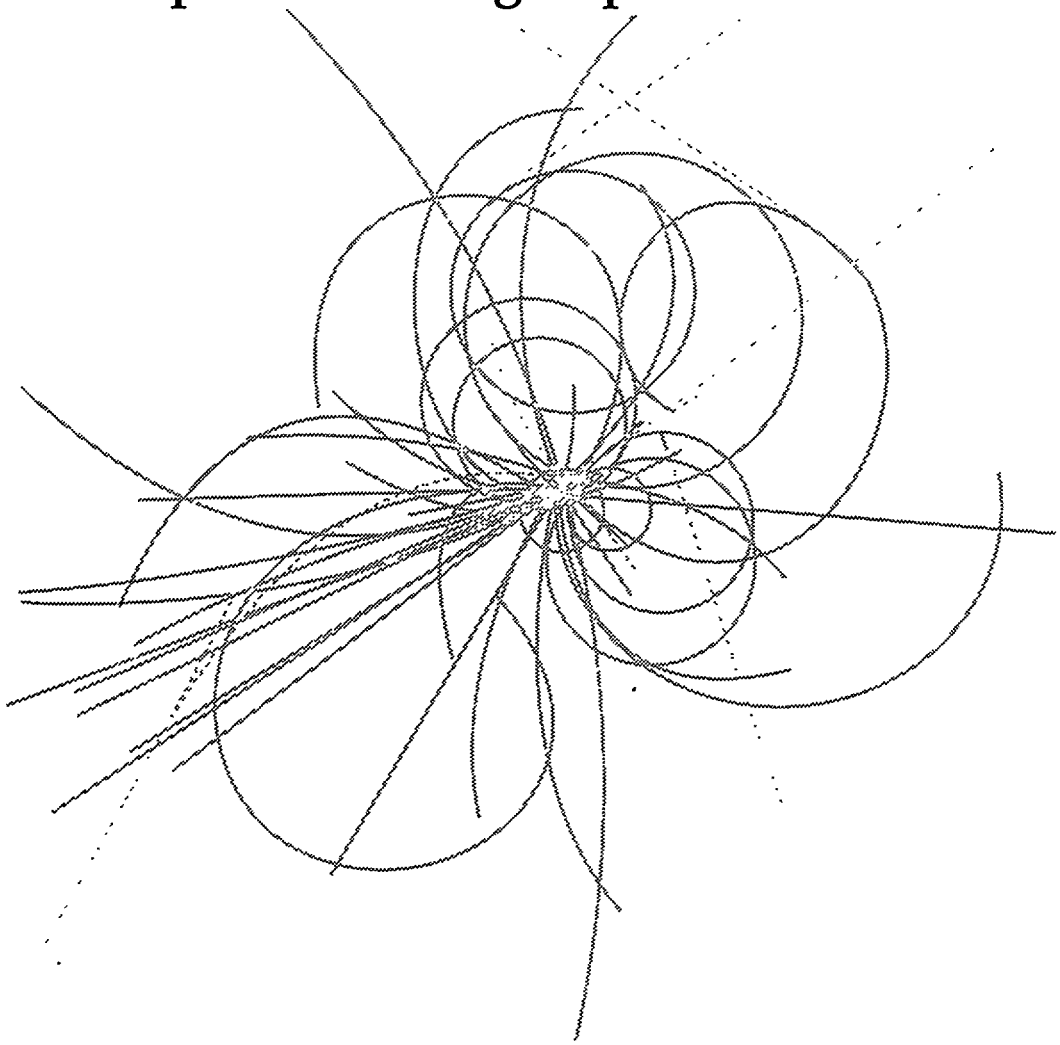
SSCL-Preprint-148

Superconducting Super Collider Laboratory

SSCL-Preprint-148



Object-Oriented Simulation for the Superconducting Super Collider

J. Zhou and M.-J. Chung

October 1992

SSCL-Preprint-148

Object-Oriented Simulation for the Superconducting Super Collider

J. Zhou

Superconducting Super Collider Laboratory*
2550 Beckleymeade Avenue
Dallas, TX 75237

