Graphical Programming of Simulation Models in an Object-Oriented Environment

Mufit Ozden
Miami University, commons-admin@lib.muohio.edu

TECHNICAL REPORT: MU-SEAS-CSA-1988-005

Graphical Programming of Simulation Models In an Object-oriented Environment

Mufit Ozden

GRAPHICAL PROGRAMMING OF SIMULATION MODELS IN AN OBJECT-ORIENTED ENVIRONMENT

by

Mufit Ozden
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #88-005 July 1988

1988 USAF-UES SUMMER FACULTY RESEARCH PROGRAM/

GRADUATE STUDENT RESEARCH PROGRAM

Sponsored by the

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

Conducted by the

Universal Energy Systems, Inc.

FINAL REPORT

GRAPHICAL PROGRAMMING OF SIMULATION MODELS IN AN OBJECT-ORIENTED ENVIRONMENT

by

Mufit H. Ozden

Department Systems Analysis

MIAMI UNIVERSITY

Oxford, OH 45056

Phone: (513) 529 - 5937

Research Location:	AFHRL / LRL Wright Patterson AFB Dayton, OH 45433
USAF Researcher :	Douglas A. Popken
Date :	May 9 - July 15, 1988
Contract No:	F49620-87-R-0004

GRAPHICAL PROGRAMMING OF SIMULATION MODELS IN OBJECT-ORIENTED ENVIRONMENT

by

Mufit H. Ozden

ABSTRACT

Graphical programming has been used in conjunction with conventional simulation languages via block diagrams or activity networks. Its beneficial effects on programming and modeling in simulation have been accepted by everyone involved in these languages. However, none of these conventional techniques is truely interactive. Given the level of the current hardware and software technology, it is possible to design a very good graphical programming system which supports an interactive incremental programming style in specifications of simulation models. The benefit of such a visual system would go beyond the modeling phase of a simulation study and it might as well be realized in understanding the behavior of complex problems, in being a communication and training medium for the user and developers, and finally in presenting the simulation results.

In this study, the graphical programming methodology has been investigated from the perspective of object-oriented simulation. The truely interactive and graphical orientation of some of the object-oriented languages (e.g., Smalltalk-80) has opened up new avenues of research in this very important topic. Today, the nature of this type of research will be not whether it can be done but how the known techniques should be combined to yield the highest benefit.

Acknowledgements

I wish to thank the Air Force Systems Command and Air Force Office of Scientific Research for sponsorship of this research. I also would like to thank Universal Energy Systems for their administrative work and concern for arranging a suitable environment for the summer faculty.

I found my summer experience totally enriching and stimulating. Capt. Douglas Popken was most informative and helpful in defining the research problem area and in discussing the developing ideas. The concern of The Branch Chief Ms. Wendy Campbell is most sincerely appreciated. The family atmosphere at AFHRL/LR was most welcome, and was an enjoyable environment for my summer research for which I would like to extend my thanks and appreciation to the division commander Col. Donald C. Tetmeyer and the technical director Mr. Bert Cream.

I. INTRODUCTION

AFHRL/LRL is currently undertaking a study that will expand the capabilities of the Air Force in analyzing logistics support systems. As a part of the Productivity Improvements in Simulation Modeling (PRISM) project, the system currently under study is an Integrated Model Development Environment (IMDE) which will create a state-of-art development and test environment for the various simulation models of capability assessment. The IMDE will consist of an integrated set of hardware and software tools which support model specifications, model development, and model verification as well as specific function such as data retrieval and update. An important feature of such an environment is the user-friendly interface programs between the user and the simulation language. To this end, the development of a graphical programming facility will be evaluated for object-oriented simulation. The graphical elements should be manipulated with friendly hardware tools, such as a mouse or touch sensitive screen. The graphical models thus created will be translated into executable simulation programs automatically. A running simulation program should be observed in several views focused on different aspects of the simulated world.

II. OBJECTIVES OF THE RESEARCH EFFORT

The main goal of the summer research has been directed towards an exploratory investigation of graphical programming for object-oriented simulation. Graphical programming for simulation in the object-oriented environment is very new and has not been studied specifically in the research literature. At the current conceptual development stage of the PRISM project, it is considered to be the most suitable goal to study the general interface features of the object-oriented languages and the graphical programming in the conventional simulation languages and to recommend future research directions on a promising graphical methodology for the IMDE.

In the light of the above research goal, the following activities were identified for the summer research study:

i) Review of the related literature on object-oriented programming and graphical programming of conventional simulation languages.

ii) Evaluate Smalltalk-80 for graphical programming techniques.

iii) Formulate a graphical programming methodology that will be investigated further in a future research effort.

III. SIMULATION IN AN OBJECT-ORIENTED ENVIRONMENT

Although the object-oriented paradigm is a relatively new popular concept in software engineering, the idea of programming based on objects was first developed in Simula (Dahl and Nygaard, 1966), which is a simulation extension to Algol-60 language. The basic idea is to modularize the programming tasks on the basis of abstract or physical objects of the system. The data structures and methods associated with an object are encapsulated within the object so that the only way its data can be accessed or changed, or one of its methods can be invoked, is via sending an appropriate message to the object. Thus, programming in this paradigm involves creating the proper sequence of the messages to be passed among the objects as well as creating the object themselves. An object-oriented language comes with its own abstract classes of objects to provide a programming environment. The inheritance of the data structures and methods from the superior classes is one of the most important characteristics of the object-oriented environments. This provides a flexible programming environment that is organized in a hierarchical structure of object classes with reusable programs.

The object-oriented paradigm creates an excellent simulation environment in which the objects of the simulated world can interact with one another according to the predetermined behavior patterns in the closest way to their physical nature. This is a higher level of abstraction and more natural than it is possible with the procedure-oriented simulation languages, (Shannon, 1987). Any system that we try to simulate can be viewed as a collection of interacting objects . These objects can be categorized into classes of different kinds of objects. Objects created from each class will be similiar but not necessarily identical. Simulation in the object-oriented environment therefore entails decomposition of the problem into a set of object classes with simple interactions. Modularity is strongly supported because the internal implementation of these objects need not concern the modeler, (Unger, 1986).

An object-oriented simulation approach would contain three types of objects: domain independent abstract classes of objects, domain dependent general classes and application specific objects. Domain independent objects provide a simulation environment which describes the behavior of the generic model components common to all simulations, such as probability distributions that generates the random events. Domain dependent objects describe the general model components which correspond to the domain of application. They provide the templates for creation of instances specific to the application area. Application specific objects provide information about the particular configuration of the components, and their processes that are unique to a single application.

The object-oriented simulation programs make excellent use of modularization, extensibility, and exploratory style of programming as it is supported by the object-oriented environment, Stairnmand and Kreutzer, 1988. The recent developments in software engineering and hardware technology have enabled the interactive programming techniques to be employed on the work stations and some high-end personal computers, which form a base for exploratory and incremental style of programming. These are all essential elements of a rapid model development environment for complex simulation problems. With the advent of the parallel computers, speedups of several orders of magnitude should be possible for large simulations if they are constructed with concurrency in mind without hiding the parallelism inherent in problems. It has been suggested that the future simulation environments will be built on the object-oriented paradigm in which this concurrency is a natural extension, Jefferson, 1984 and Unger, 1987.

