

AL-TR-1992-0161

AD-A261 257



SYSTEM CONCEPT OF OPERATIONS EVALUATOR (SCOPE)

Terry R. Raymond

MYSTECH ASSOCIATES, INCORPORATED
438 EAST MAIN ROAD
MIDDLETOWN, RI 02840

David A. Hathaway, Capt, USAF

HUMAN RESOURCES DIRECTORATE
LOGISTICS RESEARCH DIVISION

NOVEMBER 1992

FINAL TECHNICAL REPORT FOR PERIOD MARCH 1991 - SEPTEMBER 1992

Approved for public release; distribution is unlimited.

98 3 1 024

AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6573

DTIC
ELECTE
MAR 02 1993
S E D

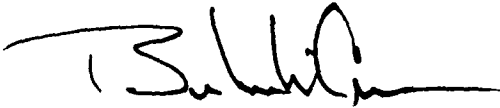
NOTICES

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this report and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.


DAVID A. HATHAWAY, Capt, USAF
Contract Monitor


BERTRAM W. CREAM, Chief
Logistics Research Division

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE November 1992		3. REPORT TYPE AND DATES COVERED Final - March 1991 - September 1992
4. TITLE AND SUBTITLE System Concept of Operations Evaluator (SCOPE)			5. FUNDING NUMBERS C - N66604-90-D-0104 PE - 62205F PR - 1710 TA - 00 WU - 76	
6. AUTHOR(S) Terry R. Raymond David A. Hathaway, Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Mystech Associates, Inc. Armstrong Laboratory 438 East Main Road Human Resources Directorate Middletown, RI 02840 Logistics Research Division Wright-Patterson AFB, OH 45433-6573			8. PERFORMING ORGANIZATION REPORT NUMBER D-N01-92	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Armstrong Laboratory Human Resources Directorate Logistics Research Division Wright-Patterson AFB, OH 45433-6573			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AL-TR-1992- 0161	
11. SUPPLEMENTARY NOTES Armstrong Laboratory Technical Monitor: Capt David A. Hathaway, AL/HRGA, (513) 255-9945				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Command and control centers and operations centers are environments in which information must be collected, processed, integrated and analyzed in order to accomplish specific missions. To be operationally and economically effective, the system design of these centers and the design of their concept of operations must be coordinated. This is a complex task due to the myriad of information available, the variety and limits of technologies that can be incorporated into the center, and the very large number of activities and information that the technology must support. The System Concept of Operations Evaluator (SCOPE) provides the designer with a tool to address this problem. It enables the designers to capture the complex relationships between various system objects using an object-oriented representation. Furthermore, it facilitates the comparison of alternatives by providing a layered construction process where all the alternatives in a layer utilize the same knowledge defined in the underlying layer. The designer can then analyze either the static relationships of a model or simulate a scenario using discrete event simulation. This report describes the model building concepts and tools concepts of SCOPE.				
14. SUBJECT TERMS modeling object-oriented simulation			15. NUMBER OF PAGES 39	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	

Contents

	<u>Page</u>
Figures	v
Preface	vi
Introduction.....	1
Purpose.....	1
History.....	2
Modeling Concepts	4
Objects and Attributes	4
Object Identity	4
Object Inheritance	5
Object Hierarchy.....	6
Object Relationships	7
Object Behavior (User-Defined Functions)	8
Object Pictures.....	8
Knowledge Reuse.....	9
Tool Concepts.....	11
Tool Overview	12
Knowledge Base Definer.....	12
Knowledge Base Manager	14
Picture Definer.....	16
Script Builder	18
Tool Design.....	20
Smalltalk Environment.....	20
Knowledge Base Architecture	20
Knowledge Base	20
Knowledge Base Objects (KObject)	21
Attributes	21
Attribute Specifications	21
Attribute Value Holders	22
Cross-Linking	22
Object Pictures.....	23
KObjectPicture.....	23
Attribute Pictures	23

Primitives.....	24
Selectors.....	24
Interfaces.....	24
Frame.....	24
Common Objects.....	24
LabeledFormListView.....	25
ClassSelectionList.....	25
KBObjectInspector.....	25
SimpleMenuView.....	25
Tools.....	26
Knowledge Base Definer.....	26
Knowledge Base Manager.....	26
Script Builder.....	27
Picture Definer.....	28
Simulation.....	28
Smalltalk Simulation Component.....	28
Knowledge Base Simulation Template.....	29
Script Construction.....	30
Script Execution.....	30

Figures

<u>Figure</u>		<u>Page</u>
1	SCOPE "Human" Object	4
2	"Supervisor" and "Employee" Objects	6
3	SCOPE Knowledge Base Object Hierarchy.....	7
4	Knowledge Base Definer	13
5	Knowledge Base Manager	15
6	Picture Definer	17
7	Script Builder	19
8	Knowledge Base Manager Composition	27

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

NOT INSPECTED

Preface

This document describes the development and application of the System Concept of Operations Evaluator (SCOPE) tool. SCOPE was developed by Mystech Associates of Middletown, RI, under Naval Undersea Warfare Center (NUWC) contract N66604-90-0104. The development effort described in this document was funded by Armstrong Laboratory, Logistics Research Division (AL/HRG), Wright-Patterson Air Force Base, OH, under the technical direction of Captain David A. Hathaway. Mr. Terry Raymond of Mystech Associates was the SCOPE systems engineer.

Prior to Armstrong Laboratory taking financial responsibility for the program in March 1991, the SCOPE development and technical direction was managed by Mr. James Cary of NUWC in Newport, RI. NUWC invested over three years in the SCOPE program (then known as the Submarine Concept of Operations Evaluator) prior to Air Force involvement. Their effort successfully produced an exploratory development (6.2) "proof-of-concept demonstrator" which illustrated the potential features and capabilities an operational SCOPE prototype could employ. Unfortunately, the demonstrator was "hardwired" to depict a submarine attack center and could not be used for other applications. Armstrong Laboratory's objective was to build on the concepts proven by NUWC and develop a fully functional tool which could be used in a variety of application areas outside of the submarine domain.

Modeling and simulation technologies have been identified as extremely cost-effective means to answer questions about complex systems. SCOPE is an integral part of the HRG efforts in this area. As a 6.2 program, SCOPE is generating technical ideas in the areas of user interface and model building capabilities for larger 6.3 programs. As a stand-alone software tool, SCOPE is actively providing a needed capability to operational users through field testing at NORAD and U.S. Space Command in Colorado Springs, Colorado.

Introduction

The task of designing weapon systems and command systems is becoming increasingly difficult. While the primary reason for this is the increasing sophistication of the threat environment, the shrinking defense budget and the addition of new missions also serve to complicate the process. Modern-day weapon systems are being asked to be more capable, address more missions, require fewer personnel, and cost less. To the system designer, this can be summarized as increased complexity. The problem of complexity has been addressed by the introduction of various Computer-Aided System Engineering (CASE) tools. Most, if not all, CASE tools have been designed to address only a portion of the system design. There are tools that address system requirements analysis, software design, and real-time system behavior, but none that address the use of and the human interaction with a system. SCOPE was conceived to address this deficiency.

The design team decided to use the object-oriented modeling paradigm to represent the information or knowledge about a system model. The object-oriented representation remained as the SCOPE tool evolved, but the designers decided to make the tool into a more general-purpose tool which would allow the user to define his/her own basic object types and object relationships. This change to the SCOPE tool significantly expanded its potential to model complex systems and provides the basic foundation of the current SCOPE tool.

Purpose

SCOPE is an engineering tool, designed to be a repository of engineering knowledge and to manage and analyze that knowledge. SCOPE began strictly as a tool to model a submarine concept of operations but has since evolved into a much broader engineering tool with a greater capability to model the operations of a system. As a modeling tool, SCOPE provides two kinds of system models: static and dynamic. The object-oriented representation of a static model is where SCOPE's strength lies. The static model captures a considerable portion of the complete system model and serves as the foundation of the dynamic model. The static model by itself serves as a database upon which queries and analysis can be performed. Dynamic analysis is currently performed by creating an object sequence for a particular scenario and running the sequence using a discrete event simulator. The simulation results can be subsequently analyzed using standard analysis techniques.

