A Visual Simulation Support Environment Based on the DOMINO Conceptual Framework*

E. Joseph Derrick and Osman Balci

TR 92-44

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

August 17, 1992

Cross-listed as Systems Research Center report SRC-92-008.

ABSTRACT

The purpose of this paper is to present a Visual Simulation Support Environment (VSSE) based on the multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO). The ever-increasing complexity of visual simulation model development is undeniable. There is a need for automated support throughout the *entire* visual simulation model development life cycle. This support is furnished by the VSSE which is composed of *integrated* software tools providing computer-aided assistance in the development and execution of a visual simulation model. The VSSE has been jointly developed with the DOMINO. Its architecture consists of three layers: hardware and operating system, kernel VSSE, minimal VSSE, and VSSEs. This paper focuses on the minimal VSSE toolset. Evaluation of the VSSE shows that it adequately satisfies all of its 13 design objectives.

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, visual simulation, visual simulation model development environments, visual simulation support environments

TABLE OF CONTENTS

1. INTRODUCTION

The ever-increasing complexity of visual simulation model development is undeniable. A simulation programming language supports only the programming process—one of 10 processes in the life cycle of a simulation study [Balci 1990]. There is a need for automated support throughout the *entire* visual simulation model development life cycle. This support can be provided in the form of an environment composed of *integrated* software tools providing computer-aided assistance in the development and execution of a visual simulation model.

A collection of computer-based tools makes up a development environment if and only if the tools are highly integrated and work under a unifying Conceptual Framework (CF). Therefore, to develop such an environment, a CF is needed. However, the development of a CF requires an environment for experimenting with and evaluating the CF. Hence, the CF and the environment need to be developed jointly. This joint development has spanned between 1984 and 1992, and resulted in: (1) creation of the multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO) [Derrick 1992; Derrick and Balci 1992a], (2) Visual Simulation Support Environment (VSSE), (3) Visual Simulation Model Specification Language (VSMSL) [Derrick 1992; Derrick and Balci 1992b], and (3) achievement of the automation-based software paradigm [Balci and Nance 1987b].

This paper describes the VSSE developed under the DOMINO based on the experience gained in the Simulation Model Development Environment (SMDE) research project [Balci 1986; Balci and Nance 1987a, 1992]. The 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, especially the companion paper on DOMINO [Derrick and Balci 1992a], to obtain the related information including a survey of the literature.

The rapid prototyping technique has been used in the VSSE's evolutionary joint development with the DOMINO. Many VSSE tool 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.

The VSSE is developed by using the C programming language, SunView graphical user interface [Sun Microsystems 1988], Sun programming environment, and INGRES relational database management system Embedded QUEry Language/C (EQUEL/C) [Sun Microsystems 1986]. It encompasses more than 50,000 lines of documented code and runs on a Sun color workstation.

The purpose of this paper is to describe the VSSE based on the DOMINO CF. The design objectives of the VSSE are introduced in Section 2. Section 3 presents the VSSE architecture. Minimal VSSE toolset is described in Section 4. Section 5 contains the evaluation of the VSSE. Concluding remarks are given in Section 6.

2. VSSE DESIGN OBJECTIVES

This section delineates the VSSE design objectives in no particular order. The design objectives for the VSSE are multi-directed. Some relate to the evaluation of the DOMINO; others relate more directly to VSSE capabilities, apart from the CF. Objectives are grouped by an associated tool when appropriate.

VSSE (Complete toolset):

Objective 1: Provide a fully functional and highly integrated toolset under the DOMINO conceptual framework. Accomplish the integration and communication among the tools using (primarily) a relational database (e.g., INGRES) representation of the specification.

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

Objective 2 supports usability of the VSSE.

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 [Nance 1981, 1987].

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 the ability to translate model component logic specifications directly into 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 model specification and documentation. Use the relational database representation as the basis for analysis. Tailor the analysis to the unique features of the DOMINO.

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 and retrieval 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, achieving the Automation-Based Paradigm [Balci and Nance 1987b].

This objective supports maintainability, reusability, 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 model in the background without animation, producing statistical analysis reports.

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

3. VSSE ARCHITECTURE

Figure 1 depicts the VSSE architecture in four layers: (0) Hardware and Operating System, (1) Kernel VSSE, (2) Minimal VSSE, and (3) VSSEs.

3.1 Layer 0: Hardware and Operating System

A Sun computer workstation constitutes the hardware of the VSSE. The UNIX SunOS operating system and utilities, SunView graphical user interface, and INGRES relational database management system constitute the software environment upon which the VSSE is built. Nance et al. [1984] evaluate the UNIX operating system as a foundation for building a simulation environment.

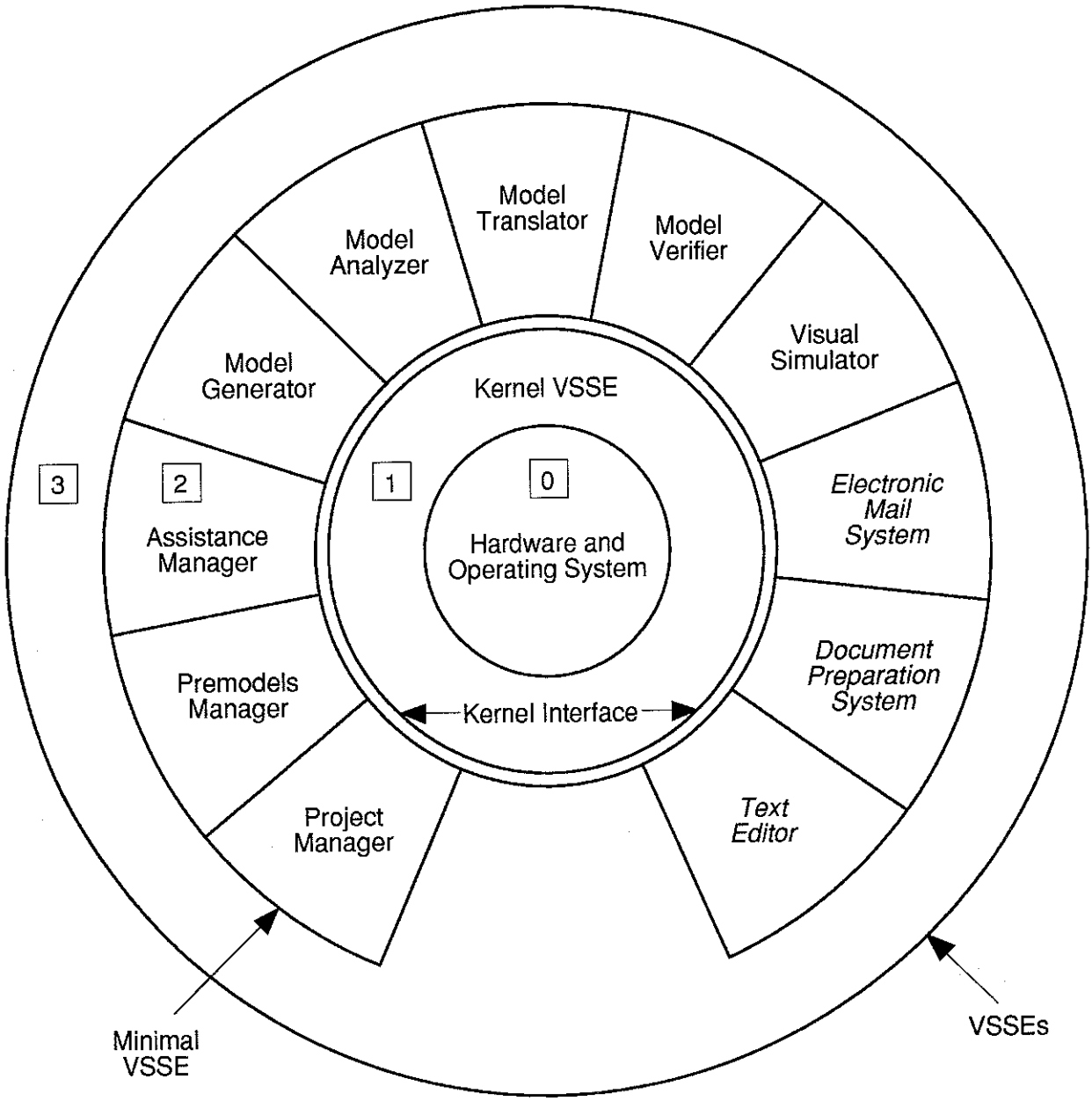


Figure 1. Visual Simulation Support Environment Architecture

3.2 Layer 1: Kernel Visual Simulation Support Environment

Primarily, this layer integrates all VSSE tools into the software environment described above. It provides INGRES databases, communication and run-time support functions, and a kernel interface. Three INGRES databases occupy this layer, labeled project, premodels, and assistance, each administered by a corresponding manager in layer 2. All VSSE tools are required to communicate through the kernel interface. Direct communication between two tools is prevented to make the VSSE 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.