IV. DIFFERENT SIMULATION STRATEGIES

The most important characteristic of a simulation approach is the strategy employed in selecting the next event to be executed and the time management. For this purpose, Three different types of world views have been used to model simulation problems _ event scheduling, activity scanning and process-interaction. Each world view emphasizes a different type of locality _ the property when all the relevant parts of a program are found in the same place, Overstreet, 1987. Event scheduling emphasizes locality of time. Each event routine describes a collection of actions which may all occur in one instant. Activity scanning emphasizes locality of state. Each activity routine describes a collection of actions which will occur once the certain conditions are reached. These resulting actions may occur at different time points, but they must all occur. Process interaction emphasizes the locality of object. Each process routine describes all actions taken by one object.

The conventional simulation languages use one or allow a combination of the world views. In general, the simulation programs of US origin use either the event scheduling or process interaction whereas those of British origin tend to prefer the activity scanning view. However, it has been illustrated that each world view allows simpler model specifications for some problems, no one particular view is superior to the others, Overstreet 1987, Hooper 1986, and O'Keefe 1986. Some conventional simulation languages and their wold views are shown in Figure 1.

Event Scheduling	Process Interaction	Activity Scanning
GASP IV	GPSS/ H	AS
SIMSCRIPT II.5	SIMPSCRIPT II.5	CSL
SLAM II	SLAM II	ECSL
SIMAN	SIMAN	SIMON
	Q-GERT	ESP
	SIMULA	

Figure 1: Some simulation languages and their wold views.

According to Hooper 1986, the three world views including the process interaction view that is usually employed by the object-oriented simulations have the following characteristics:

Event Scheduling:

* efficient execution with relatively independent entities
* very flexible with respect to scope and standard features.
* considerable model development effort is required_ some event routines may have to be written in a general-purpose language

Process Interaction:

* model representation is close to problem
* straight forward model development and modification
* greatest support from the simulation executive
* maybe, inefficient execution time

Activity Scanning:

* efficient execution for highly dependent entities
* considerable work is done by simulation executive
* inefficient execution for relatively independent entities

Any world view of simulation can be formulated in an object-oriented environment. However, the process interaction approach is most compatible with the object orientation where behavior patterns can be written into the object definitions as another method. This is the approach which has been employed by object-oriented simulation languages, Demos (Birtwistle, 1979) and Smalltalk-80 (Goldberg and Robson, 1983).

V. SIMULATION IN THE OBJECT-ORIENTED ENVIRONMENT OF SMALLTALK-80

Here in this section, we would like to briefly describe how the Smalltalk-80 environment (Goldberg and Robson, 1983) supports discrete event simulation. Everything in Smalltalk is an object and every object is an instance of a class. Classes are arranged in a tree structure with each class having exactly one parent class. The root class of the tree structure is "Object". A subclass inherits all the variables and methods of parent class.

Simulation in Smalltalk is facilitated with the use of a small set (a dozen or so) of abstract object classes. The modeler uses some of these classes directly and/or may extend them through creating their subclasses. In a simulation study, a set of instances of these classes are formed to act according to the behavior patterns ascribed to the objects in the particular simulation situation as a combination of class methods inherited and the instance methods added during modeling. The abstract simulation classes can be grouped together into five categories: simulation executive; simulation objects; resources; statistics-gathering and monitoring classes ; basic support classes. These domain independent simulation classes and their relationship are shown in Figure 2.

The simulation executive has a single class, (Simulation) which creates and oversees all the objects in simulation as well as managing the time-ordered list of the future events over the simulated clock time. Usually, the modeler creates one instance of this class to manage a particular simulated world. This simulation executive creates resources and provides access to them. At initialization and during simulation run, it schedules the creation of instances of other simulation objects according to a probability distribution of interarrival times or at specified times.

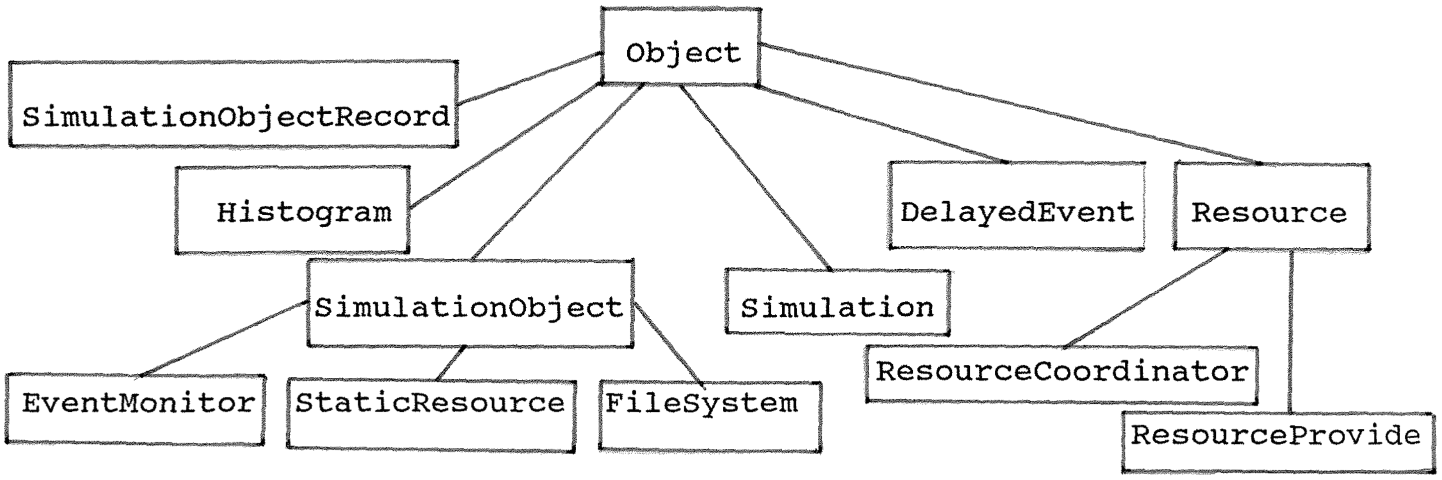


Figure 2: The domain independent simulation classes for the Smalltalk-80.