One other recommended use of SCOPE is as an engineering notebook. If the engineers on a design team would continually enter their comments or thoughts about an object or aspect of the system, the usefulness and understanding of the system model would be considerably enhanced. Furthermore, the design of large complex systems evolves as the customer's requirements change and the designers' understanding grows. But these long-lived, changing systems have problems with continuity when design team personnel leave the team and take their experience with them. If designers use SCOPE as a notebook, their knowledge about the system will remain when they leave.

History

SCOPE began in Fiscal Year (FY) 1987 as a Naval Underwater Systems Center (NUSC) sponsored project called Concepts Assessment for Combat Control (CACC) Encyclopedia. The purpose of the CACC Encyclopedia was to prototype a number of concepts underlying CACC and its approach to modeling systems. Initially, this was restricted to just submarine combat systems. The first concept is the separation of domain knowledge (i.e., submarine warfare) from knowledge about both technology and specific system implementations. Second, these knowledge bases should be reusable. Third, the tool should support all levels of system description from conceptual to engineering specification; this should be accomplished by enabling the user to add layers of new types of specifications. Fourth, the tool should describe hardware, software, and people elements of a system. Fifth, the tool should support comparisons of alternative designs. These requirements led to an object-oriented representation. At the conclusion of FY87, the project had produced an initial representation of the submarine domain and a tool to manage the knowledge base, which was developed in Smalltalk/V and hosted on an IBM XT clone. The project was successful in demonstrating the utility of this representation, but it also demonstrated the need for more computer power than the XT could provide. Subsequently, in FY88 a new tool called the Submarine Concept of Operation Evaluation was developed in Smalltalk-80 on a Tektronix color workstation. This tool explored various user interface and analysis concepts. In FY89, the tool was modified to investigate scripting of scenarios and the analysis of the scripts. The previous knowledge representation did not adequately support representing a script so a different knowledge representation, for which each binary relationship between two objects was represented as a separate object, was investigated. This representation, although more flexible at representing system dynamics than the object-oriented representation, was shown to require significantly more computer resources. In FY91, the Air Force Armstrong Laboratory, Logistics Research Division (AL/HRG), funded a new effort that is the foundation of the current SCOPE tool. AL/HRG

recognized the potential of the tool to represent and evaluate a much wider range of system models than a submarine combat system, such as the command and control activity of a ground-based operations center. The purpose of this effort was to build a broader, more flexible tool using the “lessons learned” from the previous versions. The broadened focus of the tool led to its new name: System Concept of Operation Evaluator.

Modeling Concepts

A SCOPE model is constructed by describing its objects, their behavior, and their visual representations. In the following discussion, we describe objects and their attributes, the concepts of grouping them into convenient hierarchies, attaching behavior to the objects, and effectively reusing models.

Objects and Attributes

The SCOPE knowledge base (KB) comprises a collection of modeling objects. These objects are referred to as knowledge base objects (KBOjects). KBOjects represent entities in a user's system or the real world. They are characterized by a set of attributes which are defined by the user. The attributes, the characteristics of an object, provide a meaningful description of the object and are important in the subsequent model analysis. In Figure 1, the object called "Human" has an attribute called "Age," the value of which would be a number. The value of an attribute can also be another KBOject, thus allowing the user to define relationships between objects in the model.

Human Object	
Attribute	Values
Name	
Age	

Figure 1
SCOPE "Human" Object

Object Identity

One other important but implicit attribute of an object is its identity. For example, we can refer to a person by basically two methods: using their name or pointing to them. SCOPE also uses the same two methods to refer to a KBOject but, unlike our world, SCOPE also guarantees that each KBOject name will be unique (except for clones). Within the Smalltalk image, an object is referenced only by a pointer. Even though a pointer is used to reference an object, the correct way to think about the process is that the attribute or variable contains the object. Furthermore, because the objects are referenced using pointers, more than one attribute can contain the same object. When a new object is placed into the attribute, the attribute contains the new object but the previous

object is not changed or destroyed by the new assignment. As previously mentioned, KObjects have unique names. Unique names are required in order to save the KObjects in a file external to the Smalltalk image. Objects that reside only in a file do not have a Smalltalk identity so they require some other globally unique identifier, which we have chosen to be their name.

Object Inheritance

When creating a model, we frequently encounter a situation in which many different object types have only slight differences in their attribute definitions. If the tool allowed the user to create object templates then specialize them for each object variation, both the management of the knowledge and the understandability of the model would be improved. Object-oriented modeling provides this capability through inheritance. In the inheritance structure, each attribute that is defined for an object is promulgated to the object's children or subclasses, so that each successive child only needs to define how it differs from its parent. This process is referred to as specialization. Figure 2 illustrates the inheritance process. In this figure, a root object, "Human," is defined with "Name" and "Age" attributes. Next, its child, "Employee," adds a "Supervisor" attribute to the definition. At this point, the model may have many types of employee objects but we have shown only a "Supervisor" object, which adds an "Employees" attribute. One other useful way to view the inheritance relationship is as an "is a" relationship. Again referring to Figure 2, we can see that the "Supervisor" object is an "Employee" object, which is a "Human" object.

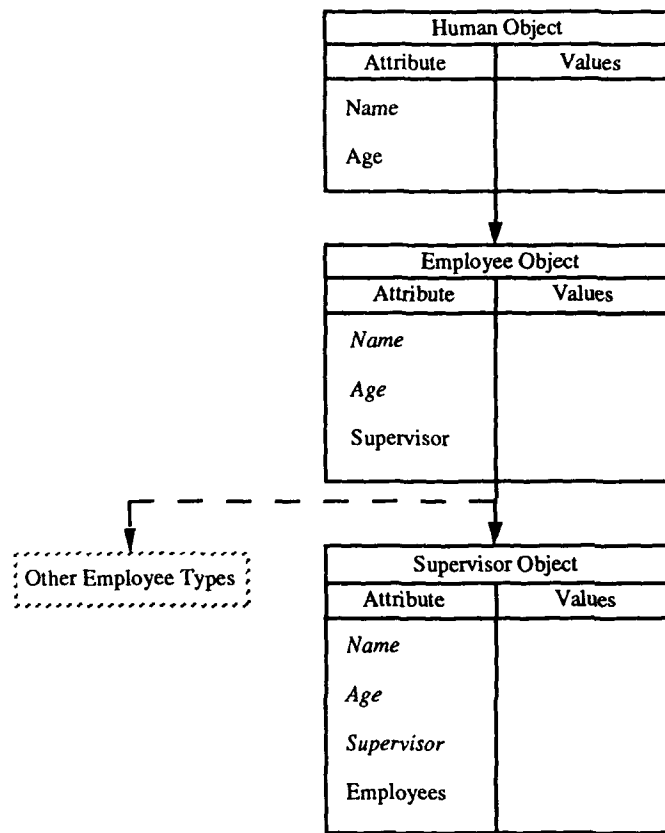


Figure 2
"Supervisor" and "Employee" Objects

Inheritance has uses other than the inheritance of attribute definitions. SCOPE also uses inheritance for behavior, object pictures, and attribute values (object behavior and object pictures are explained in detail later). To reiterate, inheritance can be used for aspects about an object that relate to object specialization within an "is a" relationship.

Object Hierarchy

Model construction in SCOPE begins by first defining the root objects and attributes in the model. Hierarchies are then created from these roots using the "is a" relationship. The meanings of the objects become more specialized as one progresses down a hierarchy, even though they may not have any additional attributes. This is useful from the perspective of managing the information in the model; that is, classifying the objects makes it easier to find existing objects and identify similarities between objects. Novice users frequently mix other relationships into the inheritance hierarchy. They may progress down a few levels using the "is a" relationship then, near the

bottom, switch to a composition relationship. This may not create any immediate problems, but will eventually cause confusion and analysis problems. A substitution test can help determine if the relationships in an inheritance hierarchy are truly "is a" relationships. If a relationship is an "is a" relationship, any child object can substitute for any of its ancestors in any situation.