3.3 Layer 2: Minimal Visual Simulation Support Environment

This layer provides a "comprehensive" set of tools which are "minimal" for the development and execution of a visual simulation model. "Comprehensive" implies that the toolset is supportive of all model development phases, processes, and credibility assessment stages. "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 VSSE without significant inconvenience. Generality is claimed in the sense that the toolset is generically applicable to various simulation modeling tasks.

Minimal VSSE tools are classified into two categories. The first category contains tools specific to simulation modeling: Project Manager, Premodels Manager, Assistance Manager, Model Generator, Model Analyzer, Model Translator, Model Verifier, and Visual Simulator. The second category tools (also called assumed tools or library tools) are expected to be provided by the software environment of Layer 0: Electronic Mail System, Document Preparation System, and Text Editor. The minimal VSSE tools are described in Section 4.

3.4 Layer 3: Visual Simulation Support Environments

This is the highest layer of the environment, expanding on a defined minimal VSSE. In addition to the toolset of the minimal VSSE, it incorporates tools that support specific applications and are needed either within a particular project or by an individual modeler. If no other tools are added to a minimal toolset, a minimal VSSE would be a VSSE.

The VSSE tools at layer 3 are also classified into two categories. The first category tools include those specific to a particular area of application. These tools might require further customizing for a specific project, or additional tools may be needed to meet special requirements. The second category tools (also called assumed tools or library tools) are those anticipated as available due to use in several other areas of application. A tool for statistical analysis of simulation output data, a tool for designing simulation experiments, a tool for documentation and credibility assessment, and a tool for input data modeling are some example tools in layer 3.

A VSSE tool at layer 3 is integrated with other VSSE tools and with the software environment of layer 0 through the kernel interface. The provision for this integration is indicated in Figure 1 by the opening between Project Manager and Text Editor. A new tool can easily be added to the toolset by making the tool conform to the communication protocol of the kernel interface.

4. MINIMAL VSSE TOOLS

The Project Manager, Premodels Manager, Assistance Manager, Model Generator, Model Analyzer, Model Translator, Model Verifier, and Visual Simulator constitute the first category mini-

mal VSSE tools as shown, except the first three, in Figure 2. The Premodels Manager and Assistance Manager, described in Section 4.6, have been developed but have not yet been integrated into the VSSE. The Project Manager is not yet functional. Electronic Mail System, Document Preparation System, and Text Editor constitute the second category minimal VSSE tools which are described in Section 4.6.

4.1 Model Generator

The Model Generator is the software tool that guides a modeler under the DOMINO conceptual framework in creating a visual simulation model specification and documentation. It fully implements the DOMINO and is the most crucial tool within the VSSE.

Activating the Model Generator by clicking on its icon on the VSSE top level menu (Figure 2) displays the dialog panel in Figure 3a. A modeler can construct a new model, work on the existing model the name of which is displayed ("transact"), or retrieve another model to work on using the appropriate option of the dialog panel in Figure 3a.

The Model Generator consists of two principal parts: the Image Editor and the Model Editor. Both parts are graphically-oriented and maximize the use of the direct manipulation interface. The Image Editor is used to create graphical representations for the visualization of simulation models. 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).

4.1.1 Image Editor

Currently, the Image Editor (Figure 4) is purely a "draw" facility. However, it is designed to enable the modeler to create 2 or 3 dimensional images using drawing, painting, and manipulating scanned images and digitized video or photo images.

The various draw operations are available with the buttons from left to right across the top of the Image Editor window shown in Figure 4. 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 image 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 and the middle button activates the eraser. Using the SCREEN OPERATIONS button, a modeler can save a new drawing, load a previous drawing, and clear the drawing screen. The TO MODEL

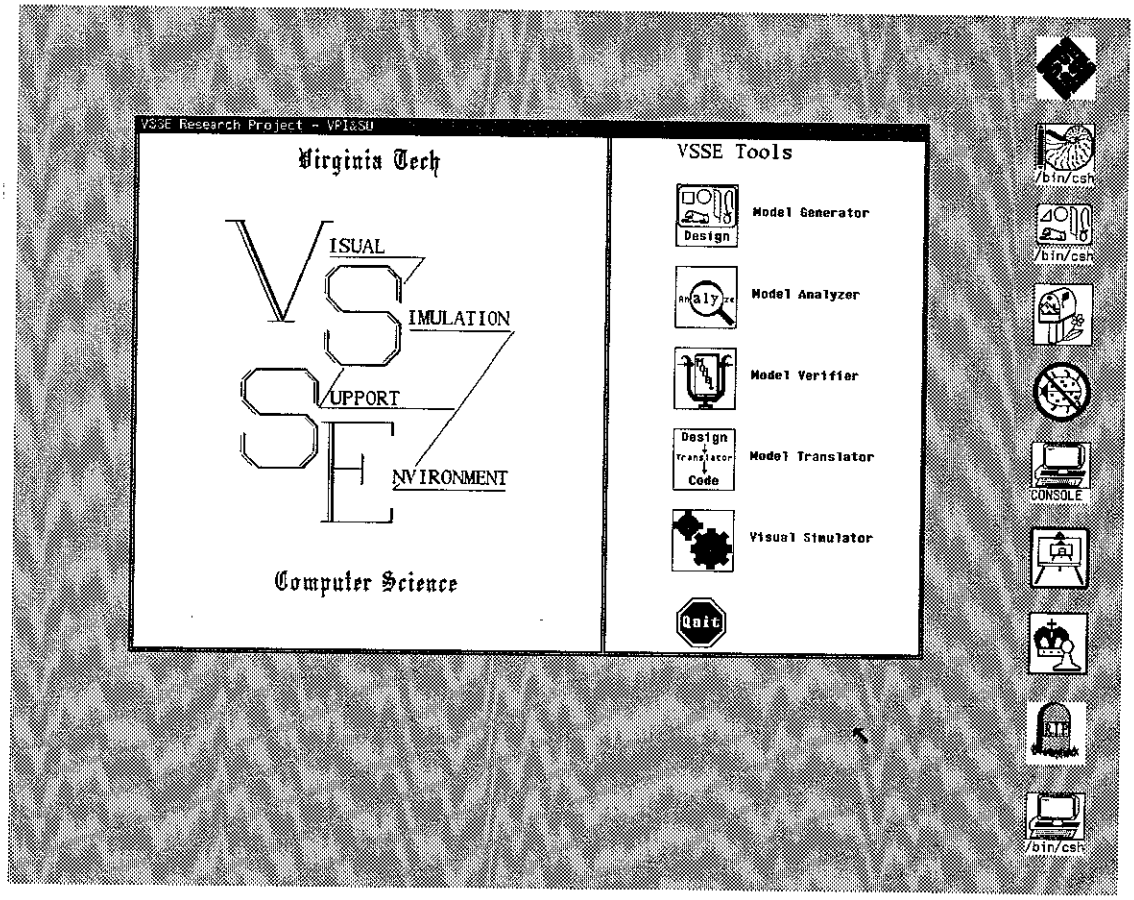
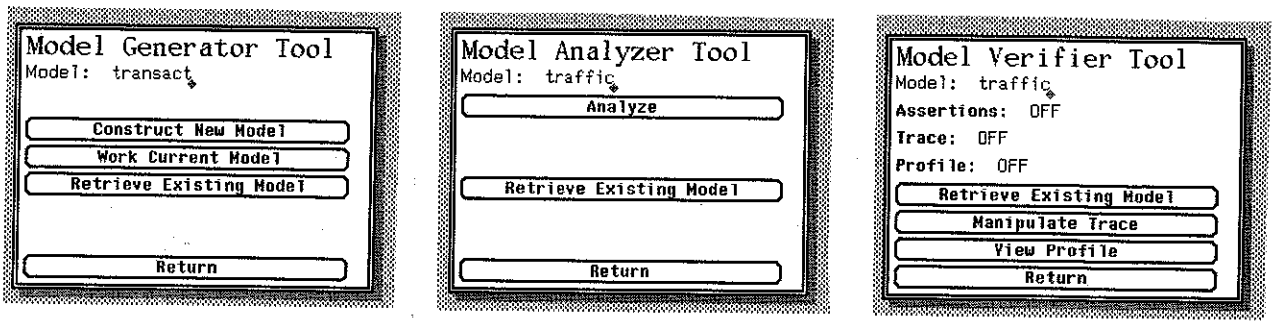
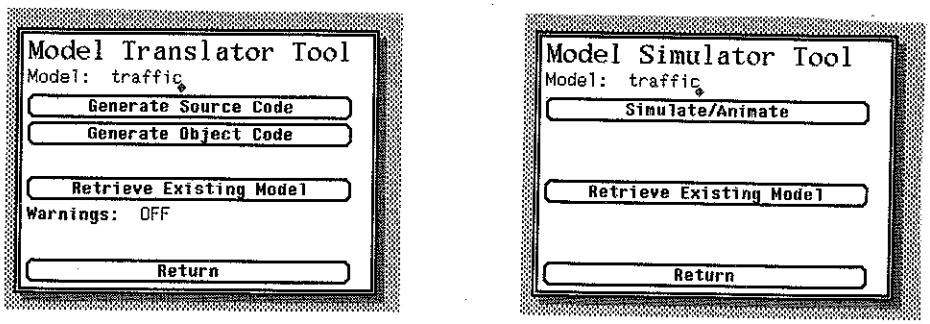