The class of the simulation objects (SimulationObject) represents a general kind of entity (object) that has a set of activities to perform during simulation. This class provides a skeleton which the modeler will use as a basic template to describe the objects which have some role to play in the simulation. A simulation object may be a temporary entity such as a 'customer' who must receive a list of services, or a permanent entity, such as a server who can perform some activities. These two types of simulation objects must coordinate the common activities through the next category of simulation classes. When a simulation object is denied the resource it has requested for some operation, its process is suspended until the resource becomes available

The third category of simulation classes is the resources (Resource) which have some general methods to manage and coordinate resources to be used in a simulation situation. Its subclasses are ResourceProvider and ResourceCoordinator. The class ResourceProvider manages both the resources that are created in fixed amounts during initialization phase by the simulation executive, and the resources that are produced and consumed during simulation . When a simulation object needs a resource to carry out its activity, it sends a message to the current simulation executive to that effect. The simulation executive in turn establishes the connection between the simulation object and the appropriate instance of the ResourceProvider which manages the type of the resource requested. The resource provider queues up the request with respect to the priority and within the same priority class it processes the requests on the basis of FCFS as the resource becomes available.

The subclass ResourceCoordinator provides synchorinization of the tasks among more than one simulation objects, e.g., among the customers and a server. It does this by means of keeping the tracts of the customers' requests and servers' availabilities in a queue. At any one time this queue will consist either entirely of servers or entirely of customers.

There are various ways of collecting statistics and monitoring the activities of the objects. Goldberg and Robson,1983 created four abstract classes for this purpose _ SimulationObjectRecord, Histogram, EventMonitor and Tally. These are the subclasses of the SimulationObject or create a file in order to store data. The type of the statistical data can be gathered is the entrance, exit and processing times of the simulation objects, the number of simulation objects that spend time within prespecified limits during simulation. In addition, the tallies of special events can be kept and any major events can be monitored completely.

The last fifth category of simulation classses is the remaining support classes which make possible the functions of the first four categories. The class DelayedEvent is used to delay the actions of the blocked objects as a package of suspended processes to a future time until the appropriate resource becomes available or synchronization among the simulation objects are possible. The various distribution classes are also defined to generate events from a set of probability distributions.

In summary, Smalltalk seems to provide an excellent support for discrete event simulation with its reusable classes and the graphics input and output capabilities. Smalltalk simulation environment provides the user with very powerful coding and debugging tools, leading to high productivity in writing and modifying simulation applications. This same conclusion has also been arrived by different researchers, (Knapp, 1987; Bezivin, 1987; Ulgen and Thomasma, 1986). A small simulation problem programmed in Smalltalk is given in Appendix B.

VI. VISUAL INTERACTIVE SIMULATION

A visual interactive simulation (VIS) is a term for a simulation which has features for specification of the model graphically, produces a dynamic display of the system model, and allows the user to interact with the running program, (O'Keefe, 1987; Hurrion, 1986). Thus, a VIS system typically provides facilities for:

i) Graphical Programming: where a model can be created visually on the screen in an interactive style.

ii) User Interaction: allows the user to interact with the running program. Interaction can be such that the simulation halts and requests information from the user, or the user stops the simulation at will and interacts with the program.

iii) Visual Display: portrays the dynamic behavior of the system on the screen. Figure 3 depicts the roles that these features play in a simulation study.

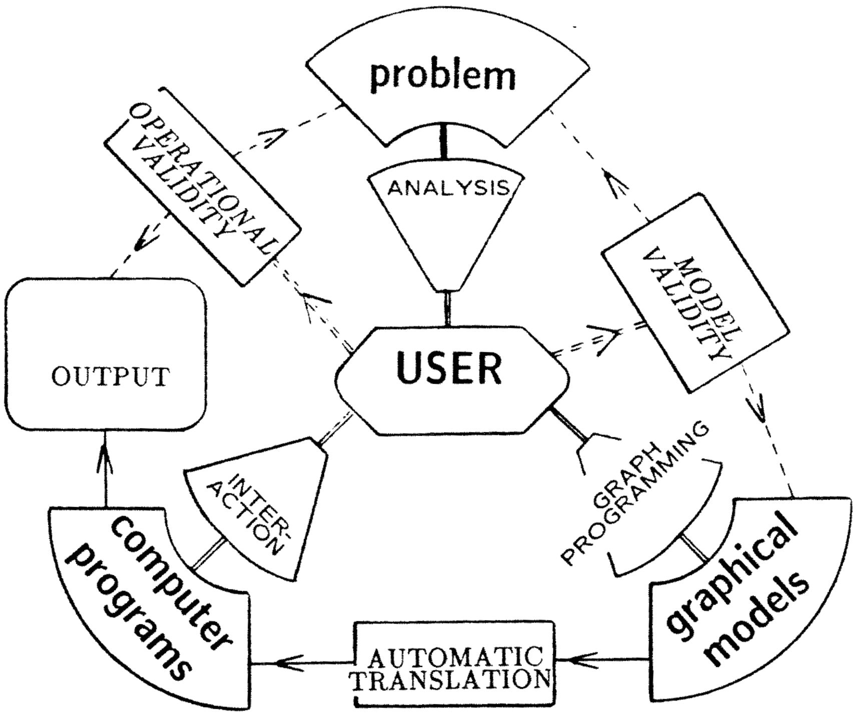


Figure 3: The role of Visual Interactive Simulation in a typical simulation study.

The General Benefits Attributed to VIS: The research attributes various observed benefits to VIS, (Hurrion, 1986; O'Keefe, 1987; Sargent, 1986; Ozden, 1988; Browne et al. 1986). The following are the most frequently cited benefits:

a) The graphical display becomes a communication medium that provides a common base between the developer and the user for discussion on development and experimentation. It is an excellent presentation medium for the results.

b) There is a lot to learn in understanding the behavior of a complex system by experimenting with the simulation model. Thus, VIS can be a teaching tool as well as an analysis tool.

c) The user can be incorporated into the model with the model determined interactions. In this way, the decisions that are too difficult to be handled by the model alone can be referred to the user.

d) The graphical techniques can be a useful means of detecting faults in coding and logic. The visual monitoring of the simulated behavior makes use of the powerful ability of the human brain to recognize logical and spacial relationships in detecting aberrant behavior.

e) VIS can increase the model validity and thereby the model credibility. Especially for the unobservable systems, where comparison between the system and model behavior is impossible, VIS may play an important role in building user confidence.

VII. GRAPHICAL PROGRAMMING

Modeling a simulation problem is a complex task demanding both the creative ability of the modeler and the support tools of the development environment . This step basically involves translation of the conceptual problem into a program which is executable by the computer. In simulation jargon, the modeler first needs to identify the temporary and permanent entities, and their data structures, as well as the behavior patterns that will closely resemble the particular problem setting under study, and then define appropriate representation forms for them in the simulation language.

Graphical programming is expected to meet the following objectives when used in a simulation environment:

a) Facilitate easy use of the simulation environment.

b) The graphical programming should itself be easy to use.

c) Modeler's productivity should be increased.

d) It should minimize programming error.

e) It should facilitate easy visualization of the conceptual problem.

These objectives are certainly not in conflict with each other, and an improvement of one may mean some betterment of the others.