In creating a hierarchy, the user must select a taxonomy for describing the objects and their relationships to each other. Several typical taxonomies are illustrated at the top of Figure 3; however, the taxonomy selected is less important than maintaining consistency of taxonomy throughout the modeling process. Hierarchies can be of any level; three are shown in Figure 3. As one moves down through a hierarchy, the description of an object becomes more refined and more faithfully represents its real-world counterpart. One advantage of SCOPE is that it allows the user to define and revise the breadth and depth of his/her real-world model, as required.

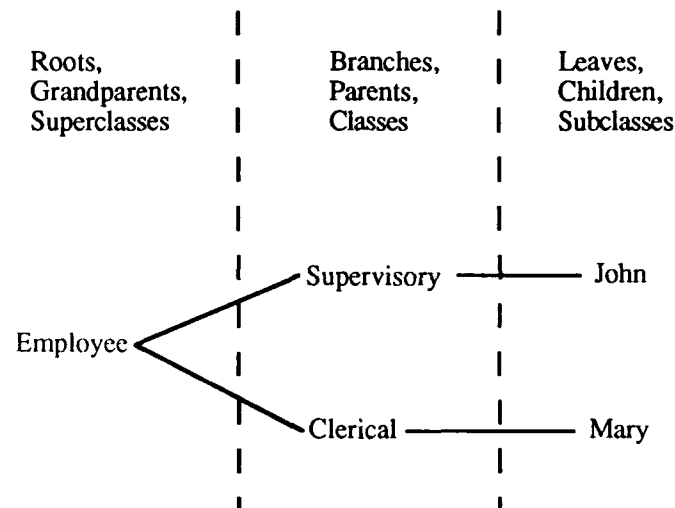


Figure 3
SCOPE Knowledge Base Object Hierarchy

Object Relationships

In the previous discussion about hierarchies we focused on the "is a" relationship; however, there are many other useful hierarchical relationships. For instance, a model may represent the organizational hierarchy of a company or a composition hierarchy of a complex object. These other types of hierarchical relationships can be represented in SCOPE by defining attributes that represent these relationships. Furthermore, these object relationships can be much more than a hierarchy. Basically, they describe a graph. When defining relations between objects, it is

important to also define the reciprocal relationships, primarily because it greatly simplifies analysis and improves understanding. For example, if John has an "Employees" attribute with one of its values being Mary, the reciprocal attribute for Mary would be a "Supervisor" attribute with a value of John. Furthermore, if Mary did not have the attribute we would have to search all supervisor objects to find Mary's supervisor. SCOPE also facilitates the management of the reciprocal relationships by completing the reverse entry for the user.

Object Behavior (User-Defined Functions)

SCOPE provides the capability for the user to define and modify object behavior through user-defined functions. User-defined functions have been incorporated into SCOPE to provide the user with a flexible simulation and analysis capability. A function is attached to a particular object or class in a hierarchy. Therefore, due to inheritance, the function is available to all the object's subclasses and may also be overridden by a subclass to provide more specialized behavior. The functions are Smalltalk methods but they are not kept in a method dictionary of a Smalltalk class. Instead, they are kept in a dictionary associated with the selected KObject. The functions are written in Smalltalk so the full power of Smalltalk is available to the user, although the vast majority of user functions will only use a small subset of the Smalltalk language. One goal of the SCOPE tool development is to keep the tool as easy to use as possible; therefore, our future plans include the investigation of techniques to provide a simplified facility for user functions.

Object Pictures

Object pictures are utilized by SCOPE in several places to provide a visual representation of a KObject. Object pictures are created by the user with the Picture Definer tool. A picture is attached to an object in a hierarchy which means it is available to all its subclasses through inheritance. The intent of creating a picture is to provide a means for the user to depict a KObject in a fashion that conveys something more about the object other than a simple listing of its attributes. If an object is properly modeled, all its important characteristics will be contained in the attributes. Therefore, a picture of an object displays some or all of the object's attributes and their values. The utility of a picture is its ability to visually accentuate particular attributes of an object and/or show the relationship between attributes. Pictures may also display just a subset of the attributes because the user may be interested in only a particular aspect of an object. Therefore, a picture must be capable of displaying the attributes of an object and additional graphical figures such as lines and rectangles.

The next consideration is how to present the values of an attribute. The method chosen for SCOPE is to represent the values with "subpictures" that are displayed in a region of the picture designated for the attribute. The result is a picture composed of primitive figures and a region for each attribute in which the values of the attribute are displayed. For the purpose of composing a picture, the region in which the values of an attribute are displayed is called an attribute picture. Furthermore, an attribute picture must accommodate different arrangements of its subpictures in the region. Two obvious arrangements are a vertical list and a horizontal list. The display location of each subpicture in the list is determined by the attribute picture. A third arrangement is for each value in the attribute to specify the location of its subpicture. This is called an area attribute picture. An example of an area picture would be a layout of a command center. When a command center is displayed as an area picture instead of a monolithic drawing, the components of the command center are displayed using the icon (subpicture) that represents the component. The default attribute from which the area picture obtains the location to display each subpicture is "location," which can be changed from a pull-down menu in the Picture Definer.

Currently, the biggest restriction of attribute pictures is their inability to display complex relationships, such as a line between a particular value of one attribute and a particular value of a different attribute or a circle whose radius is a function of an attribute value. Future plans include investigating alternative solutions to this problem.

As previously stated, object pictures are attached to particular objects in a hierarchy which makes them available to all of the object's subclasses. This implementation of picture inheritance provides the user with two very useful capabilities. First, a picture can be defined in the root class of a hierarchy thereby providing one uniform template for all objects in the hierarchy to display themselves. Second, if a user wants unique pictures for each object in a hierarchy (such as a component hierarchy) and wants to display but not immediately create the pictures for all the objects, this can be easily accommodated by creating a default picture in the root class that would be inherited by each object that does not have its own picture defined.

Knowledge Reuse

Model construction is rarely a one-shot exercise. Usually, it is characterized by an evolutionary process in which several alternative models are constructed and evaluated. Each model may represent a different aspect of a system or variations of a system. Therefore, it would be a big

advantage if the modeling tool provided the capability to easily reuse parts of existing models. SCOPE has been designed to do this by providing the capability to build a model on top of other KBs. These foundational KBs are referred to as basis KBs. This is done by an import process similar to C include files. When a KB or model is loaded, if it is defined upon some basis KB(s), SCOPE first loads the basis KB(s). After the basis KB has been loaded, SCOPE then loads the knowledge unique to the defining KB. Conversely, when a KB is saved, only the knowledge unique to it is saved. When a basis KB is modified, the modification will be reflected in the dependant KBs only when they are reloaded.

Tool Concepts

A primary goal in the design of SCOPE was to minimize the occurrence of knowledge management errors. This basically translates to: minimize user actions required to accomplish a goal. One technique chosen to accomplish this is to have the user define a knowledge framework within which the SCOPE tool will interact with the user. For example, attributes must be able to contain more than one value; therefore they are designed to contain any number of values. The user manages this type of attribute by adding or removing values. However, if an attribute has only one value (such as age), one generally thinks of replacing the value not adding and removing it. Thus, to simplify the maintenance of attributes, the user defines whether an attribute has one or many values. If the attribute has been defined to have one value, whenever the user selects a new value SCOPE will automatically remove the old value; if the attribute has been defined to have many values, SCOPE will add the value. Also, the process of defining the framework is kept to a minimum by using inheritance. Basically, the user defines most of the framework at the root of each hierarchy. The one disadvantage to this approach is that the setup process is more complex; however, the gains are well worth the extra effort.

