Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

11-22-1991

A distributed object-oriented discrete event-driven simulation environment-DODESE

Min Chen
Florida International University

DOI: 10.25148/etd.FI14060172
Follow this and additional works at: https://digitalcommons.fiu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Chen, Min, "A distributed object-oriented discrete event-driven simulation environment-DODESE" (1991). *FIU Electronic Theses and Dissertations*. 2140.
https://digitalcommons.fiu.edu/etd/2140

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

ABSTRACT OF THE THESIS

A Distributed Object-Oriented Discrete Event-Driven
Simulation Environment–DODESE

by

Min Chen
Florida International University, 1991
Miami, Florida
Professor Raimund K. Ege, Major Professor

A new distributed object-oriented discrete event-driven simulation environment, DODESE, is developed to provide a common framework for simulation model design and implementation. The DODESE can be used to define a simulation including all the simulation objects participating in the simulation while the execution of the simulation can be interactively monitored on DODESE. The DODESE system has combined the strengths of both object-oriented paradigms and data base technology to make computer simulation more powerful and has achieved the goals of object-orientation, distribution, reusability, maintainability and extensibility. The system runs on two Sun workstations concurrently connected by an Ethernet. One of the workstations performs the simulation tasks while the other workstation displays the status of the simulation interactively. Both workstations communicate through the GemStone data base, thus a mechanism is designed for synchronization and concurrency control. The DODESE is implemented using OPAL, GemStone's data definition and manipulation language, C and Xlib.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A Distributed Object-Oriented Discrete Event-Driven Simulation Environment–DODESE

A thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science
in Computer Science

by

Min Chen

1991

To Professors: Raimund K. Ege, John C. Comfort, Masoud Milani

This thesis, having been approved in respect to form and mechanical execution, is referred to you for judgement upon its substantial merit.

Dean Arthur W. Herriott
College of Arts and Science

The thesis of Min Chen is approved.

John C. Comfort

Masoud T. Milani

Raimund K. Ege, Major Professor

Date of Examination: November 22, 1991

Dean Richard Campbell
Division of Graduate Studies

Florida International University, 1991

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to all those who have been involved in the preparation of this thesis: to my professors for their continuous guidance and supervision throughout the study and for the time they had spent in discussion and carefully reviewing the manuscript, and to my family for all their support, patience, love and understanding.

I would also like to thank Mr. Wenlue Huang, who had spent lots of hours to review the manuscript.

Contents

1 Introduction **1**

2 Object-Oriented Concepts **3**

2.1 Object 3

2.2 Class 3

2.3 Methods and Messages 4

2.4 Static and Dynamic Binding 5

2.5 Inheritance 6

2.6 Encapsulation 8

2.7 Reusability 9

3 Simulation Models and Applications **11**

3.1 Model and Simulation 11

3.2 Object-Oriented Distributed Simulation 13

3.3 Applications 14

4 The DODESE System **16**

4.1 GemStone System 16

4.2 DODESE System Configuration 19

4.3 Functions of DODESE System 22

4.3.1 Defining the Simulation Objects 23
4.3.2 Defining the Simulation 24
4.3.3 Running the simulation 24
4.3.4 Graphical User Interface 25
4.4 The DODESE System Design 25
4.4.1 Event Queue 26
4.4.2 Simulation Resources 26
4.4.3 Process View of Simulation 27
4.4.4 Timing 28
4.4.5 User Interface Design 29
4.4.6 Concurrency Control 29
4.5 An Example 31
4.5.1 Example Description 31
4.5.2 Simulation Design 32
4.5.3 User Interface Design 37
4.5.4 Running Server simulation 39

5 Implementation **42**

5.1 The OPAL Classes 42
5.1.1 Random Number Generator and Mathematical Functions 43
5.1.2 Probability Distribution 45

5.1.3 Ordered and Sorted Collection 47
5.1.4 EventCell Class 48
5.1.5 Process Class 49
5.1.6 Resource Classes 53
5.1.7 Simulation Class 54
5.1.8 Simulation Object 60
5.2 Implementation of User Interfaces 61
5.3 Implementation of Lock and Unlock Functions 62
5.4 Experiments and Discussions 64

6 Conclusion and Further Enhancement **66**

6.1 Conclusion 66
6.2 Further Enhancement 67

List of Figures

1 Single Inheritance Hierarchy 8

2 Multiple Inheritance Hierarchy 9

3 GemStone System Architecture 17

4 A Distributed Simulation System Configuration 20

5 DODESE System Configuration 21

6 Definition of Class User 33

7 Definition of Class Grad 33

8 Definition of Class Server 35

9 Algorithm of the Example Simulation 36

10 Snap Shot of the User Interface at some time instant 37

11 Algorithm for User Interface 38

12 Snap Shot of the User Interface at time instant 1 39

13 Snap Shot of the User Interface at time instant 2 40

14 Snap Shot of the User Interface at time instant 3 41

15 Overview of OPAL Classes in DODESE 42

16 Definition of The Class Mathematics 44

17 Definition of The Class Random 45

18 The Class Hierarchy of Probability Distribution 46

19 Definition of The Probability Distribution Class 46

20 Definition of The Class EventCell 49

21 Definition of The Class Process 50

22 Implementation of The Method forContext: 51

23 Definition of The Class BlockHolderSuperClass 51

24 Definition of The Class BlockHolderA 52

25 Definition of BlockHolderA Class Methods 53

26 Definition of The Classes Resource and ResourceProvider 54

27 Definition of The Class Simulation 55

28 Implementation of The Method startUp 57

29 Definition of The Kernel Simulation Scheduling Methods 59

30 Algorithm For Lock Function 63

31 Algorithm For Unlock Function 63

1 Introduction

Object–oriented concepts and object-oriented programming languages and environments have been widely used in developing simulation languages and environments. Examples of the typical simulation languages and environments are: **SIMULA** [BhDMN73], the first object-oriented simulation language; **MODSIM II** [Her90], an object-oriented general purpose programming and simulation language with built-in graphics; **DOSE**, a modular and reusable object-oriented simulation environment implemented in C++; **RASE**, RADC's advanced simulation environment designed to facilitate rapid development of simulations needed to support the Air Force Command and Control; and **Synapse**[Wag91], an object-oriented conservative parallel simulation environment. However, all these existing environments do not incorporate data base technology. Thus they generally lack of support for persistent objects–objects that survive the process or programming sessions.

On the other hand, data base systems support the persistence, but they do not have the expressibility of object-oriented languages. Both persistence and expressibility are essential for application development. As a result, the object-oriented data bases have emerged.

In order to combine the strengths of both object-oriented paradigms and data base systems to make computer simulation more powerful, it is essential to develop a simulation environment based on an object-oriented data base. Though the well known object-oriented data base system GemStone [1] is widely used, it does not provide the facilities for simulation

[1] GemStone is a registered trademark of Servio Logic Corporation

applications. This thesis is to develop a distributed object-oriented simulation environment DODESE on GemStone which will provide a framework for discrete event-driven simulation applications in a distributed manner by using GemStone data base. The thesis combines object-oriented concepts, data base technology and simulation techniques together to reach the goals of object-orientation, distribution, efficiency and information sharing.

This thesis contains six chapters. In Chapter 2, related object-oriented concepts are introduced along with some examples. In Chapter 3, different simulation models are described, and the advantages of the proposed object-oriented distributed simulation are illustrated. Also application areas of simulation are summarized. In Chapter 4, we first briefly introduce the GemStone system, then the design of DODESE system configuration, system functions and design principles are discussed. In Chapter 5, the implementation of DODESE will be explained in detail. In Chapter 6, future potential enhancements of the system are discussed and a conclusion is made.

2 Object-Oriented Concepts

Some important terms in object-oriented programming like object, method, inheritance, encapsulation etc. are introduced in this chapter. They together characterize the object-oriented paradigm.

2.1 Object

An object is an entity used to represent a real world concept which exists in time and space. An object has its own local data and a set of operations that can manipulate that data [Cox87]. An object will be affected by or can affect some other objects in a global system by the means of receiving or sending messages. For example, an object **eventQueue** represents a collection which holds a set of elements called event cells. When the **eventQueue** is used in the DODESE simulation environment, it stores all the delayed events during the proceeding of the simulation. It has its private data *sortCondition*. It also contains a set of operations such as adding an event cell to the queue, removing an event cell from the queue, and making comparison between two event queues etc.

2.2 Class

A class describes a set of objects that have the same attributes and behaviors. Each individual object described by a class is referred to an 'instance' of that class. A class

consists of methods and instance variables necessary to perform its instance's operations. On the other hand, all the instances of a class have the same set of instance variables, and the instances can respond to the same set of messages sent to them by other objects. For example, the class **EventCell** defined in this simulation environment specifies a set of similar objects which contain part of the simulation information. The class **EventCell** declares four instance variables: *resumeTime*, *type*, *instance* and *procOrblock*. It also has a set of operations to obtain and change the value of its instance variables which may be in different values retained by different instances. All instances of the class **EventCell** can respond to the message **resumeTime** (which will get the value of *resumeTime*) in the same manner but their returning results may be different from each other.