In an object-oriented simulation where the domain independent-object and domain dependent-object classes exist in the programming environment, the modeling phase amounts to creating the application-specific classes and the instances of all relevant classes at the proper simulated time, and defining the behavior patterns (processes) of the objects in terms of the methods that already reside within the objects. Although existence of the object classes with the proper data structures and methods for simulation is a very convenient environment for modeling, it is still a challenging job to define the application specific classes and objects with the correct processes in terms of the programming language. It requires a good deal of working knowledge with the underlying language. A better interface is a graphical programming in which the user deals with the underlying language indirectly in an easier and more natural way. By means of graphs, icons, menus, windows and forms, a graphical programming interface can lead the user to the model specification with a sequence of visual and textual cues minimizing deviations from the correct translation of the conceptual problem.

In this respect, graphical programming is an unrivaled aid for human beings to observe the spatial and logical relationships among the simulation objects. Graphical programming may take various forms depending on the domain, and the hardware and software being used. The graph of spatial symbols, icons, menus and forms, and their combination is frequently used . We will review the graphical programming approaches developed and being considered in some simulation systems currently under development below.

Ideally, a graphical programming of simulation should be performed in such a natural and simple manner with (visual tools and perhaps aided with a natural language processor) that the modeler will be faced with a task compatible to the human cognitive process necessary to expain the conceptual problem to another human being. At the same time, an intelligent workstation should oversee this process to catch the bugs and inconsistencies in programming. Probably, a generalized modeling environment will be realized in the late 1990's. In the near future, we have to be content with the domain specific systems where we can achieve comparibly good results.

The current graphical programming approaches can be classified in three groups: i) Network and block diagrams; ii) Icons, menus, forms and windows; iii) Dialogs and tree structured menus. We are going to describe these systems in the following sections.

Network and block diagrams have been used as a modeling and communication aid in conventional simulation languages, such as GPSS, SLAM, SIMSCRIPT, and SIMAN etc. Here, the activities which each temporary entity (customer, transaction, or job) performs with the permanent entities (facilities, resources or stages) are described by the use of a sequence of blocks or a network of nodes. Each block or node represents a macro-code in the host language. In this way, computer programming is facilitated in chunks of codes taken at each step in addition to the visual help. Most of these simulation languages are oriented towards simulating queueing systems and they have wide application areas. They tend to have the view of the facilities in programming rather than the individual transactions. The transactions are usually dealt with in aggregate and probabilistic manner if possible, and the average facility performance (such as the percentage busy-time or the average waiting time at a facility etc.) is the main concern. Almost all of them now have an automatic translation feature which loads the network or block diagram as an executable code. Some even offer an interactive graphical programming feature for the restricted domains (e.g., flexible manufacturing systems) that can also display the animated view of the simulation, such as SIMAN/CINEMA and SLAM/TESS .

The number of allowable blocks or nodes may be quite large, (e.g., over 60 for GPSS). So the modeler's job is to find the right sequence of these macro elements with the correct parameter assignments. This is usually not a straight forward task and it may even require some external subroutines to be written in another programming language. SIMNET (Taha, 1987) reduces the number of these macro elements to a set of only four essential ones, and claims that this eliminates the need for external programming since it is possible to program the physically parallel processes in a parallel manner, and it is friendlier since one has to deal with only a few elements. This is a step in the right direction. But none of these simulation languages has a truly interactive graphical programming yet.

Icons, menus, forms and windows are the interface mode that has originated from the past artificial intelligence research because of the critical need for friendlier interfaces. This type of programming is the usual programming style for some modern languages, such as Smalltalk-80. But, it is better known as the MacIntosh interface mode by the general public. Especially with a pointing device such as a mouse, it is a lot friendler than key-board entry of data. Icons facilitate easy programming for the frequently used pieces of computer codes, with a single pointing action. Menus, on the other hand, offer alternative choice of operations on fixed menu items as a pull-down menu, or as a pop-up menu on the user request for a different course of action. Windows are usually used for parallel views and programming of the different parts of a computer code. Forms are for inputing standard information in a template form. Any combination of these interface modes are possible and widely used, e.g., icon/menu or window/form. This turns out to be especially a very convenient interface mode for the object-oriented languages in which flexible, and reusable parts of codes form the main program structure of the language,(as used in the paper by Cox and Hunt 1986, these are the Software-ICs just like the silicon chips in an electronic circuitry.)

In simulation, this type of programming style has been used in specific application areas, such as computer performance evaluation or manufacturing, (e.g., Melamed and Morris, 1985; Browne, et al. 1986; Sinclair et al. 1985; Duersch and Laymon, 1985; Stanwood et al. 1986).

Dialog-based programming is new in simulation. It has been developed as a part of a simulation environment, (Unger et al. 1984; Birtwistle and Luker, 1984). It originated from the idea that all simulation programs have a structured form of specifications no matter what the application area is. Therefore, a structured dialog with the user can be prepared beforehand to obtain the necessary information for any simulation model. In the dialog, the user is first asked for the process types and other global data and then requested to input the process details. Through the dialog, an intermediate representation of the model is built and is then run under an interpreter which may present different graphical views of the simulated world with icons and windows for verification purpose. Once the verification phase is completed the intermediate form is used to generate a compiled code for speed of execution. A different form of dialog style programming is also developed for simulation in a restricted area by Ingalls, 1986. Here, the dialog is based on a set of menus structured in the form of a tree. The user chooses a path of model specification from the root of the tree towards the lower branches pointing his choices from the menus.

VIII. GRAPHICAL PROGRAMMING FOR OBJECT-ORIENTED SIMULATION

It is important to understand that the software technology for graphical programming and object-oriented systems in general is fairly new and therefore the ideas and methodologies need to be tested in prototype systems before a full scale production system is attempted for developement. However, this orientation has many fruits to bear for the systems under development as well as for the future other systems. It has become apparent now that the software systems are the bottlenecks in modern technologies and the old paradigms do not comprise a solution to this problem. In this sense, graphical programming in the object-oriented simulation environment is future oriented and experimental in nature. In the light of all these facts, we propose a tentative graphical programming methodology below that needs to be examined with prototype problems close enough to the domain area. This methodology may have to be modified or expanded as discrepences are observed with these experiments. For the prototyping environment, a typical object-oriented language with nice and full features of the object-oriented enviroment should be chosen so that the transfer of the methodology into the ultimate language of the IMDE will be least painful.

In an object-oriented simulation, the global simulation data (such as, the number of temporary and permanent entities involved, and the total simulation duration, etc.), the topology (objects and their relationships), and the behavior of each object have to be defined during the model specification phase. Here in this section, we will describe a possible graphical programming for an object-oriented simulation environment in a rather speculative style. The exact form of a graphical programmming application would depend upon various factors including how a set of graphical and textual features will be selected on the basis of the objectives stated in the previous section as well as the hardware and software being used. Of course, a specific application domain can be supported better than a general purpose simulation environment. Here, we assume that the IMDE will primarily be a rectricted simulation domain of capability assessment of logistics support systems.

