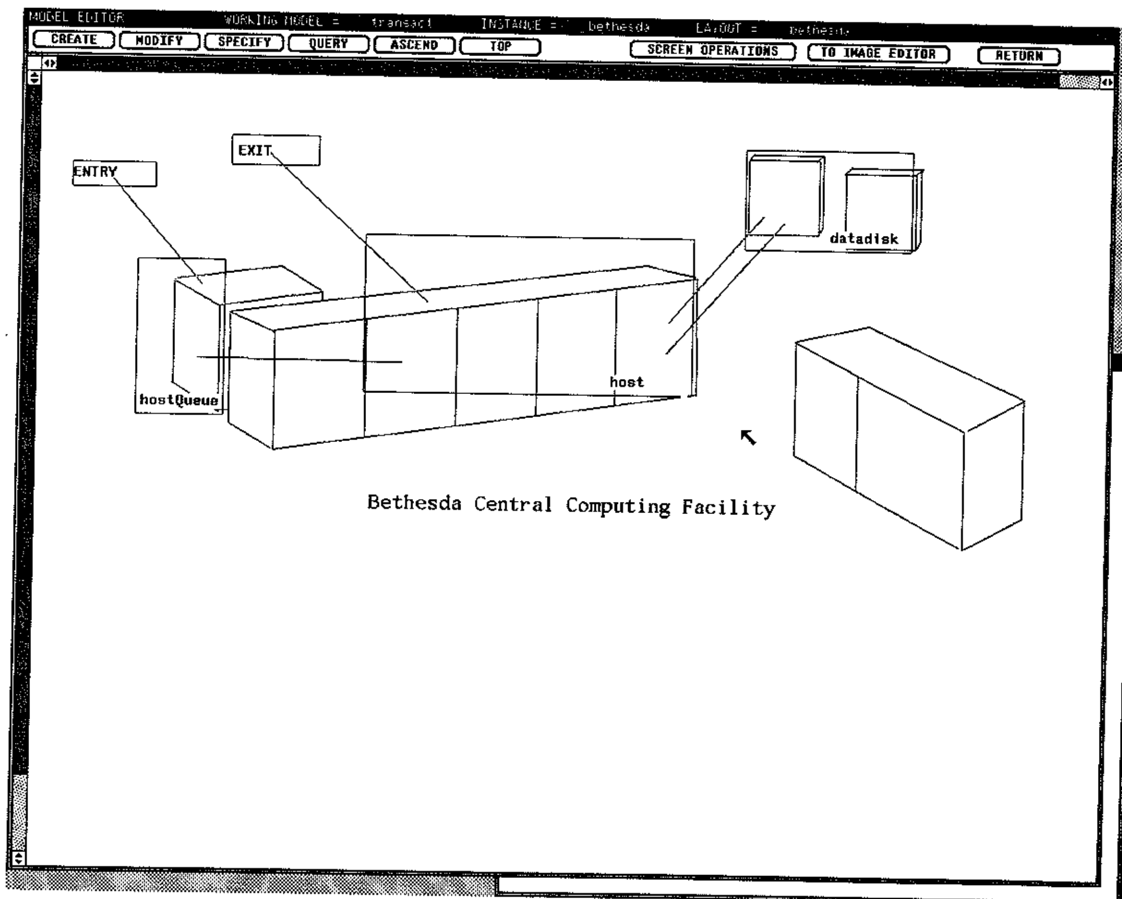

Figure 9. Layout Definition of Bethesda Central Computing Facility

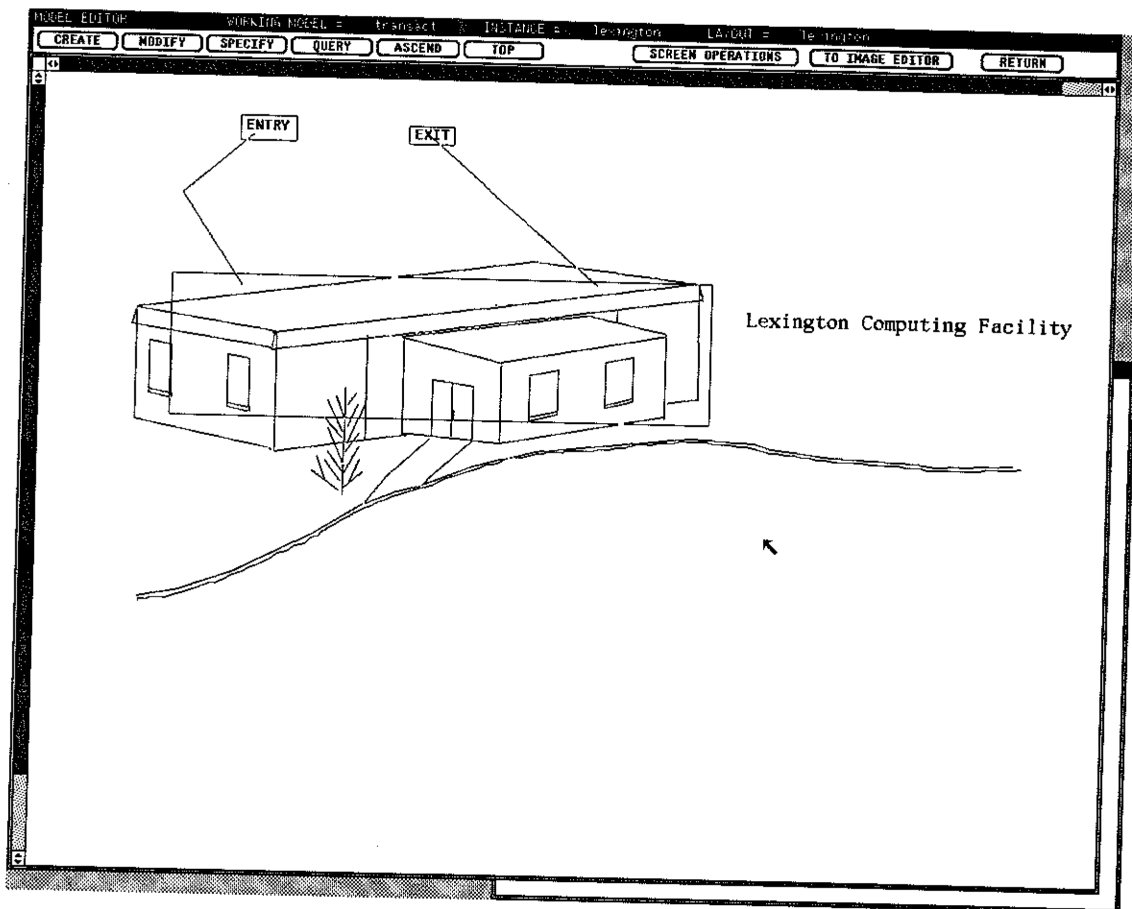

Figure 10. Layout Definition of Lexington Computing Facility

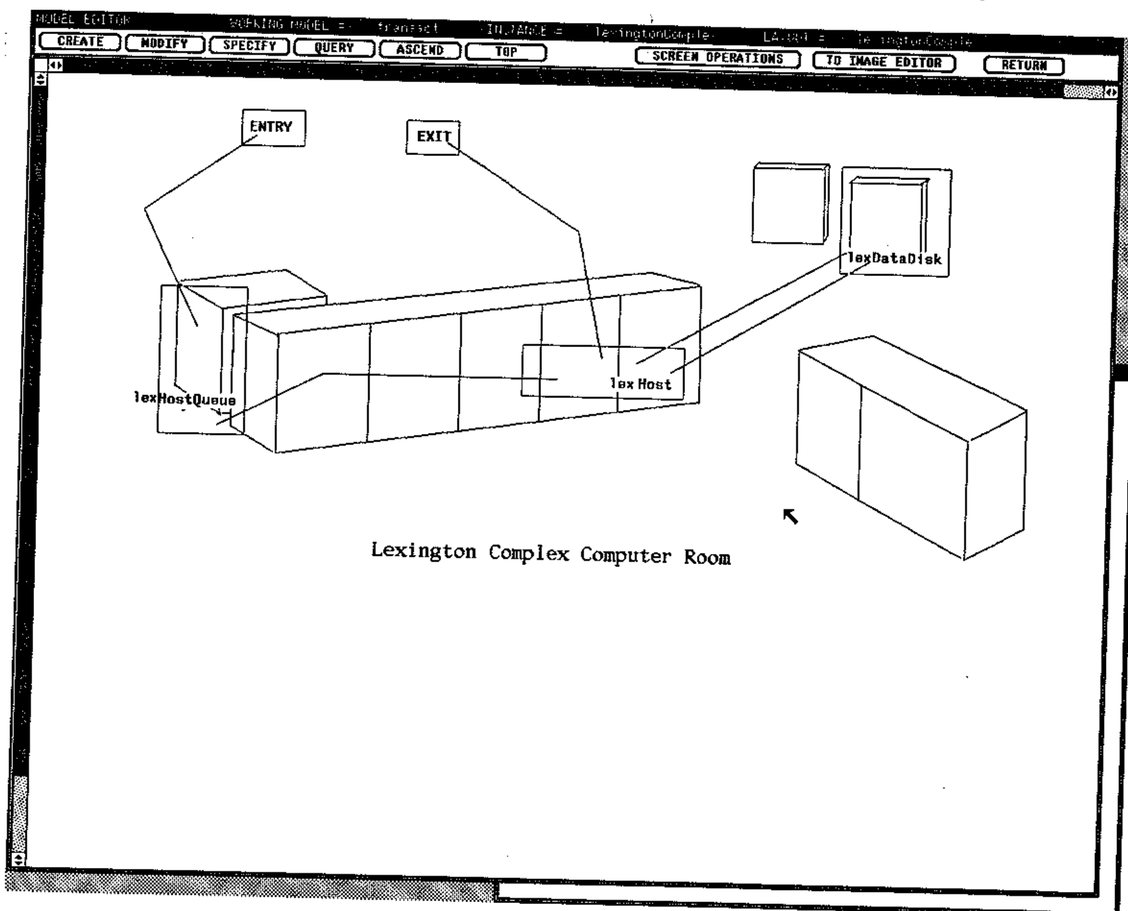

Figure 11. Layout Definition of Lexington Computer Room

4.1.4 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:

① A submodel or a subdynamic object can have only one entry point and only one exit point.

② Submodels and interaction points can share the entry or exit points of other model components (e.g., two submodels can have the same entry point in a layout; two interaction points can have the same exit point in a layout; a submodel and an interaction point can have the same entry or exit point in a layout.)

③ A static object or a base dynamic object can have only one interaction point.

④ A static object or a base dynamic object can have entry and exit points via its interaction point.

⑤ In a single direction of movement, only one linkage can exist between two components (e.g., only one path must exist to move from submodel A to submodel B).

4.1.5 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 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 available only 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 to associate a new class layout image and definition with the submodel or subdynamic object component decomposition. The instantiation process 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. The same hierarchical traversal facility is available under the VSSE during model execution and animation [Derrick and Balci 1992a].

4.2 Object Orientation

The benefits of the Object-Oriented Paradigm (OOP) for model design and implementation have been long recognized [Nance 1981b, 1987; Ülgen and Thomasma 1986]. The attendant benefits of the OOP which promote good model design are available when building models under the DOMINO (modularity, information hiding, weak coupling, strong cohesion, abstraction, extensibility, and reusability). In concert with these benefits, perhaps the most significant advantage 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 currently available under the DOMINO via the VSSE Model Generator software tool [Derrick and Balci 1992a]: 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 and polymorphic referencing among objects will be implemented in the future.