Another approach to minimize user actions is for the user to identify the object to operate on by selecting it with a mouse instead of typing its name. The only time an object's name must be entered is when it is defined. The obvious shortcoming to this approach is that locating the desired object could be cumbersome if there are a large number of objects. To overcome this problem, the objects should be displayed in some kind of order. Displaying them alphabetically is one obvious choice; another option (more consistent with the modeling methodology) is to display them in a manner that reflects the inheritance hierarchy. Furthermore, if objects are displayed in a hierarchy, it is possible to selectively expand or contract branches in the hierarchy; this can significantly reduce the number of unwanted objects on the display. This second approach was taken in SCOPE. The next issue is to select the form in which to display the hierarchy. Two obvious choices are a graphical tree and an indented list. SCOPE presently displays the objects only as an indented list. This was chosen for two reasons. First, a graphical tree requires more display area to present the same information; thus the user must perform more scrolling to locate an object. Second, an indented list is less complicated to implement and provides greater flexibility and reuse. A future version of SCOPE will provide the user with the option of displaying a hierarchy as a tree.

Tool Overview

Knowledge Base Definer

The Knowledge Base Definer (shown in Figure 4) is used to define the framework of the KB. The KB definition involves creation of the root classes, class hierarchy (to the necessary depth), and class attributes as well as the specification of the attributes. Two attribute specifications, Choices and Cross Links (explained later), refer to existing KB classes and attributes. Therefore, it is necessary to create the appropriate classes and attributes before specifying Choices and Cross Links. The following steps should be followed to define a KB framework.

1. Create all root classes.
2. For each root class, create its attributes.
3. For each root class, create the class hierarchy to the necessary depth. It is not necessary to add all the classes to the hierarchy at this point. The classes that should be added are the ones that either override their superclass attribute definitions or are referred to by the Choices specification of some attribute of another object.
4. For each subclass that incorporates an additional attribute, create that attribute in the subclass.
5. Define the attribute specifications for the attributes. Attributes and their specifications are inherited from the superclass of their class, so the specification should be defined at the highest possible class in the hierarchy.

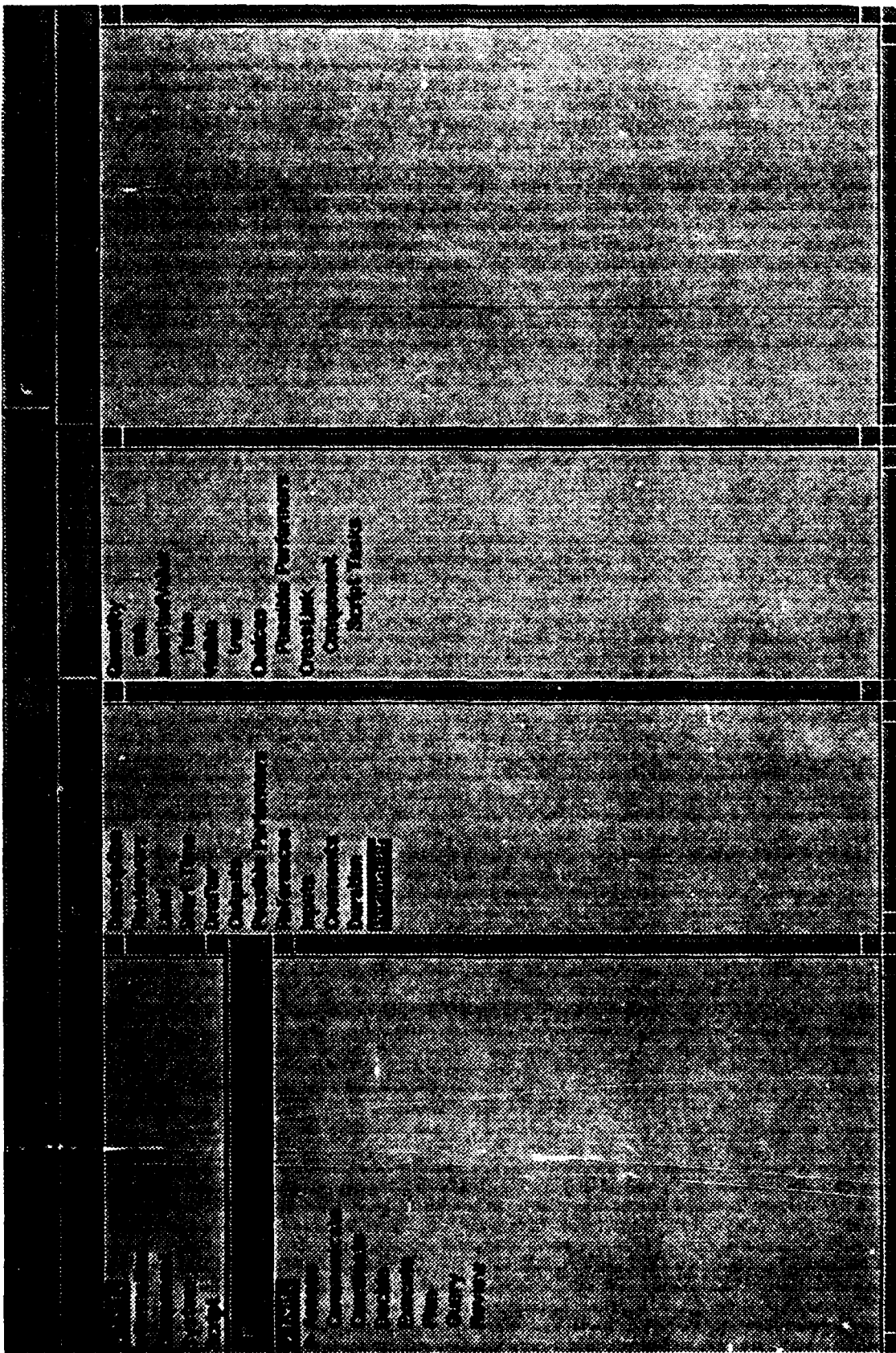


Figure 4
Knowledge Base Definer

Knowledge Base Manager

When a user is building or modifying a model, most of his/her time will be spent using the Knowledge Base Manager. The Knowledge Base Manager and its subfeatures are the primary vehicle for entering and modifying model data in the KB.

The Knowledge Base Manager (shown in Figure 5) consists of four list panes: Roots, Hierarchy, Attributes, and Choices (from left to right).

The process of modifying the KB can be broken into three logical steps:

1. select the object to be described or modified,
2. select the attribute to be modified, and
3. add or remove an attribute value.

The roots and hierarchy panes provide the primary means to select the object of interest. The panes used to view or modify the attributes of an object depend on how the user has configured the Knowledge Base Manager. Furthermore, the Knowledge Base Manager subfeatures (Examiner, Diagrammer, and Reassignment Manager) provide the same functionality as the Knowledge Base Manager to view and modify an object, but do so as an independent pop-up window. The Reassignment Manager also provides additional functionality to easily reassign values from an attribute of one object to the same attribute of another.

Activity Assess Assess Environmental Conditions Assess Target Intelligence Assess Target Situation Assess Weapon Profiles Communicate Coordinate Decide Execute Plan Query Review	Description Reviews Load 10 Start Time 0 Draft Outputs Target Classification Possible Performers Executive Officer Senior Supervisors References Inputs Acoustic Signal Intelligence Information Comments Duration 10 Performer	Information Acoustic Signal Environmental Status Signals Environmental Status Intelligence Information Target Classification Target Situation Weapon Profile
---	--	--

Figure 5
Knowledge Base Manager

Picture Definer

The Picture Definer (shown in Figure 6) is used to construct the object picture templates that a Picture Viewer uses to display an object. It is also employed by the user to aid in the design of the physical layout of a system. The Picture Definer consists of a set of list-selection panes on the left side of the display, a drawing region, and a set of pull-down menus above the drawing region.

After entering the Picture Definer, the user must first select the object for which the picture is being defined or modified, then select the picture being defined or modified. At this point, the picture definer enters the drawing mode. Once in the drawing mode, the user can draw primitive figures or define attribute pictures. Primitive figures are created by selecting the primitive from the list view and then shaping it. When a user selects a particular figure in the drawing region, mouse handles appear on the figure. The figure can subsequently be reshaped by dragging a mouse handle. An attribute picture is created by selecting, from the list view, the attribute the picture is to represent, then shaping its display area in the drawing region. A multivalued attribute picture may be one of three types: a vertical list, a horizontal list, and an area. The picture used to display the values of an attribute in its attribute picture is selected from the subpicture selection list. An area picture displays the attribute's value pictures distributed in an area instead of a list. The actual location at which each picture is displayed is specified by an attribute in each value. The default attribute used by the attribute picture is named "location." This can be changed by the user via a pull-down menu.