Figure 2. The VSSE Top Level Menu



(a) (b) (c)



(d) (e)

Figure 3. Dialog Panels of the VSSE Tools

EDITOR button toggles the Model Generator to the Model Editor facility.

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 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 the 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 (e.g., saving a submodel component image to database).

The particular class association must be identified, although the class may possibly be not 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). 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.

Loading an image is handled by the same conventions as saving. Images may be loaded from file or from the database, and always by component type such as a submodel, static object, or dynamic object. 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 mouse and "dropped" by clicking the middle mouse button.

4.1.2 Model Editor

Model definition and specification are completed using the Model Editor (Figure 5). All images do not have to be drawn before entering any Model Editor function. Certain aspects of model definition and specification (e.g., 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.

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),

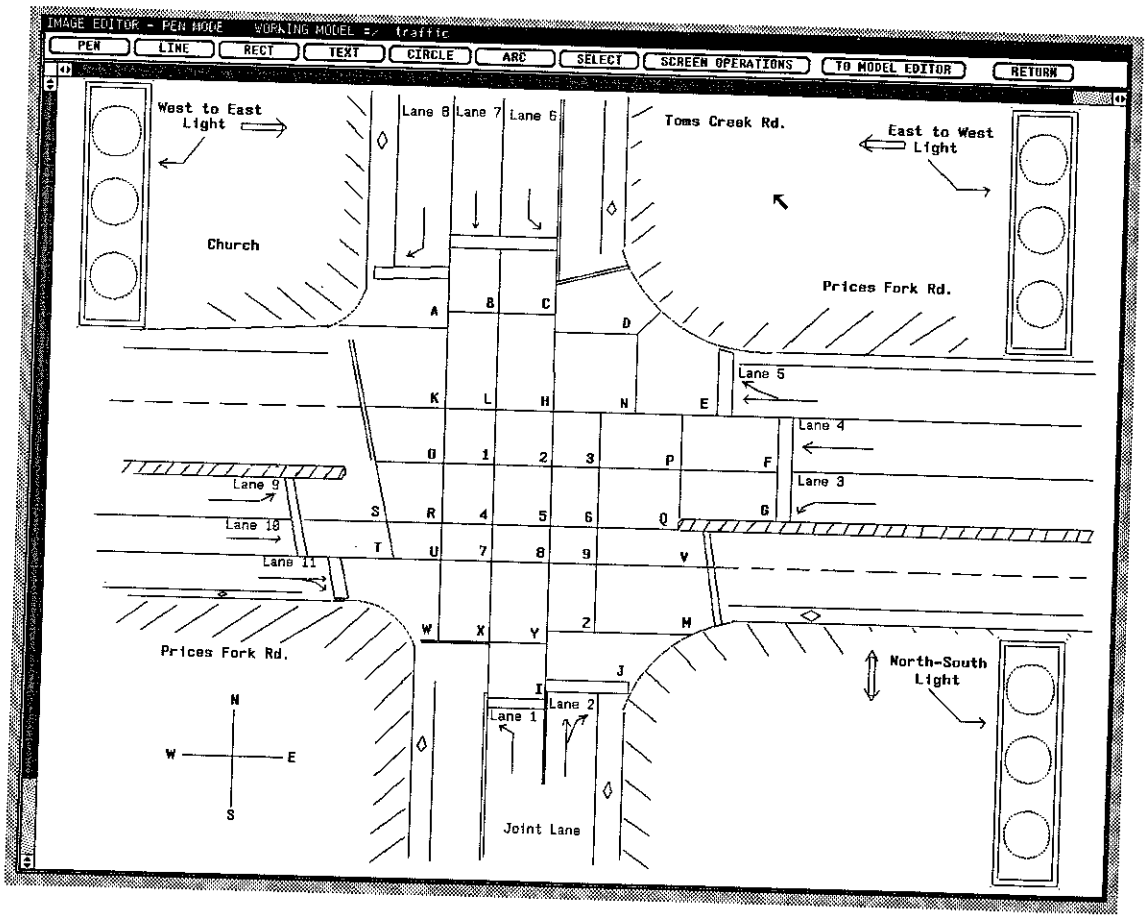


Figure 4. The Model Generator's Image Editor

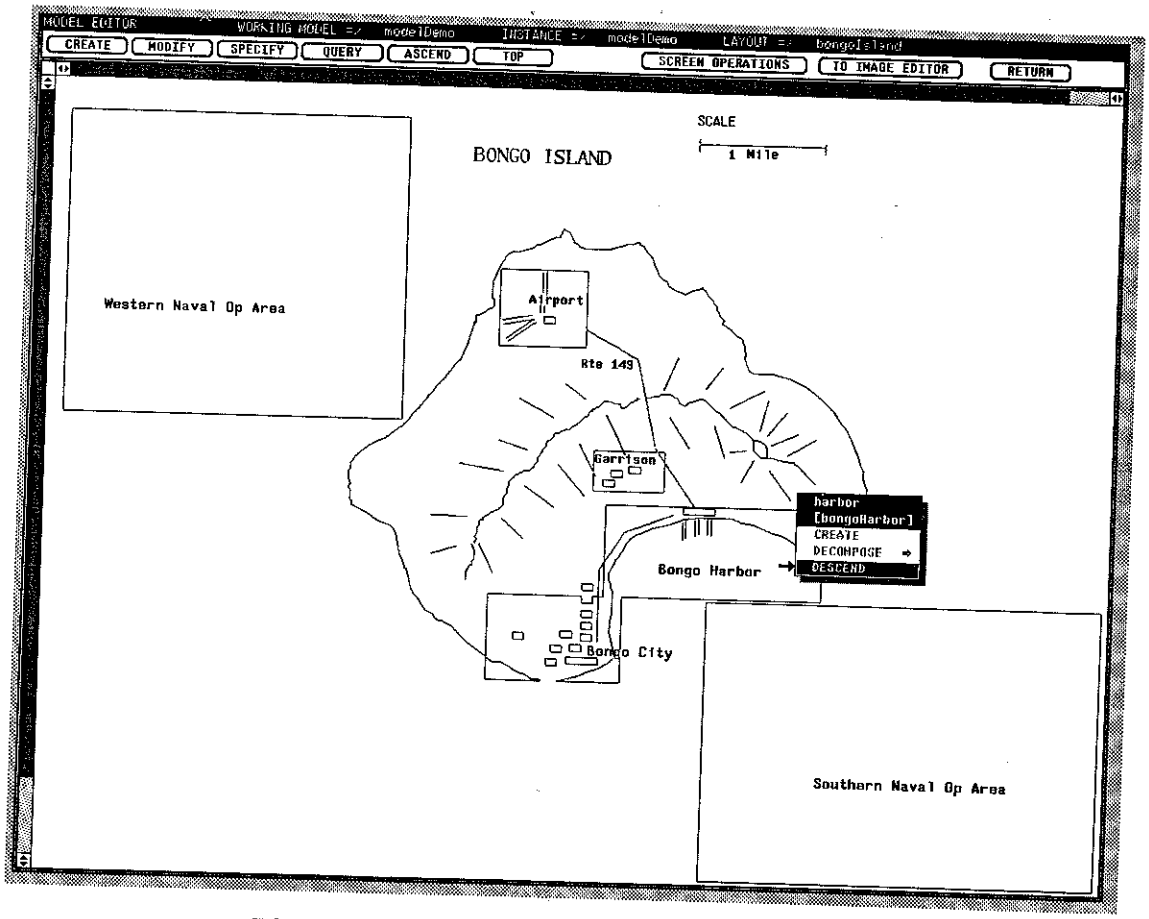


Figure 5. The Model Generator's Model Editor

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

Action 3. The identification of paths, connectors, and interactors (if 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 (Figure 5). 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.