Under the DOMINO, modelers specify class information (for real and virtual objects) 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, self logic, and hybrid logic (See Section 4.3) are developed using the VSMSL specification language [Derrick and Balci 1992b]. 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 run-time.

If class objects are decomposable in the model hierarchy, a layout image set is also associated with the class. 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 the other 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.

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 described in the next section. Object instances which are generated during model execution are accommodated by the run-time 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.

4.3 Multi-view Specification of Model Component Logic

The DOMINO provides three different types of logic specification: supervisory logic, self logic, and hybrid logic. In addition to object-oriented model design, these specifications provide multiple views or modeler perspectives and the rules for model component interaction and model dynamics. Each type of specification can be created using the VSMSL within the VSSE Model Generator. The details of the VSMSL is presented elsewhere [Derrick and Balci 1992b]. This section gives an overview of the model component logic specification.

4.3.1 Types of Logic Specification

Models built exclusively with supervisory logic are *machine-oriented* [Tocher 1965; Kreutzer 1986]. The supervisors (machines) are the principal influences 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.

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 [Tocher 1965; Kreutzer 1986]. 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.

Hybrid logic is the logic that combines both supervisory and self logic.

4.3.2 Implementation Views

Various combinations of the logic specification types are described below using several versions of a Bus Route model.

All Supervisory Logic: The logic is spread among the supervising components and is executed by the 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 object executes 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 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 1: 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 2: 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.

4.3.3 Implications of Compositional Equivalence

Figures 5 through 8 show a set for the submodel class branch for which compositional equivalence is implied. Using the variable names for the branch layout components (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. Due to the ability to use a single supervisory logic specification with variable component referencing in tandem with multiple layout configurations, the specification effort is considerably shortened and the coding requirements are substantially reduced.

A summary of graphical and model component logic specification terminology is presented in Table 2.

5. EVALUATION

The DOMINO's evolutionary development has spanned between 1984 and 1992. Using the rapid prototyping approach, many DOMINO prototypes have been developed, implemented, experimented with, and documented. Some prototypes have been discarded; however, the experience and knowledge gained through experimentation with those prototypes have been kept.

An experimental testbed was required for the prototyping and evaluation of the DOMINO. Therefore, the DOMINO and some of the VSSE software tools (Model Generator, Model Translator, and Visual Simulator) have been developed jointly for eight years. The proposed concepts, ideas, and approaches for the DOMINO have been implemented within the Model Generator. Accordingly, the Model Translator and Visual Simulator tools have been modified. Then, using the three VSSE tools, the DOMINO's proposed concepts have been experimented with for a variety of modeling problems. Based on the experience gained, the proposed concepts have been revised and experimented with again. This iterative process has continued until all design objectives of Section 2 are satisfied.

The DOMINO, reported in this paper, has been applied to the modeling and visual simulation of

Table 2. Graphical and Model Component Logic Specification 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.
Supervisory Logic	Logic attached to the class of the supervising component. The supervisors are the primary influences 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.
Hybrid Logic	Logic that combines both supervisory and self logic specification.

an order processing system of a large computer vendor, a complex traffic intersection in Blacksburg (Virginia), a Navy combat system, a bus transportation system, and many others. Based on these applications, the DOMINO has been evaluated with respect to the ten design objectives of Section 2 and has been found to effectively satisfy its design objectives. The evaluation is given below for each design objective.

Objective 1: Provide both design and implementation guidance to furnish a broad range of support during the model development life cycle.

A DOMINO model: (a) is structured graphically; (b) is defined and specified under the OOP in terms of submodels, static objects, dynamic objects, subdynamic objects, and base dynamic objects; and (c) component logic is specified using the VSMSL. (Note that the VSMSL is used only for model component logic specification; not for the whole model.) These concepts are applicable throughout the entire visual simulation model development life cycle, from requirements specification to low level implementation. However, the modeler is not concerned with the implementation

details since the graphical, OOP, and VSMSL model specification is automatically translated into a low level, ready-to-run implementation using the VSSE Model Translator tool.