The pictures defined by the Picture Definer are designed to be displayed by a Picture Viewer. To support the desired behavior of the Picture Viewer, the Picture Definer also enables the user to create selector and action buttons. How these are used by the Picture Viewer is explained in the KLObjectInspector description.

The Picture Definer also has a mode designed to facilitate the layout of a system. This mode is used in conjunction with an area picture. The system layout is normally drawn as a background diagram containing the walls and other fixed structures and an attribute area picture overlay which contains the system components. Each system component is displayed with its own drawing, which is another object picture, at the component's desired location. Using this mode, the user can select a system component in the area attribute picture and drag it to a different location. This ability to rearrange the layout in SCOPE aids in the assessment of alternative system layouts.

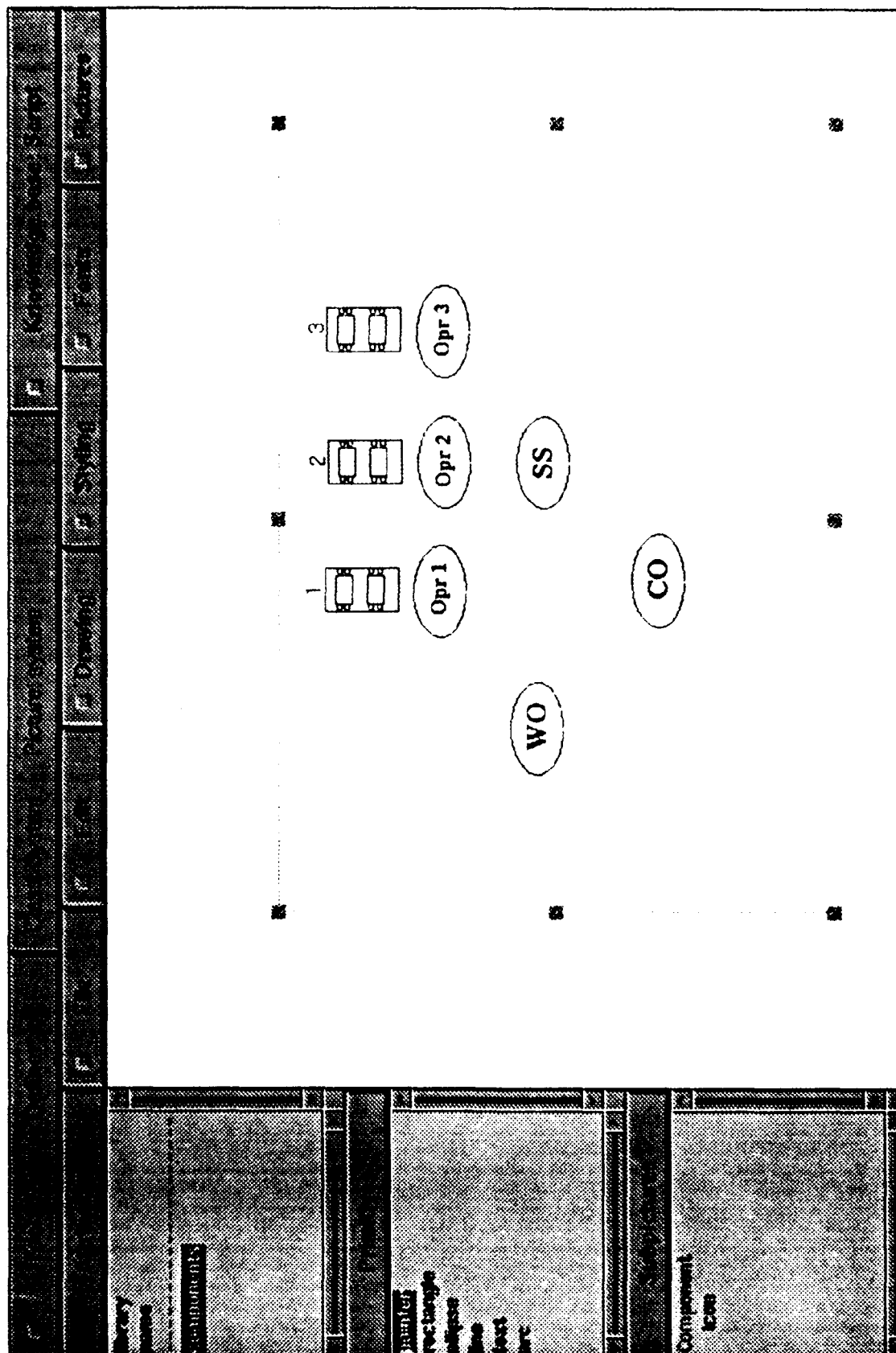


Figure 6
Picture Definer

Script Builder

The Script Builder (shown in Figure 7) provides the user the ability to define a system task script. A system task script is divided into separate resource scripts, one for each component used in the script. Each resource script is a set of time-sequenced activities performed by the resource. Currently, the Script Builder provides the user two ways to construct a script. The first is to lay out the activities on a time line by specifying the start time and duration of each activity. The second is to specify only the activity-sequence dependencies; the start times are then determined by discrete event simulation. At the completion of the simulation, the script has all the start times specified and used or analyzed in the same manner as a script produced using the first technique.

The precise operation of the Script Builder is tailored by the KB being used. This permits the script builder to be used for a wider range of situations than would otherwise be possible and allows the user to specify the types of modeling objects placed in a script (not just Activities) and how they are placed in the script. When a user selects an object to be placed in a script, the object is sent a message telling it to place itself into the script. This message is defined in a user-defined function. Therefore, what is placed in the script and how it is done is specified within the KB.