4.1.2.1 Specifying Class Attributes and Model Component Logic

Under SPECIFY menu option (Figure 5), class information is specified (Action 1) by first identifying the component type of the particular class such as submodel classes and static object classes. Using the class specification window, the class name is entered. Error checking on the name is performed to determine if the class name has been previously used for the component type. (If already 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 menu 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. 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 is similar to the class specification window. The name must be typed and is error checked. 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) 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.

Like class and attribute names, The Model Generator checks for the existence of a previous logic specification for the class being worked. The logic specification window is a text subwindow in which the modeler enters the model component logic using VSMSL (Visual Simulation Model Specification Language) [Derrick 1992; Derrick and Balci 1992b]. 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. With the symbol tables, the translator performs static analysis of the logic specification. Appropriate error messages are displayed and syntax errors are identified.

Once translated, the modeler returns to the logic specification. Of interest, any logic specification of the same component type can be loaded from the logic specification window. 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. In addition, methods require that a return type be indicated (methods often return data to the sender of the message which activates the method). Parameter specification 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 using the Image Editor. The goal is to identify the class instances in each class layout image in a general way. This section describes Actions 2 and 3 discussed earlier in Section 4.1.2. The class layout is classified as 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 layout image, we identify component locations: submodels and static objects for the static structure or subdynamic objects and base dynamic objects for dynamic structures. The locations are set using the mouse (left button) to define a rectangular area in the desired component position. The modeler must give the component a variable name to allow the class layouts to be reusable. The status of progress in a class layout can be checked using the COMPONENT STATUS button.

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 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 with a defining rectangle. Interactors must be associated with a static or base dynamic object. 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 button) 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 a specific way using the INSTANCE LAYOUT option. 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 as shown in Figure 5. 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 as shown in Figure 5. The blackened option is selected.

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, 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 ("bongoHarbor" in Figure 5). 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. After this, the indicated class layout (the decomposition level) is displayed. 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. Navigation around the structure is easily accomplished. The ASCEND button across the top row of buttons moves the instantiation context up one decomposition level. If at a deep level in the structure, the TOP button (on the top buttons bar), takes the modeler to the root context of the structure. The combination of descend/ascend/top features gives extreme flexibility and ease of model definition and specification. 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. 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 submodels are created via the CREATE/VIRTUAL/SUBMODEL sequence of pullright menus. The creation window for virtual submodels contains entries which must be entered or performed by the modeler: name, class information, documentation, and 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. The QUERY feature allows the modeler 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.). A graphics query can be oriented to component image, layout image and definition, layout member, or layout path information. Under hierarchy queries, a modeler can view class inheritance hierarchies or model static and dynamic structure hierarchies.

4.1.2.6 Modifying the Model Database

The MODIFY operation is 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 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). 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. Definition files can be viewed and deleted; image files can only be deleted.

4.2 Model Analyzer

The Model Analyzer provides a modeler with the means to perform consistency and completeness checks on various aspects of the model specification and documentation by using the model representation in the relational database. Activating the Model Analyzer by clicking on its icon on the VSSE top level menu (Figure 6) displays the dialog panel in Figure 3b. A modeler can analyze the model the name of which is displayed ("traffic") or retrieve another model to analyze using the appropriate option of the dialog panel in Figure 3b. Clicking on the Analyze button of the dialog panel displays the Analyzer window shown in Figure 6.

Analysis is possible on the class specification, logic specification, image specification, definition and instantiation information, and documentation. The type of analysis is selected by clicking on the corresponding Analyze button (Figure 6). If a problem is found during analysis, the "NO" item toggles to "YES". At that time, the modeler can view the error report by selecting the corresponding View Report button. 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 a 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 specification) providing additional flexibility during definition and specification. Therefore, a "YES" report on analysis for any

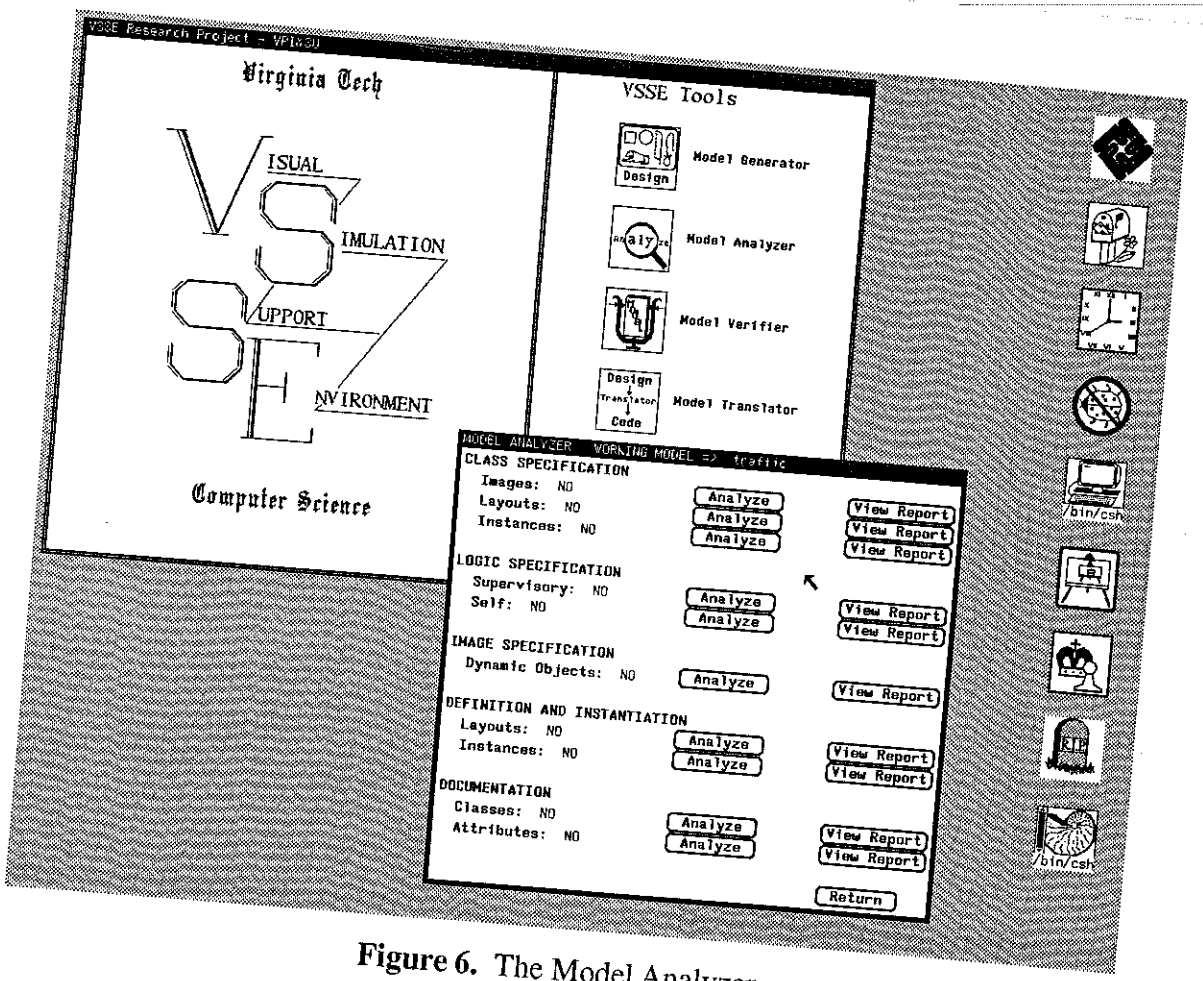


Figure 6. The Model Analyzer

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. The analysis reports provide notes to assist in this determination.