The minimum requirements of a graphical programming for an object-oriented simulation system should cover the following:

a) A graphical programming editor to create new object classes and graphical elements (icons, menus and forms etc.) to be stored in the simulation "library" (data base for persistent objects, and to edit the old objects from the library and the simulation applications saved in the form of graphical models. It should have a "dictionary" access to this library of objects. The dictionary could be for most part iconic and organized in some hierarchical fashion for easy access.

b) An interpreter to translate the graphical models to be created with the editor into the computer executable form .

c) View builder: the style of model development in this environment will be mostly exploratory and incremental. The objects and their relationships as created in part (a) should be able to be viewed graphically in a static manner (e.g., activity cycle diagrams). When the programs are run, the simulation with different object views should be observed possibly dynamically to facilitate verification of the model created so far. If the programs need to be modified the graphical models stored in a file should be reloaded.

A typical scenario of graphical programming in the object-oriented simulation will be depicted below. Since the global simulation data input will be conducted in a standard way for all simulation applications , this phase can be facilitated filling standard forms interactively on the screen. If the simulation has already been created or it is going to be some modified version of an old simulation program, the graphical model or the compiled code of the simulation will be reloaded from a file.

Since it is assumed that the simulation environment will have a restricted domain of application, most of the classes and subclasses of the objects needed for the particular application will be found in the objects library that can be accessed with the dictionary. Thus, when one identifies a temporary entity that will take place in simulation ,say a particular type of airplane, the icon representing that object is looked up in the dictionary of temporary entities, and the appropriate button of the mouse is clicked on the icon. This action will load the object into the graphical programming environment from the data base and at the same time on the screen the user sees a form that prompts him to fill the related information, such as the number of airplanes of the selected type, how they will enter the simulated world (e.g., type of the interarrival distributions) and the number and types of processes in which this particular object will be involved, etc. If some particular information about an item of the object refers to some existing data in the environment, the help can be obtained by means of a menu which offers alternatives and in turn when one is selected asks for more detailed information on that particular item, e.g., the distribution and then its parameters.

All the elementary operations necessary to define a process that an object may perform will already have been defined at the creation of the object. During the process definition phase of an object in graphical programming, all the methods that may take part in process definition and the resources defined so far can be presented as menus. When a method is selected from this menu, the proper parameter settings may be asked automatically. If a new type of resource is needed from the resource dictionary it is loaded into the simulation world and an account of used resources is kept. Later on, the necessary parameter definitions of these resources will be asked from the user automatically. The simulation world thus defined is converted into an internal representation form.

The behavior pattern (processes) of object cannot be defined unless the object always behaves in the same manner in which case this can also be incorporated in the library definition. For example, the mission of an airplane will most probably change from application to application. However, as noted elsewhere (Birtwistle and Luker, 1984), the structure of behavior patterns of objects is regular and is composed of a set of processes, each of which in turn demands a certain amount of a resource, holds it for some time and then returns some amount of the same resource. In fact this well structured behavior forms the basis for the graphical representation known as the activity cyle diagrams, Birtswistle, 1979.

The view builder of the graphical programming should be able to display different views of the simulated world. For example, an activity cycle diagram for each primary object could display the processes and its relationship with the resources graphically on the screen ; or a resource view could present all the temporary entities that use a particular resource graphically on the screen. Such orthoganal views of a simulation application may reveal a lot for verification purpose. At the same time, these views may be used to observe the dynamic behavior of the objects. For example, the resource view may show all the objects that are using different types of resources and the objects that are blocked due to lack of resources at each discrete time as animated graphics. Once the simulation modeling is completed, the intermediate form (graphical model) would be saved and be compiled for experimental runs.

In a real-life simulation case, there may be thousands of objects that need to be defined for the simulation world. But most of these objects will probably remain the same from one application to another and will not get involved with the other objects in complex interactions. Therefore, it may be very helpful to modify a copy of the closest simulation application stored on a file rather than creating it from scratch. This modifiability (reuseability) of the old programs is another asset of the object-oriented paradigm which will affect programmer's productivity a great deal.

IX. RECOMMENDATIONS

As a result of this summer research on graphical programming in object-oriented simulation, the following main points of opinion are formed:

a) Interactive graphical programming should be an integral part of the model development environment and it should support incremental programming, and load the graphical model automatically for execution. The object-oriented environment is compatible with and fully supportive of such a feature.

b) In the graphical programming, the model specification effort should be guided with icons, menus, forms and windows. There are many ways of combining these visual aids. The best design for a particular domain can be achieved through protototyping small problems in the domain.

c) Another important component of the graphical programming is a facility which will display different graphical views of the simulated "world". This will especially be helpful for verification purpose as well as reviewing typical applications saved on the files.

REFERENCES

Bezivin, J., "Timelock: A Concurrent Simulation Technique and its Description in Smalltalk-80,Conf. of 1987 W. Simul.Conf.,1987.

Birtwistle, G.M.,Discrete Event Modelling on Simula, MacMillan, 1979.

Birtwistle, G. M. and P. Luker, "Dialogs for Simulation," Proc. of Conf. on Simul. in S. Typed Lang., San Diego, 1984.

Birtwistle, G. M., B. Wyvill,D. Levinson and R. Neal, "Visualising a Simulation Using Animated Pictures," Proc. of Conf. on Simul. in S. Typed Lang., SanDiego, 1984.

Browne, J. C., J. E. Dutton and D. M. Neuse, "Introduction to Graphical programming of Simulation Models," Proc. of 1986 S. Simul. Comp. Conf., 1986.

Cox, B. and B. Hunt,"Objects,Icons, and Software-ICs," Byte, 1986.

Dahl, O. J. and K. Nygaard, "SIMULA: An Algol Based Simulation Language," Comm. of ACM, Vol. 9, 1966.

Duersch, R. R. and M. A. Laymon, "Programming-free Graphic Factory Simulation with GEFMS/PC, "Proc. of 1985 W. Simul. Conf. 1985.

Goldberg, A and D. Robson, Smalltalk-80: The Language and Its Implementation, Addison Wesley, Reading , Mass, 1983.

Hooper, J. W., "Srategy-related Characteristics of Discrete-event Languages and Models," Simulation, Vol. 46, 1986.

Hurrion, R. D., "Visual Interactive Modelling," EJOR,Vol.23,1986.

Ingalls, R. G., "Automatic Model Generation," Proc. of 1986 W. Simul. Conf., 1986.

Jefferson, D., "Future Directions in Simulation at the Conference on Simulation in Strongly Typed Languages (Panel)," Proc. of Conf. on Simul. in S. Typed Lang., SanDiego, 1984.

Knapp, V.,"The Smalltalk Simulation Environment," Proc. of 1987 W. Simul. Conf., 1987.

Mathewson, S. C., "Simulation Program Generators: Code and Animation on a P.C.," J. of Opl. Res. Society, Vol.36, 1985.

Melamed, B. and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation,"Computer, IEEE, Vol.18, 1985.

O'Keefe, R. M., "What is Visual Interactive Simulation ? " Proc. of 1987 W. Simul. Conf., 1987.

Overstreet, C. M., "Using Graphs to Translate Between World Views," Proc. of 1987 W. Simul. Conf., 1987.