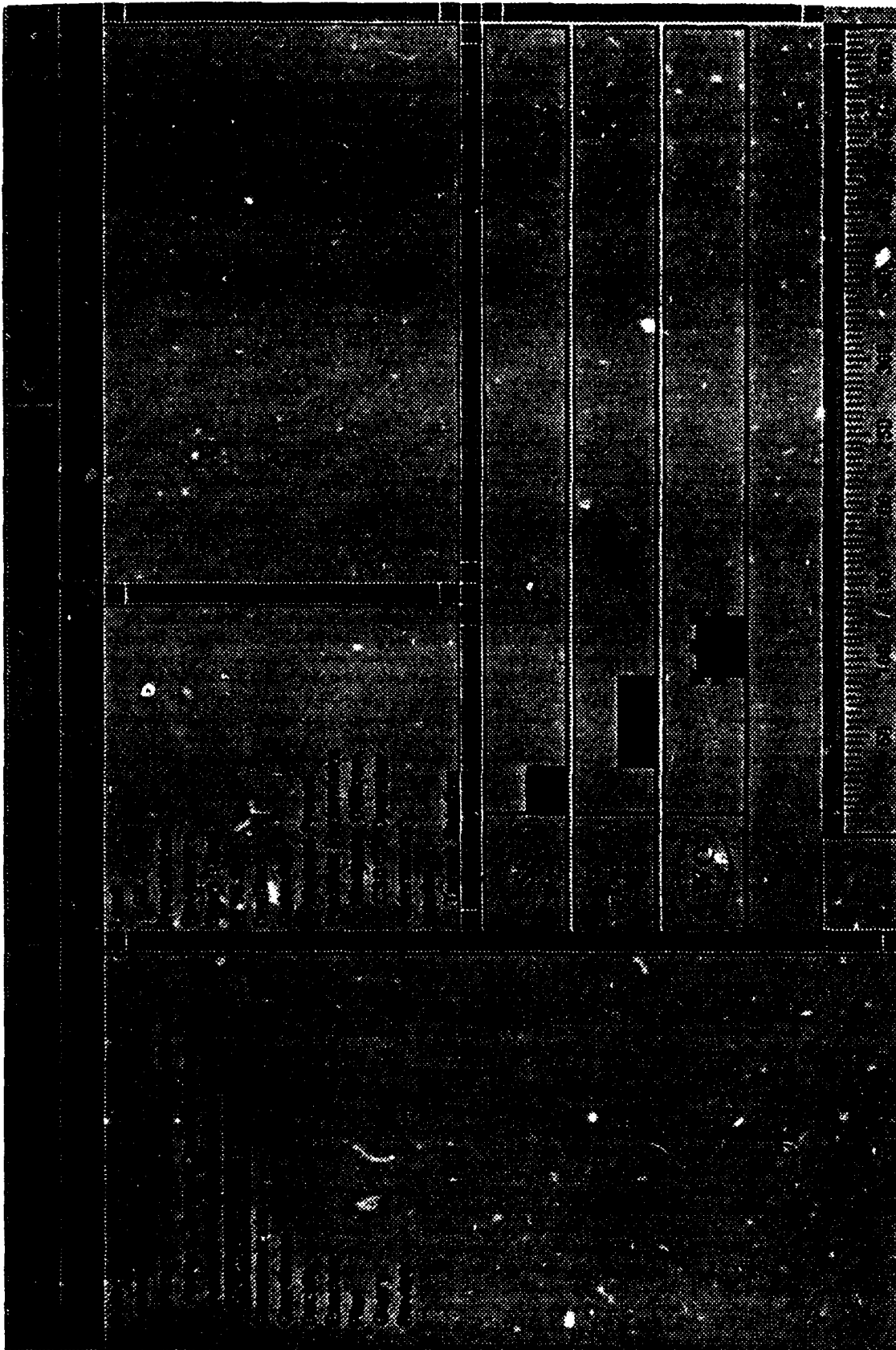


Figure 7
Script Builder

Tool Design

This tool design description focuses on the tool architectural features that are closely related to the tool operation. The KB architecture is the foundation of the SCOPE tool; therefore, any tool description should first begin with the KB.

Smalltalk Environment

Smalltalk was chosen as the implementation language because it is consistent with the object-oriented modeling methodology chosen for SCOPE and it provides an excellent environment for prototyping SCOPE. Smalltalk is the implementation language for the most recent version of SCOPE for the following reasons.

- It is consistent with the object-oriented modeling methodology.
- It provides a very cost-effective medium in which to develop the tool.
- It is important to this version of SCOPE to provide a powerful capability for the user to specify functions. If "C" or "C++" were chosen as the implementation language, providing user functions would have been much more difficult.
- It is easily ported to a wide variety of platforms. The Smalltalk design makes most platform differences transparent to SCOPE.

Knowledge Base Architecture

The SCOPE KB architecture is composed of several types of objects. The first is the KB itself.

Knowledge Base

The SCOPE KB is the basic repository of the objects in the user's model. The KB has been designed so that several KBs can exist in a Smalltalk image without interfering with each another. This was not always the case. Previous versions of SCOPE used the Smalltalk class hierarchy to support the object hierarchies in a SCOPE model. This meant that, for all practical purposes, there could only be one SCOPE model resident in the Smalltalk image at any given time. The KB provides basically two kinds of services: object accessing and record keeping (which is used for maintaining, loading, and saving the KB).

Knowledge Base Objects (KObject)

There are two kinds of KObjects: KObjectClass and KObjectInstance. KObjectClass objects are the basic KB objects. KObjectInstance objects are actually a kind of KObjectClass copy. KObjectClasses are available in the object hierarchies whereas KObjectInstances are not. The copies (instances) were designed to support the model simulation because, in the context of a simulation, a particular object may have to occur more than once with modified attribute values. The most practical way to accomplish this within the current SCOPE architecture is to use copies of the particular object. These copies are accessible only as an attribute value of some other object. Both the KObjectClass and the KObjectInstance have a name and an attribute dictionary. The KObjectInstance also has an identifier and a reference to the original KObjectClass. The identifier is needed in addition to its name because the KB loading and saving process requires globally unique identifiers. The KObjectClass contains references to its superclass and subclasses, a dictionary of its pictures and functions, and a list of its copies.

Attributes

Attributes have a name, value holder, specification dictionary, and dependents collection. The value holder keeps value(s), performs value inheritance, and, depending on the attribute's quantity specification, adds or replaces a value. The specification dictionary, which holds the attribute specifications, will probably be removed in a future version as the specifications become instance variables of the attribute. The dictionary was used initially because requirements were unclear and the dictionary offered more flexibility than instance variables. The dependents collection is a collection of objects, usually a tool component, that must be notified when the attribute value changes. This mechanism enables an object examiner to update its display when the attribute value is changed in a different window.

Attribute Specifications

An attribute specification is basically a passive object. Its responsibility is to hold its values. There are currently five specifications for an attribute; they are as follows.

Quantity	Specifies whether the attribute is single- or multivalued. The possible values of this specification are the Symbols "one" and "many." "one" is the default value.
InheritedValue	Specifies whether the value is inherited. The possible values of this specification are "true" and "false." "false" is the default value.
Visible	Specifies whether the attribute is visible in an object examiner. The possible values are "true" and "false." "true" is the default value.

Choices	Specifies the choices from which the user can select a value. The possible values of this specification are: any KObject class; any Attribute; and the Symbols "Numeric," "Text," "Any Knowledge Base Object," and "Attribute name." It is very likely that this list will expand in the future.
CrossLinks	Specifies the attribute within the value to which a cross-link will be made. Cross-links are stored as Associations, with the object being the association key and the attribute being the association value. The object also represents all of its subclasses.

Attribute Value Holders

The attribute value holders are subclasses of OrderedCollection. Originally, they were developed to prevent a user-defined function from inadvertently changing the value of an attribute without sending the request through the attribute. This was possible because the value of a multivalued attribute is a collection object which can be directly modified. The values are protected by creating a subclass and overriding the methods that would be most commonly used to add or remove a value. The overridden methods are then renamed so that normal operation is still provided. While this allows a user function to directly modify a value collection, doing so would be a deliberate action; therefore, presumably, it would be acceptable. There are now two types of value holders for multivalued attributes: ValueCollection and ValueSet. ValueCollection simply stores values as an ordered collection. ValueSet, on the other hand, stores values as a set so that elements are not repeated. A single-value attribute originally did not have a value holder; it simply stored the value. This created problems for routines that did not care if the attribute was single- or multivalued. The solution was to create another value holder called UniValueCollection. However, this created another problem: routines that know a particular attribute is single-valued do not want to treat the value as a collection. This was solved by creating a method, "realValue," that returns just the value if the holder is a UniValueCollection and returns the collection if the holder is multivalued. When a routine accesses an attribute value through the KObject, the "realValue" message is sent to the value holder. If a routine is to access all values regardless of value type, it should first get the attribute object and send it a "value" message, which will then return the value holder object.

Cross-Linking

One important feature of the SCOPE KB is the automatic cross-linking of objects and attributes. This cross-linking requires cooperation between the KObject, the attribute, and the value holder. The process for doing this is as follows.

1. A message to put a value into an attribute is received by the KObject.

2. The KBOject sends a message to the specified attribute requesting that it put the value into itself. The message also includes the KBOject as an argument.
3. The attribute obtains its cross-link specifications. If there is a specification that relates to the value, the value is sent a message telling it to cross-link to the KBOject.
4. The attribute tells its value holder to add the new value.
5. If the value holder is a UniValueCollection, its current value is replaced with the new value, otherwise the value is added to the value holder.
6. To conclude, the attribute next informs all its dependents that its value has changed.

Object Pictures

Refer to the subsection on Object Pictures under Modeling Concepts (page 8) for an introduction of the object pictures. An object picture is not an image but a structure composed of objects that know how to display themselves. There are three basic types of objects: primitive display object, selector, and Attribute Picture.