Under the DOMINO, the conceptual mapping from one model representation into another is straightforward. A model component (submodel, static object, dynamic object, subdynamic object, and base dynamic object) representing a system component can easily be traced through all model representations: depicted graphically, specified as a class with methods and attributes under the OOP, and implemented as a data structure in C programming language.

The ease of mapping from one representation form to another has enabled us to map model execution errors all the way back to the graphical specification. Thus, when an execution error occurs during the simulation run, the modeler is notified about the locations of the errors in the model specification. The modeler maintains the specification and never deals with the implementation as enunciated by the automation-based software paradigm [Balci and Nance 1987].

Objective 2: Enable the modeler to work under the object-oriented paradigm.

As presented in Section 4.2, the DOMINO provides full OOP support for the model design and enables the modeler to achieve the following software quality characteristics: modularity, information hiding, weak coupling, strong cohesion, extensibility, maintainability, reusability, and modifiability.

Objective 3: Guide the modeler in graphically structuring a visual simulation model at multiple levels of abstraction.

The DOMINO's graphical model construction concepts, as described in Section 4.1, are implemented by the VSSE Model Generator [Derrick and Balci 1992a]. Using the Image Editor of the Model Generator, the modeler creates a graphical representation of the model at the highest level of abstraction. This representation is then broken down into submodels at the next level of abstraction using a hierarchical and functional decomposition. Using the stepwise refinement principle, the modeler creates a graphical representation of each submodel and each submodel can further be decomposed into other submodels at the next level of abstraction. This decomposition continues until the desired level of abstraction is achieved. The dynamic model structure is similarly constructed. The dynamic objects are graphically represented.

Objective 4: Enable the extraction of sufficient information from the modeler so that the model execution can be visualized.

The Image and Model Editors, the OOP specification, and the VSMSL of the VSSE Model Generator software tool employ the DOMINO concepts to enable the extraction of sufficient information needed for visualization from the modeler in an efficient manner with a well designed human interface [Derrick and Balci 1992a].

Objective 5: Embody a WYSIWYR (What You See Is What You Represent) philosophy and enable the modeler to represent a system as it is naturally perceived.

The DOMINO advocates building the model static and dynamic architectures graphically in a "what you see is what you represent" manner. The modeler can create images of model static and dynamic components as digitized video or photo images, scanned images, paintings, and 2 or 3 dimensional drawings in color or black and white. Therefore, the model can be built to pictorially correspond to the system at multiple levels of abstraction.

Objective 6: Adhere to the principles of the Conical Methodology [Nance 1987].

The DOMINO fully supports the principles of the Conical Methodology (CM):

Abstraction: The DOMINO, through its graphical model construction capabilities and by employing the OOP concepts, enables the modeler to achieve multiple levels of abstraction.

Concurrent documentation: The DOMINO advocates the CM principle that the documentation and specification are inseparable. The modelers are prompted 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. Using the underlying INGRES relational database, the Query facility of the Model Generator creates documentation from the specification information that is stored in the database. Reports generated are informative about many aspects of the specification. 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.

Functional decomposition: The DOMINO advocates the model decomposition into submodels, static objects, dynamic objects, subdynamic objects, and base dynamic objects. Each model component represents a different functionality of the system being modeled. 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 hybrid logic is another form of modularization.

Hierarchical decomposition: The CM advocates top-down definition with bottom-up specification. The hierarchical traversal and definition features of the DOMINO 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, the modeler can freely alternate between the definition and specification tasks.

Information hiding: The object orientation of the DOMINO facilitates 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.

Iterative refinement, stepwise refinement, and progressive elaboration: Under the DOMINO, development of a model specification can incrementally proceed as desired by the modeler. Incomplete specifications can be saved for work at a later time. Iterative refinement, stepwise refinement, and progressive elaboration apply 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 component. 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.