Ozden, M., "A Simulation Study of Multi-load-carrying Automated Guided Vehicles in a Flexible Manufacturing System," Int. J. Prod. Res., Vol.26, 1988.

Pegden, C. D., Introduction to SIMAN, Systems Modelling Corp.,1985.

Popken, D. A., "The Productivity Improvements in Simulation Modelling (PRISM) Project: Concepts and Motivation," Draft Report, AFHLR / LRL, WPAFB, 1988.

Pritsker, A. A. B., Introduction to Simulation and Slam II, Halsted Press, 1984.

Rothenberg, J., "Object-oriented Simulation: Where Do We Go From Here ?", Proc. of 1986 W. Simul. Conf., 1986.

Sargent, R. G., "The Use of Graphical Models in Model Validation," Proc. of 1986 W. Simul. Conf., 1986.

Schriber, T. J., Simulation Using GPSS, J-Wiley and Sons,NY, 1974.

Shannon, R. E., "Models and Artificial Intelligence," Proc. of 1987 Winter Simul. Conf. , 1987.

Sinclair, J. B., K. A. Doshi and S. Madala, "Computer Performance Evaluation with GIST: a Tool for Specifying Extended Queueing Network Models," Proc. of 1985 W. Simul. Conf., 1985.

Stairmand, M. C. and W. Kreutzer, "POSE: a Process-oriented Simulation Environment Embedded in Scheme," Simulation, Vol. 50, 1988.

Stanwood, K. L., L. N. Waller and G. C. Marr, "System Iconic Modeling Facility," Proc. of 1986 W. Simul. Conf. 1986.

Stefik, M. and D. G. Bobrow, "Object-oriented Programming: Themes and Variations," The AI Magazine, Vol. 6, 1986.

Taha, H.A.,"SIMNET Simulation Language,"Proc.of 1987 W.Simul.Conf.,1987.

Thomasma, T. and O. M. Ulgen, "Modeling of a Manufacturing Cell Using a Graphical Simulation System Based on Smalltalk-80, Proc. of 1987 W. Simul. Conf., 1987.

Ulgen, O. M. and T. Thomasma, "Simulation Modelling in an Object-oriented Environment Using Smalltalk-80," Proc. of 1986 W. Simul. Conf., 1986.

Unger, B., G. Birtwistle, J. Cleary, D. Hill, G. Lomow, R Neal, M. Peterson, I. Witten, and B. Wyvill, " JADE: A Simulation and Software Prototyping Environment," Proc. of Conf. on Sim. in S. Typed Lang., SanDiego, 1984.

Unger, B. W., "Simulation Environments & Parallelism," in Simulation Environments of the 1990's (Panel), Proc. of 1987 W. Simul. Conf. , 1987.

APPENDIX A

THREE WORLD VIEWS OF SIMULATION AND THEIR RELATIONS

In order to clarify the distinction between the different word views assumed by simulation programs, we demonstrate them with an example problem . This explanation is basically adopted from Overstreet's paper (Overstreet, 1987): the problem is somewhat simplified and some mistakes in the original paper are corrected and the pseudocode is also modified. Lets consider the following simple production problem.

Problem: Parts arrive for processing by a single machine. Processing is sometimes interupted due to machine failure. The parts that have their processing interupted will be finished once the machine resumes operation. Interarrival times, machine processing times and inter-machine-failure times are randomly distributed. Estimate the amount of time required to process a fixed number of parts.

The formulation of this problem into the models of the three world views are achieved in the same three steps: i) identification of valid model actions, ii) grouping these into event, activity, or processes (depending on the world views), and iii) simplification of the individual event, activity or process descriptions.

After analysis of the problem, for the first step of formulation, the following actions groups can be identified:

Step i)

conditions (events)	actions

initialization (IN)	Create(parts)
	Create(machine)
	Initialize(statistics)
	Setup(partArrival)
	Setup(machineFailure)

termination (TE) #finishedParts = N	Report(statistics) Stop
partArrival (PA)	Setup(partArrival) Add+1(queue)

beginService (BS)	Add-1(queue)
	Setup(endService)

endService (ES)	Add+1(#finishedParts)

machineFailure (MF)	DetermineStatus(machine)
	busyFailure= BF
	or idleFailure= IF

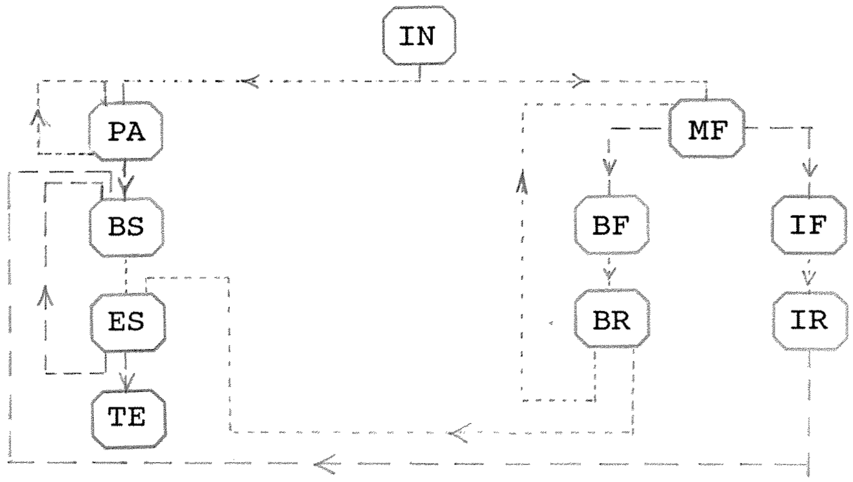
busyFailure (BF)	remainTime <-- endService - clock
	Cancel(endService)
	Setup(endBusyRepair)

idleFailure (IF)	Setup(endIdleRepair)

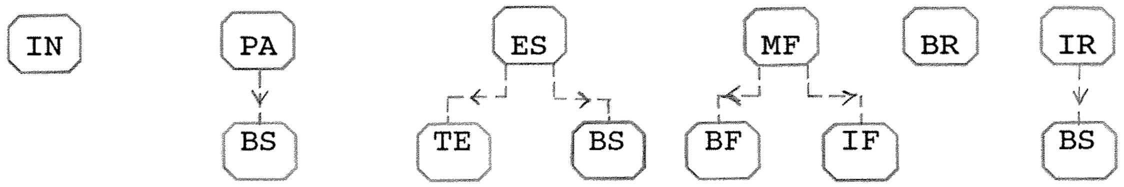
endBusyRepair (BR)	Setup(machineFailure)
	Setup(endService)

endIdleRepair (IR)	Setup(machineFailure)

Step ii) Grouping of the events into the appropriate event, action or processes can be facilitated with the event-incidence-diagram. The nodes represent events and there are two types of links conecting these nodes: "...." denotes the ability of one event to cause the occurence of another at a future instance; "----" denotes the ability of one event to cause the intantaneous occurrence of another.



a) To create the event scheduling world view, identify all the events can occur at the same time point_ the events that are linked with the dashed lines in the event-instance-diagram.



