AN OBJECT-ORIENTED SIMULATOR
FOR THE
VISUAL SIMULATION ENVIRONMENT

by

P. Anders I. Bertelrud

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for a degree of

Master of Science

in

Computer Science

APPROVED:

Osman Balci

Osman Balci, Chairman

Richard E. Nance

Marc Abrams

September 1995

Blacksburg, Virginia
AN OBJECT-ORIENTED SIMULATOR
FOR THE
VISUAL SIMULATION ENVIRONMENT

by
P. Anders I. Bertelrud

Chairman: Osman Balci
Computer Science

(ABSTRACT)

This thesis presents an object-oriented visual simulator for the Visual Simulation Environment (VSE). VSE allows graphical model definition and provides an object-oriented Visual Simulation Language (VSL) for specifying model logic. The Visual Simulator executes VSE models and displays the visual representation of the execution. It provides a class library used by modelers' VSL logic specification, and lets users collect statistics.

Four major requirements of a visual simulator for the VSE have been identified. The simulator needs to support the object-oriented approach used in the VSE, support hierarchically defined dynamic objects, let modelers view model behavior graphically, and allow modelers to easily navigate the model.

The Visual Simulator satisfies these requirements. It fully supports the object-oriented approach used throughout the VSE by providing an object-oriented class library through which modelers access simulator features. It also provides support for hierarchically defined dynamic objects. The Visual Simulator helps users understand the structure and behavior of VSE models by providing easy viewing and navigation among model components, and by allowing inspection of object attributes.
ACKNOWLEDGMENTS

I wish to express my thanks to the people who have helped me during this research, each in their own way.

I would like to thank Chuck Esterbrook for his friendship and many great ideas.

I am also grateful to Dr. Balci for his support and efforts on my behalf, and to Dr. Nance and Dr. Abrams for serving on my committee.

Thanks to the staff of the Computer Science department and Systems Research Center for their help in administrative matters.

I am deeply grateful to my parents for giving me a great start in life, for their constant love and support, and also for their financial help during my education. Also, many thanks to my sister for her kind words of encouragement.
TABLE OF CONTENTS

ABSTRACT .................................................................................................................. ii

ACKNOWLEDGMENTS ............................................................................................... iii

TABLE OF CONTENTS .............................................................................................. iv

LIST OF FIGURES ..................................................................................................... viii

LIST OF TABLES ........................................................................................................ ix

CHAPTER 1: INTRODUCTION .................................................................................. 1
  1.1 Problem Statement ............................................................................................ 1
  1.2 Objectives .......................................................................................................... 1
  1.3 Thesis Overview ................................................................................................ 2

CHAPTER 2: RELATED WORK ............................................................................... 3
  2.1 Background ....................................................................................................... 3
    2.1.1 Object-Oriented Paradigm .......................................................................... 3
    2.1.2 Conical Methodology ............................................................................... 5
    2.1.3 Automation-Based Paradigm ...................................................................... 6
  2.2 SMDE .................................................................................................................. 7
    2.2.1 SMDE Architecture .................................................................................... 7
    2.2.2 The Minimal Toolset .................................................................................. 9
  2.3 VSSE ................................................................................................................... 10
    2.3.1 The DOMINO Conceptual Framework .................................................... 11
    2.3.2 VSSE Architecture .................................................................................... 13
    2.3.3 The Minimal Toolset ................................................................................ 13
  2.4 VSE ..................................................................................................................... 16
    2.4.1 Conceptual Framework .............................................................................. 16
      2.4.1.1 Visual Simulation Language ............................................................... 17
    2.4.2 VSE Architecture ...................................................................................... 19
    2.4.3 The VSE Toolset ....................................................................................... 20
CHAPTER 3: THE VISUAL SIMULATOR .............................................. 26

3.1 User Interface ...................................................................... 26
  3.1.1 Overview ...................................................................... 27
  3.1.2 Visualization ................................................................ 31
  3.1.3 Navigation ................................................................... 31
  3.1.4 Object Inspection .......................................................... 34
  3.1.5 Experiment Design .......................................................... 36

3.2 Visual Simulation Class Library ............................................. 36
  3.2.1 Concepts and Conventions .............................................. 38
    3.2.1.1 Communication Through Messaging ......................... 38
    3.2.1.2 Notification-Driven Logic ...................................... 39
    3.2.1.3 Machine-Oriented Approach vs. Material-Oriented Approach ...... 40
    3.2.1.4 Movement of Dynamic Objects .................................. 41
    3.2.1.5 Dynamic Object Templates ........................................ 42
    3.2.1.6 Statistics Collection .................................................. 43
  3.2.2 Classes ........................................................................ 43
    3.2.2.1 VSOBJECT ............................................................. 43
    3.2.2.2 VSLIST .................................................................. 45
    3.2.2.3 VSMODEL ................................................................ 45
    3.2.2.4 VSComponent .......................................................... 46
    3.2.2.5 VSDYNAMICOBJECT ............................................. 46
    3.2.2.6 VSIMAGE ................................................................ 47
    3.2.2.7 VSTYPE ................................................................. 47
    3.2.2.8 VSTITEM and Its Subclasses ...................................... 47

3.3 Simulation Engine ................................................................ 48
  3.3.1 Overview ...................................................................... 48
  3.3.2 Random Variate Generation .......................................... 49
  3.3.3 Error Handling ............................................................... 49
LIST OF FIGURES

Figure 2.1: Architecture of the SMDE ................................................................. 8
Figure 2.2: Architecture of the VSSE ................................................................. 14
Figure 2.3: Model Editor’s layout editor window, tool palette, and inspector .......... 21
Figure 2.4: Model Editor’s class browser and class editor window ...................... 23
Figure 3.1a: Visual simulator’s main menu ......................................................... 28
Figure 3.1b: Visual simulator’s main menu and model menu .............................. 28
Figure 3.2: Visual simulator’s Open panel ......................................................... 29
Figure 3.3: The simulator’s main model window ................................................. 30
Figure 3.4: A component viewer (shown viewing the TopLevel component) ........ 32
Figure 3.5: Navigation browser and viewer (ComputingCenter is being viewed) .... 33
Figure 3.6a: Simulator’s inspector with nothing selected ..................................... 35
Figure 3.6b: Simulator’s inspector with the Waiting Line component selected ....... 35
Figure 3.7: The visual simulator’s experiment design panel ............................... 37
Figure 3.8: Inheritance structure of the Visual Simulation Class Library .............. 44
Figure 3.9: The visual simulator’s runtime error panel (with trace panel activated) ... 51
Figure 4.1: Static hierarchy of the Computer System model .................................. 54
Figure 4.2: Layout of top level component in the model editor ............................ 55
Figure 4.3: Layout of “Computing Center” component ....................................... 57
Figure 4.4: Class browser showing the model-specific component classes .......... 59
Figure 4.5: Choosing the class for a component ............................................... 60
Figure 4.6: Dynamic object template browser after creating ComputerJob ........... 62
Figure 4.7: Code editor for the DynObjEnteredModel: method of TerminalUser .... 63
**LIST OF TABLES**

Table 3.1: Built-in random variate generation functions ........................................ 50
CHAPTER 1:  INTRODUCTION

1.1  Problem Statement

Simulation continues to gain ground as a solution technique for problems that have no feasible analytical solution. As use of simulation increases, so does the need for a software environment that supports all aspects of a simulation study; not just the programming and execution phases. Phases such as experiment design, model validation, and model verification should also be supported by the simulation software.

To this end, the Visual Simulation Environment (VSE) was developed. It is an environment intended to provide computer support for the entire simulation study life cycle.

One of the essential tools in the Visual Simulation Environment is the Visual Simulator. It is the tool that executes the model created by the user. The Visual Simulator is the topic of this thesis.

1.2  Objectives

The objective of the research presented in this thesis was to design and implement a simulator for the Visual Simulation Environment. A simulator for the VSE must meet several requirements, and it must do so at speeds that let users run sizable simulations in a reasonable amount of time. It has to:

- fully support the object-oriented approach taken by the VSE
- provide full support for hierarchical decomposition of dynamic objects
- let modelers see the behavior of the model graphically
- let modelers easily navigate the model during execution
1.3 Thesis Overview

Chapter 2 provides background information and descriptions of research that predated the development of the Visual Simulator. Among other things, it describes the *Simulation Model Development Environment* by Balci and Nance [Balci and Nance 1992a], the *Visual Simulation Support Environment* by Derrick and Balci [Derrick 1992], and *Visual Simulation Environment* by Bertelrud, Esterbrook, and Balci [Bertelrud, Esterbrook, Balci 1995].

Chapter 3 provides a functional description of the Visual Simulator. Chapter 4 describes a sample model that illustrates some of the Visual Simulator’s features. Chapter 5 evaluates the Visual Simulator with respect to the objectives outlined in section 1.2 above. Chapter 6 summarizes the contributions, and provides suggestions for future research on this topic.
CHAPTER 2: RELATED WORK

In order to understand the characteristics of the VSE Simulator, one must first look at some of the fundamental concepts used in the creation of the other VSE tools and in the work that led up to the development of the Visual Simulation Environment. This chapter describes the background concepts, the Simulation Model Development Environment (SMDE), the Visual Simulation Support Environment (VSSE), and the other tools in the VSE.

2.1 Background

This section describes three of the fundamental paradigms that underlie the Visual Simulation Environment: the Object-Oriented Paradigm, the Conical Methodology, and the Automation-Based Paradigm. More complete descriptions can be found in [NeXT Computer 1993], [Nance 1981, 1987, 1994] and [Balzer et.al. 1983], respectively.

2.1.1 Object-Oriented Paradigm

The Object-Oriented Paradigm (OOP) had its beginnings in SIMULA 67 [Birtwistle et.al. 1973], and was extended and refined in SmallTalk [LaLonde and Pugh 1990]. There are many variations of OOP, but underlying each are the fundamental concepts of objects, classes, and message passing [Sebesta 1993].

In the Object-Oriented Paradigm, a software system is viewed as a set of objects. Each object contains information and has the ability to operate on that information. The information is stored in the object’s attributes, and the ability to operate on the attributes is embodied in the object’s methods. The attributes are accessible only by the methods; no
other object may access them directly. This is the notion of data encapsulation; the attributes are encapsulated within the object and are only accessible through the object’s methods. An object is thus a form of abstract data type.

The objects in the system communicate by sending messages to each other. A message is a simply a label that identifies the operation the receiving object is requested to perform. A message may optionally contain parameter values. When an object receives a message, the method corresponding to the received message is invoked. The message identifies the behavior, and the method performs the actions necessary to implement that behavior.

Objects of the same type are said to be instances of the same class. The class defines the behavior of its instances by defining the attributes, the messages that instances can respond to, and the methods that implement the behavior of those messages. Two classes can specify different methods for the same message. In this case, that same message sent to instances of the two classes will result in different behavior. The notion that each object independently determines the behavior that corresponds to a message is called polymorphism.

Classes are organized in an inheritance structure. Each class inherits the attributes and methods defined in the classes above it in the structure. It can also define additional attributes and methods. If a class that inherits a method definition for a particular message also contains its own method definition, the latter definition overrides the inherited definition. As part of its implementation of the method, the class can invoke the behavior of the overridden method definition.

The classes from which a class directly inherits are called its superclasses, and the classes that directly inherit from it are its subclasses. In many versions of the Object-Ori-
mented Paradigm, each class can have only one *superclass*; this is called *single inheritance*. In this case the class inheritance structure is a tree hierarchy; in the case of *multiple inheritance*, it is a directed acyclic graph.

OOP is very well suited to discrete-event simulation; a model is naturally viewed as a set of interacting objects. It is no coincidence that the roots of OOP lie in SIMULA 67.

2.1.2 Conical Methodology

The Conical Methodology (CM) was defined and developed in the late 1970s by Naace [1994]. It is an object-oriented, hierarchical modeling methodology that emphasizes model correctness, testability, adaptability, reusability, and maintainability. The separation between model specification and model execution is also stressed.

The CM mandates top-down definition of the components of a simulation model, followed by bottom-up specification. During the definition phase, the model is decomposed into its constituent submodels, which can in turn be decomposed into smaller submodels. This hierarchical definition can be carried out to as fine a level of granularity as desired. During the definition of each submodel, its attributes are defined, as well as the valid ranges of values for those attributes. The definition phase can be either depth-first (where each submodel is decomposed as far as needed before any of its peer submodels) or breadth-first (where all submodels at a particular level are completely defined before moving on to the next level). This decision is left to the discretion of the modeler.

After the definition of a submodel, its behavior is specified. This is done during the specification phase, which proceeds from the bottom and up. A simulation model specification and documentation language (SMDSL) is used to specify the time-based changes in the attributes defined during the definition phase.
In this manner, successive model representations are developed, each more detailed than the previous. At each level of abstraction, the model can be analyzed for correctness, completeness, and consistency. When a sufficiently detailed specification has been achieved, the model can be translated into an executable representation.

2.1.3 Automation-Based Paradigm

In the early 1980s, Balzer et.al. recognized two fundamental flaws in the software maintenance process as performed at that time [Balzer et.al. 1983]. First, there was a lack of automated support for all phases of the software development life cycle except implementation. This caused severe problems, particularly in the maintenance phase. Second, maintenance was performed on the implementation rather than the specification. Source code is generally much more difficult to understand than the specification, because global dependencies have been introduced and complex optimized implementations substituted for simple concepts. This exacerbates the maintenance problem.

The solution Balzer et.al. proposed to this problem was to perform all maintenance on the specification rather than the implementation. This requires reimplementation and reverification of the implementation each time the specification is changed, a time-consuming and costly task. Balzer et.al. thus argued that this task must be at least partially automated. They gave two reasons why fully automated implementation is not feasible, and outlined a process of automated support that yields fast, reliable, and inexpensive reimplementation [Balzer et.al. 1983]. They put forth the characteristics of an "automated assistant" that would support software maintenance.

While some of the automated support called for by Balzer et.al. [1983] has been realized, much is still lacking in modern software development systems.
2.2 SMDE

In 1983, Balci and Nance started a research project aimed at developing a domain-independent discrete-event Simulation Model Development Environment (SMDE). This work resulted in the SMDE prototype described in [Balci and Nance 1992a]. Some of the following description is taken from their overview.

The goal of the SMDE project was to create an integrated and comprehensive collection of computer-based tools to: (1) offer cost-effective, integrated and computer-aided support of model development throughout the entire model life cycle; (2) improve the model quality by effectively assisting in the quality assurance of the model; (3) significantly increase the efficiency and productivity of the project team; and (4) substantially decrease the model development time [Balci and Nance 1987a].


2.2.1 SMDE Architecture

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

Layer 0, the Hardware and Operating System, originally consisted of a VAX 11/780 minicomputer accessed through a character terminal. In later versions, the hardware/operating system layer consisted of a Sun computer workstation with the UNIX SunOS 4.0 operating system. The INGRES relational database management system was used for model storage.
Figure 2.1: Architecture of the SMDE
Layer 1, the Kernel SMDE, is the layer that integrates all SMDE tools into the operating system’s programming environment. To achieve a high level of maintainability and extensibility, all tools communicate through the kernel interface.

Layer 2, the Minimal SMDE, provides a minimal set of tools required to support all phases of model development in the SMDE. The minimal toolset is general-purpose, in that it is designed to be applicable to any discrete-event simulation model. The minimal toolset is described in more detail in section 2.2.2.

Layer 3, the SMDEs, expands on the defined minimal SMDE. This layer incorporates tools that have specific applications and that are needed either for a particular project or by an individual modeler. This layer is optional; the minimal toolset is sufficient for the development of simulation models in the SMDE. The open space at the bottom of the circle representing layer 2 in Figure 2.1 emphasizes that level 3 tools have access to the kernel; hence they can be fully integrated into the rest of the environment.

2.2.2 The Minimal Toolset

The minimal SMDE toolset consists of the following tools: Project Manager, Premodels Manager, Assistance Manager, Command Language Interpreter, Model Generator, Model Analyzer, Model Translator, and Model Verifier. The Project Manager manages the administrative tasks of the model development project. The Premodels Manager manages and organizes libraries of reusable model components, while the Assistance Manager provides context-sensitive help to the user. This help includes tutorials, user manuals for the tools, and documents describing simulation techniques. The Command Language Interpreter was originally a text-based tool that let the user interact with the SMDE, but in later versions it was replaced by a graphical SunView user interface. The Model Generator, identi-
fied by Balci and Nance [1987a] as the most critically important tool in the SMDE, lets the user define the architecture of a model and specify its logic. It thus supports both the definition and specification phases of the Conical Methodology. Four prototypes of the Model Generator were developed for the SMDE. The Model Analyzer performs static model verification. The Model Translator translates the high-level model specification to C code, which is then compiled to executable code. Finally, the Model Verifier performs dynamic verification of the executable model.

Various prototypes of the Model Generator, Model Analyzer, Model Translator, Model Verifier, and Command Language Interpreter tools have been developed. The Pre-models Manager and Assistance Manager were also prototyped. The Project Manager was never prototyped, because it had the lowest priority in the environment. The tool prototypes were not integrated to the extent intended by the design of the environment.

2.3 VSSE

The Visual Simulation Support Environment (VSSE) was a fully functional research prototype of the SMDE Project. It was developed on a Sun workstation. It was built based on the experience gained from the SMDE prototyping efforts. However, it had a new conceptual framework, called DOMINO, developed by Derrick and Balci [Derrick and Balci 1995b; Derrick 1992]. It is briefly described in section 2.3.1 below. In addition to the new conceptual framework, the VSSE contained improved versions of the basic simulation tools that were first prototyped in the SMDE. Especially the Model Generator was significantly improved in the VSSE. Also significant was the addition of a simulator tool.
2.3.1 The DOMINO Conceptual Framework

The DOMINO conceptual framework [Derrick 1992] was developed between 1984 and 1992. It adheres to the principles of the Conical Methodology, allows the user to work in the Object-Oriented Paradigm, and embodies a WYSIWYG (What You See Is What You Represent) philosophy. In the DOMINO conceptual framework, a model is comprised of submodels, static objects, dynamic objects, subdynamic objects, and base dynamic objects. As in the Conical Methodology, the model is decomposed into a hierarchy of submodels. In addition, submodels can contain static objects, which are considered the smallest elements of interest in the model. Hence, static objects are not decomposable. Together, submodels and static objects comprise the static structure of the model.

While a model has only one static structure, it has zero or more dynamic structures. The dynamic structures of the model consist of dynamic objects, subdynamic objects and base dynamic objects. These represent the dynamic aspects of the modeled system. The root of every dynamic structure is a dynamic object, which can be further decomposed into subdynamic objects and base dynamic objects. Subdynamic objects can be further decomposed, while base dynamic objects are considered the smallest elements of interest in the dynamic structure. In this manner, subdynamic objects are the dynamic counterparts of submodels, while base dynamic objects are the dynamic counterparts of static objects. While the static model structure exists from the start of the simulation until the end, the dynamic structures are created, move through the static structure (or other dynamic structures), and are finally destroyed.

In addition to the categorization as static or dynamic, any model component in the DOMINO conceptual framework can also be categorized as real or virtual. Real components represent objects that have a real representation in the system being modeled. In
contrast, virtual components represent abstract concepts such as random variate generators.

**DOMINO** also specifies that each submodel and subdynamic object has a *layout* that contains its graphical representation. The modeler creates these layouts in the Image Editor window of Model Generator (briefly described below) by using a pointing device to draw graphical shapes representing the logical submodel components. In this manner, the top-down definition phase mandated by the Conical Methodology is performed graphically.

The **DOMINO** conceptual framework also introduces the concept of *paths* that connect submodels and static objects (or their dynamic counterparts, subdynamic objects and base dynamic objects). The paths allow dynamic objects to move from one submodel to another; no movement is allowed outside of paths. There are several rules that restrict the use of paths [Derrick 1992, pages 57, 59].