KBOjectPicture

The KBOjectPicture, the primary picture object, encapsulates the other objects. The KBOjectPicture is stored in a picture dictionary in the KBOjectClass which it was defined to represent. When a KBOject is requested to provide a picture of itself, the picture must be instantiated for the KBOject because the picture is most likely defined in an ancestor to the KBOject. This is done by first making a clone (an intelligent deep copy) of the picture, then connecting the attribute pictures to the actual associated attributes in the KBOject. The picture is now ready to be displayed. The picture displays itself by directing all its components to display themselves.

Attribute Pictures

As previously explained, there are three types of attribute pictures. When a KBOjectPicture is instantiated, all its attribute pictures are connected to their associated attributes for the object the picture is portraying. This "connection" process establishes a bidirectional reference between the attribute and the attribute picture. The attribute uses its reference to the attribute picture to notify it whenever the attribute value(s) is changed. The attribute picture uses its reference to the attribute to obtain the value(s) to be displayed. Before displaying its values, each type of attribute picture first obtains a KBOjectPicture from each value and stores it in a cache. Whenever the attribute picture

is notified of a change in the attribute's value(s), the cache is revalidated and the new state of the attribute is displayed.

Primitives

Primitives are simple display objects such as lines, rectangles, and ellipses. Also, several primitive objects may be grouped together to form a more complex graphic. This is done either to serve as an image for a selector or simply for ease of creating the picture.

Selectors

A selector is actually a wrapper around a primitive or group of primitives. The selector serves as a support object for the PictureViewer. Basically, when a user presses the mouse button on a selector, the selector informs the picture view what action to take. The action can be either to perform a user-defined function or to act as though an attribute or value had been selected for a subsequent operation.

Interfaces

The SCOPE modeling tool operates as a frame that holds various components; each of these components is itself a tool. These individual tools are further composed of common interface objects.

Frame

The SCOPE tool has been designed to operate within the confines of a window frame instead of having overlapping windows. The tool is designed using the Smalltalk Model-View-Controller (MVC) windowing paradigm, although it is a complex multilayered MVC. The SCOPE frame is an MVC that provides for KB creation, selection, loading, and saving. It also provides the means to swap in the different SCOPE tools. Each SCOPE tool is a separate MVC, some of which are further composed of more than one MVC.

Common Objects

Wherever possible the SCOPE tool design uses common interface objects. This has two major advantages. First, the interface operates consistently in different tools. Whenever an improvement is made in a common object, all the tools using the object receive the improvement. Second, development time and effort are saved. The important common interface objects are presented in the following paragraphs.

LabeledFormListView

LabeledFormListView is composed of a FormListView and SimpleMenuView. When originally designed, LabeledFormListView used a LabelView instead of a SimpleMenuView; however, this configuration is no longer used. The SimpleMenuView provides a pull-down menu, a button to indicate that a menu is present, and an associated label. The FormListView displays instances of ListElement or its subclasses. A ListElement is used because it encapsulates the value it represents; the value's display object, origin, and color; whether it is selectable; and other information used by the model to manage the list. There may be other techniques that do not consume as much memory, yet achieve close to the same functionality as a ListElement. However, the ListElement is probably the least complex technique.

ClassSelectionList

ClassSelectionList is used to display a class hierarchy selection list. It is actually a model that uses a LabeledFormListView to display the hierarchy. The ClassSelectionList provides the means to expand and contract branches in the displayed hierarchy to reduce the amount of unwanted information on the display.

KObjectInspector

The KObjectInspector is another model. It was designed to provide a consistent interface to a client SCOPE tool, such as the KnowledgeBaseManager, yet take on different forms. The KObjectInspector has three major components: itself, an objectViewer, and a choicesViewer. To provide a reasonable degree of reusability, the objectViewer and choicesViewer are also models. The objectViewer may actually be one of two different models. The first is the KObjectExaminer. It displays an object textually using a LabeledFormListView. It creates the list for the list view and serves as a translator between the list view and KObjectInspector. The second model is a PictureViewer. The PictureViewer uses a PictureView to display a picture of a KObject. The PictureView does most of the work. The PictureViewer basically serves as a translator between the PictureView and the KObjectInspector.

SimpleMenuView

The SimpleMenuView provides a client with a labeled pull-down menu. The availability or unavailability of a menu is indicated by the presence or absence of a button to the left of the label. As designed, this view serves two purposes: it provides both a title and a menu for the list views (where the title does not reflect the menu contents) and serves as a labeled pull-down menu within a menu bar.

Tools

The following paragraphs describe the basic software construction of the major SCOPE tools.

Knowledge Base Definer

The Knowledge Base Definer is composed of four LabeledFormListView and a ClassSelectionList. The definer is basically a straightforward design; therefore, it will not be discussed further.

Knowledge Base Manager

The Knowledge Base Manager is composed of five models and many views and controllers. The Knowledge Base Manager itself is a model called KBManager. Figure 8 shows the composition of the Knowledge Base Manager. In the figure, each box represents an object in the composition. The bold labeled lines leaving each box are the instance variables of the object. The class of each object is indicated by the normal text in each box. The italicized labels in some of the boxes indicate the type of object (i.e., model, view or controller). The dashed lines refer to the expanded diagram of the object. The gray line from the ChoicesViewer to the ClassSelectionList indicates that the ClassSelectionList is created by the ChoicesViewer but, once created, the viewer has no direct reference to it. The diagram also shows the amount the interface depends upon the LabeledFormListView. Furthermore, the KBOBJECTInspector and the ClassSelectionList are used elsewhere in SCOPE. One other variation of the KBOBJECTInspector substitutes a PictureViewer for the KBOBJECTExaminer. The PictureViewer uses a KBOBJECTPicture to display an object pictorially. To summarize, the overall structure of the manager is much more complex than a normal MVC. But, because the KBManager object delegates a large portion of its functionality to other objects, it is actually simpler than if the entire functionality were placed in it.