and

M.-J. Chung

Department of Computer Science
Michigan State University
East Lansing, MI 48824

October 1992

* Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

OBJECT-ORIENTED SIMULATION FOR THE SUPERCONDUCTING SUPER COLLIDER

Jiasheng Zhou
Superconducting Super Collider Laboratory†
2550 Beckleymeade Avenue, MS 4011
Dallas, Texas 75237
Tel#: 214-708-3461, email: zhouj@poplar.ssc.gov

Moon-Jung Chung
Department of Computer Science
Michigan State University
East Lansing, MI 48824
Tel#: 517-353-4392, email: chung@cpswh.cps.msu.edu

ABSTRACT

This paper describes the design and implementation of an object-oriented simulation environment called OZ for the Superconducting Super Collider (SSC). The design applies object-oriented technology to data visualization, behavior modelling, dynamic simulation and version control. A meta class structure is proposed to model different types of objects in large systems by their functionality. OZ provides a direct-manipulation user interface which allows the user to visualize the data as an object in the database and interactively model the component of the system. Modelling can be exercised at different levels of the class hierarchy and then can be dynamically bound into a system for simulation. Inheritance is used to derive new configurations of the system or subsystem from the existing one, and specify an object's behavior. Delegation is used to construct a system by instantiating existing objects and "stealing" their methods by delegators.

The implementation uses C++, GLISTK[Kan91] library, InterViews 2.6[Linto90], ISTK [Saltm91] library, GNU C++ library[Stall90], GLISH event sequencer[Paxso91], NIH class library[Gorle91], and ObjectStore[Objec91].

1. INTRODUCTION

This paper describes the mechanisms used to build an integrated environment for dynamical modelling and simulation of large complex systems using object-oriented methods. The SSC is a complex machine constructed to perform high energy physics experiments. Each machine design has a configuration based on structured data residing in a database. Our goal is to build an environment which enables visualization of design data, aids interactive modelling and simulation to exercise the accelerator before it is really built. To achieve our goal, we propose an object-oriented paradigm. In our paradigm, data are modelled as objects that can be manipulated through graphical interfaces. The dynamic behavior of particles and the accelerator are modelled using these data objects in a particular configuration. Simulation results are created by applying the specified model to the simulator. Designs are managed using version control schema, and persistent object. Class hierarchy greatly facilitates the decomposition of a large complex system. Inheritance allows behavior sharing among objects while still focusing on their difference, thus making the system simpler at each level of the hierarchy. Dynamic binding makes dynamic modelling possible and delegation simplifies the aggregation of the system for simulation.

A meta class structure is proposed in this paper for organizing class hierarchies by their functionality to aid large system design (here meta class is different from SmallTalk's metaclass. We emphasize the structure and relations among classes). Deriving every class from the same root is undesirable, especially for large, complex systems such as the Super Collider. *Window* classes simply cannot be derived from *animal* classes because they are totally different in nature. A multi-level of inheritance is also confusing and low in efficiency. In a large simulation system, objects of various kinds will likely be designed, developed, and debugged in a different environment by different people in their knowledge domains. Each type of object needs its own inheritance hierarchy. The relations between these hierarchies are described by the meta class structure.

Objects in such a system can be classified into four categories:

- DATA: objects handling data transmission and providing services for modelling and simulation. These objects make the details of data transmission transparent to the rest of the system. At the SSC, data describing the structure and attributes of the accelerator (called lattice structure for each accelerator) is stored in a relational database management system (Sybase [Trahe91]). A Self-Describing Standard (*SDS*) [Saltm91] is used as a vehicle to move data structures between the application and database. DATA maps generic structured data into application oriented data for other parts of the system such as simulator and graphic plotter.
- MODELER: objects organizing the information from DATA based on their relations. It creates meta data which specify the configuration of data objects. Class hierarchy can be used to decompose a large system by their inter-component relationships: *is-a* — an attribute hierarchy, and *part-of* — an aggregation hierarchy. Delegation can be used to represent a complex system by their component structures. Class hierarchy facilitates inheritance and makes dynamic binding possible. But sometimes hierarchy structure is less efficient as the object becomes "heavier" (memory intensive). Delegation is a better way to create a "light" and "cheaper" object by instantiating its component objects through delegators (such as pointers to an object).
- SIMULATOR: objects to practice dynamic simulation. Simulation algorithms are likely to be developed independently by domain specialists. It is not necessary to design, test and debug those parts with the entire system. They can be built separately and connected to the system later. For example, it is not necessary to change the terminal each time the CPU is upgraded. For the same reason, when you design your new CPU, you don't need to worry about the type of terminal you will use if a standard interface is defined between them. Both the CPU and terminal can have their own class hierarchies and design procedures. A simulator (instance of SIMULATOR) can be built by deriving it from an existing one, or by aggregating existing ones through delegation.
- INTERFACE: objects providing a man-machine interface. It can be shared among systems with little modifications. A well-established INTERFACE class library can make interface prototyping easier and faster. A predefined look-and-feel is also important to the user to learn new applications. INTERFACE class can be built independently from its applications.

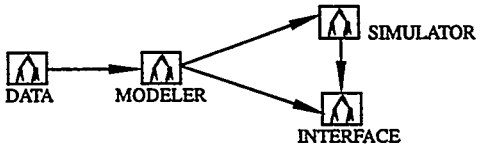


Figure 1: a meta class structure

The relations among those four class hierarchies are shown in Figure 1 (arrow points the dataflow). Each node represents one or possibly several class hierarchies. Application users can derive their own domain specific classes from high-level generic classes. A simulation application can be built by using objects from those four node libraries. Notable features of meta class structure are encapsulation and code reusability. A well-encapsulated object can be instantiated to build a more complicated object while the original object need not be modified and understood. Different applications may use similar objects to save coding effort. Once the interfaces among the nodes are clearly specified, development can proceed in parallel among class hierarchies. Independent development also makes testing and debugging

†Operated by the Universities Research Association, Inc., for the U. S. Department of Energy under Contract No. DE-AC35-89ER40486.

much easier and more efficient.

As mentioned earlier, the SSC is an accelerator built to perform high energy physics experiments. It mainly consists of magnets with various attributes. Experimental particle beams are injected from a linear accelerator (Linac), then further accelerated at different energy levels through a low energy booster (LEB), medium energy booster (MEB), and high energy booster (HEB) which are connected by beam transfer lines. Beams are then injected in opposite directions into two collider (TC and BC) rings. These 20GeV beams finally collide in the interaction region (IR).

Simulation uses both static and dynamic data. Static data created in the design are stored in the database by different versions. These data can be manipulated using a particular model to create simulation results. Dynamic data is the footprint of such results subject to a particular configuration of the lattice. So simulation is a process of manipulating static data based on a model to create dynamic data. The goals of the OZ project consist of four parts:

- A graphical browser for visualizing lattice database. This browser includes: (1) a geometric view of the accelerator complex at three dimensions (top, side, and front views) with zooming and scrolling functions, (2) a symbolic representation of the lattice structure and configuration, (3) a beamline locator which locates a beamline in the selected lattice with a name and expands it into its components, and (4) a plotter for examining various lattice optics functions.
- A dynamic optics function simulator. Users can change the attributes of the accelerator (such as initial settings), strength of the magnet and injection position of the particles. A feedback can be obtained from the dynamic optics function simulator which tells the user the effect of these changes.
- A particle tracking simulator which simulates a bunch of particles distributed in a predefined pattern passing through each accelerator several turns. It can also simulate particles passing transfer lines between accelerators. The simulator aids research in beam synchronization, timing and transfer of a trajectory within a given aperture in the accelerator.
- A basic problem in accelerator physics is how to keep beam on the correct trajectory, i.e., to avoid losing the beam. Beam is basically guided by magnets. Most magnets have fixed strength and are designed to bend the beam in a certain angle at specified locations. To correct dynamic errors which may affect the beam trajectory, hundreds of adjusting magnets (kickers) are placed among the built-in magnets. There are also hundreds of detectors (beam position monitors, BPMs) near those kickers to monitor the result of the correction and to locate the beam position. For a particular BPM reading, a model and simulator are needed to predict the adjusting value for each kicker, especially those kickers near the BPM being monitored.
- Version control for dynamic data. How to record the evolution of the design, to capture the configuration which produces an good result. Version control has two goals:

  1) Version control for attribute hierarchy to record different versions of basic building objects. For example, an accelerator can operate at different energies such as injection and collision. The magnet field strengths will be different in these different operation modes.

  2) Version control for configuration of different system or subsystem designs. For example, an accelerator can be linear or circular.

Currently, most simulation and modelling tools are designed either for small applications or static batch mode simulation. Such tools generally are not object-oriented and lack graphical and interactive capabilities. Most are not supported by object-oriented databases or persistent object management. ABLE[Rou89] is a knowledge-based simulator for particle acclerator control developed at Stanford University. ABLE does not support interactive modelling and simulation. Its simulation capability is limited to beam trajectory fitting. It is difficult to change the lattice configuration. DIMAD[Servr85] is another lattice development tool created at TRIUMF in Vancouver, Canada. DIMAD is based on FORTRAN and its graphical interface is based on C and X. It does not have the capability to directly interface with a database. It also does not support direct behavior modelling.

The rest of the paper will discuss OZ, an SSC project for doing object-oriented dynamic simulation.

2. OBJECT-ORIENTED DATABASE VISUALIZATION

Two problems need to be solved in data visualization: (1) how to retrieve data from the database, and (2) to map data for visualization. The first problem is about data modelling, and the second is about visualization.

2.1.Object-oriented data modelling

At the SSC, static data for each lattice are stored in several database tables. Each table consists of rows and columns. There is an index number (ID#) associated with each row (also called an entry, or a *record*) in the table. Each column corresponds to a particular attribute in that table. A table called GEO records all the geometrical information from the first to the last magnet through the lattice. Each magnet has an entry through GEO. Attributes could be pointers referencing to other tables that contain detailed information about that magnet such as length, and strength. Sybase is used to manage the database through multi-threaded client-server model.

Data are shipped among database and different platforms of workstations throughout the network in SDS. SDS can pack a record in a database with its attributes into a C++ structure, assemble them into an object, and load it to a SDS file. So a database table will correspond to an array of persistent structures in the SDS file. DATA object will map these SDS files into objects in C++.

Data in the database are modelled at three levels. Each level can be treated as an object derived from DATA. A data retrieval is equivalent to a message passing to that object.

- `Database` object. When a particular design version of static data is selected, the corresponding database is loaded as an object D_o. D_o actually loads a particular SDS file into memory and returns a pointer to that SDS object.
- `Table` object. If necessary, a particular table can be loaded as an object T_o. This object is dynamically created and pointed by a member variable of D_o. In SDS, the table is an array of C++ structures.
- `Column` object. An attribute (corresponding to a column) is an object A_o which can be loaded when necessary. A_o is pointed by a member variable of D_o. A_o is able to extract a particular field from an array of structures. Usually only some of the attributes are involved in the simulation at one time. Loading a database table into memory takes time and space, and is not efficient for such simulations. So making an attribute as an object is very useful.