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 [1987] 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 Model Verifier

Activating the Model Verifier by clicking on its icon on the VSSE top level menu (Figure 2) displays the dialog panel in Figure 3c which indicates the verification features that are available within the VSSE. The toggles ("ON" and "OFF") in Figure 3c enable assertion checking and the production of 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 of the model in which assertion statements (within model 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 code that can produce runtime trace data or an execution profile. Once a runtime trace or execution profile is created, then the Manipulate Trace or View Profile buttons (Figure 3c) can be used to start up their respective verification features, each of which is discussed below.

4.3.1 Trace Manager

Execution tracing, a dynamic analysis technique, provides valuable assistance in the verification of a model. During the execution of the model (which is specifically created for building a runtime trace), trace data is stored in the relational INGRES database. The database structure accommodates 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, a modeler can manipulate the trace data in order to relate runtime errors to the specification.

4.3.1.1 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 are those used by the system for storing 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 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.

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 "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.

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. The modeler can 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. The flow

information is used to "pretty print" and appropriately indent the display for better readability of the 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.

Figure 7 shows the presence of 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 button (Figure 7) provides access to these system-generated codes and enables one to decipher them.

4.3.2 Execution Profiler

After an execution profile is set up and the View Profile button (Figure 3c) is activated, the profile results are displayed. The execution profile is created using the prof library routine under the Sun Unix 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.

4.4 Model Translator

Activating the Model Translator by clicking on its icon on the VSSE top level menu (Figure 2) displays the dialog panel in Figure 3d. The Model Translator accomplishes the final automatic generation of code for model execution. This is done in a two-step process: generating source code and generating object code. *Generate Source Code* (Figure 3d) creates the modules and system definitions to accommodate 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.

Generate Object Code (Figure 3d) takes the source modules and definitions created as explained above and combines them 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 reports progress in the compilation effort to the modeler.

Compilation errors are reported. If desired, compilation warnings can be turned on (default: off) using the designated toggle (Figure 3d). As mentioned in Section 4.3, toggles in the Model Verifier cause different effects of this final translation process in producing an executable for different purposes (activation of assertions, production of runtime trace or execution profiles, or normal execution for simulation and animation by the Visual Simulator).

The automatic translation and production of an executable visual simulation model supports the Automation-Based Paradigm [Balci and Nance 1987b]. Modelers are prevented from having to get involved with the low-level details of a programming language (e.g., C, Fortran, Pascal). 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. 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."

4.5 Visual Simulator

Activating the Visual Simulator by clicking on its icon on the VSSE top level menu (Figure 2) displays the dialog panel in Figure 3e. The Visual Simulator provides two key features: (1) the runtime inspection facility for model component attributes and performance measures, and (2) the contextual visualization of the executing model.

The RUN button (Figure 8) is used to execute the model with no visualization. Clicking on the RUN button displays a pop-up menu for the modeler to enter data collection information (e.g., transient period length, steady state length, random number generator seed, number of replications, etc.) under the Method of Replications technique. Following model execution, statistical summaries of performance data are made available in a file.

The ANIMATE button (Figure 8) is used to execute the model with visualization. Before