**DOMINO** allows users to create classes for dynamic objects, submodels, static objects, subdynamic objects, and base dynamic objects [Derrick 1992, p.148]. There is a class hierarchy for each of the five component types. The logic of each class is specified using the *Visual Simulation Model Specification Language* (VSMSL), a language based on Apple’s HyperTalk Language [Winkler and Kamins 1990]. Three types of logic can be attached to a class: *supervisory logic, self logic*, and *method logic* [Derrick 1992, p.70]. Supervisory logic is attached to components such as submodels to allow them to “supervise” (i.e. control) dynamic objects contained in them. Self logic is attached to dynamic objects to let them control their own motion. Finally, method logic is logic in the form of conventional object-oriented methods associated with the class.
2.3.2 VSSE Architecture

The architecture of the VSSE is very similar to the architecture of the SMDE; the same four-layer approach is used. The main difference in the overall design is the addition of a Visual Simulator tool to the minimal toolset, a tool made possible by the use of a graphical user interface. Also, the tools present in the SMDE are improved in the VSSE; most notably the Model Generator. They are also integrated to a greater extent. Figure 2.2 shows the architecture of the VSSE.

Another difference is that SMDE’s Command Language Interpreter tool has been replaced by the SunView graphical user interface, thereby greatly improving ease of use.

Like the SMDE, the VSSE uses an INGRES database to store models. The VSSE does not reuse any source code from the SMDE project.

2.3.3 The Minimal Toolset

The VSSE’s minimal toolset is essentially the same as the toolset of the SMDE, the main difference being the addition of a Visual Simulator tool.

The VSSE Model Generator consists of the Image Editor and the Model Editor [Derrick 1992]. The Image Editor provides various tools for editing layouts, while the Model Editor lets the user perform definition and specification of the model hierarchies. It lets users specify object class information, associate layouts created in the Image Editor with instances of the classes, and identify the paths, connectors, and interactors in each class layout [Derrick 1992, p.83]. The user specifies class logic using VSMSL. All information entered by the user is stored in the model database.
Figure 2.2: Architecture of the VSSE
The Model Analyzer examines the model specification in the database, and checks its consistency and completeness. Various aspects such as logic specification, layout definition, and documentation are analyzed.

The Model Translator converts the user's VSMSL class logic to C source code, which is then compiled to produce the executable simulation program. When generating the source code, the Translator takes into account options set by the user through the Model Verifier tool. These options are assertion checking, runtime tracing, and execution profiling [Derrick 1992, p.118]. The Model Translator also lets the user examine the trace data and execution profile.

The VSSE Visual Simulator executes the compiled code generated by the Model Translator, and draws the animation on the screen. Technically, the Model Translator produces a stand-alone executable program that has the necessary simulation time-flow logic linked in. In other words, the translated model logic is fused with simulation facilities to produce a visual simulator program that is particular to the model. While the VSSE Visual Simulator tool contains the features necessary for simulation, it has certain shortcomings. For example, it does not support movement of decomposed dynamic objects across component hierarchy boundaries [Derrick 1992, p.156].

The Premodels Manager and Assistance Manager are similar to their SMDE counterparts, and have been implemented outside the VSSE environment (the previously mentioned tools are all integrated).
2.4 VSE

Based on experience gained from the SMDE and VSSE prototypes, development of the production version of the model development environment, named the Visual Simulation Environment, started in August 1992. The platform chosen for this final version was the Unix-based NeXTSTEP operating system. The main reason for this choice was NeXTSTEP’s excellent object-oriented software development environment. The source code for VSE was written entirely in Objective-C in order to take full advantage of NeXTSTEP’s powerful software development features.

2.4.1 Conceptual Framework

The conceptual framework used in the VSE is based on the DOMINO conceptual framework. Like DOMINO, the VSE conceptual framework is object-oriented. Also as in DOMINO, a model is viewed as consisting of a static structure and a set of dynamic structures. In DOMINO, these static and dynamic model structures consist of submodels, static objects, subdynamic objects, and base dynamic objects (as described in section 2.3.1 on page 11). In VSE, these types of constructs have been unified into a single type of building block: a component. This significantly increases reusability, because a modeler can define a single component type, for example an MM1Queue type, which can then be used in both the static and dynamic model structures. Note that the distinction between static and dynamic model components remains, since each instance of MM1Queue is either static or dynamic. The unification simply means that the class is easier to reuse.

Modelers graphically define the static hierarchy of a model using the VSE Editor tool (described below), by decomposing the model into its components in a process of
stepwise refinement. Each component has a layout where graphical shapes can be placed. Also, every component is represented by a shape in its parent component’s layout.

Similarly, modelers define the dynamic hierarchies by defining and decomposing dynamic objects. Since dynamic objects do not exist until the model is executed, modelers actually create *dynamic object templates* in the VSE Editor. At runtime, the simulator instantiates these templates to create the dynamic object hierarchies.

As each component in the model structure is defined, the modeler identifies and defines the conceptual behavior of that component. In accordance with the object-oriented paradigm, component behavior is encapsulated in *classes* that are organized into a single class hierarchy. This is different from DOMINO’s five class types (dynamic object, submodel, static object, subdynamic object, and base dynamic object). VSE still provides distinction between components and dynamic objects, but this is accomplished within the single class hierarchy. Although accessible from the Editor, the VSE class hierarchy is an integral part of the Visual Simulator; see section 3.2 on page 36 for a detailed description.

When a class representing the component has been defined, the modeler declares the component to be an instance of the class, meaning that it exhibits the behavior of that class. Several components can use a single class definition, thus achieving a high degree of reusability.

When the modelers are ready to specify the class logic, they do so in the *Visual Simulation Language (VSL)*.

2.4.1.1 Visual Simulation Language

The Visual Simulation Language is the language in which modelers specify logic for the classes they have defined. It is an object-oriented imperative language that has statements,
operators, expressions, selection and iteration constructs, and message passing. It provides
statically typed basic data types (character, integer, real, string, boolean, and object refer-
ence). References to objects are first-class data types; they can be passed as parameters to
other methods, and returned as results. It is important to note that it is the object refer-
ences, not the objects themselves, that are passed as parameters and returned as results.
The message recipient receives another reference to the same object, not a copy of it.

Though a complete description of VSL is beyond the scope of this thesis, a brief
description of its messaging facilities is given here. Complete information can be found in
the VSE Reference Manual. [Bertelrud, Esterbrook, and Balci 1995].

One of the most frequently used VSL constructs is the message send. There are var-
ious forms of this operation; modelers choose the form that is most readable in each par-
ticular situation. The tell statement is used to send a message to an object, either to instruct
it to perform an action:

\[
\text{tell anObject to performAnAction}
\]

or to notify it of an event:

\[
\text{tell anObject that somethingHappened}
\]

The \text{<message> of <object>} form is used when the message returns a value:

\[
\text{set aVariable to [aValue of anObject]};
\]

Although a variety of language forms can be used when sending a message, the semantics
are always the same. The message, along with its optional parameters, are passed to the
target object, which decides (based on its class) which method to invoke. This method is
then executed synchronously, after which control (and possibly a return value) is returned
to the sender. Execution of the sender’s logic continues with the statement following the
message send.
The parameter *anObject* in the *tell* statement above is an *object reference*. An object reference is a label that uniquely identifies a particular object in the model. Object reference is a basic data type in VSL, like integer and string. Object references are used whenever there is a need to refer to an object; e.g. when specifying the receiver of a message, or the destination of dynamic object movement. Object references correspond to object pointers in C++ and Objective-C.

Object references are either *generic* or *specific*. Generic object references are declared as “obj ref” in VSL. They can be used to refer to objects of any class. Specific object references, declared as “classname ref”, provide a means for modelers to declare that the referred-to object is expected to be an instance of the named class. When a specific object reference is used during model execution, the class of the referred-to object is checked. If it doesn’t match the modeler’s declaration, a run time error occurs.

The sending of messages is the sole means of communication between objects in a VSE model (and indeed between the Simulator proper and the model objects).

### 2.4.2 VSE Architecture

At the core of the VSE architecture is the hardware and the operating system (NeXTSTEP). VSE currently runs on Intel processors and NeXT workstations, and is expected to run without source code modification on any other platform to which NeXTSTEP is ported\(^1\). NeXT Computer’s open operating-system standard, *OpenStep*, is very similar to NeXTSTEP. When OpenStep-compliant system software becomes available, modifying the VSE to use OpenStep instead of NeXTSTEP as the core should require only moderate

---

\(^1\) NeXTSTEP currently runs on Intel processors and NeXT, Sparc and PA-RISC workstations.
effort. Then VSE will be able to run on any OpenStep-compliant operating system, e.g. Microsoft Windows NT.

On top of the core is the environment layer. This layer provides the shell into which the VSE tools are “plugged”. The shell provides an icon bar that lets users launch the various tools.

Above the environment layer is the tool layer, the layer containing the VSE toolset. It is an open layer; new tools can be developed and easily integrated into the tool layer.

2.4.3 The VSE Toolset

The VSE toolset consists of the Model Editor, Visual Simulator, Model Analyzer, Model Tester, Data Analyzer and Learning Support System (LSS).

2.4.3.1 Editor and Compiler

The VSE Editor lets modelers view and edit all aspects of a simulation model. It features a graphical user interface that adopts a direct-manipulation approach. Modelers select information shown on the screen using the mouse, and can then change the information using mouse clicks and keystrokes.

The layout editor window, shown in Figure 2.3, lets the user construct and modify the static model hierarchy, using the tools on the tool palette to the left of the window. These tools allow the modeler to draw circles, rectangles, text fields and freehand shapes, to decompose the current component into subcomponents, to add entry and exit points, and to connect the components using paths. Users can also include pictures in the layout by simply dragging the icon of an image file from another NeXTSTEP application into the layout editor window.
Above the editor window is a browser (a standard NeXTSTEP user interface device) that lets the modeler navigate the static hierarchy. Modelers use this browser to choose which layout is shown in the editor window. To the right of the window is the inspector, which lets the user examine and modify the attributes of the currently selected layout item. Attributes such as the name, class, and visual appearance of the layout item can be changed.

To create the Dynamic Object Templates that the Visual Simulator uses to instantiate dynamic objects at runtime (see Chapter 3), the modeler uses similar editing facilities. In fact, dynamic model hierarchies are structurally identical to the static model hierarchy, except that whereas the static hierarchy is constant during simulation, dynamic hierarchies are created and destroyed.

Figure 2.4 shows the class editor window. This window lets a user create and edit both class definition (attributes and method names) and specification (logic). As with graphical decomposition, there is a browser window above the class editor. However, this browser shows the model’s class hierarchy, letting the modeler choose the class to be edited.

In the SMDE and VSSE, the Model Translator was a separate tool in the environment. In VSE it is an integral part of the Editor, invoked by choosing “Prepare for Simulation” from the Editor’s “Model” menu. In this way, modelers need not invoke a separate tool to translate and compile the model code, nor need they even be aware that translation is being performed. In the future, model preparation and VSL script compilation will be completely transparent to the user. A script will be compiled as soon as the user is finished editing it, the user being notified only of scripting errors. This is the philosophy used in
other scripts systems such as AppleScript [Apple Computer 1993], where users need not even be aware that their script is being translated to some lower-level form.

2.4.3.2 Visual Simulator

The Visual Simulator is the tool responsible for executing and animating simulations. As it is the topic of this thesis, it is described in detail in Chapter 3.

2.4.3.3 Model Analyzer

The Model Analyzer performs static model analysis, checking completeness and consistency. For example, if a modeler defines an instance variable but never uses it, or forgets to specify the class of a component, the Model Analyzer warns the modeler of this problem. This static analysis is actually performed by the Compiler while the model is being compiled, and thus occurs automatically when the modeler chooses “Prepare for Simulation”. Nevertheless, it is regarded as a separate tool for conceptual reasons.

The Model Analyzer tool has not yet been developed in the VSE, although most of its functionality is already performed by the Compiler.

2.4.3.4 Model Tester

The Model Tester is the dynamic counterpart of the Model Analyzer. It fills the function of the tool called the “Model Verifier” in the SMDE and VSSE. It performs the dynamic analysis of the model by executing it and performing assertion checking, bounds checking, etc. The Model Tester’s functionality is actually in the Visual Simulator; the Model Tester is a separate tool for conceptual reasons.

Much of the dynamic model analysis has been implemented in the Visual Simulator and is further discussed in Chapter 3, even though the Model Tester tool per se has not yet been developed.
2.4.3.5 Data Analyzer

After executing the prepared simulation models and collecting statistics, modelers use the Data Analyzer to construct confidence intervals and to plot histograms of the collected statistics. The Data Analyzer is described in detail in [Tuglu 1995].

2.4.3.6 Learning Support System

The Learning Support System (LSS) is a tool that lets modelers browse documents containing text, graphics, and hypertext links. Users can search for keywords in the documents, and can also view a picture of the simulation life cycle according to Balci [Balci 1994b]. This tool, combined with an appropriate set of documents, lets modelers browse the VSE documentation, read papers about the field of simulation, and to learn how to conduct simulation studies. All VSE documentation has been entered in electronic form into the LSS document database. LSS is further described in detail in [Harrichunder 1994].
CHAPTER 3: THE VISUAL SIMULATOR

This chapter describes the Visual Simulator, the tool in the VSE toolset that executes the user’s model and shows the visualization on the screen.

The Visual Simulator can be thought of as consisting of three tightly interconnected components: the user interface, the runtime class library, and the simulation engine. These components are all designed to work together; each component is dependent on the other two. These dependencies are described below.

Users who simply run simulation models built by someone else need only be aware of the user interface. Modelers wishing to build models using VSE need to be aware of the user interface and the runtime class library. They also need to be aware of the effects of the simulation engine, but need not be aware of the details of its operation.

3.1 User Interface

The Visual Simulator’s user interface serves two purposes. It presents the execution of simulation models graphically, and it lets users control the simulation. Because NeXT-STEP was chosen as the operating system for the VSE software, the Visual Simulator’s user interface adheres to the guidelines for user interface design established by NeXT Computer [NeXT Computer 1994a]. This increases ease-of-use, because users of the VSE are already familiar with the NeXTSTEP user interface conventions. Where applicable, the Visual Simulator also uses the same user interface components as the other VSE tools, again in order to increase ease-of-use and to reduce the learning curve.

The following description assumes familiarity with the NeXTSTEP graphical user interface. In particular, familiarity with tasks such as launching applications, selecting
menu options, resizing and closing windows, and manipulating scroll bars and switches is assumed. These tasks are described in [NeXT Computer 1994b].

3.1.1 Overview

The Visual Simulator’s main menu is shown in Figure 3.1a. When the simulator is first activated, only the main menu is visible. This is the main point of control in the user interface. Through its submenus (brought forth by clicking on the menu items marked with \[\text{\textbullet}\]), users perform actions such as opening models and activating the object inspector.

A simulation model is opened by choosing “Open” from the “Model” menu (shown activated in Figure 3.1b). This brings up NeXTSTEP’s standard Open Panel, shown in Figure 3.2. This standard feature of NeXTSTEP, which lets the user navigate the Unix file system and select a document for opening, is used by almost all NeXTSTEP applications. Its operation is thoroughly described in [NeXT Computer 1994b].

After the user chooses a model the main model window appears, as shown in Figure 3.3. This window displays information about the state of the simulation, and it also lets users control the simulation. The buttons in the upper right corner let the user, from left to right, stop, start, and pause simulation. The button labeled with an eye allows users to open component viewers on the model. Component viewers are described in section 3.1.2. The browser beneath the simulation controls shows the model’s static hierarchy. It is described in section 3.1.3.

In Figure 3.3 the model is not being simulated; therefore, the labels that show the current replication, the simulation clock, and the number of departed dynamic objects are blank. During simulation, these fields are continuously updated to reflect the current state.
Figure 3.1a: Visual simulator’s main menu

Figure 3.1b: Visual simulator’s main menu and model menu
Figure 3.3: The simulator's main model window
The switch labeled “Visualization” controls whether or not visualization (as described in section 3.1.2) is on.

3.1.2 Visualization
Clicking on the picture of an eye opens a component viewer. This is illustrated in Figure 3.4. A component viewer is a window that graphically displays the contents of a component, along with the dynamic objects that are currently in that component. As the simulation progresses, the dynamic objects shown in the viewers are animated such that they always reflect their current location in the model. The icons that represent the dynamic objects can also be changed during the simulation to show object state. This is described further in section 3.2.2.5 on page 46.

Viewer windows can be resized and closed. If the layout being viewed is larger than the viewer window, the scroll bars on the left and bottom of the viewer can be used to show different parts of the layout. At any time, the user can save the current size, position, and viewed component of all open viewers by choosing “Save Viewers” from the Simulator’s “Model” menu. The next time the model is opened in the Simulator, viewers will be opened to match the saved configuration.

Visualization can be switched on or off at any time during model execution. With visualization switched off, simulation speed is significantly increased. The outcome of the simulation is the same regardless of whether simulation is on or off.

3.1.3 Navigation
The browser above the visualization area (shown in Figure 3.5) shows the model’s static hierarchy. The currently viewed component is highlighted. Components that are decomposed into subcomponents are marked with the symbol. Clicking on such a component
shows a list of its subcomponents. Doubleclicking on a component changes the viewer to display the chosen component.

Subcomponents can also be descended into by doubleclicking on their shapes in the component viewer. Similarly, doubleclicking on an exit point causes the viewer to switch to the current component’s parent component. In both of these cases, holding down the Alternate key while doubleclicking opens a new viewer rather than replacing the contents of the current one.

Dynamic hierarchies can also be viewed. If the user doubleclicks on a dynamic object that is decomposed (i.e. that has a hierarchy of components), the viewer changes to show the top level of the selected dynamic object’s hierarchical structure. This dynamic structure can then be viewed and navigated in exactly the same manner as the static structure. If the dynamic object being viewed departs from the model (i.e. goes of out existence), any viewers showing its hierarchy become blank.

### 3.1.4 Object Inspection

Choosing “Inspector” from the Visual Simulator’s “Tools” menu shows the Inspector Panel. This panel shows information about the currently selected object. The appearance of the panel changes depending on the type of the selected object. If nothing is selected, the panel simply displays the message “No Selection” (Figure 3.6a). If a model object is selected, its attributes are shown, as illustrated in Figure 3.6b.

While the Simulator’s Inspector Panel is similar to the Editor’s, there is one important difference: in the Simulator, the user may not modify the attributes. Such modification could compromise the validity of statistics collected during the simulation run. The VSE does not currently support user interaction with the simulated model.
Figure 3.6a: Simulator's inspector with nothing selected

Figure 3.6b: Simulator's inspector with the Waiting Line component selected
3.1.5 Experiment Design

Upon starting the simulation (by clicking on the button marked with a VCR-style “Play” icon), the user is presented with the dialog panel shown in Figure 3.7. This panel lets the user enter the number of simulation runs to be performed, and the lengths of the warm-up and steady-state periods. These lengths can be specified either in terms of simulation time units, or in terms of number of dynamic objects leaving the model. The latter capability is useful if, for example, the modeler wants to simulate exactly 10000 jobs being processed by a computer, after 1000 jobs of warm-up. The user can also specify the number of replications to run.

When the user clicks “OK”, the simulation starts. The simulation will run until completed, or until the user stops it by clicking the stop button. The user can pause and resume the simulation without affecting the outcome of the simulation in any way. As described earlier, visualization can also be turned off at any time to speed up the simulation.

3.2 Visual Simulation Class Library

VSE’s Visual Simulation Class Library (VSCL) is a set of classes that provides the modeler with the basic functionality required for any simulation model. It contains classes that represent simulation-specific concepts, such as components and dynamic objects, as well as general-purpose concepts, such as lists of objects. It forms the upper part of VSE’s single-inheritance class hierarchy; modelers create their own subclasses of the VSCL classes in order to add their model-specific logic.

When a modeler creates a new model in the Model Editor, the new model’s class hierarchy consists of only the Visual Simulation Class Library. As the modeler identifies
Figure 3.7: The visual simulator's experiment design panel
new classes specific to a model, they are defined by creating subclasses of the VSCL classes. The custom classes thus inherit default behavior from the VSCL classes.