2.3 Methods and Messages

A procedure defined for an object is called a method. A method in an object-oriented programming language is similar to a definition of a procedure or a function in conventional languages such as C or Pascal. A method consists of a name, also called selector, a set of formal parameters(may be empty), and the implementation details of the method. Unlike procedures or functions in C or Pascal, a method is invoked by an object through "message sending". Message sending is just like the procedure or function calls in conventional languages. Because each invocation of a method is the result of a message sending, there is always a sender and a receiver. The sender sends a message to invoke a method

while the receiver responds to the message by executing the method and may or may not return its values. For example, the message in this simulation environment "**EventCell resumeTime:time type:eventType instance: instN proc: aProcOrblock**" invokes the method associated with the selector *resumeTime:type:instName:procN:* defined in the class **EventCell**. This message is sent to the class **EventCell** which receives the message, performs the operation of creating an instance of EventCell with *aTime*, *aType*, *instN* and *aProcOrblock* as its private data values and returns that instance as its return value.

2.4 Static and Dynamic Binding

In conventional programming languages, the types of all variables being used have to be declared before compile time. Consequently, procedure calls are statically bound to specific procedures. Such a binding of a procedure call to a specific procedure at compile time is known as static binding. The advantage of static binding is that it can speed up the execution of the program and provide type checking at compile time. However, static binding has less flexibility of sending the same message to different objects, and it is difficult to change programs with static binding.

Unlike static binding, dynamic binding does not bind a message to a specific method at compile time, so that similar messages can be interpreted by various objects. On the other hand, each object will be the one to determine what method to invoke when receiving a specific message. For example, there are several numbers which are the instances of class

Integer and class **Float**. Suppose these instances receive the same message, division "/", then each number itself will determine which method is needed to perform the division function. Dynamic binding has more flexibility and thus it is easier for programmers to modify the programs. But dynamic binding slows down executions because each method will have to be bound to a message when every time the message is sent.

Polymorphism is a feature closely related to dynamic binding[Her90]. It allows multiple implementations of a method, where each object can implement its own operations in different ways. Therefore, when different objects receive the same message, the results may be different. A typical example of polymorphism in Smalltalk-80 [2] is the message: *add: anObject*. The sending of the message *add: anObject* will behave differently depending on the receiver. For instance, if the class **Bag** receives the message *add: anObject*, it just add the argument *anObject* to the receiver. Whereas if the class **Set** receives the same message, it will check whether the argument– *anObject* is already in the receiver. If the argument is not in the receiver, the argument will be added to the receiver. Otherwise *anObject* will not be added to the receiver.

2.5 Inheritance

Inheritance enables programmers to create objects, such that they can be the specialization or extension of existing objects. The newly created object inherits the data structures and

[2]Smalltalk-80 is a trademark of Xerox Corporation

behaviors of its ancestor object. Therefore, inheritance makes it possible to reuse software components and share data/code.

In languages such as Smalltalk-80 or OPAL [3], an object-oriented data definition and manipulation language for GemStone data base, inheritance occurs among classes. New classes can be created by using the existing classes. In this case, the new classes are called "subclass", and the existing class is the "superclass" of the new class. A subclass inherits instance variables, class variables, and methods from its superclass. New instance variables, class variables and methods, or reimplementation of the methods of a superclass may be added to the subclass to make the subclass behave appropriately. In general, it is clear that inheritance forms a multi-level class hierarchy. If a new class is created from a single existing class, then the inheritance of the new class from the existing class is called single inheritance. In fact, it is possible to create a new class based on more than one existing class, which is called multiple inheritance. Clearly, single inheritance yields a tree structure for the class hierarchy, while the multiple inheritance generates a graph structure for the class hierarchy. Figure 1 is an example of single inheritance and Figure 2 is an example of multiple inheritance.

[3]OPAL is a trademark of Servio Logic Corporation

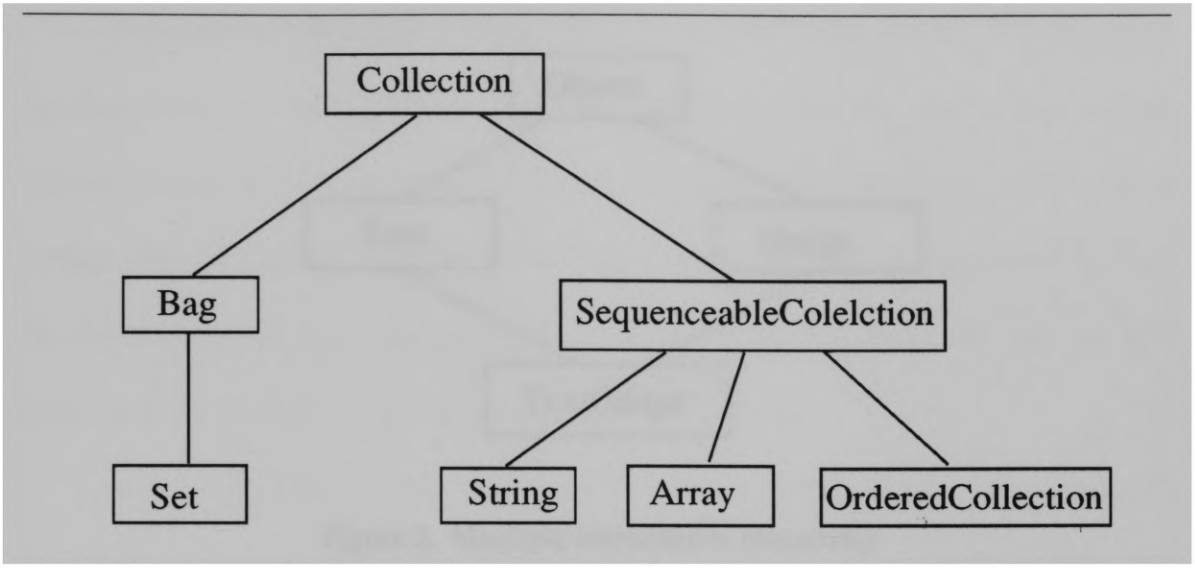

Figure 1: **Single Inheritance Hierarchy**

2.6 Encapsulation

Encapsulation is a technique for minimizing interdependency among separately-written modules by defining their external interfaces [Sny86]. This technique makes the so called "Software IC" possible. The external interface links the module to its clients so that the clients depend only on the external interface when the module is employed. Encapsulation is closely related to *information hiding* because the implementation details of the module are hidden within the module. One of the important features of encapsulation is that a module can be reimplemented without affecting any other module. Another important feature of encapsulation is that the internals of a module are protected from outside access so that correct functioning is guaranteed, and thus program debugging can be done easier in case

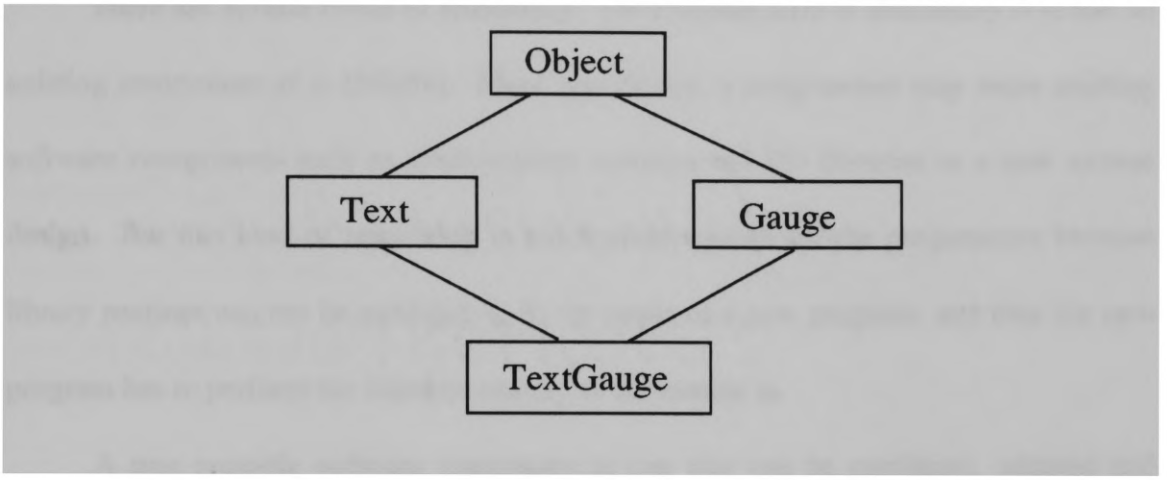

Figure 2: **Multiple Inheritance Hierarchy**

of a malfunctioning program [Mic88].

Encapsulation and information hiding in an object-oriented programming language are supported as classes. The methods of a class hide the implementation details from the outside, so that other classes can send messages to invoke them only through their external interface or set of messages.

2.7 Reusability

In an ideal software development environment, new systems are built by selecting software components from catalogs of software modules and combining them, rather than reinventing the wheel [Mey88]. Whether this idea can be realized depends on the reusability of software components or modules. Therefore, the reusability of a software is always considered as a important factor.

There are several forms of reusability. The simplest kind of reusability is to use an existing component *as is* [Mic88]. More specifically, a programmer may reuse existing software components such as mathematical routines and I/O libraries in a new system design. But this kind of reusability is not flexible enough for the programmer because library routines can not be modified to fit the needs of a new program, and thus the new program has to perform the function exactly as the routine is.

A true reusable software component is one that can be combined, adapted and modified to fit in a new application, it's function must be clear to a programmer to determine if it is reuseful in his application. In object-oriented programming languages, reusability is supported by inheritance. Any class can be a true reusable component because it can be inherited by another class. The inherited methods can be modified to meet the new applications. For example, in OPAL, there is a class representing a set of sequence objects named **SequenceableCollection**. This class can be reused and modified to become a new class **OrderedCollection** in which elements of the collection can be accessed via an index. The **OrderedCollection** then can be reused and modified to create another new class **SortedCollection** in which elements are stored in certain order, according to some sorting conditions. Furthermore, the **SortedCollection** is used in this simulation environment to implement an event queue which stores all the delayed events during the proceeding of a simulation. It is obvious that reusability of software components is a powerful feature in software development.