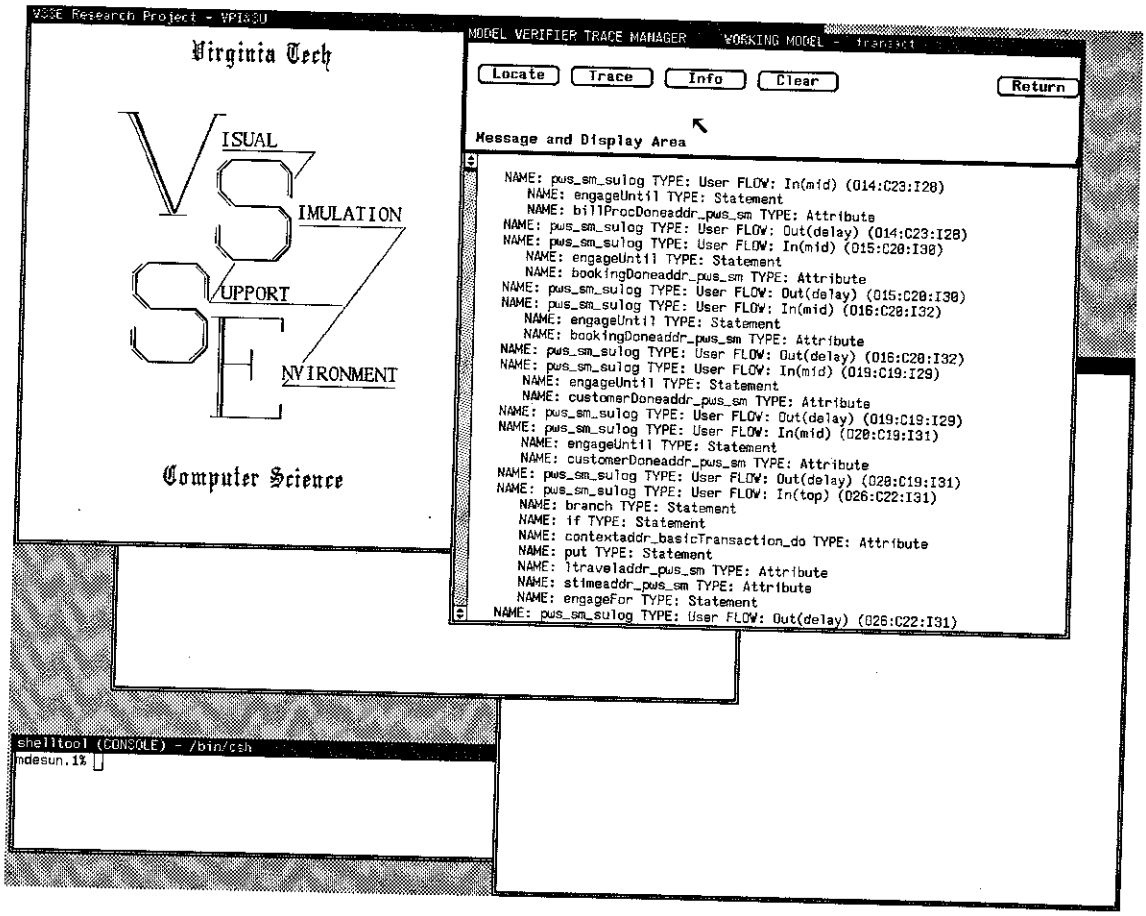


Figure 7. The Model Verifier's Trace Manager

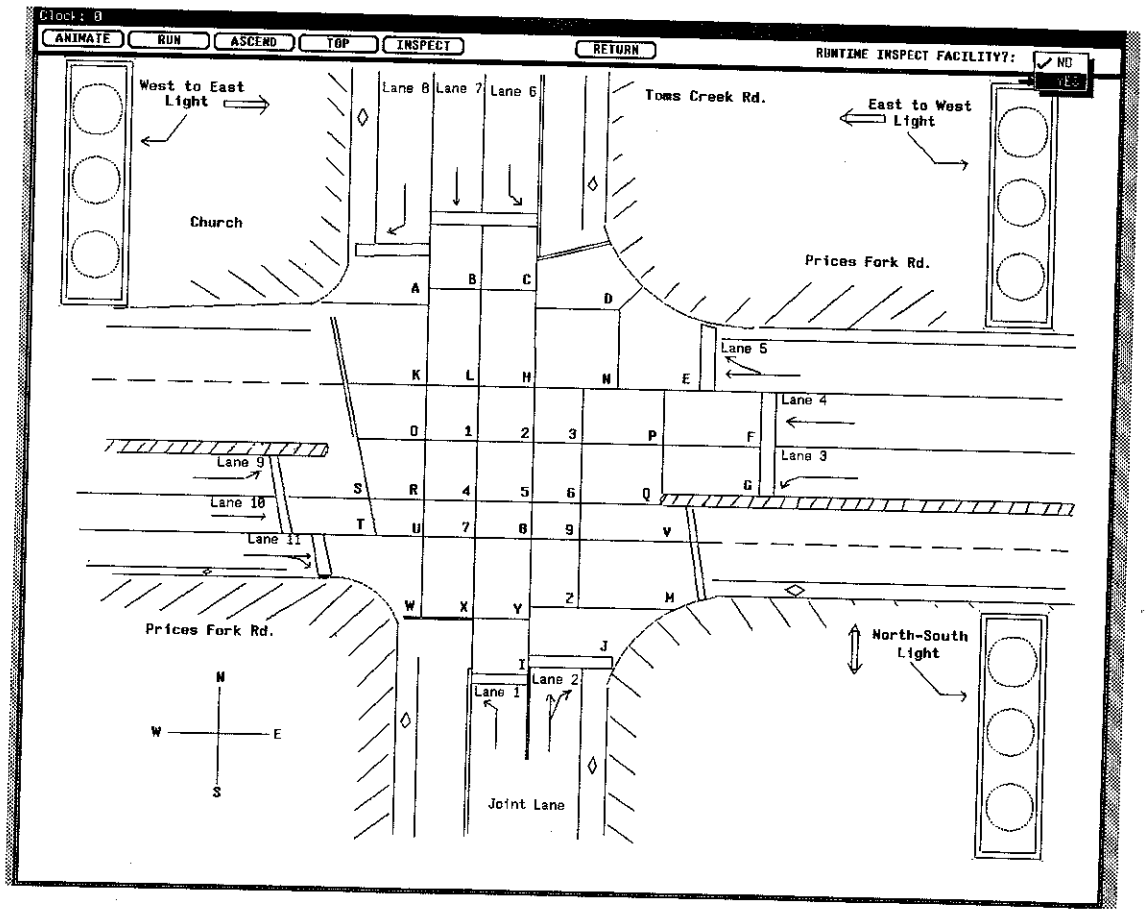


Figure 8. The Visual Simulator

animation, the modeler turns the Runtime Inspection Facility on or off by selecting "yes" or "no" from the pull-down menu in the upper-right corner of Figure 8.

The Runtime Inspection Facility enables the modeler to view the values of the following attributes during visualization: (1) model component attributes (for non-dynamic object components which are instantiated within model class layouts), (2) global attributes (attached to the model itself; i.e., model attributes) or the attributes of virtual submodels, and (3) attributes of dynamic objects (both decomposed and non-decomposed). Using this facility, a modeler can set up performance or statistical measures of interest and observe their changes during the course of visualization.

The contextual visualization capability of the Visual Simulator allows a modeler to view the animation from any layout context. Figure 9 shows the visualization at the top level of the Branch Operations Model in which a transaction called "BillProc" is exiting Branch 1 submodel with some information displayed on it. Similar to the ability to navigate throughout 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.

If a component is decomposed, the DESCEND option of its popup menu 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 menu is displayed which lists all model layouts (Figure 9). By selecting one, the viewing context can be changed to a deep level of the hierarchy, without having to incrementally get there.

Within any layout, pointing (using the mouse) at a model component on the layout and then pressing the right mouse button pops up a menu containing INSPECT and DESCEND options. Selecting INSPECT and pulling down its menu, the component's attributes and their current values are displayed as shown in Figure 10. The INSPECT button in the top horizontal bar of Visual Simulator buttons (Figure 10) provides the capability for inspecting the model attributes and also virtual submodel 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. The attributes of decomposed dynamic objects are also available using the INSPECT button.