3.2.1 Concepts and Conventions
This section describes some of the concepts underlying the Visual Simulation Class Library, and also some of the conventions used throughout the VSCL.

3.2.1.1 Communication Through Messaging
As described in section 2.1.1 on page 3, messaging is one of the fundamental concepts of the Object-Oriented Paradigm. In the VSE, all inter-object communication is performed by sending messages; not only communication between the modeler’s objects, but also communication between the simulator and the custom objects.

A key difference between VSE and the VSSE lies in this single method of communication. For example, the VSSE’s VSMSL language has a special statement, the move statement, which is used to move a dynamic object from one location to another. By contrast, VSE modelers use the generic tell statement to send a moveTo: message to the dynamic object. This reflects the different outlook used in the VSE: all objects are entities in their own right that are told to perform an action, instead of being acted on by some other entity.

This method also has the benefit of letting the modeler intercept any system method, for example in order to collect statistics. By implementing the moveTo: method in their subclasses of the VSDynamicObject class (described in section 3.2.2.5 on page 46), modelers can write logic that will be executed whenever instances of the implemented class are asked to move. As part of their implementation modelers can invoke the default behavior of moveTo:, which is to move the object.
3.2.1.2 Notification-Driven Logic

As described in section 3.2.1.1, modeler-written logic asks the runtime system to perform actions by sending system-defined messages (the behavior of which can be redefined by the modeler). Similarly, the runtime system notifies model-specific logic when certain events occur. This notion of notification-driven logic is used extensively throughout the Visual Simulation Class Library.

For example, when a dynamic object enters into a component, the **dynObjArrived:** notification message is sent to the component into which the dynamic object has arrived. By implementing the **dynObjArrived:** method in his subclass of the VSComponent class (described section 3.2.2.4 on page 46), the modeler can specify logic to move the dynamic object that just arrived, to collect statistics, or to perform any other action.

Whenever a notification message is sent to a dynamic object, a corresponding notification message is sent to the component containing the dynamic object. Thus, in the case of a dynamic object arriving at a component, the **dynObjArrived:** message (sent to the component) is accompanied by a **arrivedAt:** message to the dynamic object.

Not all notification messages occur as the result of dynamic object movement. For example, the **startUp** message is sent to every static object when model execution starts, and to every dynamic object when it is created. Similarly, the **shutDown** message is sent to static objects when the model has completed execution, and to dynamic objects when they are destroyed.

While modelers are free to implement notification methods in order to respond to events in the model, they are not required to do so. They may choose to ignore any notification message by simply not implementing the corresponding method.
3.2.1.3 Machine-Oriented Approach vs. Material-Oriented Approach

When building a model, two general approaches can be taken. In the machine-oriented approach, the static components of the model contain the logic necessary to control the dynamic objects. When a dynamic object enters a static component a notification message is sent to the component, as described in section 3.2.1.2. The modeler would implement the method for that notification message, and specify logic to tell the newly arrived dynamic object to perform some action. In this approach, the dynamic entities of the model have no logic of their own; they are told what to do by the static components.

In the material-oriented approach, on the other hand, the dynamic objects contain the logic to move themselves through the static hierarchy. In this approach, it is the static components who do not have any logic to be executed. A dynamic object enters the model in a certain component; it will receive a notification message (as described in Appendix A) informing it of its entry into the model. In the implementation of the corresponding method, the modeler includes code that moves the dynamic object to its first destination. When it arrives, it will receive another notification message, to which it responds by moving to its second destination. The dynamic object alone controls its movement through the model.

In the VSE, modelers can use either the machine-oriented or the material-oriented approach. Alternatively, a mixture of the two could be used. The dynamic objects could determine their own actions for many cases, leaving it up to the static components to handle the rest. Because notification messages are always sent in pairs, one to the dynamic object and one to the component the dynamic object resides in, the modeler is free to choose either approach for each specific case.
3.2.1.4 Movement of Dynamic Objects

During model execution, dynamic objects move throughout the model’s static and dynamic hierarchies. As described in section 2.4.1 on page 16, each hierarchy is a tree structure in which the nodes are components. The component a dynamic object resides in is known as its location. The part of the location’s layout it occupies is called its position. A dynamic object can reside in only one location at a time.

Movement of a dynamic object always occurs from its current location to a destination component which is either specified or implied. The destination component may be the current location’s parent component, one of the current location’s child components, or the root component of a hierarchically defined dynamic object. Movement is initiated by sending one of the moveTo... messages to the dynamic object; the exact message sent depends on the nature of the destination. The available messages are described in detail in the class documentation for VSDynamicObject in Appendix A.

All dynamic object movement occurs through doorways, as described in Appendix A. If the destination is the current location’s parent, the dynamic object moves graphically to an exit doorway (which is either specified or implied). It leaves its current location through the doorway, and appears on the previous location’s representation in the parent component. A dynamic that moves up from the top level of the static model hierarchy is automatically destroyed. A dynamic object moving up from the top level of a dynamic object appears on the dynamic object’s representation. This is described in further detail in Appendix A.

If the destination is a child component or a dynamic object also residing the in the moving dynamic object’s location, the dynamic object moves graphically to the destination’s representation in the current location, after which it appears on an entry doorway in
the destination component. Movement to a dynamic object that is not hierarchically decomposed is not allowed.

Modelers can associate a duration with each movement. The duration is the amount of time (in simulation time units) needed to complete the move. When the dynamic object is told to move, it initiates the move. It arrives at its destination after the duration has passed.

Whenever a dynamic object departs from a component, notification messages are sent to all involved objects. Similarly, when it arrives at its destination, another set of messages is sent. These messages are described in detail in the "Notification Messages" section of the VSDynamicObject class documentation in Appendix A.

3.2.1.5 Dynamic Object Templates

The model's static component hierarchy exists throughout execution of the model; it is defined graphically in the Model Editor, and remains unaltered during execution of the model. Dynamic objects, on the other hand, are created and destroyed as model execution progresses. Since there are no dynamic objects in the model when execution is first started, they have to be created by the static architecture when execution begins. For example, the \texttt{startUp} method of an Airport component may contain logic to create the aircraft originally at the airport.

In order to facilitate the runtime creation of complex hierarchical dynamic objects, VSE uses a construct called \textit{Dynamic Object Templates (DOTs)}. Modelers identify the various types of dynamic objects that will be needed during simulation, and defines a dynamic object template for each type. At runtime, dynamic object instances can be created from the templates. Each dynamic object is an identical copy of the DOT it was cre-
ated from. This mechanism allows arbitrarily complex dynamic objects to be easily created at runtime.

3.2.1.6 Statistics Collection

In the VSE, statistics collection is carried out using the notification mechanism. For example, the `startUp` method of a component can contain logic to open a statistics file for writing (see the description of the VSFile class in section 3.2.2.7 on page 47). During the simulation run, the component then records the arrival and departure times of dynamic objects (by implementing the `dynObjArrived` and `dynObjDeparted` methods, respectively). When the component receives the `shutDown` message, indicating the end of the simulation run, it computes a statistical value and writes it to the file (by sending a message to the VSFile object opened earlier).

Modelers can then use the VSE Data Analyzer tool to view the collected statistics.

3.2.2 Classes

The classes in the Visual Simulation Class Library are shown in Figure 3.8. Inheritance is also indicated, with classes to the right inheriting from those on the left. Each class in now described briefly. Detailed class descriptions, along with the definition of each of the messages each class can respond to, are given in Appendix A.

3.2.2.1 VSOBJECT

VSOBJECT is at the root of the class hierarchy. It represents the most general type of object, and contains only those methods that are common to all types of objects. All other classes in the VSE model’s class hierarchy (both those in the VSCL and the modeler-defined classes) inherit either directly or indirectly from VSOBJECT. Therefore, all messages defined in VSOBJECT can be sent to any object in the model.
Figure 3.8: Inheritance structure of the Visual Simulation Class Library
All objects in a VSE model have a name. The object name is a string that can be set by the modeler using the **setName:** message and queried using the **name** message. The name is used only as an aid to help the modeler identify the object. For example, it’s displayed in the Inspector Panel (described in section 3.1.4).

New objects (of any class) are created by sending the **newInstance** message to the appropriate class. This method returns an instance of the receiving class. The **initNewInstance** message will have been sent to the new instance by the runtime system.

The **destroy** message is sent to an object when the object is to be freed.

3.2.2.2 VSLlist

Instances of the VSLlist class are used to store ordered lists of objects. Various methods such as **addObject:** and **removeObject:** allow modification of the list, and method such as **objectAt:** allow access to objects in the list. Lists are used by the Class Library (for example to store the objects in a component), and modelers can create VSLlist instances for their own purposes.

3.2.2.3 VSMModel

The VSMModel class provides global services to all objects in the model. It serves as a “front end” to the Simulation Engine (described in section 3.3 on page 48), letting other model objects access the current simulation time (by using the **clock** message) or creating new random number streams.

Each model has one and only one instance of VSMModel (or of a user-defined subclass). The class of this single instance can be specified in the Model Editor.
3.2.2.4 VSCOMPONENT

All components, both static and dynamic, are instances of classes that inherit from VSCOMPONENT. It is one of the key classes of the VSCL; together with the VSDYNAMICOBJECT class and the Simulation Engine, the VSCOMPONENT class contains the mechanism that drives the simulation.

Modelers create components in the Model Editor using the graphical decomposition tools. Components cannot be directly created by modeler logic at runtime, although (as described in section 3.2.1.5) they can be created as part of DOT instantiation.

VSCOMPONENT defines several notification messages that modelers can override in the subclasses of VSCOMPONENT. They are described in detail in Appendix A.

3.2.2.5 VSDYNAMICOBJECT

VSDYNAMICOBJECT is the other key class of the VSCL. All dynamic objects in the model are instances of VSDYNAMICOBJECT. Each dynamic object may optionally have a hierarchy of dynamic components.

Dynamic objects can move through the model’s static hierarchy, and can also enter into other (decomposed) dynamic objects. Dynamic objects are told to move by sending them one of the several variants of the moveTo: message. These are described in detail in Appendix A.

Dynamic objects can also be told to engage in an activity that takes a certain amount of simulation time. This is done by sending them the engageIn: message, as described in Appendix A.

Dynamic objects are represented during visualization by modeler-defined images. A dynamic object’s image representation can be changed at any time, and images may be
rotated to represent rotation of the dynamic object. Images are described briefly in section 3.2.2.6, and are described in detail in Appendix A.

3.2.2.6 VSImage

In VSE, components and dynamic objects can display pictures as part of the visualization of the simulation. These images are encoded in instances of the VSImage class.

In the Model Editor, modelers can associate a list of named images with each component class and dynamic object class he/she defines. Images are inherited, just as attributes are. At runtime modeler VSL logic can send messages to dynamic objects and components, telling them to set their images to the defined image with the given name. Alternatively, model logic can ask an object for its image. It will receive an object reference to an instance of VSImage. It can then use the received image as a parameter to a useImage: message sent to another component or dynamic object.

3.2.2.7 VSFile

Instances of VSFile are used to write data (usually statistics) to a file. The modeler instantiates a new VSFile object, and sends it the message openForWriting:. Model logic can then write values to the file by using one of the write... messages (described in more detail in Appendix A). When all the data has been written, the close message closes the file.

Currently, only file output is supported. In the future, the modeler should be able to use VSFile objects to read information from an input file.

3.2.2.8 VSItem and Its Subclasses

VSItem and its subclasses are defined so that users can communicate with the graphical item objects in component layouts. There is normally little need to do this, since the role of the graphical items in the simulation is solely for visualization. Each image in a com
nent layout has a name unique within that component, and one of the VSCComponent messages (`itemNamed:`) can be used to obtain an object reference to the named item.

Certain item attributes, such as line width and color, can be set at runtime to convey state information during visualization. Also, VSTextItem instances (which display text to the user) contain methods to set the text shown. This can be very useful for model debugging purposes.

Although modelers can technically create subclasses of VSIItem, they cannot instantiate their own item objects. These classes are defined only so that modelers can send messages to the existing graphic items that are part of the model hierarchy. For example, at model execution time, a VSTextItem created in the Model Editor can be told to display an integer value by sending it the `displayInteger:` message. The class documentation for VSTextItem in Appendix A describes methods that can be sent to VSTextItem objects.

### 3.3 Simulation Engine

This section describes the Simulation Engine; this is the mechanism that advances the simulation clock and carries out the instructions given to simulation objects through system messages. For example, it handles dynamic object creation, movement, engagement in activities, and destruction. It also provides access to random variate generation functions, and handles VSL script runtime errors.

#### 3.3.1 Overview

At the heart of the Simulation Engine is an event list and an event loop. Events are ordered by event time (in simulation clock units). The Process Interaction framework is used to drive the simulation.
With visualization switched on, the Engine uses a fixed-increment time flow mechanism in order to provide smooth animation. An option enables the refresh of viewers after each event. With visualization off, a variable-increment time flow mechanism is used.

3.3.2 Random Variate Generation

The Simulation Engine supports an arbitrary number of random number streams, each with its own 32-bit integer seed. A new random number stream is allocated by sending the \texttt{newRndStreamWithSeed:} message to the global VSM\textit{Model} instance (as described in section 3.2.2.3 on page 45, VSM\textit{Model} is the modeler’s “front end” to Simulation Engine facilities). The initial seed is specified as the parameter. The seed can be changed at any time using the \texttt{setSeed:ofRndStream:} method.

The Engine uses the multiplicative linear congruential random number generator, 
\[ Z_i = 7^5Z_{i-1} \pmod{2^{31} - 1} \]. Several random variate generation functions are provided, each of which takes a stream number (identifying the random number stream to use) as the first parameter. The random variate generation functions are listed in Table 3.1 on page 50. They are described in more detail in Appendix A.

As a convenience for those models that only use one random number stream, stream zero is created by default at the start of the model execution.

3.3.3 Error Handling

The Simulation Engine also intercepts and handles runtime errors caused by a modeler’s incorrect VSL logic. For example, if a negative duration is specified for a dynamic object movement operation, the Simulation Engine displays the runtime error panel, shown in Figure 3.9 on page 51. By clicking the button labeled “Show Trace”, the user can open a panel showing a trace of the VSL methods that led up to the runtime error. This panel
<table>
<thead>
<tr>
<th>Function</th>
<th>Random Variate Generator Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>Bounded continuous Beta distribution</td>
</tr>
<tr>
<td>binomial</td>
<td>Discrete Binomial distribution</td>
</tr>
<tr>
<td>discreteUniform</td>
<td>Bounded discrete Uniform distribution</td>
</tr>
<tr>
<td>exponential</td>
<td>Continuous Exponential distribution</td>
</tr>
<tr>
<td>extremeValueA</td>
<td>Unbounded continuous Extreme Value Type A distribution</td>
</tr>
<tr>
<td>extremeValueB</td>
<td>Unbounded continuous Extreme Value Type B distribution</td>
</tr>
<tr>
<td>gamma</td>
<td>Non-negative continuous Gamma distribution</td>
</tr>
<tr>
<td>geometric</td>
<td>Discrete Geometric distribution</td>
</tr>
<tr>
<td>inverseGaussian</td>
<td>Non-negative continuous Inverse Gaussian distribution</td>
</tr>
<tr>
<td>invertedWeibull</td>
<td>Non-negative continuous Inverted Weibull distribution</td>
</tr>
<tr>
<td>johnsonSB</td>
<td>Bounded continuous Johnson $S_B$ distribution</td>
</tr>
<tr>
<td>johnsonSU</td>
<td>Unbounded continuous Johnson $S_U$ distribution</td>
</tr>
<tr>
<td>laplace</td>
<td>Unbounded continuous Laplace distribution</td>
</tr>
<tr>
<td>logistic</td>
<td>Unbounded continuous Logistic distribution</td>
</tr>
<tr>
<td>logLaplace</td>
<td>Non-negative continuous Log-Laplace distribution</td>
</tr>
<tr>
<td>logLogistic</td>
<td>Non-negative continuous Log-Logistic distribution</td>
</tr>
<tr>
<td>lognormal</td>
<td>Non-negative continuous Lognormal distribution</td>
</tr>
<tr>
<td>negBinomial</td>
<td>Discrete Negative Binomial distribution</td>
</tr>
<tr>
<td>normal</td>
<td>Unbounded continuous Normal distribution</td>
</tr>
<tr>
<td>pareto</td>
<td>Non-negative continuous Pareto distribution</td>
</tr>
<tr>
<td>pearson5</td>
<td>Non-negative continuous Pearson Type V distribution</td>
</tr>
<tr>
<td>pearson6</td>
<td>Non-negative continuous Pearson Type VI distribution</td>
</tr>
<tr>
<td>poisson</td>
<td>Discrete Poisson distribution</td>
</tr>
<tr>
<td>randomWalk</td>
<td>Non-negative continuous Random Walk distribution</td>
</tr>
<tr>
<td>triangular</td>
<td>Bounded continuous Triangular distribution</td>
</tr>
<tr>
<td>uniform</td>
<td>Bounded continuous Uniform distribution</td>
</tr>
<tr>
<td>weibull</td>
<td>Non-negative continuous Weibull distribution</td>
</tr>
</tbody>
</table>
Figure 3.9: The visual simulator’s runtime error panel (with trace panel activated)
shows a history of the methods invoked immediately before the error occurred. System-provided methods (those implemented in the VSCL) are shown in gray, and modeler-specified methods in black. Clicking on a method listed in black invoked the Model Editor and lets the user edit the VSL script of the method that caused the runtime error.
CHAPTER 4:  EXAMPLE: COMPUTER SYSTEM MODEL

This chapter describes a sample VSE model, “Computer System.” This is a simple model that illustrates component decomposition, dynamic object motion, and statistics collection.

4.1 Model Definition

The example model represents a simple computer system with three remote users submitting batch jobs to one central computing center. Each user uses a terminal to submit the batch jobs.

This model has only a static hierarchy; the dynamic objects (the batch jobs) are not decomposed into dynamic hierarchies. In a more complex model, the dynamic objects might be decomposed into tree structures of their own.

4.1.1 Graphical Decomposition

First, the model’s static hierarchy is defined in the Model Editor. This is done using the decomposition tools in the Editor’s tool palette. Using simple click-and-drag mouse operations, the modeler draws shapes representing the components. As components are created, each is given a unique name that will be used to refer to it from the VSL specification.

The Computer System model’s static hierarchy is shown in Figure 4.1. The top level contains the three terminal users and the computing center. These are named “Terminal 1 User,” “Terminal 2 User,” “Terminal 3 User,” and “Computing Center,” respectively. Figure 4.2 shows the layout of the top level component in the Model Editor.
Figure 4.1: Static hierarchy of the Computer System model
Figure 4.2: Layout of top level component in the model editor
The “Computing Center” component is further decomposed into the “Computer” and the “Waiting Line.” Figure 4.3 shows the layout of the computing center component in the Model Editor. Also shown (above the layout) is the Model Editor’s component browser after all of the components have been defined.

Batch jobs move from the terminals to the computing center, and once in the computing center they move through the waiting line to the computer. In this model, movement of information through the actual wires is not modelled. The information is simply viewed as somehow moving from one component to another, taking a certain amount of time. Therefore, no paths are used in this model. In a model of a physical system such as a traffic intersection, paths are used to define the exact motion of dynamic objects. Paths affect only visualization; whether or not there is a path connecting the source and the destination does not affect the simulation results. If there is no path, the dynamic objects are visualized as moving in a straight line from the center of the source component to the center of the destination.

4.1.2 Class Definition

After defining the components, their behavior must be defined. This is done by identifying and defining classes for the components. In this case, the terminal users all exhibit the same behavior, so a TerminalUser class is defined. Similarly, a ComputingCenter class is defined for the “Computing Center” component. In addition, a class is defined for controlling the computer job movement at the top level of the static hierarchy. This class is called TopLevel.