3 Simulation Models and Applications

In this chapter, some basic concepts of modeling and simulation are discussed. They are fundamental to the design of an object-oriented simulation environment. Because we focus on discrete event simulation, emphasis will be placed on discrete event driven simulation. Then applications of simulation will be discussed also.

3.1 Model and Simulation

In order to study a real world entity or a system scientifically and systematically, an assumption about the system in form of mathematical relationship should be made, and this mathematical assumption is called the *model* of the system. Because of the complexity of most of the real world systems, it is significantly easier to build models of the systems by the means of computer simulations than to use any other means. In fact, a simulation is the process of designing a model, the conduction of experiments with this model for the purpose of understanding the behavior of the system and the evaluation of various strategies for the operation of the system.

Three major types of simulation models can be distinguished[Smi68]: continuous change model, fixed-period model and discrete event model. In a continuous change model, the system is represented by a set of continuous variables whose value changes continuously with time according to a set of differential equations. In a fixed-period model, time is broken up into a set of finite periods, and the value of the variables is allowed

to change at the end of periods only. The variables are continuous quantities which are specified by a set of difference equations relating values at the current time period to those at the immediately previous time-period. In contrast to these two types of simulation models, a discrete-event model employs variables in discrete quantities representing states of entities in the system. Interactions between entities take place at discrete points of time only, separated by intervals of inactivity. Such interactions are usually known as "events". There usually is an event queue to keep all the events waiting to be scheduled to occur in the simulation. The actual timing of occurring events is usually affected by stochastic factors. Time is advanced from the current occurring event to the next event being scheduled [Smi68], which is called next-event time advance. The process of advancing the simulation time from one event to another is continued until some pre-specified stopping condition is satisfied.

In discrete event simulation, the following components are usually found [LK82]:

- **System state.** The collection of state variables necessary to describe the system at a particular time.
- **Simulation clock.** A variable giving the current value of simulated time.
- **Event queue.** A list containing the next time when each type of event will occur.
- **Statistical counters.** Variables for storing statistical information about the system performance.

- **Initialization routine.** A subroutine to initialize the simulation model at time zero.
- **Timing routine.** A subroutine to determine the next event from the event queue and then advances the simulation clock to the time when that event is to occur.
- **Event routine.** A subroutine to update the system state when a particular type of event occurs (there is one event routine for each event type).
- **Report generator.** A subroutine compute estimates(from the statistical counters) of the desired measures of performance and prints a report when the simulation ends.
- **Main program.** A program to call the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately.

3.2 Object-Oriented Distributed Simulation

There are strong reasons to combine simulation with data base techniques. Rich data structuring facilities would make simulation modeling much easier. The data of a simulation run can be easily managed by a data base management system (DBMS), and simulation models may require access to the data stored in a data base [MLPW90]. On the other hand, there are many advantages of object oriented paradigm as mentioned in chapter 2. Obviously, to construct a powerful simulation environment, it is desired to incorporate an object-oriented data base into a simulation environment.

Furthermore, an object-oriented simulation environment can be designed with a distributed method by using a data base, so that the following benefits can be obtained:

- **Decentralization.** Different processors can run different simulations and their related parts simultaneously, such as statistic collection and user interface.
- **Efficiency.** The speed and efficiency of a simulation can be increased when two or more processors are used to process the simulation, where the simulation tasks are divided and performed by these processors. Each process can run independently on a processor and the process is not impeded by other parts of the simulation run. Thus high efficiency is expected.
- **Information Sharing.** All the processors involved in the simulation can share all the information stored in the data base.

After all, adherence to the paradigm of object-oriented distributed programming will result in a high quality simulation environment.

3.3 Applications

Simulation can be used in a wide variety of fields and disciplines. Some examples are given as follows [Smi68, Sha75].

1. **Management Science.** Simulation has been used to solve problems of queueing which affects the operations of individual firms, enterprises and customer service

points. The data, such as queue lengths, waiting times, customers turned away of otherwise canceled, and server utilization, can be collected via the simulation.

2. **Engineering Design.** Simulation has been used to accomplish a variety of engineering tasks in areas such as control systems, VLSI systems, communication systems and power systems.

3. **Econometrics.** Simulation models have been used to analyze and predict the behavior of firms, industries, national economic systems, and to study the effect of different control policies available to the authorities.

4. **Education and Training.** The development and use of a simulation model allows experimenters to get familiar with the system. In such a way, a experimenter will be assisted in understanding and gaining a feel of the system, thus aiding the process of innovation.

In recent years, object-oriented simulation has been successfully applied to these areas, and more and more programmers are employing the object-oriented technique for their new applications.

4 The DODESE System

In the DODESE, object-orientation is realized by using the object-oriented data base system–GemStone. In this chapter, the GemStone system is briefly reviewed first, then the DODESE system configuration, design principles and user interface are introduced. Finally, an example is given to explain how DODESE functions.

4.1 GemStone System

GemStone is an object-oriented data base system which merges object-oriented programming concepts with those of data base systems. It is a set of tools providing the facilities to create data base application programs. Figure 3 [Ser91c] is the overall architecture of the GemStone system version 2.0. As shown in Figure 3, a host computer runs a single data base monitor process. The monitor handles user login and logout operations and coordinates concurrent access by multiple users. Individual GemStone session processes communicate with the monitor through interprocess channels. Each GemStone session process incorporates a data management kernel and an interpreter for GemStone's data definition, data manipulation and general computation language—OPAL [Ser91c].

GemStone system provides a powerful object-oriented programming language, OPAL, a descendent of Smalltalk-80. Unlike non-object-oriented data base systems, OPAL combines data definition, manipulation, and query facilities together to provide flexible data modeling facilities which can model various kinds of objects and their relationships not suitable for

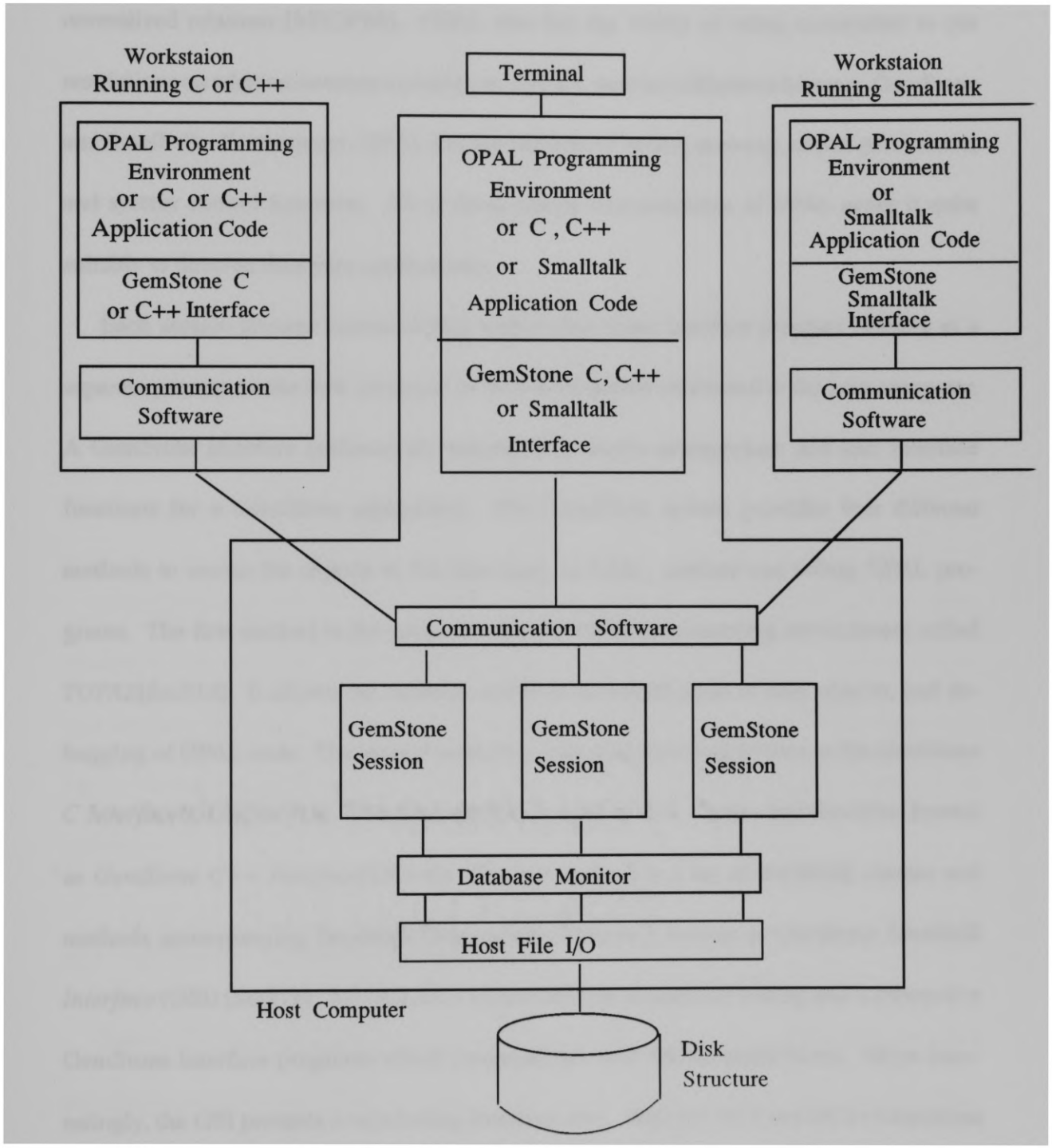

Figure 3: GemStone System Architecture

normalized relations [MSOP86]. OPAL also has the ability of using *constraints* to put restrictions on what an instance variable can contain, another difference between GemStone and Smalltalk. Furthermore, OPAL has the features of virtual memory, object persistence, and system control functions. All of these unique characteristics of OPAL make it quite suitable to develop data base applications.