Life-cycle verification: Verification begins with model specification and continues throughout the development life cycle. Using the INGRES relational database representation of the model specification (complete or not), diagnostic analysis can be performed, by using the Model Analyzer (both automated and semi-automated), on the representation form from the very early stages of development to the executable model representation. The level of diagnostic capabilities are further evaluated under Objective 10. The Model Verifier provides continued verification capabilities (assertion checking, trace data, execution profiles) on executable forms at the later stages of the development life cycle.

Separation of concerns: The DOMINO advocates this principle by separating real model components from the virtual ones, model static structure from the dynamic one, and by employing the OOP concepts of classes, attributes, methods, etc.

Objective 7: Contain a rich and expressive terminology applicable for any discrete-event simulation problem domain.

Based on the DOMINO's application to a variety of problems, it has been found that the DOMINO's terminology is rich and applicable for any discrete-event simulation problem domain. The use of DOMINO's terms is quite general purpose: submodels, static objects, dynamic objects, subdynamic objects, and base dynamic objects. The meaning of a model component is by analogy. It may represent whatever the modeler wishes to represent within the WYSIWYR philosophy.

Objective 8: Differentiate a model component having a representation in the system from the one that does not have a representation in the system.

This is achieved by classifying model components as real versus virtual. The separation is important not only to remove the contradiction in the definition of a model (Section 2, Objective 8), but also for model verification and validation. Real versus virtual separation guides the modeler in identifying the model components for which validation or verification needs to be emphasized.

Objective 9: Enable the extraction of sufficient information from the modeler so that the model specification can be automatically translated into executable code following the automation-based software paradigm [Balci and Nance 1987].

The automation-based software paradigm has been achieved to a large extent. Using the VSSE Model Generator, a modeler can structure a model graphically, specify it under the OOP, and provide the model component logic specification using the VSMSL English-like language. Thereafter, the Model Translator, reading in the whole model specification from the INGRES database, converts the model into an executable code which provides visual simulation upon execution. The modeler is not concerned about the translation process, the implementation language, or the fact that INGRES is being used in the background. If execution errors occur, the modeler is informed about the locations of the errors within the model specification. Hence, the modeler maintains the specification at all times and the model specification is automatically translated.

However, the Model Translator does not currently use artificial intelligence techniques for producing an optimized executable code that would provide an efficient execution of the model. Except this part, the automation-based software paradigm has been achieved.

Objective 10: Enable the creation of a model specification which lends itself for formal diagnostic testing [Nance and Overstreet 1987].

Nance and Overstreet [1987] present a wide variety of diagnostic capabilities (analytical, comparative, and informative) under the Condition Specification. 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 VSSE Model Analyzer, Model Generator, and Model Translator [Derrick and Balci 1992a].

The Model Generator and Analyzer demonstrate attribute initialization and attribute consistency from among the analytical measures. Attributes are required to be initialized during class specification; the Model Generator automatically performs these initializations at model startup. The Model 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 Model 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.

The remaining diagnostic measures, which include all comparative measures, deal with action cluster diagnosis under the Condition Specification representation. In order to employ 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 converting 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.

The 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 [1987] analytical, comparative, and informative measures. Consistency and completeness checks are possible on various aspects of the model specification. 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 currently available under the Condition Specification. However, new and additional techniques are available under the DOMINO.

6. CONCLUSIONS

A multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO) is presented. The DOMINO possesses many talents and has many aspects or phases, and therefore it is called multifaceted. Providing both design and implementation guidance, the DOMINO guides a modeler over a broad range of tasks in the simulation model life cycle. The DOMINO contains many desired characteristics (e.g., object-oriented, graphical-oriented, and activity-based) and new approaches which generate a fruitful and nurturing environment within which a modeler is assisted in the creative processes.

Incorporating the OOP, the DOMINO contains excellent support for good software-engineered model designs. 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 DOMINO fully supports the guiding principles of the Conical Methodology.

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 (e.g., dynamic versus static, real versus virtual, etc.) of the DOMINO.