In the “Computing Center” component, there is a waiting line and a computer. Each exhibit distinct behavior, so the WaitingLine and Computer classes are created as well.
Figure 4.3: Layout of "Computing Center" component
Classes are created by selecting the superclass (in this case VSCOMPONENT) in the Model Editor's class hierarchy browser, and choosing "New..." from the "Class" menu. This action presents the modeler with a panel asking for the name of the new class. Because the VSCOMPONENT class was selected when the "New..." command was activated, the new class will be a subclass of VSCOMPONENT.

In this case, all of the model-specific classes inherit from VSCOMPONENT. In a complex model, the model-specific classes may in turn have subclasses. For example, if there were small differences in the behavior of the terminal users, the modeler may have created classes such as ExpertTerminalUser and NoviceTerminalUser under the TerminalUser class.

Figure 4.4 shows the class hierarchy browser after the five component classes TerminalUser, ComputingCenter, TopLevel, WaitingLine, and Computer have been defined. When a class is selected in the hierarchy browser, the class editor window below it shows the class definition. As the figure illustrates, modelers can enter documentation describing the class. This should be done during class definition.

When the component classes have been defined, each defined component must be associated with one of the classes. This is done by selecting each component in turn, and choosing "Set class..." from the "Component" menu. This brings up the panel shown in Figure 4.5, where the modeler selects the subclass of VSCOMPONENT to associate with the current component. When the model is executed, each component will exhibit the behavior of the associated class.

With the model's static hierarchy defined, the dynamic aspect of the model is defined. This is done by creating Dynamic Object Templates (DOTs) for the different types of dynamic objects that will be used in the model. In the case of the Computer System
Figure 4.4: Class browser showing the model-specific component classes
Figure 4.5: Choosing the class for a component
model, there is only one type of dynamic object: computer jobs. Therefore, a dynamic object template called ComputerJob is created. Computer jobs aren't decomposed in this sample model, so no graphical decomposition is performed. Figure 4.6 shows the DOT browser after creating the ComputerJob dynamic object template.

Just as for the static model hierarchy, the behavior of the dynamic model parts (in this case the ComputerJob DOT) needs to be defined. This is done by creating a class named ComputerJob as a subclass of the VSDynamicObject class. The name spaces of classes and DOTs are distinct, so when desired, the same name can be used for both. After creating the ComputerJob class, the ComputerJob DOT is associated with the ComputerJob class in the same manner as for components. At model run time, instances of ComputerJob will then exhibit the behavior that will be specified for the ComputerJob class.

4.2 Model Specification

With the definition complete, the VSL script logic of each class is specified. The class editor window of the Model Editor is brought forth. This window shows the definition and specification of the class selected in the class hierarchy browser above. At the top of the editor window, modelers select the aspect of the class they want to edit: constants, enumerations, instance variables, instance methods, etc. As an example, the editor window for the dynObjEnteredModel: method of the TerminalUser class is shown in Figure 4.7.

The VSL constants, attributes, and methods attached to each class of the Computer System sample model are now briefly described.
Figure 4.6: Dynamic object template browser after creating the ComputerJob class.
Figure 4.7: Code editor for the `dynObjEnteredModel` method of the `TerminalUser` class

```plaintext
returns
nothing

parameters
Computer.Job ref job;

declarations
none

logic
tell job to moveToExt;
create Computer.Job at "Entrance" of self at time
   ((VSEModel clock) + exponential(0, 0.0, MEAN_THINKTIME));
```
4.2.1 TerminalUser

The TerminalUser class specifies the behavior of the terminal users. Each terminal user submits batch jobs with a time interval sampled from an exponential distribution with a constant mean time. This constant is defined in the TerminalUser class’ Constants section:

```cpp
const real MEAN_THINKTIME = 10.0;  // Mean time users spend thinking (in seconds).
```

The double slashes (//) indicate that the rest of the line is a comment. Comments can be included in any section of VSL logic.

The constant is specified in simulation time units. All time values in VSE are specified in this generic time unit. For each model, the modeler chooses which time unit in the system being modeled should correspond to the VSE simulation time unit. In the ComputerSystem model, simulation time units correspond to seconds in the real system.

TerminalUser instances have no attributes, because there are no variable characteristics of terminal users in this model. The mean of the exponentially distributed job submission time is a constant, as declared above.

4.2.1.1 The startUp method

The TerminalUser class implements the startUp method, which is the method that responds to startUp messages received by TerminalUser instances. The startUp message is a system message sent to each model component at the start of each simulation run. It is described in detail in the VComponent class documentation in Appendix A. The TerminalUser implementation of the startUp method is:

```cpp
method startUp
logic
create ComputerJob at "Entrance" of self
at time exponential(0, 0.0, MEAN_THINKTIME);
```
This method creates the first computer job generated by this terminal. In VSE, the time of
creation of a dynamic object and the time at which it enters into the model are not neces-
sarily the same. VSL’s create statement allows modelers to specify the entry time of the
dynamic object. This time may be the same as the current value of the simulation clock or
it may specify a later time. When the create statement is executed, a dynamic object is
created from the Dynamic Object Template named “ComputerJob”, and is scheduled to
enter the model at the specified time. In this case, the entry time is sampled from an expon-
ential distribution with the mean time declared as a constant above. The dynamic object
will enter at the doorway named “Entrance” of “self” (i.e. the current component). The
keyword self is special in VSL. It is a reference to the object whose method is being exe-
cuted. In this case, self refers to whichever instance of the TerminalUser class received the
startUp message. Since each component in the model receives the startUp message when
model execution starts, and since there are three instances of TerminalUser in the model,
the logic shown above will be invoked three times. In each case, self will be a reference to
the TerminalUser instance that received the startUp message.

4.2.1.2 The dynObjEnteredModel: method

The dynObjEnteredModel: notification message is sent whenever a dynamic
object enters into the model. It is sent to the component into which the dynamic object
entered. It takes a single parameter: a reference to the dynamic object. The documentation
for the VSComponent class (in Appendix A) describes the **dynObjEnteredModel**: in detail. The TerminalUser class’ implementation is:

```plaintext
method dynObjEnteredModel:
  parameters
    ComputerJob ref  job;
  logic
    tell job to moveToExit;
    create ComputerJob at "Entrance" of self
      at time ([VSEModel clock] + exponential(0, 0.0, MEAN_THINKTIME));
```

The parameter declaration indicates that the single parameter called “job” is a reference to an instance of the ComputerJob class. In this model, it is known before model execution that only computer jobs will enter into TerminalUser components. Declaring the parameter to be a reference to an object of a particular class lets the VSE assert this fact each time the **dynObjEnteredModel**: method is invoked.

This implementation simply tells the dynamic object to move to the nearest exit point, which brings the dynamic object up to the parent component of the terminal. In this case, the parent level is the top level of the model. Since TerminalUser’s implementation of the **startUp** message specified that newly created computer jobs arrive at the terminal, they must be sent to the top level component in order to be able to move to the computing center. As described in section 3.2.1.4 on page 41, dynamic objects may only move up or down in the hierarchy; they may not move directly to a sibling component.

Because no movement time is specified in the **moveToExit** message, the movement is instantaneous.

After the computer job has been told to move up, the next compute: job to be submitted by this terminal user is generated. The entry time is specified as the current simulation clock plus a random variate from an exponential distribution. When that job enters the
model, \texttt{dynObjEnteredModel}: is again invoked. This cycle repeats throughout the execution of the model.

4.2.2 \textit{TopLevel}

The top level component in the Computer System model is an instance of the \texttt{TopLevel} class. When computer jobs are sent up to it from the terminals (see section 4.2.1.2), the top level passes them on to the computing center. Computer jobs that come up from the computing center are sent to the model exit.

The \texttt{TerminalUser} class defines one constant:

\begin{verbatim}
const real TRANSMISSIONTIME is 3.0;  // Job transmission time in seconds.
\end{verbatim}

This constant represents the time needed to transmit a job from a terminal to the computing center. This constant is used in \texttt{TopLevel}'s \texttt{dynObjArrived}: method.

4.2.2.1 The \texttt{dynObjArrived}: method

The \texttt{dynObjArrived}: method is sent to a component when a dynamic object has arrived at the component. This is described in detail in Appendix A. The \texttt{TopLevel} class' implementation of the \texttt{dynObjArrived}: method is:

\begin{verbatim}
method dynObjArrived:
  parameters
    ComputerJob ref job;
  logic
    if [previousLocation of job] is [componentNamed:"Computing Center"] then
      tell job to moveToExit:TakingTime:TRANSMISSIONTIME;
    else
      tell job to moveToComponent:[componentNamed:"Computing Center"]
        takingTime:TRANSMISSIONTIME;
\end{verbatim}

If the previous location of the job is the computing center then the job is moved to the top level exit. When the job arrives at the top level exit, it is automatically destroyed (as described in section 3.2.1.4 on page 41). If the previous location is not the computing cen-
ter, it must be one of the terminals, so the computer job is sent to the computer center. In either case, the movement takes a constant time (as defined by the TRANSMISSIONTIME constant). Any numerical VSL expression could have been used in place of the TRANSMISSIONTIME constant. For example, a random variate generation function could have been invoked.

In the current version of the VSE, the dynObjArrived: message is sent to a component whenever a dynamic object arrives. Modelers must examine the dynamic object and choose what to do with it. For components into which dynamic objects can enter from many sources, the implementation of the dynObjArrived: method can become complex. For this reason, a planned future improvement is to make the simulation engine send different messages based on some criteria chosen by the modeler. For example, modelers should be able to specify a different notification message for each entrance doorway of the component. Another benefit of this improvement would be that modelers could choose more meaningful names for the notification messages. For example, busPassengerArrivedFromAirportTerminal: may be more appropriate than dynObjArrived: in a model of an airport bus shuttle.

4.2.3 ComputingCenter

The ComputingCenter class specifies the behavior of the computing center component. It routes incoming computer jobs to the waiting queue, and routes jobs from the waiting queue to the computer (see Figure 4.1 on page 54). Jobs that leave the computer are sent to the top level component.

ComputingCenter defines one constant:

```
const real TRANSMISSIONTIME is 3.0;        // Job transmission time in seconds.
```
This constant represents the time needed to transmit a job between the waiting line and the computer. The same constant is declared in the TopLevel class; there is no name conflict because the scope of constants is limited to the class specification. The VSL keyword `public` can be used to indicate that a particular constant is visible to all class specifications in the model.

The `ComputingCenter` class defines several attributes:

```
// For calculating average number
real totalArea, previousTime;
integer previousNumber;

// For calculating average time
real totalWaitingTime;
integer totalNumberOfJobs;

// Object reference declarations
WaitingLine ref waitingLine;
Computer ref computer;
```

All but the last two of these are used for statistics collection. They object references are needed to allow the computing center to communicate with the waiting line and computer components.

4.2.3.1 The startUp method

`ComputingCenter`'s implementation of the `startUp` method is:

```
method startUp
logic
  set waitingLine to [componentNamed:"waiting line"];  
  set computer to [componentNamed:"computer"];  
```

This implementation uses the Class Library-defined message `componentNamed:` to retrieve object references for the waiting line and computer components. The object references are stored as attributes, and are used in the other `ComputingCenter` methods. The
documentation of the VSComponent class in Appendix A describes the `componentNamed:` method.

No receiver is specified for the componentNamed: message. If no receiver is specified, `self` (the object reference to the object for which the logic is being executed) is assumed.

In future versions of the VSE, the Model Editor will allow modelers to graphically specify the referents of object reference attributes. This will eliminate the need for looking up object references to other components by name in the `startUp` method. It will be a significant improvement, because the use of names of individual instances to determine class logic behavior is very detrimental to reusability.

4.2.3.2 The `dynObjArrived:` method

The `dynObjArrived:` method implementation is:

```plaintext
method dynObjArrived:
parameters
  ComputerJob ref job;
logic
  if [previousLocation of job] is waitingLine then
    tell job to moveToComponent:computer takingTime:TRANSMISSIONTIME;
  else if [previousLocation of job] is computer then
    tell job to moveToExitTakingTime:TRANSMISSIONTIME;
  else {
    if [VSEModel warmedUp] then {
      // Collect statistics
      tell job to recordArrivalTimeToCCenter;
      tell self to adjustTotalArea;
    }
    tell job to moveToComponent:waitingLine takingTime:TRANSMISSIONTIME;
  }
```

Based on the previous location, the computer job is routed to the appropriate component. If the computer job entered the computing center for the first time, statics information is collected. The `adjustTotalArea` method, described in section 4.2.3.4, adjusts the cumula-
The area-under-the-curve parameter used in the computation of the average number of jobs in the computing center.

The text enclosed in brackets in the VSL code are examples of the expression form of messaging. This form can be used for any message that returns a result.

4.2.3.3 The dynObjDeparted: method

The computing center implements the **dynObjDeparted**: method as:

```vsl
method dynObjDeparted:
    parameters
        ComputerJob ref job;
    logic
        if [VSEModel warmedUp] then {
            // Collect statistics
            add ([VSEModel clock] - [arrivalTimeToCenter of job]) to totalWaitingTime;
            add 1 to totalNumberOfJobs;
            tell self to adjustTotalArea;
        }
```

This method collects statistics if the model has been warmed up. The Visual Simulator lets modelers specify a warm-up length and a steady-state length (see section 3.1.5 on page 36). To ensure correct collection of statistics, models are normally allowed to “warm up” before statistics collection starts.

4.2.3.4 The adjustTotalArea method

The **adjustTotalArea** method of the ComputingCenter class is used by the **dynObjArrived** and **dynObjDeparted**: methods to update the area-under-the-curve part of an average-number-of-jobs computation:

```vsl
method adjustTotalArea
    logic
        add (previousNumber * ([VSEModel clock] - previousTime)) to totalArea;
        set previousNumber to [numberOfDyn_objs];
        set previousTime to [VSEModel clock];
```

The expression **[numberOfDyn_objs]** is a short form for **[numberOfDyn_objs of self]**.
4.2.3.5 The shutDown method

ComputingCenter’s implementation of the shutDown method is:

```
method shutDown
declarations
    real steadyStateTime;
    real avgNumberOfJobsInCCenter, avgWaitingTimeInCCenter;
    VSF ile ref statisticFile;
logic
    set avgWaitingTimeInCCenter to (totalWaitingTime / totalNumberOfJobs);
    set steadyStateTime to [VSEM odel clock] - [warmUpLengthInTimeUnits of VSEM odel];
    set avgNumberOfJobsInCCenter to (totalArea / steadyStateTime);
    set statisticFile to [newInstance of VSF ile];
tell statisticFile to openForWriting:"Avg waiting time in comp ctr";
    tell statisticFile to writeReal:avgWaitingTimeInCCenter;
    tell statisticFile to close;
    tell statisticFile to openForWriting:"Avg num jobs in comp ctr";
    tell statisticFile to writeReal:avgNumberOfJobsInCCenter;
    tell statisticFile to close;
    destroy statisticFile;
```

This method first computes the average waiting time of jobs in the computer center and the average number of jobs, and then writes these statistics to files. This method illustrates the creation of new instances by sending the newInstance message to the class. Also illustrated is the destruction of object using the destroy statement.

4.2.4 WaitingLine

The WaitingLine class is used for the waiting line component in the computing center. It declares these attributes:

```
// For calculating average number
real totalArea, previousTime;
integer previousNumber;

// For calculating average time
real totalWaitingTime;
integer totalNumberOfJobs;

// Object reference declaration
Computer ref computer;
```
The `computer` attribute is an object reference that is used to send messages to the computer component. The other attributes are used for statistics collection.

4.2.4.1 The startUp method

WaitingLine’s implementation of the `startUp` method obtains an object reference to the computer, and stores it in the `computer` attribute:

```plaintext
method startUp
logic
    set computer to [componentNamed: "computer" of [parent]]; 
```

4.2.4.2 The dynObjArrived: method

The WaitingLine class performs the following logic whenever a dynamic object arrives:

```plaintext
method dynObjArrived:
parameters
    ComputerJob ref job;
logic
    if [VSEMModel warmedUp] then {
        // Collect statistics
        tell job to recordArrivalTimeToWaitingLine;
        tell self to adjustTotalArea;
    }
    if [computer isAvailable] then {
        // Send the job to the computer
        tell computer that jobWillArrive;
        tell [firstObject of [dynObjList]] to moveToExit;
    }
```

If the model is warmed up, statistics are collected. Then, if the computer is idle, the job is sent along to the computer (via the parent component; i.e. the computing center — see Figure 4.1 on page 54). The computer component is also notified that a job is en route, so that it doesn’t promise service to another job before this one arrives at the computer component. The description of the `jobWillArrive` method of the Computer class (section 4.2.5.3 on page 77) contains more information about this.
4.2.4.3 The `jobExecutionEnded` method

The `jobExecutionEnded` message is sent by the computer component to notify the waiting line that it can accept another job. If there are any jobs in the waiting line, the first one in line is sent to the computer:

```plaintext
method jobExecutionEnded
logic
    if ([numberOfDynObjs] > 0) then {
        // Send the job to the computer
        tell computer that jobWillArrive;
        tell [firstObject of [dynObjList]] to moveToExit;
    }
```

4.2.4.4 The `dynObjDeparted` method

When computer jobs leave the waiting line, statistics are collected:

```plaintext
method dynObjDeparted:
parameters
    ComputerJob ref job;
logic
    if [VSEMModel warmedUp] then {
        // Collect statistics
        add ([VSEMModel clock] - [arrivalTimeToWaitingLine of job]) to totalWaitingTime;
        add 1 to totalNumberOfJobs;
        tell self to adjustTotalArea;
    }
```

4.2.4.5 The adjustTotalArea method

WaitingLine's `adjustTotalArea` method is identical to the one in ComputingCenter (listed in section 4.2.3.4):

```plaintext
method adjustTotalArea
logic
    add (previousNumber * ([VSEMModel clock] - previousTime)) to totalArea;
    set previousNumber to [numberOfDynObjs];
    set previousTime to [VSEMModel clock];
```
4.2.4.6 The `shutDown` method

The implementation of the `shutDown` method in `WaitingLine` is very similar to the implementation in `ComputingCenter`:

```plaintext
method shutDown

declarations
real steadyStateTime;
real avgNumberOfJobsInQueue, avgWaitingTimeInQueue;
VSFFile ref statisticFile;

logic
set avgWaitingTimeInQueue to (totalWaitingTime / totalNumberOfJobs);
set steadyStateTime to [VSEModel clock] - [warmUpLengthInTimeUnits of VSEModel];
set avgNumberOfJobsInQueue to (totalArea / steadyStateTime);

set statisticFile to [newInstance of VSFFile];
tell statisticFile to openForWriting:"Avg waiting time in queue";
tell statisticFile to writeReal:avgWaitingTimeInQueue;
tell statisticFile to close;
tell statisticFile to openForWriting:"Avg num jobs in queue";
tell statisticFile to writeReal:avgNumberOfJobsInQueue;
tell statisticFile to close;
destroy statisticFile;
```

Statistics are computed and written to the files using an instance of `VSFile`.

4.2.5 Computer

The computer facility in the computing center model is an instance of the `Computer` class.

It is the component that accepts computer jobs from the waiting line and processes them.

The details of the job processing are not modeled in this example; processing is considered a “black box” activity that takes a random amount of time. Processing of computer jobs is not interruptible in this model.

The amount of time needed to process any job is sampled from an exponential distribution with a mean of 6.0. This constant is declared as a named constant in the `Computer` class:

```plaintext
const real MEANSERVICETIME is 6.0; // Mean service time in seconds.
```
At any given time, the computer is in one of two states: it is either idle or engaged in execution of a job. This is reflected in the `ComputerActivity` enumeration declared in the `Computer` class:

```java
enumeration ComputerActivity can be IDLE, EXECUTION;
```

The attributes of the computer component are:

- `obj ref waitingLine;` // Reference to waiting line.
- `boolean busy;` // Is the computer processing a job?
- `boolean servicePromised;` // Has service been promised to a job?
- `real sumOfExecutionTimes;` // Total sum of execution times so far.

The `waitingLine` attribute is an object reference to the waiting line associated with this computer component. It is used in `Computer`'s implementation of the `dynObjDeparted:` method to send a message telling the waiting line that the computer has become idle.