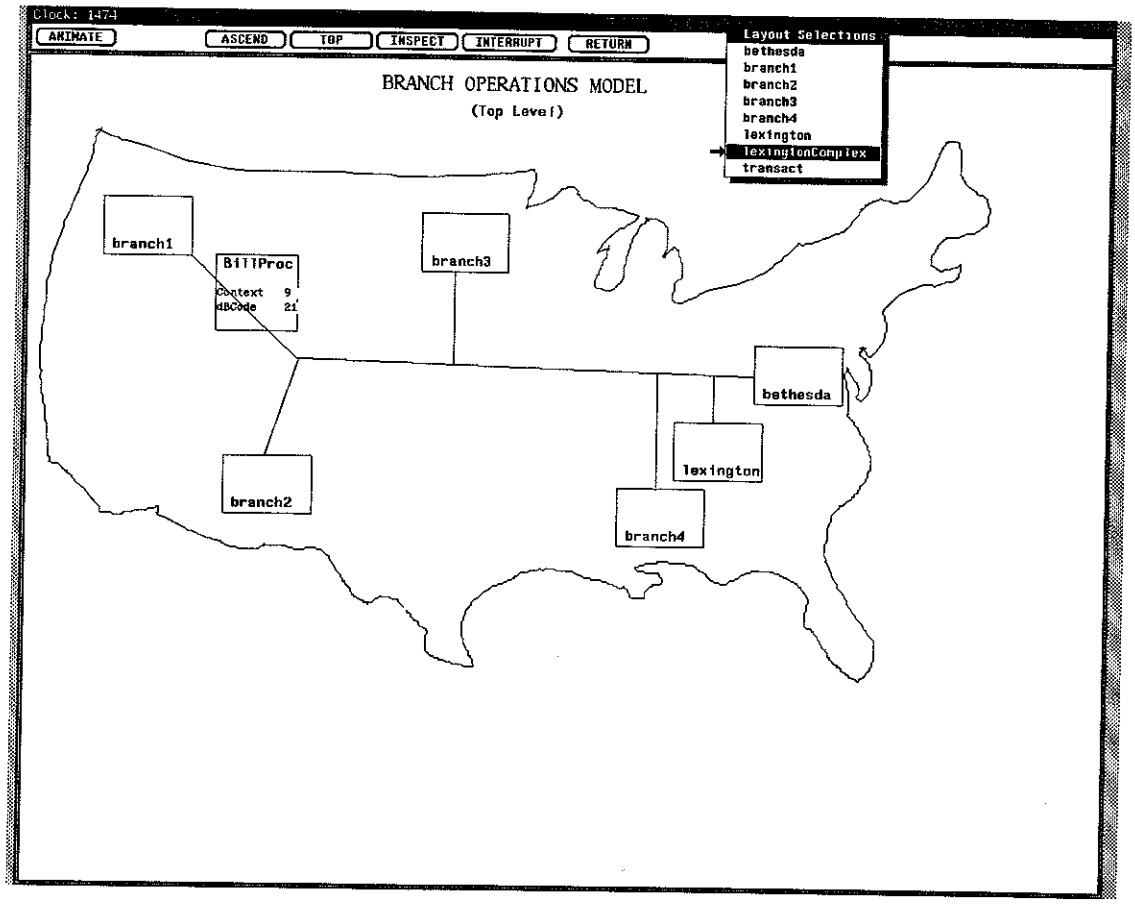


Figure 9. Visual Simulation of the Branch Operations Model

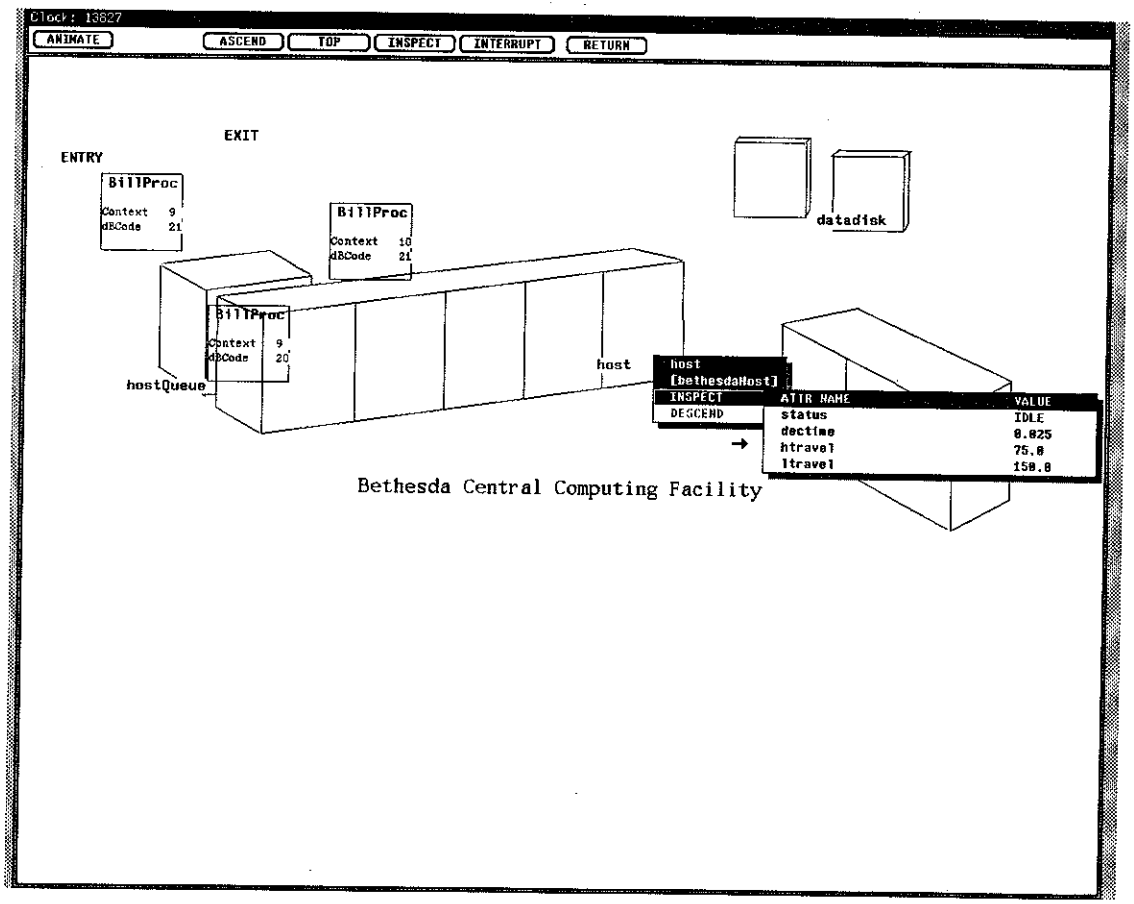


Figure 10. Visual Simulation of a Submodel of the Branch Operations Model

4.6 Other Tools

4.6.1 Project Manager

The Project Manager software tool: (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. The Project Manager is not yet functional.

4.6.2 Premodels Manager

The overall goal of the Premodels Manager (PM) software tool is to enable the user to: (a) locate and reuse components of successfully completed simulation studies, and (b) learn from past experience. The following design objectives are identified to meet this overall goal [Beams 1991; Beams and Balci 1992]:

① Provide easy methods of installing and maintaining documentation of successfully completed simulation studies [Nance 1977, 1979] in the Premodels Database.

② Provide appropriate methods of access to documentation of successfully completed simulation studies in the Premodels Database. Initiative, mechanisms, and complexity of access should vary according to task and type of user.

③ Provide a stratified display, capabilities for copying and pasting, capability for storing in a user-created file, and printing of the information located by a user in the Premodels Database.

④ Provide a user interface that satisfies the nine usability principles for interfaces: enable simple and natural dialogue, speak the user's language, minimize the user's memory load, promote consistency, provide feedback, provide clearly marked exits, provide shortcuts, supply good error messages, and prevent errors.

⑤ Provide context-sensitive help that is always available in a consistent manner. The system should use all available information on the user's state and avoid placing the burden on the user.

The PM consists of a collection of windows which work together to allow different types of interactions between users and the Premodels Database. Three types of windows are used in the PM: (1) working windows (Browser, Searcher, Installer, and Maintainer), (2) access windows (Driver, Retriever, and Administrator), and (3) support windows (File Viewer, Describer, and Helper).

The PM has been evaluated with respect to the five design objectives stated above and has been found to provide effective reusability and learning support [Beams 1991; Beams and Balci 1992]. The design objectives altogether contribute to enabling the user to locate and reuse components of successfully completed simulation studies and learn from past experience.

The rapid prototyping software engineering approach has been used in developing the PM. The first PM prototype focused on the terminology problem in searching model components in the database. Subsequent PM prototypes have been developed, evaluated, and discarded prior to the current

version described above. Knowledge gained by experimenting with one prototype PM has been used in developing the next improved PM prototype.

4.6.3 Assistance Manager

The overall goal of the Assistance Manager (AM) software tool is to provide effective and efficient transfer of assistance information to a VSSE user. "Effective" means accurate information is provided that is relevant to the user's needs. "Efficient" implies that if the user is involved in interaction with the VSSE, it is not necessary to switch tasks or modes in the process of seeking help. The following objectives are identified to meet this overall goal [Frankel 1987; Frankel and Balci 1989]:

(1) Provide general information for beginning system users. Such information would serve to acquaint new users with the environment, and establish a context for subsequent learning.

(2) Provide detailed and specific help on the use of an VSSE tool.

(3) Provide definitions and example usages of technical terms encountered in documentation and communication within the environment.

(4) Provide tutorial assistance for VSSE users. The tutorial should give the user a protected arena for limited experimentation with a tool's features.

(5) Provide help that is constantly available and immediately accessible. Methods should be available to suspend temporarily interaction with the AM, or save the current display for future reference. The user should not be required to step through an artificial protocol or syntax to access immediate assistance.

(6) Provide help that is unobtrusive; i.e., messages or prompts that are only visible when required or asked for.

(7) Provide a help system that is flexible enough to accommodate experienced users as well as novice or casual users.

(8) Provide context-sensitive help wherever possible. The system should use all available information on the user's state and avoid placing the burden on the user.

(9) Provide appropriate methods of access to the help information. Initiative, mechanisms, and complexity of access should vary according to task and type of user.

(10) Provide a straightforward and systematic method for tool developers (application programmers) to build help into tools which may be added to the environment.

(11) Provide help that is available in a consistent manner from any tool within the environment.

(12) Administer the Assistance Database by serving as an interface between the user or programmer and the database contents.

(13) Provide easy methods for update and expansion of the AM database. This is critical in order to accommodate the tailoring and updates that are inevitable in a large software environment. Updates should be enforced in a manner which helps enforce database integrity and consistency.

The AM has four components providing: (1) information on how to use an environment tool; (2) a glossary of technical terms; (3) introductory information about the environment; and (4) assistance

for tool developers for integrating help information. Evaluation of the AM with respect to the 13 design objectives stated above has found the tool able to provide effective and efficient transfer of assistance information to a VSSE user [Frankel 1987; Frankel and Balci 1989].

4.6.4 Second Category Minimal VSSE Tools

Electronic Mail System, Document Preparation System, and Text Editor constitute the second category Minimal VSSE tools. These tools are also called assumed tools or library tools and are expected to be provided by the software environment of Layer 0—Sun programming environment. Many options exist within the Sun programming environment, through Sun Microsystems and third party vendors, for providing these tools to the VSSE users.

5. EVALUATION

The VSSE's evolutionary joint development with the DOMINO has spanned between 1984 and 1992. Using the rapid prototyping approach, many VSSE tool 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.

The VSSE has served as an experimental testbed 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.

Many authors (e.g., [Bell and O'Keefe 1987; O'Keefe 1987; Paul 1989]) advocate the use of visual *interactive* simulation. We also believe in the importance of the interactive nature of visual simulation and the VSSE provides interactive capabilities through the achievement of the automation-based paradigm. During animation, if the modeler wants to change the model (e.g., components, attributes, layouts, specifications, etc.), such changes can easily be incorporated within the model specification through the use of the Model Generator's Image and Model Editors. Thereafter, a new executable model can be automatically generated via the Model Translator. A new animation is obtained by activating the Visual Simulator under the modified model. The Runtime Inspection Facility of the Visual Simulator also enables the modeler to interact with the animation by viewing

the values of the model attributes. The ability to view animation at any component level within the model hierarchy also enables the modeler to interact with the visualization of the model.

The VSSE, reported in this paper, has been applied to the modeling and visual simulation of 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 VSSE has been evaluated with respect to the 18 design objectives of Section 2 and has been found to effectively satisfy its design objectives. The evaluation is given below for each design objective. Evaluation comments are grouped by the associated VSSE tool.

VSSE (Complete toolset):

Objective 1: Provide a fully functional and highly integrated toolset under the DOMINO conceptual framework. Accomplish the integration and communication among the tools using (primarily) a relational database (e.g., INGRES) representation of the specification.