The incorporation of visualization as a prominent element of the DOMINO enhances the verification and validation tasks. 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 [Balci and Nance 1992].

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 software paradigm, with potential gains in efficiency and productivity in the model development effort, and reductions to development time. Modelers work only with the model specification which is automatically translated into an executable visual simulation model. 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 language. This burden has previously been a severe stumbling block to the development process.

ACKNOWLEDGEMENTS

This research was sponsored in part by the U.S. Navy and IBM through the Systems Research Center at VPI&SU. The authors acknowledge stimulating discussions with Richard E. Nance, Lynne F. Barger, Jay D. Beams, John L. Bishop, Valerie L. Frankel, Robert L. Moose, Jr., C. Michael Overstreet, Ernest H. Page, Jr., and Fred A. Puthoff which contributed to the research described herein.

REFERENCES

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," In *Proceedings of the 1988 Winter Simulation Conference*, M.A. Abrams, P.L. Haigh, and J.C. Comfort, Eds. IEEE, Piscataway, NJ, pp. 287-295.

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, pp. 25-32.

Balci, O. and R.E. Nance (1987), "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, pp. 495-502.

Balci, O. and R.E. Nance (1992), "The Simulation Model Development Environment: An Overview," In *Proceedings of the 1992 Winter Simulation Conference* (Arlington, VA, Dec. 13-16). IEEE, Piscataway, NJ, to appear.

Balci O. and R.G. Sargent (1981), "A Methodology for Cost-Risk Analysis in the Statistical Validation of Simulation Models," *Communications of the ACM 24*, 4 (Apr.), 190-197.

Booch, G. (1986), "Object-Oriented Development," *IEEE Transaction on Software Engineering SE-12*, 2 (Feb.), 211-221.

Derrick, E.J. (1988), "Conceptual Frameworks for Discrete Event Simulation Modeling," M.S. Thesis, Department of Computer Science, VPI&SU, Blacksburg, VA, Aug.

Derrick, E.J. (1992), "A Visual Simulation Support Environment Based on a Multifaceted Conceptual Framework," Ph.D. Dissertation, Department of Computer Science, VPI&SU, Blacksburg, VA, Apr.

Derrick, E.J. and O. Balci (1992a), "A Visual Simulation Support Environment Based on the DOMINO Conceptual Framework," Technical Report TR-92-44, Department of Computer Science, VPI&SU, Blacksburg, VA, Aug.

Derrick, E.J. and O. Balci (1992b), "A Visual Simulation Model Specification Language," (in preparation).

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, pp. 711-718.

Knuth, D.E. (1973), *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA.

Kreutzer, W. (1986), *System Simulation: Programming Styles and Languages*, Addison-Wesley, Reading, MA.

Miller, G.A. (1956), "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review 63*, 2 (Mar.), 81-97.

Nance, R.E. (1984), "Model Development Revisited," In *Proceedings of the 1984 Winter Simulation Conference*, S. Sheppard, U.W. Pooch, and C.D. Pegden, Eds. IEEE, Piscataway, NJ, pp. 75-80.

Nance, R.E. (1981a), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24*, 4 (Apr.), 173-179.

Nance, R.E. (1981b), "Model Representation in Discrete-Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, VPI&SU, 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. Published as *Simulation Series 19*, 1 (Jan. 1988), 38-43. SCS, San Diego, CA.

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 (Jan.), 33-57.

Shannon, R.E. (1975), *Systems Simulation: The Art and Science*, Prentice-Hall, Englewood Cliffs, NJ.

Tocher, K.D. (1965), "Review of Simulation Languages," *Operational Research Quarterly 16*, 2, 189- 217.

Ülgen, O.M. and T. Thomasma (1989), "Simulation Modeling in an Object-Oriented Environment Using Smalltalk-80," In *Proceedings of the 1986 Winter Simulation Conference*, J. Wilson, J. Henriksen, and S. Roberts, Eds. IEEE, Piscataway, NJ, pp. 474-484.