The `busy` and `servicePromised` attributes are used to ensure that the computer processes only one job at a time. `Busy` is true whenever the computer is actually processing a job, and `servicePromised` is true whenever the computer has accepted a computer job for processing but the job has not arrived yet. This can occur because of transmission of a job from the waiting line to the computer is not instantaneous. The `ComputingCenter` class (described in section 4.2.3 on page 68) defines the constant `TRANSMISSION` time (local to that class) which specifies the time lag between a job’s departure from the waiting line and its arrival at the computer.

The attribute `sumOfExecutionTimes` is used for computing usage statistics of the computer.
4.2.5.1 The startUp method

Computer’s `startUp` method retrieves an object reference to the waiting line component:

```plaintext
method startUp
logic
set waitingLine to [componentNamed:"waiting line" of [parent]];
```

This object reference is then used when the computer needs to communicate with the waiting line.

4.2.5.2 The isAvailable method

This method returns a boolean that indicates whether the computer is available for processing a job. It is considered available only if it is idle and if it has not already promised service to another job.

```plaintext
method isAvailable
returns
  boolean
logic
  if (busy is false) and (servicePromised is false) then
    return true;
  else
    return false;
```

4.2.5.3 The jobWillArrive method

This method is invoked by the waiting line component when it dispatches a job to the computer for execution. The implementation is:

```plaintext
method jobWillArrive
logic
set servicePromised to true;
```

Setting the `servicePromised` attribute to `true` ensures that the `isAvailable` method (described in section 4.2.5.2 above) will return `false`, indicating that the computer isn’t available. This flag is cleared when a computer job arrives.
4.2.5.4 The `dynObj:Arrived` method

This method, invoked when a computer job enters the computer, marks the computer as busy. It then chooses a processing time at random and tells the job to engage in the EXECUTION activity (declared as an enumeration value in section 4.2.5):

```pseudocode
method dynObj:Arrived:
    parameters
        ComputerJob ref job;
    declarations
        real executionTime;
    logic
        set servicePromised to false;
        set busy to true;
        set executionTime to exponential(0, 0.0, MEANSERVICETIME);
        if [VSEMModel warnedUp] then
            add executionTime to sumOfExecutionTimes;
        tell job to engageIn:EXECUTION for:executionTime;
```

When the computer job has been engaged in EXECUTION for the specified amount of time, the `dynObj:stoppedEngagement`: message is sent to the computer. See the documentation of the VSDynamicObject class is Appendix A for more information about engaging in activities.

4.2.5.5 The `dynObj:stoppedEngagement` method

This method is invoked when the computer job completes processing:

```pseudocode
method dynObj:stoppedEngagement:
    parameters
        ComputerJob ref job;
        ComputerActivity ref activity;
    logic
        tell job to moveToExit;
```

It tells the job to leave the computer component. It will move up to the computer’s parent component, which is the computing center component (see Figure 4.1 on page 54).
4.2.5.6 The dynObjDeparted: method

The **dynObjDeparted**: method is implemented as:

```
method dynObjDeparted:
  parameters
    ComputerJob ref job;
  logic
    set busy to false;
    tell waitingLine that jobExecutionEnded;
```

It marks the computer as idle, and informs the waiting line component that it is ready to process another job.

4.2.5.7 The shutDown method

The **shutDown** method in the Computer class is:

```
method shutDown
  declarations
    real steadyStateTime;
    real utilization;
    VSFile ref statisticFile;
  logic
    set steadyStateTime to [VSEMModel clock] - [warmUpLengthInTimeUnits of VSEMModel];
    set utilization to (sumOfExecutionTimes / steadyStateTime);
    set statisticFile to [newInstance of VSFile];
    tell statisticFile to openForWriting:"CPU utilization";
    tell statisticFile to writeReal: utilization;
    tell statisticFile to close;
    destroy statisticFile;
```

Like the implementations in the ComputingCenter and WaitingLine classes, this implementation computes statistics and writes the result to a file. For the computer, usage (as a percentage of total time) is computed.

4.2.6 ComputerJob

All computer jobs in the model are instances of the ComputerJob class. It contains methods to record and retrieve statistics.
ComputerJob instances have two attributes, both of which are used for statistics collection:

```plaintext
real arrivalTimeToCenter;    // Time when this job arrived at computing center.
real arrivalTimeToWaitingLine;  // Time when this job arrived in waiting line.
```

They record the arrival time to the computing center and the arrival time to the waiting line, respectively. Because of non-zero transmission times, these times are not necessarily the same.

### 4.2.6.1 The startUp method

The VSL logic executed at creating of computer jobs is:

```plaintext
method startUp
logic
    tell self to useImageNamed:"Computer Job Image";
```

This logic sets the image that represents computer job during visualization to be the named image. Images used to represent dynamic objects are discussed briefly in section 3.2.2.6 on page 47, and described in detail in Appendix A. By default, dynamic objects are represented by an image of a shaded sphere. Choosing a different image has purely aesthetic ramifications and in no way affects the outcome of the simulation (unless, of course, the modeler writes VSL logic that examines the image of a dynamic object and alters its behavior based on it).

### 4.2.6.2 The recordArrivalTimeToCCenter method

This method records the current time as the time of arrival to the computer center:

```plaintext
method recordArrivalTimeToCCenter
logic
    set arrivalTimeToCCenter to [VSEModel clock];
```
It is invoked from the dynObjArrived: method of the ComputingCenter (see section 4.2.3.2 on page 70).

4.2.6.3 The recordArrivalTimeToWaitingLine method

This method records the current time as the time of arrival to the waiting line

```plaintext
method recordArrivalTimeToWaitingLine
  logic
  set arrivalTimeToWaitingLine to [VSEModel clock];
```

It is invoked from the dynObjArrived: method of the ComputerCenter (see section 4.2.4.2 on page 73).

4.2.6.4 The arrivalTimeToCCenter method

This method returns the arrival time recorded in the recordArrivalTimeToCCenter method:

```plaintext
method arrivalTimeToCCenter
  returns real
  logic
  return arrivalTimeToCCenter;
```

4.2.6.5 The arrivalTimeToWaitingLine method

This method returns the arrival time recorded in the recordArrivalTimeToWaitingLine method:

```plaintext
method arrivalTimeToWaitingLine
  returns real
  logic
  return arrivalTimeToWaitingLine;
```
CHAPTER 5: EVALUATION OF THE VISUAL SIMULATOR

This chapter evaluates the Visual Simulator with respect to the four objectives set forth in Chapter 1.

Objective 1: The Simulator must fully support the object-oriented approach taken by the Visual Simulation Environment.

As described in Chapter 2, the VSE takes an object-oriented approach to simulation and modeling. Simulation-specific operations such as dynamic object movement and activity engagement have been intentionally left out of the Visual Simulation Language. This functionality is instead provided by a class library. The object-oriented approach makes VSL simpler and easier to learn, and it gives modelers more flexibility because they can override system-defined behavior. Another important benefit is associativity; modelers can associate real-world objects with simulation objects in their model. This makes models more closely correspond to the system being modeled than they would using another paradigm.

A key element in the Simulator’s support of the object-oriented approach is the Visual Simulation Class Library (VSCL). The VSCL contains both general purpose and simulation-specific classes; e.g. VSList and VSDynamicObject, respectively. Modelers can subclass the provided runtime classes, and through the inheritance mechanism they can replace or add to the standard behavior of any system-defined method.

The VSCL was briefly described in Chapter 3, and a detailed reference appears in Appendix A.
Objective 2: The Simulator must provide full support for dynamic object decomposition.

In the VSE, a model consists of a static and a dynamic part. The static part of a model is the hierarchy of static components, and the dynamic part consists of the set of dynamic objects in the model at any given time. The dynamic objects can move not only throughout the components of the static hierarchy, but also into other dynamic objects. In other words, dynamic objects can contain other dynamic objects. This is especially useful for modelling, for example, a bus that can pick up passengers. In order to take advantage of this, modelers must decompose their dynamic objects into hierarchies of components. This is done in the Model Editor.

In order to support the concept that dynamic objects can contain other dynamic objects, the Simulator’s logic for dynamic object motion is sufficiently flexible to allow movement among all sources and all destinations; between static components, between dynamic components, and between a static and a dynamic component. The VSCL contains methods specifically to support decomposed dynamic objects; for example, the **topLevelComponent**: method of VSDynamicObject returns a reference to the root component of the dynamic object’s component hierarchy. Of course, use of this method is only appropriate if the dynamic object receiving the message is hierarchically decomposed.

Simulator support for dynamic object decomposition is not limited to a few methods in the VSCL; it is integral to all Simulator activities. For example, when the Simulation engine destroys a dynamic object (because it departed from the model, for example) it also ensures that all dynamic objects contained in the one being destroyed are also destroyed.
Objective 3: The Simulator must let modelers see the behavior of the model graphically.

The benefits of visual simulation are clearly recognized [Bell and O’Keefe 1994]. The VSE strives to make visual simulation automatic; i.e. modelers should not have to write special logic (or indeed do anything else special) in order to see the execution of their model graphically. Visualization should be a by-product of model execution.

The Visual Simulator provides such automatic visualization during simulation. Because modelers define VSE models graphically, the information required for visualization is available for all models. Modelers have the option of customizing the visualization by associating images with components and dynamic objects, setting the color and line width of graphics items, and setting the text of text fields. They do this by writing VSL logic that uses the object-oriented features of VSE to send messages to the appropriate objects. For instance, the useImage: message is sent to a dynamic object or component to make it display an image.

Objective 4: The Simulator must let modelers easily navigate the model during execution of the simulation model.

A VSE model consists of a hierarchy of static components and a set of dynamic objects, each of which might have a component hierarchy of its own. For large models, the number of components and dynamic objects active simultaneously can make it difficult for modelers to get an overview of the state of the model.

The Visual Simulator alleviates this problem by allowing users to open an arbitrary number of viewer windows, each of which displays the visualization of one model component. Users can resize and arrange these viewers on the screen, and they can easily switch
focus of the viewer (change what is being viewed). This can be done through the compo-
nent browser, which shows the model components and lets users double-click on compo-
nent names to open viewers. It can also be done by double-clicking on items shown in any
viewer. For example, double-clicking on an exit changes the viewer to show the immediate
parent of the currently viewed component. This is convenient for “following” a dynamic
object up through an exit. Similarly, model components and decomposed dynamic object
can be double-clicked to “descend” into them; i.e. to display the chosen component.

This direct manipulation approach lets users easily select the portion of the model
they wish to visualize.
CHAPTER 6: CONCLUSIONS AND FUTURE RESEARCH

In this chapter, the research is summarized, major contributions identified, and future research on this topic is suggested.

6.1 Summary

In this thesis, an Object-Oriented Simulator for the Visual Simulation Environment is presented. Chapter 1 outlines the objectives of the research, while Chapter 2 gives background information and describes work related to the Simulator tool. Chapter 3 provides a functional description of the Visual Simulator’s three main components: the user interface, the Visual Simulation Class Library, and the Simulation Engine. Chapter 4 explains the details of a sample model, while Chapter 5 evaluates the Visual Simulator based on the objectives in Chapter 1.

The Visual Simulation Class Library Reference Manual (in Appendix A) provides detailed information about all classes and methods of the Visual Simulation Class Library.

6.2 Major Contributions

While the primary purpose of the research is to develop a functional Visual Simulation tool for the VSE, it also results in several other contributions.

Among the most significant of these is the design of the decomposable dynamic object feature. According to Barker [Barker 1995], the Visual Simulation Environment is the only simulation software package of eight simulation packages studied that provides hierarchical decomposition of dynamic objects. This feature of the VSE accounts for a siz-
able part of the Simulator development effort, both in the design and implementation phases. In addition to the design of the feature implementation, the manner in which the feature is presented to the user has been carefully designed.

Several issues arising from the decomposable dynamic object feature are addressed. For example, when a decomposed dynamic object is destroyed, the dynamic objects in it must also be destroyed. This must occur in a recursive manner. Much effort has been spent to make sure that model events such as dynamic object destruction cannot put the system in an inconsistent state.

Another contribution of the Visual Simulator is the design of the object-oriented Visual Simulation Class Library, the component that provides simulation features (such as dynamic object motion) to modelers. Because these simulation features are provided as object methods instead of VSL statements, the VSL language is kept compact and the modeler is given more flexibility. As described in Chapter 3, modelers can take advantage of the inheritance structure of the Runtime Class Library by overriding any system-defined methods they wish to customize.

A third contribution is the design of the navigation and object inspection facilities of the Visual Simulator. A direct manipulation approach is used, whereby users can point with the mouse and doubleclick to ascend or descend levels of the model hierarchy.

### 6.3 Future Research

While the Visual Simulator provides many features to modelers, there are many improvements that could be made. One of the more significant of these is to improve the speed of the visualization. Currently, viewer contents are redrawn after each simulation event. It is
only necessary to update the portions of the display that have actually changed. This task requires a significant amount of planning and designing such that only the minimum information necessary for a correct display is redrawn. Drawing anything not strictly necessary is wasteful, while failing to redraw an area in which the information has changed results in an incorrect visualization.

Another significant improvement would be the provision of better object inspection and debugging facilities. Currently the Simulator provides a simple object inspector that allows users to examine instance variables of individual objects. This feature was described in Chapter 3. While this can be very useful during debugging of model logic, a full-featured debugger which lets modelers set breakpoints and single-step through VSL logic would be invaluable. A trace feature, that let modelers see a log of events that occurred during a simulation, would also be very useful.

Another useful feature would be the ability to save the state of the simulation and to resume execution at a later time. In this way, a lengthy simulation run could be suspended every morning and resumed every evening, thereby taking maximum advantage of processing power. Care would need to be taken such that no modifications that invalidate the simulation state occur while a simulation is suspended. For example, modification of the model would invalidate suspended simulation runs.

A related feature would be the ability to automatically schedule a simulation run to start at a given time of day. This is another way in which, for example, simulations could be run nightly and on weekends when computing resources would otherwise be idle.

The Visual Simulator was designed with future improvements such as these in mind. Thus, incorporation of these features is quite feasible. No doubt there are many other hitherto unthought-of improvements that can also be made.
BIBLIOGRAPHY


Box, C.W. (1984), “A Prototype of the Premodels Manager,” MDE Project Memorandum, Department of Computer Science, VPI&SU, Blacksburg, VA.

Cox, B.J. (1986), Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, MA.


NeXT Computer (1993), Object-Oriented Programming and the Objective-C Language, Addison-Wesley, Reading, MA.

NeXT Computer (1994a), User Interface Guidelines, Addison-Wesley, Reading, MA.


APPENDIX A: VSCL CLASS LIBRARY REFERENCE MANUAL

The Visual Simulation Class Library (VSCL) is a set of VSL classes that provide commonly used services to modelers. It contains both general-purpose classes, such as lists of object references, and simulation-specific classes, such as components and dynamic objects.

Among the classes in the VSCL is the VSOBJECT class. This class is at the root of the inheritance hierarchy; it has no superclass. VSOBJECT defines the fundamental operations that all objects in a VSE model can perform.

The VSCL forms the upper part of the class hierarchy in a VSE model. Modelers create custom classes by subclassing from VSCL classes.

The inheritance hierarchy of the VSCL is shown below. This appendix describes each class in detail.
VSComponent

Inherits from: VSOBJECT

Class Description

The VSComponent class is one of the key classes in the VSE environment (the other key class is VSDynamicObject). It is an abstract class that defines the behavior of components, which are the building blocks of models. A model’s basic structure is a hierarchy of components. In the same manner, dynamic objects can be decomposed into hierarchies of components. The model’s static hierarchy of components comprises the static structure of a model, while dynamic objects and their hierarchies comprise the dynamic structure. You will frequently create your own subclasses VSComponent to specify the behavior that is needed for your particular model.

As dynamic objects (DOs) move around in the model, notification messages are sent to the components that they enter and leave. You will often subclass VSComponent and override some or all of these notification messages, so that your components can take the appropriate action when an event occurs. For example, every time a dynamic object enters into a component, that component is sent the notification message dynObjArrived:. You can implement that method in your subclass of VSComponent to tell the dynamic object to move to another component. You can also use the notification method to collect statistics about arrival times, or you can use it to perform any other action you need to perform whenever a DO enters a component.

Since components are a permanent part of the structure they belong to, you should never send them the destroy message. Static components belong to the static model structure and are never destroyed, and dynamic components are destroyed when the dynamic object they belong to is destroyed. Also, you can’t graft components into a structure hierarchy at runtime, so you should never send the newInstance message to the VSComponent class. When a model is loaded into the simulator, the components that make up the static model structure are automatically created. Similarly, when a dynamic object is created from a
**Dynamic Object Template (DOT)**, the components objects that make up its dynamic structure are created automatically.

If you need your subclass of VSComponent to perform some script logic whenever a simulation is started or stopped, put that logic in the methods `startUp` and `shutDown`, respectively. Never override the `initNewInstance` method in your subclass of VSComponent, and never send that message to instances that inherit from VSComponent.

### Instance Variables

- **obj ref parent**
  An object reference to the component’s parent, or nil if this is the root component.

### Instance Methods

**Methods invoked during startup and shutdown**

- **startUp**
  - **startUp**

  This message is sent to “start up” VSComponent instances for model execution. Components that are part of the static model structure receive this message at the start of each replication. The messages start at the bottom of the static hierarchy and propagate up; when each component is started up, its descendants have received the `startUp` message but its ancestors have not.

  Components that belong to the hierarchy of a dynamic object receive the `startUp` message when the dynamic object they are a part of is created. Again, messages start at the bottom of the hierarchy and propagate up.

  Note that you never send the `startUp` message yourself; it is sent for you by the runtime system. However, you may want to override this method in your subclass of VSComponent. You can use it to initialize instance variables and to perform any other tasks you need to perform at start up.

  See also: **shutDown**
**shutDown**

**shutDown**

This message is sent to “shut down” VSComponent instances. Components that are part of the static model structure receive this message at the end of each replication. The messages start at the top of the static hierarchy and propagate down; when each component is shut down, its descendants are still in a “running” state (i.e. they have not yet received the shutDown message). The ancestors of the component have received the message. Note that this is opposite from the order in which the startUp message is propagated.

Components that belong to the hierarchy of a dynamic object receive the shutDown message when the dynamic object they are a part of is destroyed, but only if the dynamic object is destroyed as a normal part of simulation. Dynamic objects that are in the model at the time a replication ends do not receive the shutDown message. This allows you to collect statistics in the shutDown method. Again, messages start at the top of the hierarchy and propagate down.

Note that you never send the shutDown message yourself; it is sent for you by the runtime system. However, you may want to override this method in your subclass of VSComponent. You can use it to destroy any objects you created in startUp, and to perform any other tasks you need performed every time a simulation run ends or a dynamic object is destroyed.

See also: **startUp**

Methods for accessing the dynamic objects in the component

**dynObjList**

*(VList ref)***dynObjList**

Returns an object reference to a list that contains the object references of the dynamic objects that are currently in the component. You receive a reference to the actual list, not a copy of it. Do not modify the returned list in any way!

See also: **numberOfDynObjsInLayout, numberOfDynObjs**
**numberOfDynobjsInLayout**

(integer) **numberOfDynobjsInLayout**

This method returns the number of dynamic objects that are in the component. This isn’t a recursive count; i.e. it does not include dynamic objects that are in child components of the receiving component.

The value returned by this method is the same as you would get if you first sent the **dynObjList** message to the component, and then sent the **count** message to the returned VSLList instance.

See also:  **dynObjList, numberOfDynObjs**

**numberOfDynObjs**

(integer) **numberOfDynObjs**

This method returns the number of dynamic objects that are in the component. Unlike the count returned by **numberOfDynobjsInLayout**, this is a recursive count; i.e. it includes dynamic objects that are in all child components of the receiving component.

See also:  **dynObjList, numberOfDynObjsInLayout**

**dynObjAt:**

(VSDynamicObject ref) **dynObjAt**(integer) **index**

Returns an object reference to the object at slot **index** in the receiver’s list of dynamic objects. This is a shorthand for using the **dynObjList** method to get a list of dynamic objects, and then sending a **objectAt:** message to that list to get the object reference at slot **index**.