Each session process communicates with a GemStone interface program running as a separate process on the host computer or on a workstation connected to the host computer. A GemStone interface performs all terminal I/O, screen management and user interface functions for a GemStone application. The GemStone system provides four different methods to access the objects in the data base, to build , execute and debug OPAL programs. The first method is the command-driven OPAL programming environment called *TOPAZ*[Ser91d]. It allows the creation, retrieval or modification of data objects, and debugging of OPAL code. The second method is a set of C functions known as the *GemStone C Interface*(GCI)[Ser91a]. The third method is a set of C++ classes and functions known as *GemStone C++ Interface*(GC++I). The last method is a set of Smalltalk classes and methods accompanying Smalltalk Objectworks Release-4 known as *GemStone Smalltalk Interface* (GSI) [Ser91b]. All GCI, GC++I and GSI can be used in creating user's interactive GemStone interface programs which communicate with OPAL applications. More interestingly, the GSI presents a windowing interface also. With the GCI and GC++I functions or GSI methods, the following tasks can be performed [Ser91c]:

- Building data objects as C or Smalltalk data structures in the interface program and transmit them to GemStone.

- Importing GemStone data objects into the interface program's memory for manipulation as C or Smalltalk data structures.

- Filing data definition written in OPAL into text files on host operating system, or vice versa, filing data definition text files on host operating system into GemStone.

- Performing system functions such as committing transactions, aborting transactions, and performing host operating system commands.

GemStone supports all object-oriented concepts and possesses all the features of object-oriented programming by using OPAL. OPAL provides the mechanism for defining classes and sending messages. Consequently, through the inheritance among classes and dynamic binding, the goal of highly software reusability and maintainability is possible to be achieved.

4.2 DODESE System Configuration

Figure 4 [EC90] depicts one of the possible architectures of a distributed object-oriented simulation environment. The centralized GemStone data base is accessible to processors which perform simulation runs, collect statistical data and provide user interfaces.

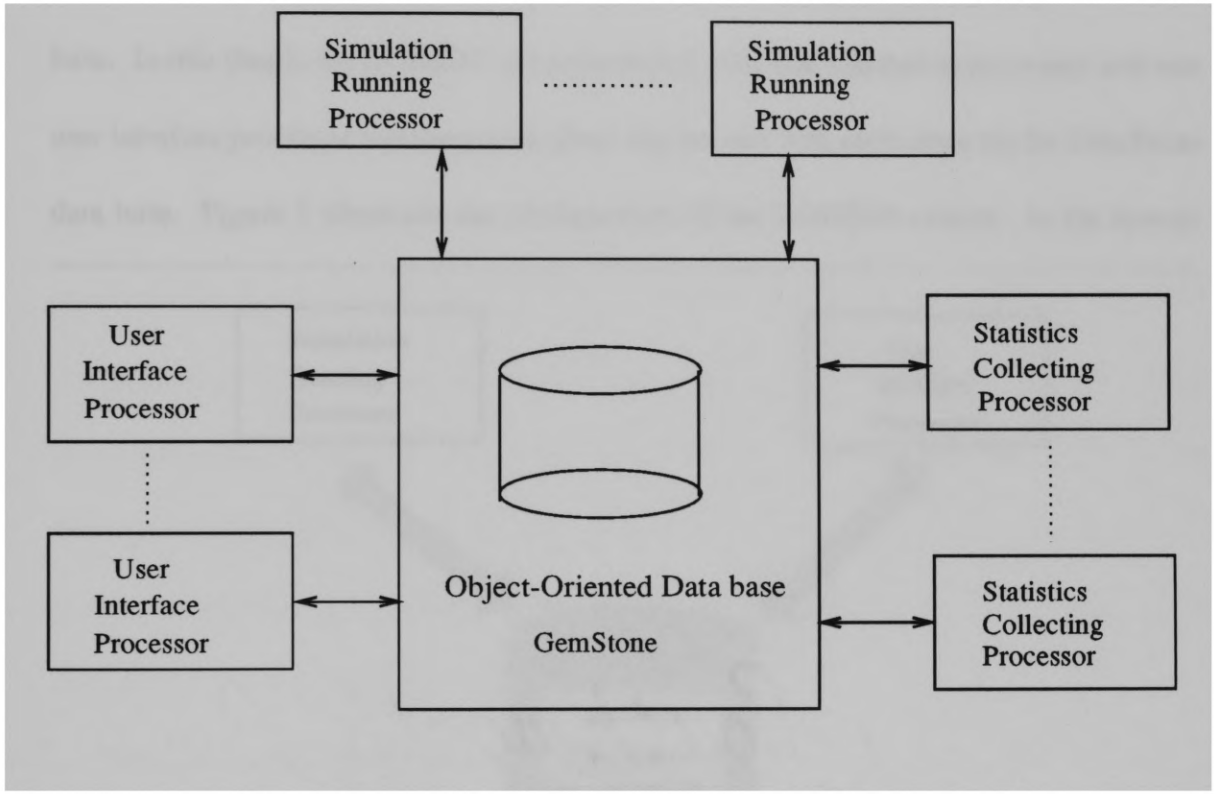

Figure 4: **A Distributed Simulation System Configuration**

Using the object-oriented data model, the GemStone data base stores all the events and objects which are related to all the simulations running on different processors. Different simulations can be run on different processors that interact with the data base. Meanwhile, statistical data about the simulations can be collected by other independent processes. The status of the simulations can be observed from user interface processors that query the data base. In this thesis, the DODESE is implemented with one simulation processor and one user interface processor and these processors can interact with each other via the GemStone data base. Figure 5 illustrates the configuration of the DODESE system. In the system

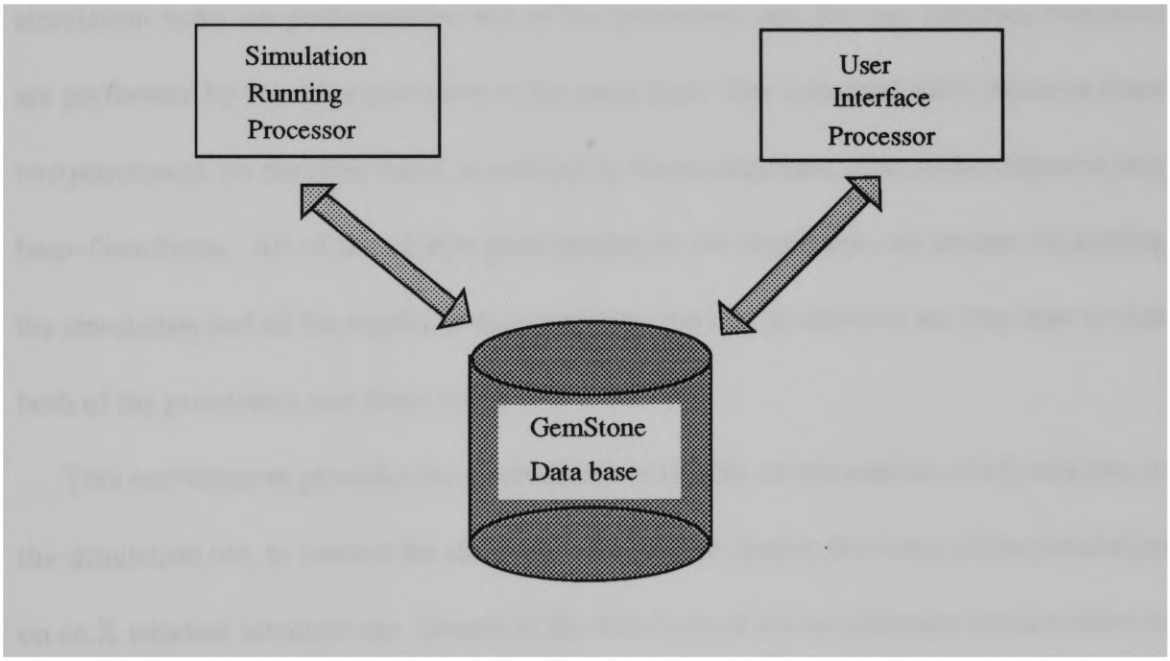

Figure 5: DODESE System Configuration

implementation, only two Sun workstations are used, without loss of generality. One of the

workstations is used to run the simulation programs, while the other workstation is used to display the status of the simulation continuously as well as updating the simulation run interactively.

4.3 Functions of DODESE System

The DODESE system is a simulation environment for discrete-event simulations. This system distributes a simulation's tasks into two independent processors and provides the control of the interprocess communication between the processors. More specifically, the simulation tasks are performed by one of the processors and the user interface functions are performed by the other processor at the same time. The communication between these two processors, on the other hand, is realized by the management of an object-oriented data base–GemStone. All of the objects participating in the simulation, all classes supporting the simulation and all the results of the simulation run can be stored in the data base so that both of the processors can share them.