Pseudo-code for the event scheduling view: as an event becomes current in the future event list that is maintained by the simulation executive, perform the associated operations shown below.

```
initialization (IN)
_____________________
Create( part)
Create( machine)

Initialize( statistics)
Setup( PA)
Setup( MF)
```

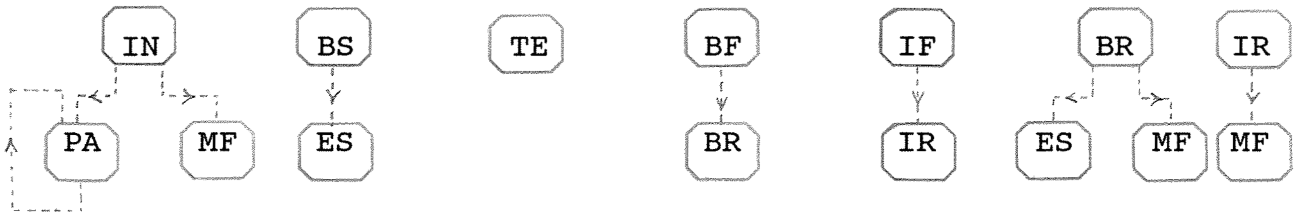
```
partArrival (PA)
__________________
  Setup( PA)
If machine(idle),Setup( ES) and
       set(machine, busy) ;
Else  Add+1( queue)
```

```
endService (ES)
_________________
Add+1( #finishedPart)
If #finishedPart   =   N,
   Report(statistics),Stop;
Else if queue > 0,
```

```
machineFailure (MF)
_______________________
If machine(busy),
   remainTime <-- endService
              - clock,
   Cancel( ES),
          Add-1( queue), Setup( ES);          Setup( BR);
          Else set(machine, idle).          Else  Setup( BR).


     endBusyRepair (BR)                     endIdleRepair (IR)
     __________________                     ______________________
     Setup( ES, repairTime),                Setup( MF)
     Setup( MF)                            If queue > 0, Add-1(queue)
                                                 Setup( ES).
```

b) To create the activity scanning world view, identify all the activities that can occur some time later once a specified event has occurred : the activities that are represented with the dotted lines in the event-instance-diagram. Scan these activities for the their occurrance times in the future event list, as the clock is updated.



Pseudo-code of the activity scanning world view will be:

```
 initialization activity
_________________________
     Create( part)
     Create( machine)
     Initialize( statistics)
     ConcurrentWait( ES)
     ConcurrentWait( MF)

 service activity (condition: queue > 0, machine(idle) )
________________________________________________________
     Add-1( queue)
     Wait( ES)
     Add+1( #finishedPart)

 termination activity (condition: #finishedPart = N)
____________________________________________________
     Report(statistics)
     Stop

 busy machine failing activity (condition: (MF), machine( busy) )
_________________________________________________________________
     remainingTime <-- endService - clock)
     Cancel( ES)
```

```
    Wait( BR)
```

idle machine failing activity (condition: (MF), machine(idle))

```
    Wait( IR)
```

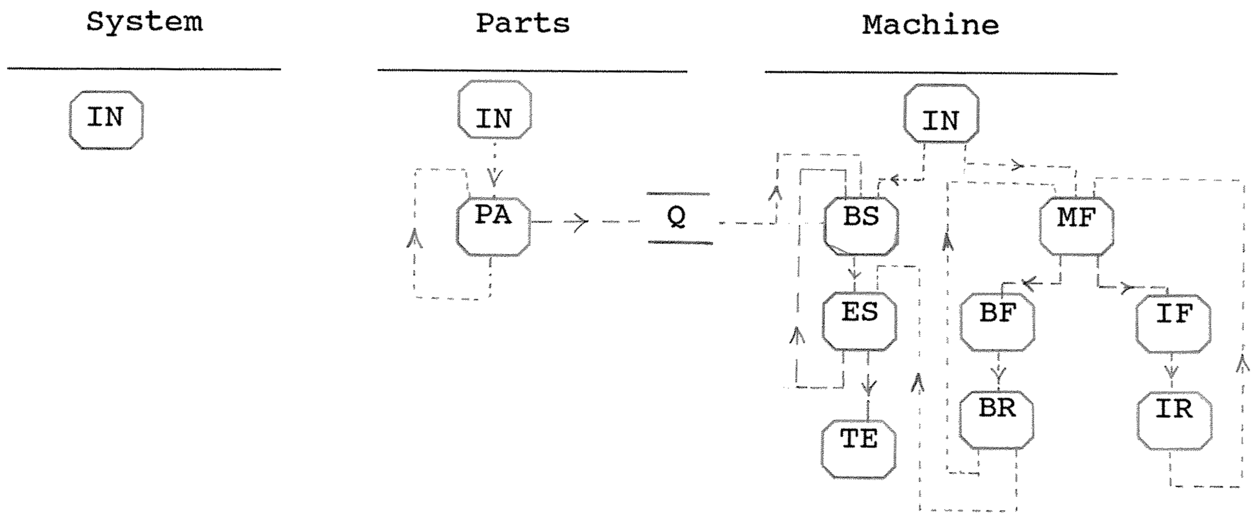
busy machine repairing (condition: (BR))

```
    ConcurrentWait(MF)
    ConcurrentWait( ES, remainingTime)
    Add+1( #finishedPart)
```

idle machine repairing (condition: (IR))

```
    Wait( MF)
```

c) To create the process interaction world view of the problem, identify the objects which are undergoing processes: system, parts and machine.



System Process

```
Create(part)
Create( machine)
Initialize( countStats)
```

Parts Process

```
Initialize( partStats)
Loop
    Add+1( queue)
    Hold(interarrivalTime)
End Loop
```

Machine Process 1

```
Initialize( machineStats)
Hold( machineUptime)
Loop
     If machine( busy),
          remainingTime <-- endService - clock,
          Passivate( machineProcess_2),
          Hold( repairTime)
          Activate( machineProcess-2)
     Else
          Hold( repairTime),
          Activate( machineProcess-2)
     End Else,
     Hold( machineUptime)
End Loop.
```

Machine Process 2

```
Loop
     WaitUntill( queue > 0 and machine( idle) )
     Set(machine, busy)
     If Passivated
          Hold( remainingTime)
     Else Add-1( queue)
          Hold( endProcessingTime)
          Set( machine, idle)
     End Else
     Add+1( #finishedPart)
     If ( #finishedPart = N)
          Report( statistics)
          Stop
End Loop
```