See also:  **dynObjList, objectAt: (VSList)**
Setting and changing the component’s image

image

(image ref)image

Returns an object reference to the image that is currently displayed for the receiving component. You cannot send any messages to the returned object reference, but you can pass it as the argument to the useImage: method.

See also:  image (VSDynamicObject), useImage:, imageName

useImage:

useImage:(image ref)newImage

Sets the currently displayed image of the receiving component to be newImage, which must be an object reference returned by the image method of either an instance of VSCComponent or an instance of VSDynamicObject.

If newImage is nil the component image is removed, just as if you had sent the removeImage message.

See also:  useImage: (VSDynamicObject), useImageNamed:, removeImage

imageName

(string)imageName

Returns a string containing the name of the image that is currently displayed for the receiving component.

See also:  useImageNamed:

useImageNamed:

useImageNamed:(string)imageName

Sets the currently displayed image of the receiving component to the image with the given name. For each subclass of VSCComponent, you can define (in the VSE Editor) a list of named images that can be used to represent the decompositions of
instances of that component class. A component’s image is displayed on the
decomposition item in the parent component.

The images in the image lists are inherited, and the inheritance mechanism works
the same way as method inheritance. That is, each class inherits the images of all
of its ancestor classes, and it can override any or all of the inherited images with its
own images.

For example, if you have a class called MyComponent which is a subclass of
VComponent, and if MyComponent has an image named “Image no. 1”, then
instances of any subclass of MyComponent can also use that image. If a subclass
MySecondComponent of MyComponent defines another image that is also called
“Image no. 1”, then that second image will override the first one for instances of
MySecondComponent and its subclasses.

If imageName is an empty string, the component image is removed, just as if you
had sent the removeImage message.

If the receiver doesn’t inherit or define an image with the given name, the message
“image not found” is displayed in the component.

See also: **useImage:, useImageNamed:** (VSDynamicObject), **removeImage**

```
removeImage
```

```
removeImage
```

Removes the current image representing the receiver in the parent’s layout. This is
the same as sending the **useImageNamed:** message with an image name of “” (the
empty string), or as sending a **useImage:** message with a nil parameter. The image
isn’t removed from the class’s image list; the receiving instance is simply told not
to display an image. At a later time, you can use the **useImage:** or **useImage-
Named:** method to make the component display an image again.

See also: **useImageNamed:, useImage:, removeImage** (VSDynamicObject)
**imageAlignment**

(integer)imageAlignment

Returns a constant that identifies the way the component displays its image. This constant is either IMG_ALIGN_CENTER or IMG_ALIGN_SCALE.

See also: setImageAlignment:, useImageNamed:

**setImageAlignment:**

setImageAlignment:(integer)alignmentType

Changes the way the component image is drawn in the component’s decomposition representation. If alignmentType is IMG_ALIGN_CENTER, the image is centered within the enclosing rectangle of the decomposition. If it is IMG_ALIGN_SCALE, the image is scaled to fit the decomposition rectangle. The default image alignment type is IMG_ALIGN_CENTER.

The image alignment doesn’t take effect until the next time an image is set.

See also: imageAlignment, useImageNamed:

**Accessing component hierarchy information**

**parentDynObj**

(VSDynamicObject ref)parentDynObj

For components that are part of a dynamic object’s hierarchy, this method returns an object reference to the dynamic object that the receiver is a part of. For components in the static model hierarchy, this method returns nil.

See also: siblings, children, parent

**parent**

(VSComponent obj)parent

Returns an object reference to the parent component object, or nil if this is the top level component of the structure it is in (either the static model structure or a
dynamic object structure).

See also: siblings, children, parentDynObj

siblings

(VSList ref)siblings

Returns an object reference to a VSList object that contains references to the sibling components of the receiving component. Sibling components are those components that have the same parent as the receiver. The returned list is a just a reference to the list of siblings, not a copy of it; you should under no circumstance modify or destroy the list.

See also: parent, children

children

(VSList ref)children

Returns an object reference to a VSList object that contains references to the child components of the receiving component. Child components are the direct descendant components of a component. The returned list is a just a reference to the list of children, not a copy of it; you should under no circumstance modify or destroy the list.

See also: parent, siblings

Methods for looking up items and child components by name

itemNamed:

(VSItem ref)itemNamed:(string)itemName

Looks up the item named itemName in the receiving component, and returns an object reference to it.

If the receiver doesn’t contain an item with the given name, this method returns nil.

If there are several items named itemName, any one of them is returned.

If the item you are looking up is an instance of VSDoorwayItem (i.e. if it is an
entrance or an exit), use either the `entranceNamed:` message or the `exitNamed:` messages instead.

See also: `entranceNamed:, exitNamed:`

**entranceNamed:**

```csharp
(VSDoorwayItem ref)entranceNamed:(string)doorwayName
```

Looks up the entrance named `doorwayName` in the receiving component, and returns an object reference to it.

If the receiver doesn’t contain an entrance with the given name, this method returns nil. If there are several entrances named `doorwayName`, any one of them is returned.

See also: `exitNamed:, entranceNamed: (VSDynamicObject)`

**exitNamed:**

```csharp
(VSDoorwayItem ref)exitNamed:(string)doorwayName
```

Looks up the exit named `doorwayName` in the receiving component, and returns an object reference to it.

If the receiver doesn’t contain an exit with the given name, this method returns nil. If there are several exit named `doorwayName`, any one of them is returned.

See also: `entranceNamed:, exitNamed: (VSDynamicObject)`

**componentNamed:**

```csharp
(VSComponent ref)componentNamed:(string)componentName
```

Looks up the child component named `componentName` of the receiver, and returns an object reference to it. If the receiver doesn’t have a child component with the given name, this method returns nil.

See also: `componentNamed: (VSDynamicObject)`
Notification methods

dynObjEnteredModel:

dynObjEnteredModel: (VSDynamicObject ref)_dynObj

When a dynamic object enters the model, this message is sent to the component at which the dynamic object entered. The dynObj parameter is an object reference to the dynamic object that entered. When this message is sent, dynObj’s current location is the component receiving this message, and both the previous and next location references are nil. The sublocation object reference of the dynamic object is the doorway that was specified in the dynamic object’s create statement.

You never invoke this method yourself, but you will often override it in your subclass of VSClone. In your implementation, you could tell dynObj to move somewhere, or you could let the enteredModelAtComponent: method of the dynamic object do that. If nobody tells dynObj where to go to next, it will simply stay put until it is told to move somewhere.

This message is sent to the component after the enteredModelAtComponent: message has been send to dynObj.

See also: dynObjDepartedFromModel: enteredModelAtComponent: (VSDynamicObject)


dynObjDepartedFromModel:

dynObjDepartedFromModel: (VSDynamicObject ref)_dynObj

When a dynamic object departs from the model, this message is sent to the component from which the dynamic object departed. The dynObj parameter is an object reference to the dynamic object that departed. When this message is sent, dynObj’s current location, next location, and sublocation are all nil, and the previous location is the component receiving this message.

You never invoke this method yourself, but you will sometimes override it in your subclass of VSClone. In your implementation you may wish to collect statistics, or you could let the departedFromModelAtComponent: method of the dynamic object do that.
This message is sent to the component after the **departedFromModelAtComponent** message has been send to *dynObj*.

See also: **dynObjEnteredModel**, **departedFromModelAtComponent**: (VSDynamicObject)

**dynObjArrived:**

**dynObjArrived:** (VSDynamicObject ref)*dynObj*

When a dynamic object arrives at a component, this message is sent to the component at which it arrives. The *dynObj* parameter is the dynamic object that just arrived. When this message is sent, *dynObj*’s previous location is the component from which it arrived (or nil if *dynObj* just entered the model), the current location is the component receiving this message, and the next location is nil. The sublocation object reference of the dynamic object is the doorway or decomposition through which *dynObj* arrived at the component receiving this message.

You never send this message yourself, but you will often override it in your subclass of VSCComponent. In your implementation, you can tell *dynObj* to move somewhere, or you can let the **arrivedAtComponent**: method of the dynamic object do that. If nobody tells *dynObj* where to go, it will simply stay where it entered the component until it is told to move somewhere.

The **dynObjArrived** message is sent to the component after the **dynObjDeparted** message has been sent to the component from which *dynObj* came, and also after the message **arrivedAtComponent**: has been send to *dynObj*.

Note that this message is sent to the component even if *dynObj* just entered the model; if that is the case, the **dynObjEnteredModel** message is sent before this message.

See also: **dynObjDeparted**, **arrivedAtComponent**: (VSDynamicObject), **dynObjEnteredModel**:
**dynObjDeparted:**

\[
\text{dynObjDeparted: (VSDynamicObject ref) dynObj}
\]

When a dynamic object departs from a component, this message is sent to the component from which it departed. The \textit{dynObj} parameter is the dynamic object that just departed. When this message is sent, \textit{dynObj}’s previous location is the component receiving this message, the current location is the component into which the object moved, and the next location is nil. The sublocation object reference of the dynamic object is the doorway or decomposition through which \textit{dynObj} arrived at the new component.

You never send this message yourself, but you will sometimes override it in your subclass of VComponent. In your implementation you may wish to collect statistics, or you could let the \texttt{departedFromComponent} method of the dynamic object do that.

The \texttt{dynObjDeparted} message is sent to the component after the \texttt{departedFromComponent} message has been send to \textit{dynObj}, but before the \texttt{dynObjArrived} message is sent to the component at which \textit{dynObj} arrives.

Note that this message is sent to the component even if \textit{dynObj} just departed from the model; if that is the case, the \texttt{dynObjDepartedFromModel} message is sent after this message.

See also: \texttt{dynObjArrived}, \texttt{departedFromComponent}: (VSDynamicObject),
\texttt{dynObjDepartedFromModel}:

**dynObj: startedEngagement:**

\[
\text{dynObj: (VSDynamicObject ref) dynObj}
\]

\[\text{startedEngagement: (integer) activityIdentifier}\]

This message is sent to a component when a dynamic object in that component starts engaging in an activity. See the documentation of the VSDynamicObject class for more information about engaging in activities.

See also: \texttt{dynObj: stoppedEngagement}, \texttt{engageIn:for}: (VSDynamicObject)
**dynObj:stoppedEngagement:**

```plaintext
dynObj:(VSDynamicObject ref)dynObj
stoppedEngagement:(integer)activityIdentifier
```

This message is sent to a component when a dynamic object in that component finishes engaging in an activity. See the description of the VSDynamicObject class for more information about engaging in activities.

See also:  **dynObj:startedEngagement:, engageIn:for:** (VSDynamicObject)
VSDoorwayItem

Inherits from: VSIItem : VSOobject

Class Description

All entrances and exits in the model are instances of VSDoorwayItem. You cannot create objects of the VSDoorwayItem class, but you can obtain object references to existing instances (by using VSComponent’s entranceNamed: and exitNamed: methods, for example). These object references can then be passed as parameters to other VSCL methods.

Instance Variables

None declared in this class

Methods

None declared in this class
VSDynamicObject

Inherits from: VSOBJECT

Class Description

The VSDynamicObject class is one of the key classes in the VSE environment, along with the VSCComponent class. Every dynamic object (DO) in the model is an instance of a direct or indirect subclass of VSDynamicObject. Dynamic objects are created, enter the model at some component in the hierarchy, move around between components (and possibly other dynamic objects), and finally depart from the model and get destroyed.

Each dynamic object has references to its current location, its previous location, and its next location. The current location is an object reference to the component the dynamic object is currently in. The previous location is the component that the DO was in immediately before it entered the current component; the previous location reference can be nil if the DO just entered the model. The next location is the component the DO is moving to (or nil if it is not moving). You can find out a dynamic object’s previous location, current location, and next location by sending it the **previousLocation**, **location**, and **nextLocation** messages, respectively.

A dynamic object also has a “sublocation”. This is a reference to the VSItem instance the dynamic object is ‘on’. For example, if a DO is traveling along a path, the sublocation is a reference to that path. Similarly, if the DO is positioned on top of a doorway, its sublocation is a reference to the doorway. The sublocation is always a reference to an item in the current location.

In the visual simulation, a dynamic object is represented by an image. By default, every dynamic object is represented by a generic image of a shaded sphere. You can use the **useImage** or **useImageNamed** methods to set the image used to represent a particular DO. The dynamic object can also be told to display the image rotated to a specific angle. Optionally, the dynamic object can automatically rotate its image such that the “up” direction of the image is kept aligned with the target to the part of a path the DO is currently on.
You make dynamic objects move by sending them **moveTo...** messages. Three types of movement are supported: movement into a child component, movement into a dynamic object located in the current component, and movement up to the parent component.

You tell a dynamic object to move to one of its child components by sending it a **moveToComponent** message. The dynamic object then looks for a path item that leads from the sublocation to the decomposition of the specified child component. If one or more such paths exist, one of them is be followed when moving the DO from the current sublocation to the child component. If no such path is found, the DO moves in a straight line from the center of the current sublocation item to the center of the decomposition item of the child component. When the DO reaches the decomposition item, it moves down into the child component and appears at an entrance in that component.

You can also move a dynamic object into another dynamic object. In this case, the destination DO must have a component hierarchy. Also, the destination DO must be located in the same component as the dynamic object that is being moved. The destination dynamic object is not allowed to move out of the component while dynamic objects are moving towards it. When the moving DO arrives at the destination DO, it enters into the top level component of the destination DO. You tell a dynamic object to move into another dynamic object by sending it a **moveToDynObj** message.

To move a dynamic object up to the parent component of the current component, send it a **moveToExit** message. The dynamic object then looks for a path from the current sublocation to the specified exit doorway. If it doesn’t find one, it moves in a straight line from the sublocation to the exit item. When the DO reaches the exit, it moves up to the parent component, appearing at the decomposition of the component the DO came from. If the DO is moving up from the top level component of a decomposed dynamic object, the DO that’s moving moves out of the dynamic object it was in, and appears in the same component the DO it was in is located at.

If the DO is moving up from the top level component of the static model hierarchy, the dynamic object automatically departs from the model and is destroyed.

Whenever a dynamic object enters or departs from the model, or when it enters or departs from a component, it is sent notification messages. See the message descriptions under the category **Notification methods** for more information.
When a dynamic object needs to carry out an action that takes time to complete, you tell it to “engage in an activity”. To do this, you send it an engageIn:takingTime: message, giving it an activity identifier you previously declared in the Enumerations section of your VSL script, and specify the time (in simulation clock time units) the activity will take to complete. In this way, you can simulate time-consuming activities in the system you’re modeling.

**Instance Variables**

None declared in this class

**Instance Methods**

Methods for asking about the dynamic object’s location

**previousLocation**

(VSComponent ref) previousLocation

Returns an object reference to the previous location of the receiver; i.e. the component the dynamic object was in before it moved to the current one. This method returns nil if the dynamic object just entered the model and hasn’t moved anywhere since then.

See also: location, nextLocation

**location**

(VSComponent ref) location

Returns an object reference to the current location of the receiver; i.e. the component the dynamic object is currently in. This method returns nil if the dynamic object isn’t in any component at the moment; this is the case before a dynamic object enters the model, for example.

See also: previousLocation, nextLocation
**nextLocation**

(VSComponent ref)nextLocation

Returns an object reference to the next location of the receiver; i.e. the component to which the dynamic object is currently moving. This method returns nil if the dynamic object doesn’t currently have a destination.

See also: previousLocation, location

**subLocation**

(VSItem ref)subLocation

Returns an object reference to the item in the current component that the dynamic object is positioned “on”. For example, when a dynamic object enters one of its child components, the sublocation is set to be the entrance doorway through which the dynamic object entered.

See also: location

**Accessing the dynamic object’s name**

**name**

(string)name

Returns a string containing the name of the dynamic object.

See also: setName:

**setName:**

setName:(string)newName

Sets the name of the dynamic object to newName. By default, new simulation objects have names of the form “SimObject #”, where ‘#’ is the number of the dynamic object; for example, the seventeenth dynamic object created during the execution of the simulation would by default be named “SimObject 17”.

See also: name
Getting information about the dynamic object

`topLevelComponent`

`(VSCOMPONENT ref)topLevelComponent`

Returns an object reference to the top level component of the receiving dynamic object, if it is decomposed. This method returns nil if the receiving DO isn’t decomposed.

`numberOfDynObjs`

`(INTEGER)numberOfDynObjs`

This method returns the total number of dynamic objects that are contained in the dynamic object. This count doesn’t include dynamic objects contained in other dynamic objects in the receiver. For example, consider the case where a dynamic object `Pilot` is contained in a dynamic object `Aircraft`, and dynamic object `AircraftCarrier` in turn contains `Aircraft`. In this case, the count returned by sending the `numberOfDynObjs` message to `AircraftCarrier` would include the dynamic object `Aircraft`, but not `Pilot`.

See also: `numberOfDynObjs (VSCOMPONENT)`

`isMoving`

`(BOOLEAN)isMoving`

Returns true if the dynamic object receiving this message is currently in motion, or false if it isn’t.

Setting and changing the dynamic object’s image

`image`

`(VSI IMAGE ref)image`

Returns an object reference to the image that is currently displayed for the receiving dynamic object. You cannot send any messages to the returned object refer-
ence, but you can pass it as the argument to the `useImage:` method.

See also: `image (VSCComponent), useImage:, imageName`

**useImage:**

`useImage:(VSIImage ref)newImage`

Sets the currently displayed image of the receiving dynamic object to be `newImage`, which must be an object reference returned by the image method of either an instance of VSCComponent or an instance of VSDynamicObject.

If `newImage` is nil, the dynamic object’s image is removed, just as if you had sent the `removeImage` message.

See also: `useImage: (VSCComponent), useImageNamed:, removeImage`

**imageName**

`(string)imageName`

Returns a string containing the name of the image that currently represents the dynamic object.

See also: `useImageNamed:`

**useImageNamed:**

`useImageNamed:(string)imageName`

Sets the image of the receiving dynamic object to the image with the given name.

For each descendant of VSDynamicObject, you can define (in the VSE Editor) a list of named images that can be used to represent instances of that dynamic object class.

The images in the image lists are inherited, and the inheritance mechanism works the same way as method inheritance. That is, each class inherits the images of all of its ancestor classes, and it can override any or all of the inherited images with its own images.

For example, if you have a class MyDynamicObject which is a subclass of VSDy-
namicObject, and if MyDynamicObject has an image named “Image no. 1”, then instances of any subclass of MyDynamicObject can also use that image. If a subclass MySecondDynamicObject of MyDynamicObject defines another image that is also called “Image no. 1”, then that second image will override the first one for instances of MySecondDynamicObject and its subclasses.

If imageName is an empty string, the dynamic object image is set to the default dynamic object image, just as if you had sent the removeImage message.

If the receiver doesn’t inherit or define an image with the given name, the message “image not found” is displayed in the component.

See also: useImage:, useImageNamed: (VComponent), removeImage

removeImage

removeImage

Removes the image currently representing the dynamic object, and makes the dynamic object display the default image for dynamic objects. This is the same as sending the useImageNamed: message with an image name of "" (the empty string), or as sending a useImage: message with a nil parameter. The image isn’t removed from the class’s image list; the receiving dynamic object is simply told to display the default dynamic object image. At a later time, you can use the useImage: or useImageNamed: method to make the dynamic object display a special image again.

See also: useImageNamed:, useImage:, removeImage (VComponent)

rotationAngle

(rotationAngle)

Returns the current rotation angle (in degrees) of the image. Positive values indicate clockwise rotation. 0° means that the image is not rotated.

See also: setRotationAngle:
setRotationAngle:

setRotationAngle:(real)newAngle

Rotates the dynamic object’s image to the given angle (in degrees). Positive angles indicate clockwise rotation. 0° means that the image is not rotated. Angles need not be limited to the range -180° – 180°; for example, –720°, 0°, and 1440° all indicate no rotation.

If autorotation is turned on, this statement has no effect.

See also: rotationAngle, setAutorotation:

autoRotated

(boolean)autoRotated

Returns true if the dynamic object automatically rotates its image as it travels along a path, false if it does not.