This environment provides the mechanisms to specify all the entities which take part in the simulation run, to control the simulation run, and to display the status of the simulation on an X window interactively. Details of the functions of the environment are described in the following subsections.

4.3.1 Defining the Simulation Objects

A simulation object is any one of the objects that presents in a simulated situation and carries out its tasks through the simulation run. A superclass **SimObject** is defined in the DODESE to describe all the objects that present in a simulated situation. In fact, the class **SimObject** models the similarities, attributes and behaviors of the simulation objects. Consequently, all the simulation objects taking part in any simulation in DODESE can be defined as the subclasses of **SimObject**. The general life cycle of a simulation object can be considered as the sequence of entering the simulation, performing the tasks and leaving the simulation. A user can specify a simulation object by defining a set of activities that are initiated when the object enters the simulation, and a set of tasks to be performed after the object entered the simulation. After finishing its tasks, the object will leave the simulation. The DODESE system provides the mechanism for each simulation object to access to the specific simulation in which it is functioning.

The tasks carried out by each simulation object might be doing a number of subjobs, acquiring access to other simulation objects or acquiring the resources shared by other simulation objects in the same simulation run. When an object enters or leaves the simulation, it will inform the simulation about its entering or leaving.

4.3.2 Defining the Simulation

An OPAL superclass **Simulation** is defined to provide a framework for defining simulations in DODESE. Any simulation can be defined as a subclass of **Simulation** by specifying the following: 1. Arrival schedule for each object that will participate in the simulation. 2. The resources that will be used by the simulation objects. 3. Probability distribution used to randomly determine any time interval during the simulation run, such as the interarrival time, job length etc.. 4. Global variables storing the simulation information which will be displayed on the user interface processor as well as the information interactively stored by the user interface processor. Some mechanisms are provided for each simulation run so that the simulation can be informed of which simulation object is entering or leaving. Also, the simulation provides the information about the resources it have, and schedule an instance of a simulation object or a **SimObject** class to occur. The time interval of scheduling simulation objects to enter the simulation is randomly generated by using a probability distribution discussed in Section 5.1.2.

4.3.3 Running the simulation

After the simulation and all the simulation objects which will participate in the simulation are defined, the simulation can be run by invoking the system provided mechanisms via several phases:

- **Instantiation**–Creating an instance of the **Simulation** class defined in the DODESE.

- **Starting up the simulation**–Specifying the initial simulation objects, the arrival schedule of new objects, the resources that simulation possesses and all the probability distributions which will be employed to generate the random sampling during the simulation.

- **Proceeding the simulation**–Proceeding the simulation until the pre-specified stopping criterion is reached.

- **Terminating**–Terminating the simulation and emptying the event queue.

4.3.4 Graphical User Interface

The running status of the simulations on one of the processors can be displayed on the user interface processor in graphical form by using X11 windows concurrently. The user interface processor will run interactively, where the user may interact with the simulation processor by adding or deleting some simulation resources or simulation events. All the information being displayed and interacted is stored in the data base. Both processors can retrieve the information periodically.

4.4 The DODESE System Design

Some principles used in the design of the DODESE system are presented in this section. These principles include the event queue design, simulation resources, the process view of simulation, timing and the concurrency control.

4.4.1 Event Queue

An event queue is designed for each simulation to store all the events that must be delayed for some increment of simulated time. All the delayed events will be sorted in ascending order with respect to the simulated time associating with each event. On the other hand, the structure of each event cell contains all the information needed to proceed the simulation, such as the resuming time of the event, the type of the event, the name of participating object at that resumed time, and the name of the corresponding process which carries out the tasks of the participating object. In this environment, a class named **EventCell** is designed, and all the event cells are the instances of this class.

4.4.2 Simulation Resources

To support a general simulation, mechanisms are needed to coordinate the use of simulation resources by simulation objects. A simulation can consist of the definition of resources and the messages for acquiring or releasing each kind of resources. Also, a simulation object will perform the tasks by acquiring the resources to synchronize with other simulation objects participating in the same simulation. In this environment, two classes named **Resource** and **ResourceProvider** are designed, and mechanisms such as acquiring, releasing, producing and consuming resources are provided. Consequently, all the resources defined in the simulation are the instances of class ResourceProvider.

4.4.3 Process View of Simulation

There are two types of discrete-event driven simulation: event-oriented and process-oriented [BFS83]. In event-oriented simulation, the simulation run is based on the sequencing of a dynamic event queue. Each event has an associated routine that determines the actions to be done when the event is sequenced. During the execution of an event routine, simulation time does not advance, however, the event queue manager of the simulation environment advances the time when the next event is scheduled. In process-oriented simulation, a process can be defined as a dynamic entity, a singularly occurring instance of execution of a set of logically related activities [Fra77]. The routine implementing the process contains all of its related activities. Each process may maintain its own activity list, and the simulation environment may maintain a master activity list containing the next activity from each process' activity list. Thus, the process-oriented technique can be used to implement the simulations with interruptible tasks, while the event-oriented technique can be used to implement the simulation with atomic tasks.

The DODESE system combines both the event-oriented and the process-oriented techniques, and the idea of virtual process is employed in implementing this system. In a virtual process, it can be considered that the process is not scheduled and executed by the host operating system process scheduler, instead by the simulation itself. The statements of one process might be the simulation scheduler which schedules new objects into the simulation, or some simulation activities performed by the simulation objects or the simulation itself.

The event that will be scheduled by the simulation might be associated with one of the virtual processes or one event routine. Each virtual process is expressed as an OPAL class, while the statements to be executed in a virtual process are designed as the contents of a set of class methods of that class, where each method contains one of those statements. The DODESE users can design their own simulations using either or both of the following approaches for different types of the tasks (interruptible or atomic). 1. Associate a simulation event with one of the virtual process, and perform the functions by executing the pre-defined class methods which are defined when the event is created. 2. Create a simulation event accompanied with an event routine, and performing the simulation tasks by executing the event routine. In addition, in this environment, a class named **Process** is designed, and all the virtual processes created during the simulation are the instances of this class.

4.4.4 Timing

Timing is a very important factor in simulations. For any simulation, there always is a clock to represent the real or simulated time which is related to all the simulated objects. For discrete event–driven simulations, the clock is moved forward according to the time at which the next event will occur. In this environment, an instance variable of the class **Simulation** named '*currentTime*' is specified to record the clock value of the simulation. A probability distribution is employed to randomly generate the time interval at which the

next event will occur.

4.4.5 User Interface Design

An X window is designed to form the user interface to display the status of the simulation runs and interact with the simulation processor. The interface program login to the GemStone data base via GemStone C Interface. After a successful login, the user interface processor will wait for the simulation to start on a simulation processor and begin to display the simulation status. Also, the user interface is designed to read the keyboard inputs so that the functions such as interaction with simulation processor and termination of the interface process running can be performed. If a simulation terminates, the user interface processor will stay in the data base and wait for the next simulation to start, and keep observing the status and interacting with the next simulation.

4.4.6 Concurrency Control

A concurrency control mechanism has to be provided to synchronize the execution of the two processors in DODESE concurrently. As one processor logs into GemStone via Smalltalk interface, and the other processor logs into GemStone via GCI, both the GemStone object locking mechanisms and the Unix locking mechanisms can be employed to reach the goal of concurrency control. However, if a lock on an object is requested, and another session already holds a conflicting lock on that object, then the GemStone will deny

the request. Although a recursive calling method can be used to request a lock repeatedly until the lock becomes available, GemStone does not assure that a lock will be acquired no matter how long the user will wait [Ser91c].

In fact, in Unix, there are some methods to perform interprocess communication and file locking. The **flock** system call will apply or remove lock on the file which allows two types of locks: shared locks and exclusive locks. The **lockf** system call places, removes, and tests for exclusive locks on sections of files. All the locks of a process are removed when that process terminates. For both **flock** and **lockf**, a locking call on an already locked file section will put the caller to sleep until that file section is unlocked, or return an error value depending on parameter setting in the lock call.

In this environment, considering the acquisition time of the lock functions, both lock and unlock algorithms are designed using Unix locking mechanism–**lockf**. The lock and unlock functions are implemented in C as two primitive routines for both the simulation processor and user interface processor. The details of the algorithms and implementation will be described in Section 5.3. In order to avoid any dead locks between the two processors, only one lock is acquired at a time. That is, for any processor, it can only acquire one lock on one object. After reading or writing the object, the processor has to release the lock before acquiring another lock on any other object, So that the system is dead-lock free.

4.5 An Example

4.5.1 Example Description

A simulation example is given here to illustrate how the DODESE system can be used to design different simulations. Suppose that the simulated system has variable number of processors and three types of users, undergraduate students, graduate students and professors. Each kind of users will log into the system randomly, where the undergraduate students log into the server about one every 5 units of simulated time, the graduate students log into the server about one every 10 units of simulated time and the professors log into the server about one every 15 units of simulated time. Suppose that each type of users starts to login to the server at simulated time 0. When a user logged in to the simulated system, a total amount of execution time for each session is randomly determined according to a random number generator and a number of subjobs will be scheduled. Suppose that the interarrival time between any two subjobs for each kind of user, also called thinking time, is an exponential distribution with a mean of 5, and the execution time of the subjobs for undergraduate students, graduate students and professors is an uniform distribution with intervals over [8, 15], [5, 10] and [4, 9] respectively. The number of processors in the simulated system is initialized to two and is able to be changed from the user interface processor, that is, a user can add or delete simulated processors from the user interface. During the simulation run, the following four kinds of information about the simulation are graphically and concurrently displayed on the user interface.