The minimal VSSE toolset (Model Generator, Model Analyzer, Model Verifier, Model Translator, and Visual Simulator) was rigorously experimented with in performing model development of a variety of problems described above. Each prototype tool performs as described in Section 4, with all the attendant capabilities listed therein. Two of the three models (Traffic Intersection and 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 (e.g., 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 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]: enable simple and natural dialogue, speak the user's language, minimize the user's memory load, promote consistency, provide feedback, provide clearly marked exits, provide shortcuts, supply good error messages, and prevent errors.

The VSSE is a fully functional prototype environment 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 for building the production version of the VSSE. Each usability principle is taken in turn for comment:

- *Enable 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. 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 10). In general, the further right that a "pull" is made, the more limited or categorized the data display becomes. Also, Section 4.3.1 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. Its 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 to guide the user. Again, context sensitive help, if available, would be extremely valuable.

- *Promote 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. 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 are made. The query facility of the Model Generator enables a user to essentially tailor his/her 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 2), the QUIT icon is unmistakable in the form of an annotated stop sign. Once a VSSE tool is entered, the RETURN button is 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 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 the use of the keyboard for supplying or typing in model information. For example, from each top level tool menu (e.g., Figure 3a to 3e), the modeler can type in the model name to be worked on, but the RETRIEVE EXISTING MODEL button eliminates the need to type. Selecting it displays a list of model names from which to choose one by clicking on the name using the mouse. Also, popup menus using the right mouse button allow easy selection of options. 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.

- *Supply 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: (a) the modeler is notified that an attribute has already been specified for the given name. (In this case, the modeler can choose to reenter the attribute name or to continue by nullifying the previous attribute data), and (b) a confirmation request is displayed for the modeler regarding the 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. 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 for the production version of the VSSE. 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's.

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

A full single inheritance mechanism (to include inheritance of attributes and model component logic specifications between objects) is available. Since INGRES does not have an imbedded query language interface for C++, Objective C, or any other object-oriented programming language, multiple inheritance was not implemented within C due to its difficulty. The production version VSSE should use a relational database management system with an object-oriented query language interface such as Sybase.

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

Attribute access is fully available with three types of attribute referencing mechanisms. Class methods and message passing are 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 nine categories of model specification information. 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.

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. The descend/ascend/top combination of "navigation" 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 example model applications [Derrick 1992]. The Traffic Intersection represents a complex (many component interactions) model. There were no noted deficiencies in expressive power. A close review of the examples of model component logic reveals the English-likeness of the VSMSL statements [Derrick and Balci

1992b]. 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 the ability to translate model component logic specifications directly into 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 Model Translator converts the VSMSL statements in the model component logic into the source code of the target language C. 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. No modeler interaction at the target C 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 Model Translator contains a facility for the dynamic creation of symbol tables. 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 Model 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, the Model Translator also provides effective syntactic analysis.

Model Analyzer:

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

The Model Analyzer diagnostic checking presents 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 the trace data and relating runtime errors to the modeler-defined specification. Use the relational database for storage and retrieval of the trace data.

Section 4.3.1 describes the Trace Manager of the Model Verifier and clearly delineates its impressive manipulation capabilities of the trace data, stored within the INGRES relational database. Additionally, useful approaches are available for relating runtime errors to the specification.

Objective 13. Provide the 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) or 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.

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, achieving the Automation-Based Paradigm [Balci and Nance 1987b].

The Model Translator, Section 4.4, accomplishes automatic full translation of the DOMINO model specification in a two-step process: creating the source code for object-oriented programming modules and compiling and linking the resulting source and object modules into the final executable model. The Automation-Based Paradigm is achieved. The modeler maintains only the model specification and does not deal with the implementation of the model at all. Model execution errors are mapped back to the specification. The modeler changes the specification in order to correct execution errors. Maintainability and reusability model quality characteristics are achieved.

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 provided. The capabilities for moving between contexts are even more powerful than the ease of movement during structure definition.

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 describes the runtime inspection facility which gives instant-by-instant monitoring capability of attributes (which can be modeler-defined performance measures) to modelers. Figure 10 demonstrates this facility and substantiates the satisfaction of this objective. Dynamic analysis of the model specification is enhanced with this feature.

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

The description of the Visual Simulator (Sections 4.5) summarizes the VSSE execution of a model without visualization while providing confidence intervals 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.

6. CONCLUDING REMARKS

A Visual Simulation Support Environment (VSSE) based on the multifaceteD cOnceptual fraMework for vIsual simulatioN mOdeling (DOMINO) is presented. The VSSE, with its fully functional and highly integrated toolset, demonstrates significant advances in the Simulation Model Development Environment (SMDE) research [Balci 1986; Balci and Nance 1987a, 1992] and automated support for visual simulation model development.

Based on the DOMINO, the VSSE brings new features to the SMDE. The automation-based paradigm is achieved. The WYSIWYR (What You See Is What You Represent) philosophy is realized. The VSSE is applicable for any discrete-event simulation problem and is domain independent. The use of a library for reusability is introduced. The Object-Oriented Paradigm is more fully implemented than in previous tools with true inheritance, methods, and message passing. Visualization and the extended use of graphical facilities for model definition and specification are explored within the SMDE research for the first time. The Model Verifier is built also for the first time.

The graphical facilities for model structure definition and contextual visualization at multiple levels of abstraction clearly offer help to the modeler in overcoming the complexity problem 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 model's hierarchical 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 model structure is successfully managed.

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. Versatile, with facilities for assertion checking, trace management, and execution profiling, the Model Verifier provides significant, additional capabilities for automated support.

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 VSSE enables the effective evaluation of the DOMINO. The non-trivial example model applications demonstrate the utility of the DOMINO for the design and implementation of visual simulation models. Guidance and support are given to the modeler through the VSSE and the underlying DOMINO for a wide variety of tasks during visual simulation model development.

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. (1986), "Requirements for Model Development Environments," *Computers & Operations Research 13*, 1 (Jan.-Feb.), 53-67.

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 (1987a), "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society 38*, 8 (Aug.), 753-763.

Balci, O. and R.E. Nance (1987b), "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.

Beams, J.D. (1991), "A Premodels Manager for the Simulation Model Development Environment" M.S. Thesis, Department of Computer Science, VPI&SU, Blacksburg, VA, Sept.

Beams, J.D. and O. Balci (1992), "Providing Reusability and Learning Support in the Simulation Model Development Environment," Technical Report TR-92-03, Department of Computer Science, VPI&SU, Blacksburg, VA, Mar.

Bell, P.C. and R.M. O'Keefe (1987), "Visual Interactive Simulation - History, Recent Developments, and Major Issues," *Simulation 49*, 3, 109-115.

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), "DOMINO: A Multifaceted Conceptual Framework for Visual Simulation Modeling," Technical Report TR-92-43, Department of Computer Science, VPI&SU, Blacksburg, VA, Aug.

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

Frankel, V.L. (1987), "A Prototype Assistance Manager for the Simulation Model Development Environment," M.S. Thesis, Department of Computer Science, VPI&SU, Blacksburg, VA, July.

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*, 699-716.

Nance, R.E. (1977), "The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models," Final Report to the National Bureau of Standards, Department of Computer Science, VPI&SU, Blacksburg, VA, June

Nance, R.E. (1979), "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards," In *Current Issues in Computer Simulation*, N. Adam and A. Dogramaci, Eds., Academic Press, New York, pp. 83-97.

Nance, R.E. (1981), "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., O. Balci, and R.L. Moose, Jr. (1984), "Evaluation of the UNIX Host for a Model Development Environment," In *Proceedings of the 1984 Winter Simulation Conference*, S. Sheppard, U.W. Pooch, and C.D. Pegden, Eds. IEEE, Piscataway, NJ, pp. 577-584.

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.

Nielsen, J. (1990), "Traditional Dialogue Design Applied to Modern User Interfaces", *Communications of the ACM 33*, 10 (Oct.), pp. 109-118.

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.

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.

Sun Microsystems (1986), *SunINGRES*, Volumes I, II, and III, Sun Microsystems, Inc., Mountain View, Calif., May.

Sun Microsystems (1988), *SunView Programmer's Guide*, Sun Microsystems, Inc., Mountain View, Calif., May.

Winkler, D. and S. Kamins (1990), *HyperTalk 2.0 The Book*, Bantam Books, New York, NY.