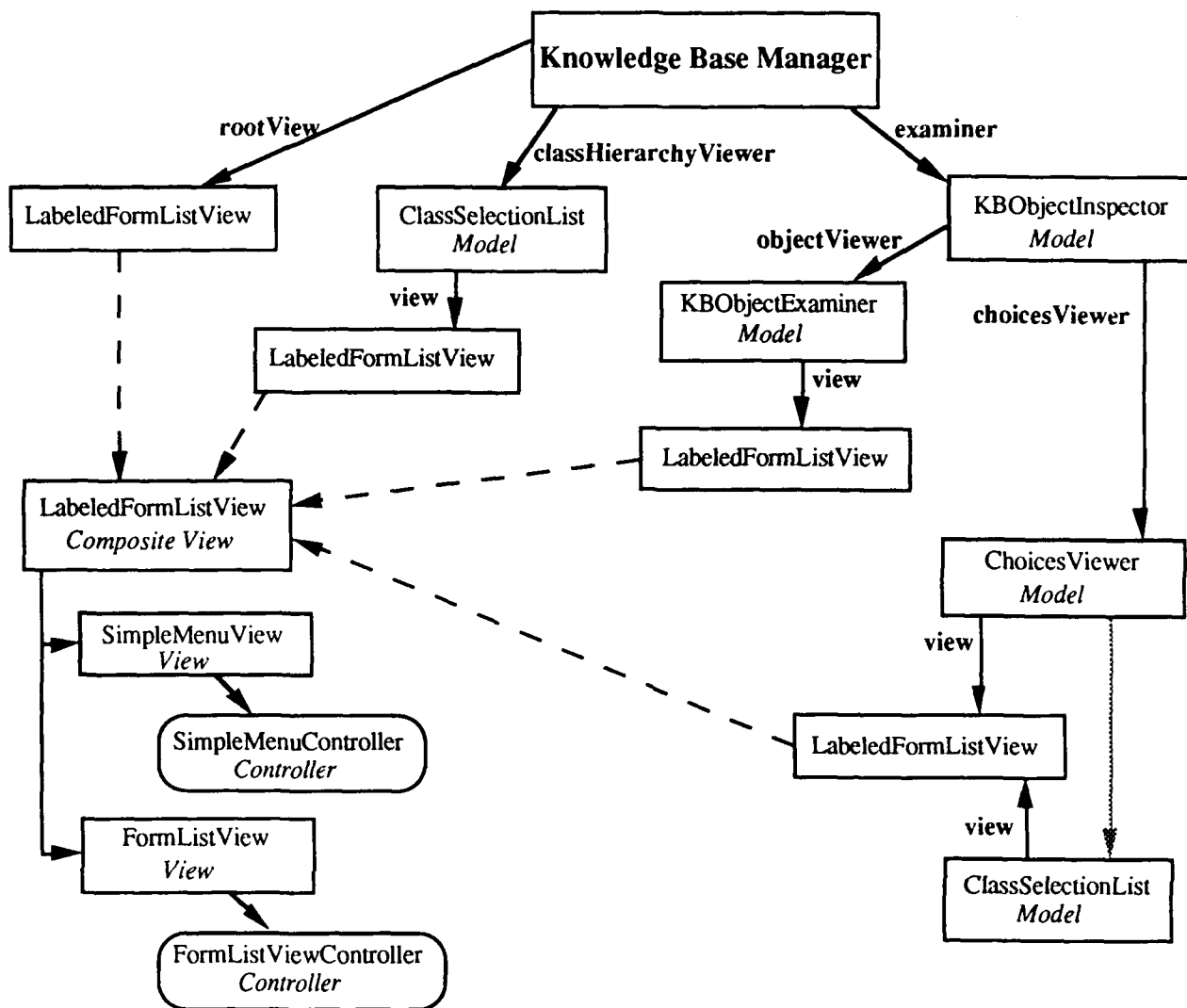


Figure 8
Knowledge Base Manager Composition

Script Builder

The Script Builder is composed of three panes. The first, a ClassSelectionList, is used to select a script, an object to place in a script, and the roots of a simulation script. The second pane, either a ScriptView or a GraphView, displays the script. A ScriptView is further composed of ResourceScriptViews, one ResourceScriptView for each resource in the script. The ResourceScriptView displays the resource icon and script for the resource. The GraphView is composed of a graph, graph composer, and node views. The third pane in the script builder is either a KBOBJECTInspector, which displays the selected activity object, or a ScriptSystemView, which displays the system containing the resources. Whenever an activity is selected and the ScriptSystemView is displayed, the communication lines required for the activity are drawn

between the resources. If a resource is selected in the ScriptSystemView, the ResourceScriptView for the resource is scrolled into the viewing area.

Picture Definer

The Picture Definer is composed of itself, four LabeledFormListView, a ClassSelectionList, a DrawingModel, a DrawingView, and a PullDownMenuBar. The PictureDefiner is a model object which is responsible for most of the user menu commands and controlling the different tool modes. The DrawingModel and DrawingView are responsible for creating, selecting, moving, and resizing the objects in a picture. The functionality of the DrawingModel was separated from the PictureDefiner because it was initially considered possible that a basic drawing capability would be included in another tool. Should the need arise, the current design would make it easy to incorporate the DrawingModel and DrawingView into a different tool to provide a basic drawing capability.

Simulation

The SCOPE simulation tool provides the capability to perform discrete-event simulation with the data contained in the KB. Simulation is closely tied to the type of model; therefore, the simulation implementation has been split into two components: a Smalltalk component and a KB component. The Smalltalk component is composed of a set of simulation-specific Smalltalk classes and functionality in the SCOPE Script Builder. The KB component is composed of a combination of KB classes and attributes and user-defined functions. The SCOPE tool is currently provided with a template KB containing a simulation model.

Smalltalk Simulation Component

The Smalltalk portion of the SCOPE simulation is implemented in several Smalltalk classes. These classes provide a set of methods that comprise a simulation language with which a modeler composes a simulation engine. The SCOPE concept of simulation places the simulation engine in user-defined functions. This allows the engine to be easily tailored to different system models. The Smalltalk simulation is provided by four primary types of objects: Simulation, SimulationObject, Resource, and DelayedEvent. The Simulation object manages the progression of time, the scheduling of SimulationObjects, and the coordination of SimulationObjects. A SimulationObject represents an activity or process that consumes resources for a period of time. A Resource represents a real-world resource, such as a person or a machine. A DelayedEvent represents an event that will occur at some point in the future. Each DelayedEvent is associated with a SimulationObject that is to be activated when the event occurs. Association of the

DelayedEvent to the SimulationObject is done through a Smalltalk Process. Each DelayedEvent contains a Semaphore that is used to initiate the process. Placing the execution of the SimulationObject within its own Smalltalk process allows the execution of the SimulationObject tasks to be suspended and later resumed should the SimulationObject perform a request that cannot be immediately satisfied.

The Simulation operates by keeping a queue of DelayedEvents and activating each in sequence. As each DelayedEvent is removed from the queue, the Simulation advances time to the time of the DelayedEvent. When the DelayedEvent is activated, it signals its semaphore which causes the process containing the SimulationObject to resume execution. The SimulationObject then performs its tasks. When a SimulationObject performs its tasks, it may request resources, do work, or initiate other SimulationObjects. The process of doing work really means that the resource is occupied for a specified length of time. When a SimulationObject is initiated, it is actually scheduled to begin at a specified time. This causes a DelayedEvent to be placed in the event queue of the active Simulation. When the event queue is empty, the simulation completes its processing and exits. The SimulationObject "tasks" method is actually a user-defined function of the KLObject that the SimulationObject represents. The "tasks" user-defined function also serves as the core of the model's simulation engine.

Currently, the Resource used by SCOPE is a ResourceProvider. It is capable of having a dynamic number of units available for consumption. Whenever a request is made for some number of units, the ResourceProvider will grant the request if enough units are available. If enough units are not available, the requestor is suspended and placed in a priority queue. Whenever more units are added to the ResourceProvider, the queue is checked for any waiting SimulationObjects. Future plans are to add the capability to have a resource type that will interrupt and suspend the SimulationObject that is using the resource when a higher priority request is made.

Knowledge Base Simulation Template

The simulation template operates on the premise that the KB contains Activity, Information, and Component classes. The relationships between these three classes are that activities require information inputs and generate information outputs. The activities are also performed by components for a given duration. The SCOPE simulation is divided into two phases: construction of a simulation script and execution of the script.

Script Construction

The SCOPE Script Builder provides the capability for a user to create scripts from the set of all possibilities contained in the KB using a graphical-tree-structured interface. The process begins by the user first selecting the activity(s) that form the root(s) of the sequence chain(s). After the activities have been placed in the script, the user selects an activity and directs the tool to place its information outputs in the script. Again, for each information item that was placed in the script, the user directs the tool to place the item's destination activities in the script. This process of expanding the activity outputs and the information destinations continues until the desired goal is reached. When the tool places an activity in the script, it first requests the user to resolve its performer. This occurs when the initial activities are placed in the script and when the information destinations are expanded. The user can also remove an activity or information item that has been placed in the script. Another attribute necessary for proper simulation of the activities is the duration attribute. The KB is currently defined so that the duration value defaults to ten. However, the user should change this to a correct value while constructing the script. Future plans are to allow the user to specify a statistical distribution instead of a single value.

Script Execution

Script execution begins by simulating the initial activities at time zero; future enhancements will permit the simulation to start later. For each activity, the discrete-event simulation will request the activity's performer, a component, to execute the activity for a specified duration. Currently, the components are simulated as simple resources that can execute only one activity at a time. If the simulation determines that an activity needs to begin execution and its performer is currently busy, the activity will be placed on a first-in-first-out queue for the component. Current plans are to provide for more complex resources that can perform multiple activities simultaneously, switch activities when interrupted, and prioritize the waiting queue. Performing multiple activities simultaneously would be accomplished by providing a "load" metric which would be allocated to the requesting activities until fully utilized, at which time an activity would then wait. After the component completes the activity, the simulation initiates all information output by the activity. Currently, the simulation does not address transfer delay or resource consumption required to perform the transfer. The new output information items then immediately initiate their destination activities. The simulation continues through the chain of activities and information in the script until they have all been exhausted. At the conclusion of the simulation, the start times of all activities will have been resolved so that a simple evaluation of the script may be performed.