- Processor Utilization–which can be defined as the number of busy processors over the total number of processors available in the simulated server system.

- Relative utilization–for each user who entered simulation and scheduled n subjobs, the relative time ratio will be defined as:

$$x = \frac{\sum_{i=1}^{n-1}(y_i + z_i) + y_n}{\sum_{i=1}^{n-1}(y_i + z_i + t_i) + y_n} \tag{1}$$

where: y_i is the execution time of $subjob_i, i = 1, \cdots, n$,

z_i is the thinking time between $subjob_i$ and $subjob_{i+1}, i = 1, \cdots, n-1$,

t_i is the waiting time of $subjob_{i+1}, i = 1, \cdots, n-1$.

At a time instant, let u be the accumulated sum of execution time of all the subjobs that are already processed, v be the accumulated sum of thinking time between any two subjobs that already got executed and w be the accumulated sum of the job waiting time. Thus, the relative utilization at that time instant is $\frac{u+v}{u+v+w}$.

- Number of jobs in the job queue on waiting for being processed.

- Number of users presented in the simulation.

4.5.2 Simulation Design

According to the described specifications, a class User is defined as the subclass of the DODESE system class **SimObject** shown as in Figure 6. Where, the class variable *rand* is initialized to the seed of the Uniformly distributed random number over the interval

```
SimObject subclass: 'User'
   instVarNames: #( 'cpuTime')
   classVars: #( 'subclasses' 'rand')
   poolDictionaries: #[]
   inDictionary: ExampleNew
   constraints: #[  ]
   isInvariant: false
```

Figure 6: **Definition of Class User**

(0.0, 1.0), while the instance variable *cpuTime* is initialized to the random number over the interval (0.0, 100.0). Whenever an instance of **User** is created, a random number is assigned to *cpuTime* to get the total amount of execution time for each user logged into the system.

Three classes, **Grad**, **UnderGrad** and **Prof**, are designed for each kind of users mentioned above. They are the subclasses of class **User**. The definition of class **Grad** is shown as Figure 7. Where the class variables *jlProb*, *thinkProb* and *arrivalProb* are initialized

```
User subclass: 'Grad'
  instVarNames: #()
  classVars: #( 'jlProb' 'thinkProb'
                'arrivalProb' 'subclasses')
  poolDictionaries: #[]
  inDictionary: ExampleNew
  constraints: #[  ]
  isInvariant: false
```

Figure 7: **Definition of Class Grad**

to the probability distributions of job length, thinking time and interarrival time. The definitions of classes **UnderGrad** and **Prof** are similar to those of class **Grad**.

The example simulation is defined as the subclass of the DODESE system class **Simulation**. The definition of class **Server** and two of its initializing methods **defineProbDistribution** and **defineResources** are shown as Figure 8. There are four types of events in the event queue listed bellow:

- Type 1 – job arriving
- Type 2 – job ending
- Type 3 – simulation ending
- Type 4 – user entering the simulation

For different types of events, the simulation will perform different processing accordingly. Several design strategies are described as follows:

- At the beginning of the simulation, only one of each type of users is scheduled to enter the simulation.
- When a user enters the simulation, the first subjob of the user is scheduled to be processed and the next user of the same type is scheduled according to the predefined probability distribution.

```
Simulation subclass: 'Server'
     instVarNames: #()
     classVars: #( 'subclasses')
     poolDictionaries: #[]
     inDictionary: ExampleNew
     constraints: #[]
     isInvariant: false


defineProbDistribution

    Grad setArrival: (Exponential mean: 10)
         setThink: (Exponential mean: 5)
         setJL: (Uniform from: 5  to: 10).

    Prof setArrival: (Exponential mean: 5)
         setThink: (Exponential mean: 5)
         setJL: (Uniform from: 4  to: 9).

    UnderGrad setArrival: (Exponential mean: 15)
         setThink: (Exponential mean: 5)
         setJL: (Uniform from: 8  to: 15).


defineResources

    self produce: 2 of: 'Processor'.
```

Figure 8: **Definition of Class Server**

- The simulation ending event is scheduled at the beginning of the simulation with the pre-specified simulation ending time. When the simulation reaches the simulation ending event, the simulation will stop immediately and disregard all the events in the event queue.

- When one simulated processor is asked to be deleted by the user while all the simulated processors are busy at that moment, the processor will be deleted right after the first subjob is finished.

The algorithm of the example simulation is illustrated in Figure 9.

```
1. Initialize the instance variables.
2. Initialize the probability distribution for
   each simulation object.
3. Initialize the global variables.
4. Define the resources used in the simulation.
5. Schedule first user of each type of users to
   enter the simulation, and schedule simulation
   ending event.
6. Check for adding or deleting processors, do
   the appropriate processing.
7. Remove next event from event queue.
8. Perform appropriate processing according to
   different types of event.
9. Repeat step 6.
```

Figure 9: **Algorithm of the Example Simulation**

4.5.3 User Interface Design

The graphical user interface of the system will dynamically display the information of the simulation run. The snap shot of the interface at an accumulated instance will have the display format as one shown in Figure 10. From Figure 10, it can be seen that four

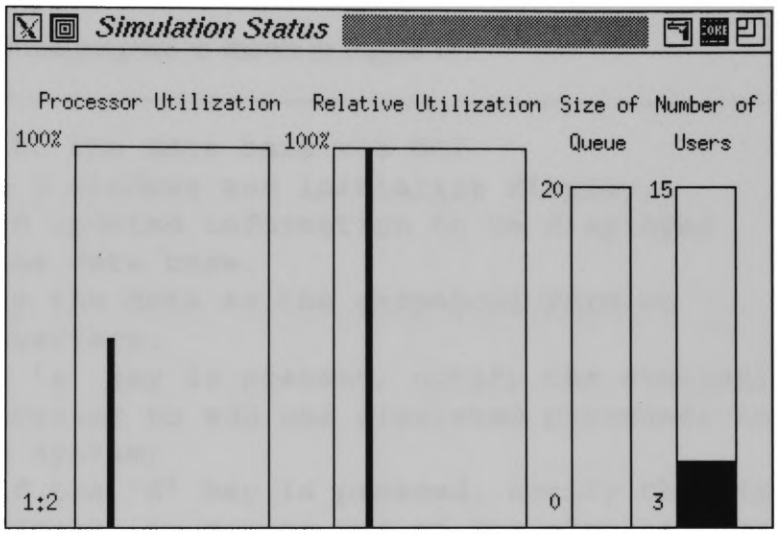

Figure 10: **Snap Shot of the User Interface at some time instant**

subwindows are designed to display four different kinds of information. The numbers in the bottom of the windows indicate the actual numbers that the display presents. For example, the two numbers on the side of the processor utilization subwindow represent that there are 2 simulated processors available and one processor is busy at that instant. Thus the processor utilization is 50 per cent. The number 3 on the side of the user subwindow

indicates that there are three users in the simulation at that time moment.

As the simulation proceeds, all the information displayed will be continuously updated. Also, the user of the DODESE system can add processors from the user interface by pressing the 'a' key, or delete processors by pressing the 'd' key. When user press the 'q' key, the user interface program will be interrupted, and the process will terminate. The algorithm for the user interface program is shown in Figure 11.

```
1. Login to the data base via GCI.
2. Set up X windows and initialize display.
3. Get the updated information to be displayed
   from the data base.
4. Display the data as the graphical form on
   the interface.
5. If the 'a' key is pressed, notify the simulation
      processor to add one simulated processor to
      the system;
   else if the 'd' key is pressed, notify the simulation
      processor to delete one of the simulated processors;
   else if the 'q' key is pressed, log out from data base
         and quit;
6. Repeat step 3.
```

Figure 11: **Algorithm for User Interface**

4.5.4 Running Server simulation

After the simulation and simulation objects are defined, the following expression is evaluated on one Sun 3 workstation to execute the example simulation, meanwhile the user interface program starts to run on the other Sun 3 workstation to login to the data base and start observing the simulation status.

Server new startUp proceed

At certain time instant, the status of user interface is shown as Figure 12. From Figure

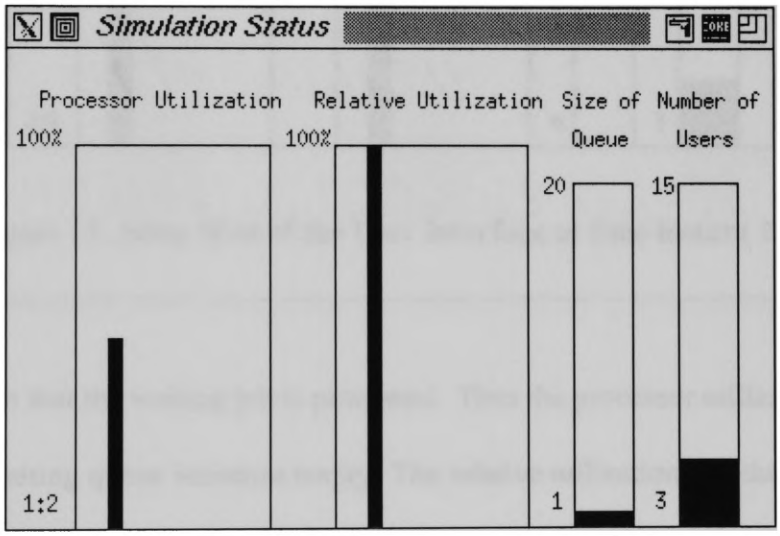

Figure 12: **Snap Shot of the User Interface at time instant 1**

12, it can be seen that the relative utilization of the simulated system is 100%, while the processor utilization is 50% which means that there is still one processor available. It also

can be seen that there are three users in the simulation and there is one subjob waiting to be processed.