See also: setAutoRotated, setRotationAngle:

setAutorotated:

setAutorotated:(boolean)autorotationSetting

Turns automatic rotation on or off. If automatic rotation is on, the dynamic object will rotate its image as it travels along a path. The image will be rotated so that its upper edge is perpendicular to the direction of travel.

While automatic rotation can make visualization look very good, it uses a significant amount of memory and computing power. Expect animation to be slower when automatic rotation is on. For this reason, automatic rotation is off by default.

See also: autoRotated
Querying and changing a dynamic object's coordinates

\[ x \]
\[(\text{real})x\]

Returns the x coordinate of the center of the receiving dynamic object's frame rectangle.

See also:  \( y, \text{setX:Y}, x \) (VSGlobal)

\[ y \]
\[(\text{real})y\]

Returns the y coordinate of the center of the receiving dynamic object's frame rectangle.

See also:  \( x, \text{setX:Y}, y \) (VSGlobal)

\[ \text{setX:Y} \]
\[ \text{setX:(real)}x \ Y:(\text{real})y \]

Sets the coordinates of the center of the receiving dynamic object's frame rectangle to \( x, y \).

See also:  \( \text{translateX:Y}, \text{setX:Y} \) (VSGlobal)

\[ \text{translateX:Y} \]
\[ \text{translateX:(real)}dx \ Y:(\text{real})dy \]

Translates the coordinates of the center of the receiving dynamic object's frame rectangle by an offset of \( dx, dy \).

See also:  \( \text{setX:Y}, \text{translateX:Y} \) (VSGlobal)

\[ \text{positionItselfOnTopOf} \]
\[ \text{positionItselfOnTopOf:(VSGlobal ref)}anItem \]

Positions the receiving dynamic object such that the center of its frame rectangle
coincides with the center of the frame rectangle of the item referred to by \textit{anItem}. The item \textit{anItem} must be in the same component as the dynamic object that receives this message.

See also: \texttt{setX:Y; translateX:Y; setX:Y; (VSItem), x (VSItem), y (VSItem)}

**Telling the dynamic object to move**

\texttt{moveToComponent:throughEntrance:usingPathNamed:takingTime:}

\begin{verbatim}
moveToComponent:(VSCOMPONENT ref)component
throughEntrance:(VSDoorwayItem ref)entrance
usingPathNamed:(string)pathName takingTime:(real)duration
\end{verbatim}

This method tells the receiver to move to the child component specified by the object reference \textit{component}. If there is a path named \textit{pathName} from the current sublocation to \textit{component}'s decomposition item, it will be used. If there is no such path, a nondeterministically chosen path from the current sublocation to the destination will be used. If there’s no path at all from the current sublocation to the destination, the dynamic object will move in a straight line from the center of the current sublocation to the center of the decomposition.

The parameter \textit{entrance} is an object reference to an entrance doorway in component. This is the entrance through which the receiver will enter into \textit{component}.

The movement will take \textit{duration} time units to complete.

See also: \texttt{moveToComponent:throughEntrance:takingTime:}

\texttt{moveToComponent:throughEntrance:usingPathNamed:}

\begin{verbatim}
moveToComponent:(VSCOMPONENT ref)component
throughEntrance:(VSDoorwayItem ref)entrance
usingPathNamed:(string)pathName
\end{verbatim}

This method is like \texttt{moveToComponent:throughEntrance:usingPathNamed:-takingTime:}, except that the time taken to complete the move is zero (i.e. the
move is instantaneous).

See also: `moveToComponent:throughEntrance:usingPathNamed:takingTime`

`moveToComponent:(VSCOMPONENT ref)component throughEntrance:(VSDoorwayItem ref)entrance takingTime:(real)duration`

This method is like `moveToComponent:throughEntrance:usingPathNamed:takingTime`, except that the path used is a nondeterministically chosen path connecting the current sublocation to `component`'s decomposition item. If there is no such path, the dynamic object will move in a straight line from the center of the current sublocation to the center of `component`'s decomposition item.

See also: `moveToComponent:throughEntrance:usingPathNamed:takingTime`

`moveToComponent:throughEntrance:

`moveToComponent:(VSCOMPONENT ref)component throughEntrance:(VSDoorwayItem ref)entrance`

This method is like `moveToComponent:throughEntrance:takingTime`, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: `moveToComponent:throughEntrance:takingTime`

`moveToComponent:usingPathNamed:takingTime:

`moveToComponent:(VSCOMPONENT ref)component usingPathNamed:(string)pathName takingTime:(real)duration`

This method is like `moveToComponent:throughEntrance:usingPathNamed:takingTime`, except that the entrance doorway is nondeterministically chosen from the set of entrance doorways in `component`. This method is especially conve-
nient to use when component is known to contain only one entrance doorway.

See also: **moveToComponent:throughEntrance:usingPathNamed:takingTime:**

**moveToComponent:usingPathNamed:**

```swift
moveToComponent:(VSComponent ref)component
usingPathNamed:(string)pathName
```

This method is like **moveToComponent:usingPathNamed:takingTime:**, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: **moveToComponent:usingPathNamed:takingTime:**

**moveToComponent:takingTime:**

```swift
moveToComponent:(VSComponent ref)component
takingTime:(real)duration
```

This method is like **moveToComponent:usingPathNamed:takingTime:**, except that the path used is a nondeterministically chosen path connecting the current sublocation to component’s decomposition item. If there is no such path, the dynamic object will move in a straight line from the center of the current sublocation to the center of component’s decomposition item.

See also: **moveToComponent:usingPathNamed:takingTime:**

**moveToComponent:**

```swift
moveToComponent:(VSComponent ref)component
```

This method is like **moveToComponent:takingTime:**, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: **moveToComponent:takingTime:**
moveToDynObj:throughEntrance:takingTime:

moveToDynObj:(VSDynamicObject ref)dynObj
throughEntrance:(VSDoorwayItem ref)entrance
takingTime:(real)duration

This method tells the receiver to move to the dynamic object specified by the
object reference dynObj, which must be in the same component as the receiver.
Also, dynObj must be decomposed into at least one component. Once the dynamic
object receiving this message starts moving toward dynObj, it is an error for
dynObj try to start moving away.
The parameter entrance is an object reference to an entrance doorway in the top
level component of dynObj. This is the entrance through which the receiver will
enter into the top level component of dynObj.
The movement will take duration time units to complete.

See also: moveToDynObj:throughEntrance:, moveToDynObj:takingTime:,
moveToDynObj:

moveToDynObj:throughEntrance:

moveToDynObj:(VSDynamicObject ref)dynObj
throughEntrance:(VSDoorwayItem ref)entrance

This method is like moveToDynObj:throughEntrance:takingTime:, except that
the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: moveToDynObj:throughEntrance:takingTime:, moveToDynObj:-
takingTime:, moveToDynObj:

moveToDynObj:takingTime:

moveToDynObj:(VSDynamicObject ref)dynObj takingTime:(real)duration

This method is like moveToDynObj:throughEntrance:takingTime:, except that
the entrance doorway is nondeterministically chosen from the set of entrance door-
ways in the top level component of dynObj. This method is especially convenient
to use when dynObj is known to have only one entrance doorway in its top level
component.

See also: moveToDynObj:throughEntrance:takingTime:, moveToDynObj:, moveToDynObj:throughEntrance:

**moveToDynObj:**

moveToDynObj:(VSDynamicObject ref)dynObj

This method is like moveToDynObj:takingTime:, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: moveToDynObj:throughEntrance:takingTime:, moveToDynObj:-
takingTime:, moveToDynObj:throughEntrance:

**moveToExit:usingPathNamed:takingTime:**

moveToExit:(VSDoorwayItem ref)exit usingPathNamed:(string)pathName
takingTime:(real)duration

This method tells the receiver to move to the exit specified by the object reference exit. This exit must be an exit doorway in the dynamic object’s current location. The path named pathName is used to move to the exit. If there is no path with that name, a nondeterministically chosen path from the current sublocation to the specified exit is used. If there’s no path at all from the current sublocation to the exit, the dynamic object will move in a straight line from the center of the current sublocation to the center of the exit.

Moving to an exit means that the dynamic object will move out of the current component and into the parent component. If the receiver’s current location is the top level component of a decomposed dynamic object, the receiver will move out of the DO, and will appear in the same component the enclosing dynamic object is in.

If this message is sent to a DO that is in the top level component, the DO will automatically depart from the model when it reaches the exit doorway.

The movement will take duration time units to complete.

See also: moveToExit:usingPathNamed:, moveToExit:takingTime:
**moveToExit:usingPathNamed:**

`moveToExit:(VSDoorwayItem ref)exit usingPathNamed:(string)pathName`

This method is like `moveToExit:usingPathNamed:takingTime:`, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: `moveToExit:usingPathNamed:takingTime:`

**moveToExit:takingTime:**

`moveToExit:(VSDoorwayItem ref)exit takingTime:(real)duration`

This method is like `moveToExit:usingPathNamed:takingTime:`, except that the path used is a nondeterministically chosen path connecting the current sublocation to the exit. If there is no such path, the dynamic object will move in a straight line from the center of the current sublocation to the center of the exit.

See also: `moveToExit:usingPathNamed:takingTime:`

**moveToExit:**

`moveToExit:(VSDoorwayItem ref)exit`

This method is like `moveToExit:takingTime:`, except that the time taken to complete the move is zero (i.e. the move is instantaneous).

See also: `moveToExit:takingTime:`

**moveToExitUsingPathNamed:takingTime:**

`moveToExitUsingPathNamed:(string)exit takingTime:(real)duration`

This method is like `moveToExit:usingPathNamed:takingTime:`, except that the exit the receiving DO moves up through on its way up to the parent is chosen nondeterministically. The component the receiver is currently in must contain at least one exit.

See also: `moveToExit:usingPathNamed:takingTime:`
**moveToExitUsingPathNamed:**

```
moveToExitUsingPathNamed:(string)pathName
```

This method is like **moveToExitUsingPathNamed:takingTime:**, except that the time taken to complete the move is zero time units (i.e. the move is instantaneous).

See also: **moveToExitUsingPathNamed:takingTime:**

**moveToExitTakingTime:**

```
moveToExitTakingTime:(real)duration
```

This method is like **moveToExit:takingTime:**, except that the exit the receiving DO moves up through on its way up to the parent is chosen nondeterministically. The component the receiver is currently in must contain at least one exit.

See also: **moveToExit:takingTime:**

**moveToExit**

```
moveToExit
```

This method is like **moveToExitTakingTime:**, except that the time taken to complete the move is zero time units (i.e. the move is instantaneous).

See also: **moveToExitTakingTime:**

---

**Telling the dynamic object to engage in activities**

**engageIn:**

```
engageIn:(integer)activityIdentifier for:(real)duration
```

This method tells the receiving dynamic object to engage in the activity specified by *activityIdentifier* for *duration* time units. See the *Class Description* for more information about engaging in activities.

See also: **startedEngagement:, stoppedEngagement:**
Methods for finding top level items and components by name

**itemName:**

(VSItem ref) **itemName**:(string)itemName

Looks up the item named itemName in the top level component of the receiving dynamic object, and returns an object reference to it.

If the receiver’s top level component doesn’t contain an item with the given name, this method returns nil. If there are several items named itemName, any one of them is returned.

If the item you are looking up is an instance of VSDoorwayItem (i.e. if it is an entrance or an exit), use either the **entranceNamed:** message or the **exitNamed:** messages instead.

See also:  **entranceNamed:; exitNamed:; itemName:** (VSCOMPONENT)

**entranceNamed:**

(VSDoorwayItem ref) **entranceNamed:**:(string)doorwayName

Looks up the entrance named doorwayName in the top level component of the receiving dynamic object, and returns an object reference to it.

If the receiver’s top level component doesn’t contain an entrance with the given name, this method returns nil. If there are several entrances named doorwayName, any one of them is returned.

See also:  **exitNamed; entranceNamed:** (VSCOMPONENT)

**exitNamed:**

(VSDoorwayItem ref) **exitNamed:**:(string)doorwayName

Looks up the exit named doorwayName in the top level component of the receiving dynamic object, and returns an object reference to it.

If the receiver’s top level component doesn’t contain an exit with the given name, this method returns nil. If there are several exit named doorwayName, any one of
them is returned.

See also: entranceNamed:, exitNamed: (VSCComponent)

**componentNamed:**

```
(VSCComponent ref)componentNamed:(string)componentName
```

Looks up the child component named `componentName` of the receiver’s top level component, and returns an object reference to it. If the receiver’s top level component doesn’t have a child component with the given name, this method returns nil.

See also: `componentNamed:` (VSCComponent)

**Notification methods**

**enteredModelAtComponent:**

```
enteredModelAtComponent:(VSCComponent ref)theComponent
```

When a dynamic object enters the model into a component, this message is sent to the dynamic object. The `theComponent` parameter is an object reference to the component at which the DO entered the model. When this message is sent, the receiver’s current location is `theComponent`, and both the previous and next location references are nil. The sublocation object reference of the dynamic object is the doorway that was specified in the dynamic object’s `create` statement.

You never send this message yourself, but you will often override it in your sub-classes of VSDynamicObject. In your implementations, you could tell `self` to move somewhere, or you could let the `dynObjEnteredModel:` method of the component do that. If nobody tells the dynamic object where to go to `next`, it will simply stay put until it is told to move somewhere.

This message is sent to the dynamic object before the `dynObjEnteredModel:` message is sent to `theComponent`.

See also: `departedFromModelAtComponent:` `dynObjEnteredModel:

(VSCComponent)"
departedFromModelAtComponent:

   departedFromModelAtComponent: (VSCOMponent ref) theComponent

When a dynamic object departs from the model from a component, this message is
sent to the DO. The theComponent parameter is an object reference to the compo-
nent from which the receiver of the message departed. When this message is sent,
the dynamic object’s current location, next location, and sublocation are all nil, and
the previous location is the component from which the DO departed.

You never invoke this method yourself, but you will sometimes override it in your
subclasses of VSDynamicObject. In your implementations you may wish to collect
statistics, or you could let the dynObjDepartedFromModel: method of the com-
ponent do that.

This message is sent to the dynamic object before the dynObjDepartedFrom-
Model: message is sent to the Component.

See also: enteredModelAtComponent:, dynObjDepartedFromModel:

   (VSCOMponent)

arrivedAtComponent:

   arrivedAtComponent: (VSCOMponent ref) theComponent

After a dynamic object arrives at a component, it receives this message. The
parameter theComponent is the component at which the receiving DO just arrived.
When this message is sent, the DO’s previous location is the component from
which it arrived, the current location is the component it just arrived at, and the
next location is nil. The sublocation object reference of the DO is the doorway or
decomposition through which the dynamic object arrived.

You never send this message yourself, but you will often override it in your sub-
classes of VSDynamicObject. In your implementations, you can tell the DO to
move somewhere, or you can let the dynObjArrived: method of the component
do that. If nobody tells the DO where to go, it will simply stay where it entered the
component until it is told to move somewhere.

The arrivedAtComponent: message is sent to the component after the dynObj-
**Departed:** message has been sent to the component from which the DO came, but before the **dynObjArrived:** message has been sent to the **Component**.

Note that this message is sent to the dynamic object even if it just entered the model; if that is the case, the **enteredModelAtComponent:** message is sent before this message.

See also: **departedFromComponent:**; **enteredModelAtComponent:**; **dynObjArrived:** (VSCOMPONENT)

**departedFromComponent:**

**departedFromComponent:**(VSCOMPONENT ref)the **Component**

After a dynamic object departs from a component, it receives this message. The parameter **the Component** is the component from which the DO just departed. When this message is sent, the dynamic object’s previous location is the component from which it just departed, the current location is nil, and the next location is the component into which the object is moving. The sublocation is nil.

You never send this message yourself, but you will sometimes override it in your subclasses of VSDynamicObject. In your implementations you may wish to collect statistics, or you could let the **dynObjDeparted:** method of the component do that.

The **departedFromComponent:** message is sent to the dynamic object before the message **dynObjDeparted:** is sent to the **Component**, and before the **arrivedAtComponent:** message is sent to the dynamic object.

Note that this message is sent to the dynamic object even if it departed from the model; if that is the case, the **departedFromModelAtComponent:** message is sent after this message is sent.

See also: **arrivedAtComponent:**; **departedFromModelAtComponent:**; **dynObjDeparted:** (VSCOMPONENT)
**startedEngagement:**

`startedEngagement:(integer)activityIdentifier`

This message is sent to a dynamic object right after it starts engaging in an activity. The `activityIdentifier` parameter is the activity identifier specified in the `engageIn:-for:` message that started the activity engagement.

You never send `startedEngagement:` messages directly, but you will often override this method to collect statistics or perform other tasks.

See also: `engageIn:-for:, stoppedEngagement:, dynObj:startedEngagement:` (VSComponent)

**stoppedEngagement:**

`stoppedEngagement:(integer)activityIdentifier`

This message is sent to a dynamic object right after it stops engaging in an activity. The `activityIdentifier` parameter is the activity identifier specified in the `engageIn:-for:` message that started the activity engagement.

You never send `stoppedEngagement:` messages directly, but you will often override this method to collect statistics or perform other tasks.

See also: `engageIn:-for:, startedEngagement:, dynObj:stoppedEngagement:` (VSComponent)
VSEllipseItem

Inherits from:  VSIItem : VSOBJECT

Class Description

Each ellipse item created in the Model Editor is an instance of VSEllipseItem. As with any item, instances of VSEllipseItem should not be created directly by VSL scripts. The VSEllipseItem class is public because references to VSEllipseItem objects may be returned by VComponent’s itemNamed: method.

Instance Variables

None declared in this class

Methods

None declared in this class
VSFile

Inherits from: VSOBJECT

Class Description

Instances of the VSFile class are used to write statistics to output files. Currently, only the basic data types integer, real, and string are supported. Support for other basic data types is planned for future versions of the VSE.

A VSFile instance is in one of two states: open or closed. You can only write information to a file when it is in the open state, so the first message you send to a file will always be openForWriting:. Then you write information by sending one of the methods for writing data: writeReal:, writeInteger:, or writeString: methods. When you are done writing to the file, you close the file by sending it the close message. If you want to write more data to the file, you can send the openForWriting: message again. Each data item is written to the file on a separate line.

Support for file input is planned for future versions of the VSE.

You will often use instances of the VSFile class, though you will rarely need to subclass it.

Instance Variables

None declared in this class

Instance Methods

None declared in this class

openForWriting:

openForWriting:(string)fileName

Opens a statistics output file with the given name for writing. The name may not contain any slash ("/") characters. The file is created in the model’s file package, where it can later be accessed by the VSE Data Analyzer tool.

If the model already contains a file with the given name, new data is appended to
the end of it. At the beginning of the first run of each model execution, the contents of statistics files are discarded. So the output file will only contain data relevant to the last model execution.

See also:  close

**close**

**close**

Closes the file, saving all data that has been written to it. After sending this message, you need to send an **openForWriting:** message again before you can write any more data.

See also:  **openForWriting:**

**Writing basic data types to open files**

**writeReal:**

```plaintext
writeReal:(real)value
```

Write the given real number value to the file.

See also:  **writeInteger:**, **writeString:**

**writeInteger:**

```plaintext
writeInteger:(integer)value
```

Writes the given integer value to the file.

See also:  **writeReal:**, **writeString:**

**writeString:**

```plaintext
writeString:(string)value
```

Writes the given string to the file.

See also:  **writeInteger:**, **writeReal:**
VSIImage

Inherits from:  VSOBJECT

Class Description
Instances of the VSIImage class are images that you can use in the VSE. Currently, the only way in which they can be used is to be passed as arguments to VSCOMPONENT and VSDYNAMICOBJECT methods that take images as parameters (such as useImage:].
You shouldn’t make a subclass of VSIImage, nor should you create VSIImage instances. The class is public because some class library methods return references to instances of VSIImage.