APPENDIX B

A SIMULATION EXAMPLE IN SMALLTALK-80

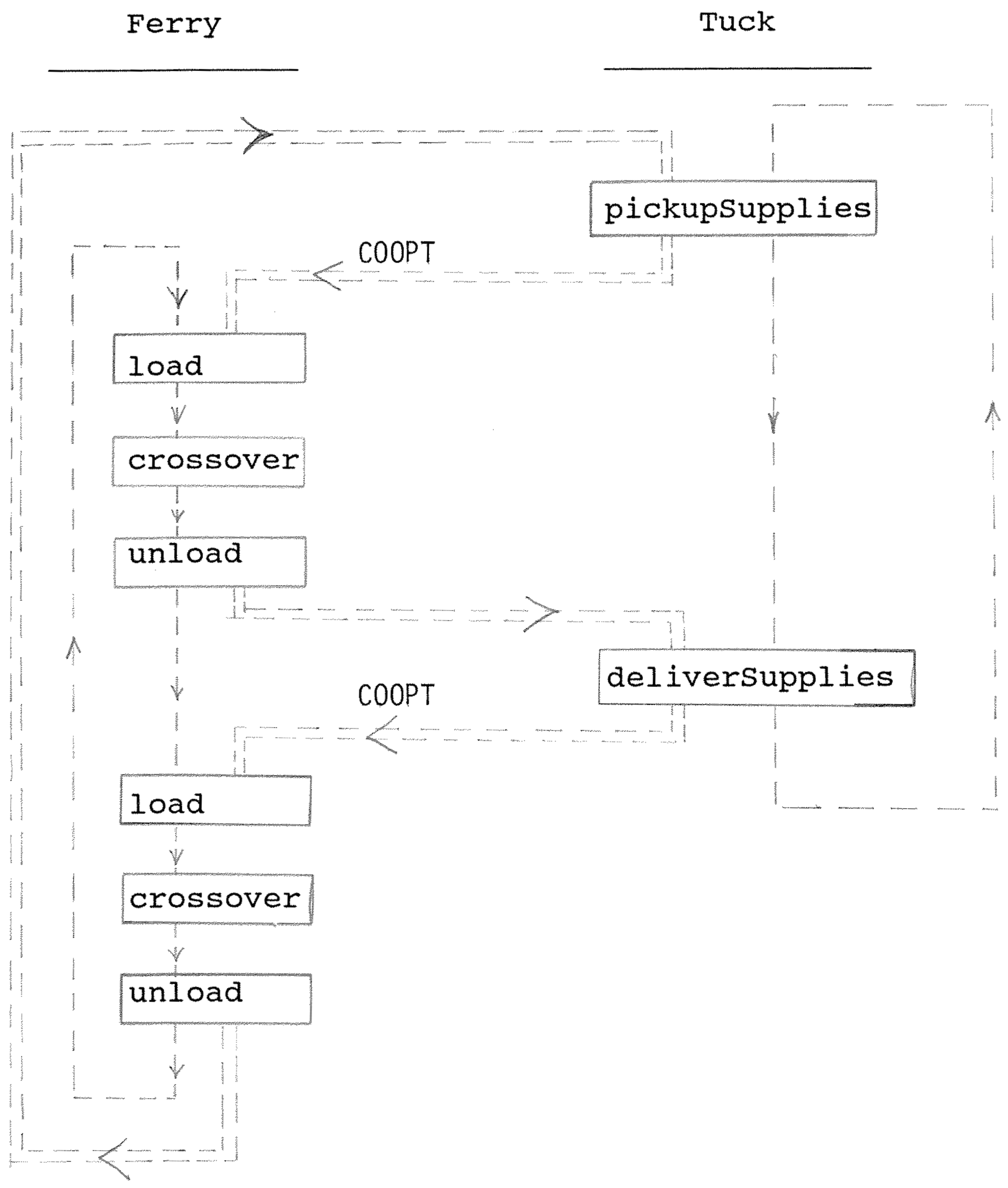
Here, we will prepare the classes and and subclasses necessary to simulate a problem in Smalltalk-80 in order to illustrate what is specifically involved in simulation in this powerful environment. Also, this illustration will shed some light on the question of how to set a simulation model in an object-oriented environment, which will be the subject of the following section.

We consider a ferry shuttling problem between an island and a mainland, which was given as an example originally in Birtwistle's Demos (Birtwistle, 1979) and also in the Smalltalk-80 book (Goldberg an Robson, 1983).

Problem: A ferry operates 1000 minutes and stops at one of the docking locations. A truck goes from the mainland to the island in oreder to make deliveries, returns to the mainland to get more supplies, and goes to the island again. The ferry can carry as many as four cars in addition to the truck, but the ferry will not cross across unless there is the truck to carry. The crossing takes approximately 8 minutes with the standard deviation of 0.5 minutes.

This example requires a coordination of SimulationObjects representing the ferry and the truck; each has its own sequence of tasks, but the truck cannot do its tasks without the assistance of the ferry and the ferry has no tasks to do in the absence of the truck. On the hand, cars will be treated as intances of the StaticResource class.

Before we start defining the object classes for the fery problem, it will be useful to prepare the activity diagram of these objects involved in the problem. The activity diagram represents the processes of two objects and their interactions .



First, we define the particular simulation executive, FerrySimulation which will manage the ferry problem as a subclass of the abstract class Simulation.

```
class name                FerrySimulation
super class               Simulation

instance methods
 initialization
     defineArrivalSchedule
          self scheduleArrivalOf: Truck new at: 0.
          self scheduleArrivalOf: Ferry new at: 0.
     defineResources
          self coordinate: 'TruckCrossing'
```

Note that all the necessary methods such as scheduleArrivalOf coordinate and many others need not be defined here since they can be inherited from the super class Simulation which resides in the simulation environment of Smalltalk-80. 'TruckCrossing' is formulated as some sort of resource that is produced by the truck object and aquired by the ferry object in order to complete their common crassing tasks. In this way, the coordination between the processes of the two objects is formulated in this problem.

```
class name                Ferry
super class               SimulationObject
instance methods
 simulation control
     tasks
          | truckRequest |
          [ ActiveSimulation time > 1000] whileFalse:
               [truckRequest <-- self acquireResource: 'TruckCrossing'.
               self load.
               self crossOver.
               self unload.
               self resume: truckRequest]
     load
          self holdFor: 5.0
     unload
          self holdFor: 3.0
     crossOver
          self holdFor: (Normal mean: 8 deviation: 0.5) next
```

Note here again that all we have to define the application specific methods (load, unload, crossOver and the sequence of tasks , i.e., the process); all other methods (acquireResource, holdFor, resume etc. are inherited from the super class), and Normal is name of a class in SmallTalk which generates events from a specified normal distribution.

Now, we have to define the third object, Truck as a subclass of the abstract class, SimulationObject.

```
class name                    Truck
super class                   SimulationObject
instance methods
 simulation control
     tasks
          [ true ] whileTrue:
               [ self produceResource: 'TruckCrossing'.
               self deliverSupplies.
               self produceResource: 'TruckCrossing'.
               self pickUpSupplies ]
     deliverSupplies
          self holdFor: ( Uniform from: 15 to: 30) next
     pickUpSupplies
          self holdFor: ( Uniform from: 30 to: 45) next
```

Here again, we define the sequence of tasks and the associated methods of the Truck specific to the problem at hand. The method, holdFor will be inherited from the super class and the distribution object (Uniform) which knows the method next, generates the process times with the associated parameters from the uniform distribution.