At the next time instant, the status of the interface is shown as Figure 13. From Figure

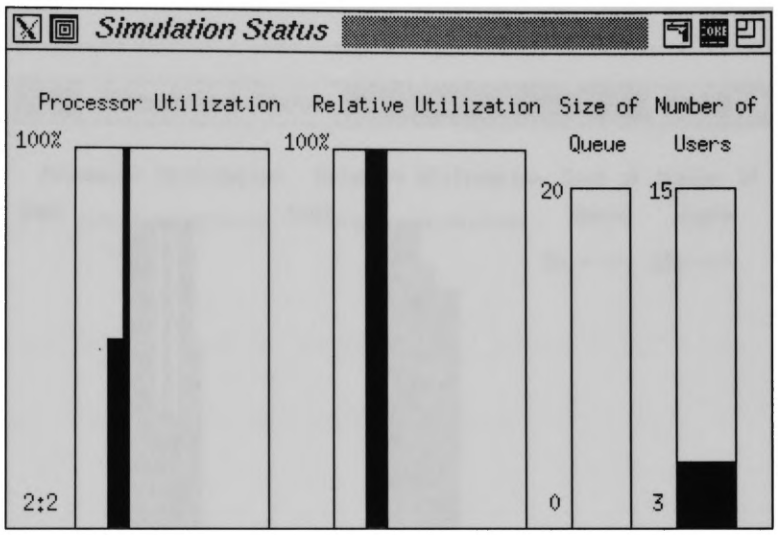

Figure 13: **Snap Shot of the User Interface at time instant 2**

13, it can be seen that the waiting job is processed. Thus the processor utilization becomes 100% and the waiting queue becomes empty. The relative utilization and the total number of users remains unchanged since the last time instant.

Now suppose the 'a' key is pressed to indicate that a processor is added to the system. In a certain amount of time, the status of the interface will become the window shown in Figure 14, Where the available processors are increased to 3, and the processor utilization is about 82%. On the other hand, one processor can be deleted by pressing 'd' key. Then, the

total number of processors will be decreased by 1. When processor utilization is low and relative utilization is high, the user can delete the simulated processors. When processor utilization is high and relative utilization is low, or job queue is not empty, the user should add more processors to increase the simulation speed.

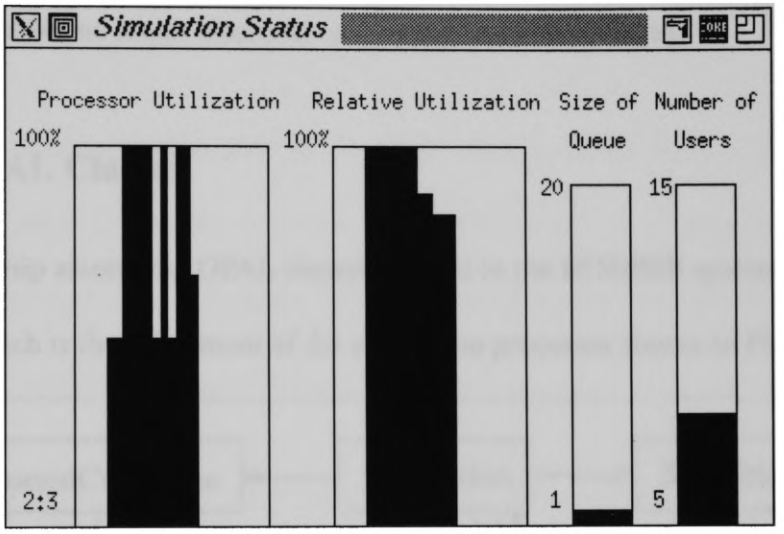

Figure 14: **Snap Shot of the User Interface at time instant 3**

5 Implementation

The implementation details of the DODESE system will be described in this chapter. This simulation environment consists of a set of GemStone classes written in OPAL and a user interface to fetch the information stored in GemStone data base. The following subsections describe the major elements composing this environment.

5.1 The OPAL Classes

The relationship among the OPAL classes created in the DODESE system is illustrated in Figure 15 which is the refinement of the simulation processor shown in Figure 5.

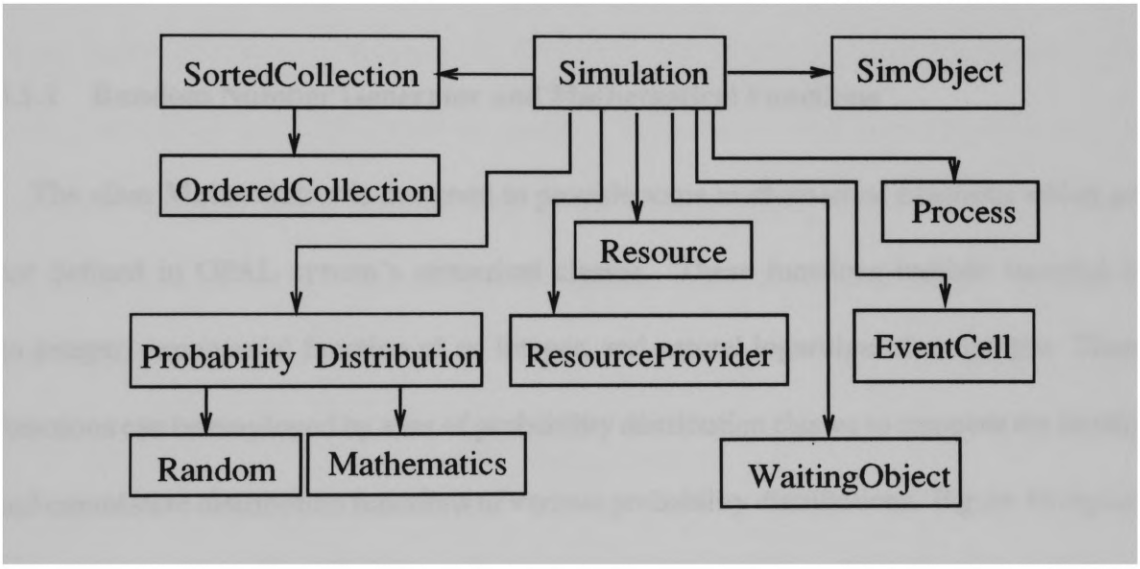

Figure 15: Overview of OPAL Classes in DODESE

From Figure 15, it can be seen that the class **Random**, a random number generator

and the class **Mathematics** will be employed by a set of probability distribution classes to obtain the random sampling of the distribution. The class **OrderedCollection** is used to construct the class **SortedCollection** which will be used to implement the event queue in the simulation, while the instances of the class **EventCell** will be the elements in the event queue. The class **Process** is created to implement the idea of virtual process described in chapter 4. The class **Resource** and the class **ResourceProvider** are implemented to provide the mechanisms to acquiring the resources provided by the simulation, while the instances of the class **WaitingObject** will be the elements in the resource waiting queue. The key classes implemented in this environment are the classes **Simulation** and **SimObject** which will accomplish most of the tasks of this environment.

5.1.1 Random Number Generator and Mathematical Functions

The class **Mathematics** is designed to provide some mathematical functions which are not defined in OPAL system's numerical classes. These functions include factorial of an integer, exponential function of an integer, and natural logarithm of an integer. These functions can be employed by a set of probability distribution classes to compute the density and cumulative distribution functions of various probability distributions. Figure 16 depicts the definition of the class **Mathematics** and the basic method specifications.

The class **Random** provides a mechanism to obtain a sequence of random numbers that will be uniformly distributed over the interval between 0.0 and 1.0. An instance of the

```
Number subclass: 'Mathematics'
        instVarNames: #()
        classVars: #( 'subclasses')
        PoolDictionaries: #[]
        inDictionary: Statistics
        constraints: #[]
        isInvariant: false

factorial: x                    return the factorial of x
base: x aisedToInteger: anInteger compute x to anInteger
exp: x                          return the exponential function of x
ln: x                           return the natural logrithm of x
```

Figure 16: **Definition of The Class Mathematics**

dclass **Random** maintains a seed from which the next random number is generated. The seed is initialized in a pseudo-random manner. The message **next** is sent to an instance of **Random** whenever a new random number is required. Figure 17 is the definition of the class **Random** and some major method specifications. A random number generator can be created with the expression:

```
rand := Random new
```

The expression **rand next** then can be evaluated whenever a new random number is needed. The result of this expression is a floating-point number between 0.0 and 1.0. The implementation of **next** is based on Lehmer's linear congruential method [Knu76].

Stream subclass: 'Random'
instVarNames: #('seed')
classVars: #('subclasses')
poolDictionaries: #[]
inDictionary: Statistics
constraints: #[]
isInvariant: false

initializeSeed	initialize the first random number
next	get the next random number

Figure 17: **Definition of The Class Random**

5.1.2 Probability Distribution

A tree structure of a set of probability distribution classes is designed and illustrated in Figure 18. The superclass **ProbabilityDistribution** provides the protocol to obtain one or more random samplings from any probability distribution, and to compute the density functions and cumulative distribution functions. Figure 19 shows the definition of this class. The class variable "rand" is initialized to an instance of the class **Random**. Whenever a random sampling is required, the message *next* is sent to the distribution. **ProbabilityDistribution** implements the method *next* by returning the result of the message **inverseDistribution: var**, where *var* is a random number generated by '*rand*'. All subclasses of **ProbabilityDistribution** have to implement **inverseDistribution:** in order to map (0, 1) onto their sample space to get the next sampling, or else they must override the definition of the method *next*.