Instance Variables
None declared in this class

Methods
None declared in this class
**VSItem**

**Inherits from:** VSOBJECT

**Class Description**

All graphic items that are shown in components during visual simulation are instances of the VSItem class (or of a subclass of VSItem). You should not create instances of the VSItem class, but during the execution of the simulation you can obtain object references to existing items. You can then change some of the attributes of these items. The VSItem attributes you change in this way only affect the way the items are drawn during visual simulation; the behavior of the simulation is not affected. You may want to query item attributes, and make decisions based on the values of those attributes. In that case, of course, simulation will be affected, whether visualization is turned on or off.

Even when visualization is turned off, items still update their attributes. The only difference between visualizing and not visualizing is that when visualization is off nothing is drawn on the screen. You can still depend on the values of item attributes.

**Instance Variables**

None declared in this class

**Instance Methods**

**Accessing the item’s name**

```csharp
name
```

*(string)*name

Returns a string containing the name of the item.

See also: **setName:**
setName:

setName: (string) newName

Sets the name of the item to newName.

See also: name

Querying and setting the coordinates of the item

setFrame:::

setName: (real) x : (real) y : (real) w : (real) h

Sets the receiving item’s frame (the rectangle enclosing the item) to have the specified x and y coordinates as the lower left corner, with the given width w and height h. The coordinates are in the coordinate system of the component in which the item is located.

Use of this method does not affect the logic of the simulation. It only affects the display of the item in the visual simulation.

See also: setX: Y:, translateX: Y:

x

( real ) x

Returns the x coordinate of the lower left corner of the receiving item’s frame rectangle.

See also: y, setX: Y:, x (VSDynamicObject)

y

( real ) y

Returns the y coordinate of the lower left corner of the receiving item’s frame rectangle.

See also: x, setX: Y:, y (VSDynamicObject)
setX:Y:

\[ \text{setX}:(\text{real})x \; \text{Y}:(\text{real})y \]

Sets the coordinates of the lower left corner of the receiving item’s frame rectangle to \( x, y \).

See also: \( \text{setFrame:::}, \text{translateX:Y:}, \text{setX:Y:} \) (VSDynamicObject)

translateX:Y:

\[ \text{translateX}:(\text{real})dx \; \text{Y}:(\text{real})dy \]

Translates the coordinates of the lower left corner of the receiver’s frame rectangle by an offset of \( dx, dy \).

See also: \( \text{setFrame:::}, \text{setX:Y:}, \text{translateX:Y:} \) (VSDynamicObject)

Changing an item’s appearance

useLineColor:::

\[ \text{useLineColor}:(\text{real})red \; (\text{real})green \; (\text{real})blue \]

Sets the color used to draw the item’s outline. The parameters specify the intensities of each of the three primary colors: red, green, and blue. Each intensity ranges from 0 (the darkest) to 1 (the brightest). For example:

```plaintext
tell item to useLineColor:0.0 :1.0 :0.0; // Bright green
tell item to useLineColor:0.0 :0.0 :0.5; // Medium blue
tell item to useLineColor:0.0 :0.0 :0.0; // Black
tell item to useLineColor:1.0 :1.0 :1.0; // White
tell item to useLineColor:0.7 :0.7 :0.7; // Light gray
```

The default line color of an item can be specified using the VSE Editor.

See also: \( \text{useFillColor:::} \)

useFillColor:::

\[ \text{useFillColor}:(\text{real})red \; (\text{real})green \; (\text{real})blue \]

Sets the color used to draw the item’s solid area. See the \( \text{useLineColor:::} \) method for information about the parameter values.
The default fill color of an item can be specified using the VSE Editor.

See also:  useLineColor:::

useLineWidth:
  useLineWidth:(real)width

Sets the width (in screen pixels) used to draw the item’s outline. The default line width of an item can be specified using the VSE Editor.
VSLList

Inherits from: VSOBJECT

Class Description

Instances of VSLList are used to store ordered lists of object references. Only the references are stored in the list; if you add the same object reference to two different lists, both lists will refer to the same actual object.

Some classes in the class library use instances of VSLList to return information to you. For example, you can ask instances of the VSComponent class for a list of all simulation objects in the component. What is returned to you is an object reference to an instance of VSLList that contains references to the dynamic objects in the component.

Objects in the list can be accessed either by object reference or by index. To access by object reference, simply pass the object reference to the appropriate method, such as `addObject:`, `removeObject:`, etc. When you access objects by index in the list, you specify an integer between 1 and the number of objects in the list. This index indicates the “slot” in the list of the object you want to access. For example, to access the third object in the list, specify index 3 to the method `objectAt:`. You can get the index number of an object reference by using the `indexOf:` method.

You cannot use lists to store object references that have the value nil. If you pass nil to methods that take an object reference as a parameter, the method will not do anything.

Remember that it is an error to try to use a reference to an object that has been destroyed. Therefore, be sure that all references to an object have been removed from lists before you destroy the object.

Instance Variables

None declared in this class
Instance Methods

Method for destroying a VSList

destroy

destroy

This method destroys the list, but does not destroy any of the objects in it. After you send this message to a list, it is an error to send any more messages to it.

See also: destroyObjects, destroy (VSObject)

Accessing objects by index

insertObject:at:

insertObject:(obj ref)anObjectRef at:(integer)index

Inserts the given object reference into the list at the specified position. All object references at greater indices are “pushed down” one slot to make room for the inserted object reference. This method does nothing if index is less than 1 or greater than the index of the last slot in the list, or if anObjectRef is nil.

See also: addObject:

removeObjectAt:

(obj ref)removeObjectAt:(integer)index

The object reference at position index in the list is removed. All object references below it are moved up one slot to close the gap. This method does nothing if index is less than 1 or greater than the index of the last slot in the list. The removed object, if any, is returned.

See also: removeObject:
replaceObjectAt:with:

    replaceObjectAt:(integer)index with:(objref)newObjectRef

This method replaces the object at index with the new object reference newObjectRef. The old object reference is lost, but the actual object that the old reference referred to is not affected in any way.

This method does nothing if index is less than 1 or greater than the index of the last slot in the list.

See also: replaceObject:with:

objectAt:

    (objref)objectAt:(integer)index

This method returns the object reference at slot index of the list, or nil if index isn’t valid. The index is valid if its value is between 1 and the number of objects in the list, inclusive.

See also: indexOf:

Accessing objects by object reference

addObject:

    addObject:(objref)anObjectRef

Adds the object reference anObjectRef to the end of the list, unless anObjectRef is nil.

See also: insertObject:at:, addObjectIfAbsent:

addObjectIfAbsent:

    addObjectIfAbsent:(obj ref)anObjectRef

This method works like addObject:, except that it only add the object reference if it is not already in the list.

See also: addObject:
**removeObject:**

\[\text{removeObject}:(\text{obj ref})\text{anObjectRef}\]

If \text{anObjectRef} is not nil, and if that reference is stored in the list, the first occurrence of the reference is removed from the list, and the object references below it are moved up one slot.

See also: **removeObjectAt:**

**replaceObject:with:**

\[\text{replaceObject}:(\text{obj ref})\text{anObjectRef with:(obj ref)}\text{newObjectRef}\]

If \text{anObjectRef} is in the list, it is replaced with \text{newObjectRef}. Otherwise, this method does nothing.

See also: **replaceObjectAt:with:**

**indexOf:**

\[(\text{integer})\text{indexOf:(obj ref)}\text{anObjectRef}\]

This method returns the index of the given object reference. This index can then be used in VSLList methods that require an index as a parameter. If \text{anObjectRef} isn’t in the list, -1 is returned.

See also: **objectAt:**

**Accessing objects by relative position**

**firstObject**

\[(\text{obj ref})\text{firstObject}\]

Returns the first object reference in the list, or nil if the list is empty.

See also: **lastObject, removeFirstObject, objectAt:**
removeFirstObject

(obj ref)removeFirstObject

Removes the first object reference from the list, or does nothing if the list is already empty. If an object reference is actually removed, that object reference is returned. Otherwise, nil is returned.

See also: removeLastObject, firstObject, removeObjectAt:

lastObject

(obj ref)lastObject

Returns the last object reference in the list, or nil if the list is empty.

See also: firstObject, removeLastObject, objectAt:

removeLastObject

(obj ref)removeLastObject

Removes the last object reference from the list, or does nothing if the list is already empty. If an object reference is actually removed, that object reference is returned. Otherwise, nil is returned.

See also: removeFirstObject, lastObject, removeObjectAt:

Asking for the number of object references in the list

count

(integer)count

Returns the number of object references in the list.

See also: isEmpty
**isEmpty**

`(boolean)isEmpty`  
This method returns true if the list contains no object references, or false if it does.

See also:  **count**

**Other VSLList operations**

**empty**

`empty`

This method removes all object references from the list. It does not destroy the objects that were in the list; if you want to do that, use the **destroyObjects** method.

See also:  **destroyObjects**

**destroyObjects**

`destroyObjects`

Destroys all objects referred to by object references in the list, and removes them from the list. The difference between this method and the **empty** method is that this method destroys the actual objects, whereas **empty** does not. After an object has been destroyed, be sure you don’t refer to it again.

See also:  **empty**
VSMModel

Inherits from: VSOBJECT

Class Description

The VSMModel class contains methods that are of general use to objects in a model. An instance of VSMModel (or a subclass thereof) is automatically created for you in every model you create. This instance is responsible for driving the simulation. It is global to all VSL scripts in the model, and you can access it by specifying VSEModel as the receiver of messages, e.g. [clock of VSEModel].

The VSMModel instance contains methods for looking up components by name, and also defines two methods startUp and shutDown that you can override to implement logic that will be executed when a model execution replication starts and ends, respectively. The global instance also contains methods to let you access the simulation clock, and to let you manipulate random number streams.

You should never create more instances of VSMModel, although you may wish to create a subclass of it to specify your own logic. If you do subclass VSMModel, you should use the Attributes panel of the VSE Editor to specify that subclass as the class of the VSEModel instance.

Instance Variables

None declared in this class

Instance Methods

Accessing model attributes

name

(string)name

Returns a string containing the name of the model. This is the same as the name of
the model file in the file system.
Accessing the state of the simulation

**clock**

(REAL) `clock`

This method returns the current value of the simulation clock, in the time units of the model. These time units are generic, and the amount of actual system time that a unit of simulation model time corresponds to is defined by the modeler.

**dynObjsDestroyed**

(INTEGER) `dynObjsDestroyed`

Returns the total number of dynamic objects that have been destroyed since the start of this simulation run. This is sometimes used in computation of statistics.

**replicationNumber**

(INTEGER) `replicationNumber`

Returns the number of the current model execution replication. This will be between 1 and the number returned by `numberOfReplications`, inclusive. Since each replication should be independent, use of this method should be extremely rare.

See also: `numberOfReplications`

**numberOfReplications**

(INTEGER) `numberOfReplications`

Returns the total number of replications that constitute the current simulation experiment. Since each replication should be independent, use of this method should be extremely rare.

See also: `replicationNumber`
warmedUp

(boolean) warmedUp

Returns true if the model has been warmed up, false if not.

See also: warmedUpLengthInTimeUnits, warmedUpLengthInDynObjs

warmUpLengthInTimeUnits

(real) warmUpLengthInTimeUnits

Returns the length in simulation time units of the warm-up period of this replication. If the model has not yet been warmed up, the current simulation time is returned.

See also: warmedUpLengthInDynObjs, warmedUp, clock

warmUpLengthInDynObjs

(integer) warmUpLengthInDynObjs

Returns the length in dynamic objects destroyed of the warmup period of this replication. If the model has not yet been warmed up, the number of destroyed dynamic objects is returned.

See also: warmUpLengthInTimeUnits, dynObjsDestroyed, clock

Looking up components by name

topLevelComponent

(VSComponent ref) topLevelComponent

This method returns an object reference to the top level component of the model’s static component hierarchy.

componentNamed:in:

(VSComponent ref) componentNamed:(string) componentPathname
in:(VSComponent ref) anchor

This method looks up and returns the component specified by the given
componentPathname. A component pathname is a string that uniquely identifies a component in the model’s static component hierarchy. The way this works is very similar to the way a file path in the Unix file system uniquely specifies a file. A component pathname consists of component names, separated by slashes (“/”), that define the ‘path’ one needs to follow from the root component (for absolute pathnames) or the anchor component (for relative pathnames) to find the specified component.

For example, the pathname “/United States/Virginia/Virginia Tech” refers to a component named “Virginia Tech”, which is located in a component named “Virginia”, which in turn is located in the top level component, named “United States”. Because the first character of the string is a slash, the pathname is said to be absolute. For absolute pathnames, the anchor parameter isn’t used, and it may be specified as nil. The first component name in the string must be the name of the root component, the second component name must be the name of a child of the root component, and so on.

You can also specify a relative component path. In this case, the path string does not start with a slash, and it is considered to be relative to the anchor component. The first component name in the path string must be the name of a child component of anchor, the second name must be the name of a grandchild component of anchor, etc. In the previous example, the component “Virginia Tech” could also be specified using a relative path. For example, the string “Virginia/Virginia Tech” could be used as componentPathname, and an object reference to the component named “United States” could be specified as anchor.

If componentPathname is a relative path, you must specify an anchor. If it is absolute, you may pass the value nil for the anchor parameter.

This method returns an object reference to the component, or nil if componentPathname doesn’t specify a valid component in the receiving model.

See also: componentNamed: (VSComponent)
Methods for using random number streams

newRndStreamWithSeed:

(integer) newRndStreamWithSeed: (integer) initialSeed

Send this message to VSEModel when you want to create a new stream of pseudo-random numbers. The initialSeed parameter is the initial seed that the random number stream will use. If you specify 0 for initialSeed, every random number in the stream will also be 0. If you specify any nonzero seed, the stream will generate pseudo-random numbers.

This method returns an integer stream identifier, which can later be used as the first parameter of the random variate generation functions (uniform(), exponential(), etc). Typically you will assign this message expression to a variable in your script, as in the following example:

declarations
  integer arrivalTimeStream;
logic
  set arrivalTimeStream to [newRndStreamWithSeed: 42 of VSEModel];

One random number stream is created for you automatically. It has stream identifier 0. Therefore, if your model only needs to use one stream, you can specify 0 as the first argument to a random variate generation function without first having to send the newRndStreamWithSeed: message to VSEModel.

See also: setSeed:ofRndStream:

setSeed:ofRndStream:

setSeed: (integer) newSeed ofRndStream: (integer) streamNumber

Use this message when you want to change the seed of a random number stream. The stream must have previously been created by sending the newRndStreamWithSeed: method (or it must be stream number 0, the automatically created random number stream).

If streamNumber isn’t the identifier of an existing stream, an error panel will be shown.

See also: newRndStreamWithSeed:
Methods invoked during simulation startup and shutdown

**startUp**

**startUp**

This message is sent to the model instance when a replication starts. You never send the **startUp** message yourself; it is sent for you by the runtime system. However, you may want to override this method if you subclass VSMModel. You can use it to initialize instance variables and to perform any other tasks you need performed every time a replication begins.

See also: **shutDown**

**shutDown**

**shutDown**

This message is sent to the model instance when a replication stops. You never send the **shutDown** message yourself; it is sent for you by the runtime system. However, you may want to override this method if you subclass VSMModel. You can use it to destroy any objects you created in **startUp**, and to perform any other tasks you need performed every time a replication ends.

See also: **startUp**
VSObject

Inherits from: none (VSObject is the root class)

Class Description

This is the root class of the Visual Simulation Class Library. All other classes in the library inherit from it, so all objects can respond to the methods declared in this class.

Instance Variables

None declared in this class

Class Methods

Method for creating a new instance of a class

newInstance
    (obj ref)newInstance

Send this message to a class to create an instance of that class. The newly created instance will be sent the initNewInstance message to let it initialize itself. This method returns an object reference to the new instance.

You shouldn’t need to override this method, but if you do, be sure to pass the message to super as the first line of your method implementation.

See also: initNewInstance, destroy, startUp (VModel), startUp (VSDynamicObject), startUp (VComponent)
Instance Methods

Method for initializing a new instance

initNewInstance

This method is implemented by subclasses of VSOBJECT to initialize the instance variables of newly created instances. An object isn’t ready to be used until it has been initialized. You should never send a initNewInstance message yourself; it is automatically sent to the new instance by the newInstance method. However, you often override this method in your own subclasses to initialize instance variables.

If you override this method, be sure to pass the message to super as the first line of your method implementation.

For subclasses of VSOBJECT that respond to the startUp message (such as VSComponent, VSDynamicObject, and VSMODEL), you should override that method instead of this one to implement your startup code.

See also: newInstance, destroy, startUp (VSMODEL), startUp (VSDynamicObject), startUp (VSComponent)

Method for destroying an object

destroy

This method destroys the receiving object, making all object references to it invalid. Once an object has been destroyed, you should never try to send messages to it again.

You may want to override this method to add logic for deinitializing the instance before it is destroyed. For example, you may wish to collect some statistics here. If you override this method, be sure to pass the message to super as the last line of your method implementation.

See also: newInstance, initNewInstance
Methods for accessing object information

class
    (obj ref)class

Returns a reference to an object implementing the class of the receiver. You can send class messages to the returned object references. You can also use the returned reference to compare the class of an object. For example:

    if [class of anObject] is VSDynamicObject then
    (            ...            
    )

Be sure to never destroy a class object.

name
    (string)name

Returns the string representing the name of the object.

See also:  setName:

setName:
    setName:(string)newName

Sets the name of the receiving object to be equal to the given string. Some subclasses of VSOBJECT refuse to have their name set. For example, component names need to be unique, so you cannot use this method to set the name of a component.

See also:  name
VSPathItem

Inherits from: VSIItem : VSOBJECT

Class Description

All paths in the model are instances of VSPathItem. You cannot create subclasses of VSPathItem, but you can obtain object references to VSPathItem instances (by using VSCComponent’s itemNamed: method, for example). These object references can then be passed as parameters to other runtime methods.

Instance Variables

None declared in this class

Methods

None declared in this class
VSRectangleItem

Inherits from: VSItem : VSOBJECT

Class Description

Each rectangle item created in the Model Editor is an instance of VSRectangleItem. As with any item, instances of VSRectangleItem should not be created directly by VSL scripts. The VSRectangleItem class is public because references to VSRectangleItem objects may be returned by VSClponent’s itemNamed: method.

Instance Variables

None declared in this class

Methods

None declared in this class
VSTextItem

Inherits from: VSIItem : VSOBJECT

Class Description

Instances of VSTextItem display text during visual simulation. You cannot create instances of the VSTextItem class, but you can send itemNamed: messages to components (specifying the names of text items) to get object references to existing VSTextItem objects. You can then send one of the display... messages to the returned object references to change the text displayed. This is useful for showing status messages, debugging messages, etc.

Instance Variables

None declared in this class

Instance Methods

Methods for changing what’s displayed in the text item

displayReal:

   displayReal:(real)value

Displays the given real value in the text item.

See also: displayInteger:, displayString:

displayInteger:

   displayInteger:(integer)value

Displays the given integer value in the text item.

See also: displayReal:, displayString:
displayString:
  displayString:(string)value

Displays the given string in the text item.

See also: displayInteger:, displayReal:
VITA

Name: P. Anders I. Bertelrud

Address: 750 Hunter Mill Rd. #8300J
          Blacksburg, Virginia 24060
          U.S.A.

Phone: (504) 552-6577

Birth date: 11 November 1971

Education: M.S., September 1995
            Computer Science
            Virginia Polytechnic Institute and State University
            Blacksburg, Virginia, U.S.A.

            B.S., May 1993
            Computer Science
            Virginia Polytechnic Institute and State University
            Blacksburg, Virginia, U.S.A.

Mr. Bertelrud graduated Magna Cum Laude from Virginia Tech in 1993. He is fluent in Swedish (his native language) and English. Apart from computers, his interests include music, travel, skiing, and martial arts. He is a cofounder of Orca Computer, Inc.

Mr. Bertelrud is a student member of the Association of Computing Machinery. He is also a member of Phi Beta Kappa honor society and Upsilon Pi Epsilon, the national Computer Science honor society.