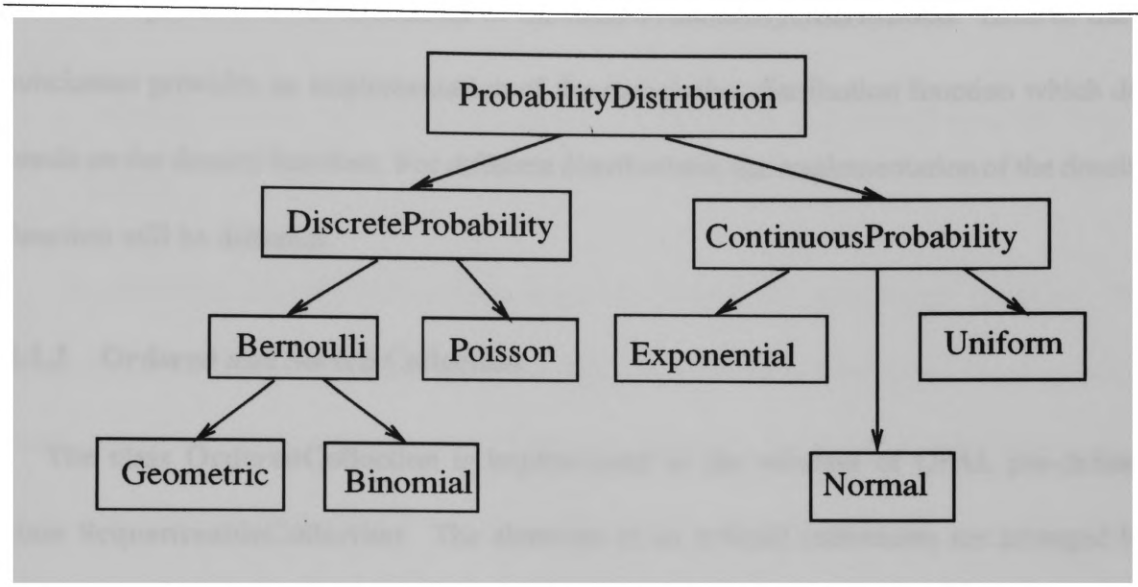

Figure 18: **The Class Hierarchy of Probability Distribution**

```
Stream subclass: 'ProbabilityDistribution'
        instVarNames: #()
        classVars: #( 'subclasses''rand')
        PoolDictionaries: #[]
        inDictionary: Statistics
        constraints: #[]
        isInvariant: false
```

Figure 19: **Definition of The Probability Distribution Class**

As shown in Figure 18, the two types of probability distribution, discrete and continuous, are specified as the subclasses of the class **ProbabilityDistribution**. Each of these subclasses provides an implementation of the cumulative distribution function which depends on the density function. For different distributions, the implementation of the density function will be different.

5.1.3 Ordered and Sorted Collection

The class **OrderedCollection** is implemented as the subclass of OPAL pre-defined class **SequenceableCollection**. The elements in an ordered collections are arranged by the sequence in which they are added and removed from the collections. The elements are accessible to external keys that are indices. On the other hand, the class **OrderedCollection** has two instance variables named *firstIndex* and *lastIndex* which indicate the position of first and last elements in the collection. The methods defined in **OrderedCollection** provide the following major mechanisms to manipulate the collection:

- Obtaining the first or last element of the collection by responding messages: **first** or **last**.
- Getting the element of the specified position by responding the message **at:**.
- Adding element to the first or last position in the collection by responding messages: **addFirst:** or **addLast:**. The method **add:** is implemented as **addLast**.

- Removing the elements from the first or last position of the collection by responding messages: **removeFirst** or **removeLast**.

- Inserting an element before the specified position by responding message **insert: before:**.

The class **SortedCollection** is implemented as the subclass of **OrderedCollection**. The elements in the **SortedCollection** are arranged by a two-argument block function called "sortCondition". The elements will be added to a sorted collection only with the message **add:**. The method **add:anObject** is implemented in such a way that *anObject* is inserted in an appropriate position in the collection with respect to the sorting function. It is also possible to initialize the sorting function or to reset the sorting function by sending corresponding messages to the instance of **SortedCollection**. When the sorting function is changed, the elements of the collection will be re-sorted, where the message *reSort* is defined to perform the collection re-sorting.

5.1.4 EventCell Class

The class **EventCell** is implemented as the subclass of **Object**, an OPAL system pre-defined class. The instance of this class represents an event in the event queue accessible to the simulation. The definition of the class EventCell is shown as Figure 20. From Figure 20, it can be seen that the class EventCell maintains four instance variables. One of these variables is *procOrblock* which keeps the name of the process associated with a

```
Object subclass: 'EventCell'
  instVarNames: #('resumeTime''type''instance'
                                  'procOrblock')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: SimulationNew
  constraints: #[#[ #resumeTime, Float],
                 #[ #type, Integer]]
  isInvariant: false
```

Figure 20: **Definition of The Class EventCell**

particular event, or a block which will be evaluated when an event is scheduled. Another instance variable named *resumeTime* stores the simulated time when an event is sequenced. The instance variable named *instance* stores the instance of participating simulation objects associated with the event, and the instance variable *type* will store an integer number indicating the type of the event. The instances of the class **EventCell** can be created by sending the **EventCell** the message **resumeTime:type:instance:proc:**, which will create an instance of the superclass and initialize the four instance variables. Furthermore, class **EventCell** provides the mechanisms such as accessing the values of the four instance variables, making comparison between two event cells and so on.

5.1.5 Process Class

As described in section 4.4.3, the class Process is designed to realize a virtual process which is appropriate to perform interruptible simulation tasks. The virtual process is

actually an OPAL class which is generated automatically each time when an instance of **Process** is created. The definition of the class **Process** is shown as Figure 21. The class

```
Object subclass: 'Process'
    instVarNames: #( 'listOfmethodName' 'className')
    classVars: #( 'numberOfInstance' 'subclasses')
    poolDictionaries: #[]
    inDictionary: SupportToSimulation
    constraints: #[  ]
    isInvariant: false
```

Figure 21: **Definition of The Class Process**

Process consists of two instance variables *className* and *listOfmethodName*. The variable *className* will store the class name generated during the instance creation. On the other hand, the variable *listOfmethodName* is an array to store all the names of the generated methods. The class variable *numberOfInstance* is used to compose the created class name. The instance of **Process** can be created by sending **Process** the message: **forContext:**, and the implementation of this method is shown as Figure 22. Before any instance is created, the process initialization has to be done by sending the message **initialize** to the class **Process** and an OPAL class named **BlockHolderSuperClass** has to be created as the super class of all the classes created by **Process**. The definition of **BlockHolderSuperClass** is shown in Figure 23, where no method is defined within the class.

From the implementation of the method forContext:aString, it can be seen that the

```
forContext: aString

  |newProcess aclassName|

   newProcess := self new.
   aclassName := String new.
   aclassName add: 'BlockHolder'.
   aclassName add: (Character withValue:
                    (64 + numberOfInstance)).
   BlockHolderSuperClass subclass: aclassName
   instVarNames: #()
   classVars: #()
   poolDictionaries: #[]
   inDictionary: UserGlobals
   constraints: #[]
   isInvariant: false.
   (numberOfInstance > 25)
   ifTrue: [ numberOfInstance := 1]
   ifFalse: [numberOfInstance := numberOfInstance + 1].
   newProcess set: aString under: aclassName.
   ^newProcess
```

Figure 22: **Implementation of The Method forContext:**

```
Object subclass: 'BlockHolderSuperClass'
    instVarNames: #()
    classVars: #( 'subclasses')
    poolDictionaries: #[]
    inDictionary: UserGlobals
    constraints: #[]
    isInvariant: false
```

Figure 23: **Definition of The Class BlockHolderSuperClass**

name of the class is automatically generated first, then the message **subclass: instVarNames: classVars: poolDictionaries: inDictionary: constraints: isInvariant:** is sent to **BlockHolderSuperClass** to create the subclass named *aclassName*. Finally, the message **set:under:** is sent to the newly created instance to construct the methods of the class. The generated class names will have the form of **BlockHolderA, BlockHolderB, ... BlockHolderZ** etc.. The definition of these classes is shown in Figure 24. The instance method

```
BlockHolderSuperClass subclass: 'BlockHolderA'
   instVarNames: #()
   classVars: #( 'subclasses')
   poolDictionaries: #[]
   inDictionary: UserGlobals
   constraints: #[]
   isInvariant: false
```

Figure 24: **Definition of The Class BlockHolderA**

set:aString under:aClassName will create methods with context of *aString* under the class *aClassName*, where each method will contain one of the statements in the string *aString*. The following example illustrates how to create a virtual process in DODESE. Suppose that the instance variable *numberOfInstance* is equal to 1, sending message **forContext: 'Simulation active startup. Simulation active finishup.'** to class **Process** will result in creating an instance of **Process** class with *BlockHolderA* and *blockA blockB* as its instance variable values. Also an OPAL class BlockHolderA is generated as defined in Figure 24, with the two class methods as in Figure 25. Under such circumstances, these statements