The database itself will not provide any application-oriented data manipulation support. The main purpose for creating an object-oriented data model is to facilitate data manipulations. A standard well-encapsulated interface between DATA and other parts of the system will keep the implementation detail transparent to the user no matter what kind of database structure is to be used. Data manipulations are supported by a set of methods. Among the available methods are:

- Loading a data object: D_o, T_o, or A_o implemented by polymorphism.
- Dynamic mapping: dynamically mapping data between lattice databases. For example, particle beam tracking can go from one accelerator to another. DATA should be able to handle such shifting automatically between different databases.
- Geometrical functions: area intersection, viewpoint transformation, zooming and scrolling, symbolic creation based of magnet type, finding a magnet based on its name or type, and selecting a magnet based on its position;
- Application-oriented functions: user-defined one dimensional scaling;

These methods can be designed as virtual functions for functional or signature sharing. Methods can also be designed to fit specific needs for DATA objects at different levels.

Magnet has two kinds of attributes. Basic attributes A_b are those attributes

which can be directly retrieved from DATA objects. Composite attributes A_c are those attributes which are function of basic attributes and exclusive to a specific application.

$A_c = f(A_b^{i1}, A_b^{i2},...A_b^{ik})$, $0<i1<i2<...<ik<=n$, $0<k<=n$. n is the number of basic attributes. f is the function.

By making DATA object persistent, composite attributes can be saved in an object-oriented database (such as ObjectStore). For large lattices or complicated compositions, loading a persistent object will usually be much faster than constructing them at the beginning of the program. Persistent objects also provide record of footprint and sharable results.

ObjectStore is used in the development. DML (Data Definition and Manipulation Language) in ObjectStore can be embedded in the existing C++ code to make object persistent. Detailed implementation is based on database version control. When a new version is loaded, DATA object will compute all composite attributes A_c with available A_b and make itself a persistent object. When the same version is loaded again later. Version control will keep the freshness of these persistent data automatically. Persistent data is only validated when it is fresh, i.e., the version of the database it is created from has not yet been modified.

In DATA object, composite attributes are persistent data members. It can be accessed by other objects in the system as well as basic attributes. An attribute modification can either directly go to the persistent memory or kept in a temporary copy A_c'. These temporary copies are updated when necessary during the modelling and simulation processes. But A_c' is transient data and will not be kept after a session. User has to make the decision whether to make them persistent or not.

2.2.Object-oriented data visualization

Data visualization is handled by objects in INTERFACE. INTERFACE classes are developed using the GLISTK and InterViews libraries. There are two important components in building a graphic interface: (1) the layout of the interactive interface and connections among *control elements* such as buttons and menus, and (2) is the interactive graphics (*view*). A static non-interactive graphics is called an *image*.

- `ControlLayout` is a class to layout graphical user interface. ControlLayout is derived from class Gorgan in GLISTK which is in turn derived from InterViews' Scene.
- `ViewPlot` is another class to plot dynamic graphics controlled by ControlLayout. ViewPlot is a GLISTK which is derived from InterViews' Interactor.

Usually ControlLayout only instantiates its control element through delegation. It only keeps a pointer pointing to its control element. Control elements are created not inside the constructor but by another virtual method called CreateAndInsert(). Different versions may derive CreateAndInsert() to create and insert their own control elements. Delegation often creates a "cheap" object, which only keeps what it needs, and is more dynamic and efficient for derivation.

Control elements are derived from LabelGlistk, which has a label, size, position, state and control action. Label can be text or attached to a bitmap. State is a variable with a valid C++ type. State is associated with an object called `Communistk`. Communistk focuses on the value of the state and has a list of ControlLayout to notify when this value, or the focus, changes.

Every Communistk has a `CommuList` which is a list of ControlLayout which is informed any time the value of state that Communistk is focused on is changed by Communistk::SetValue. Every object that attaches to a Communistk is automatically added to that Communistk's CommuList. When the value that a Communistk is focused on is changed by Communistk::SetValue, the Communistk calls its `HitCommuList` method, which informs every ControlLayout in its CommuList by calling their `CommuHit` method. Such notification can also be put on hold by calling Communistk::SetValueNoHit.

A message not only can be sent back and forth between control element and ControlLayout, but also can be sent out to another application using GLISH event sequencer. For the Communistk to notify the outside world, a message must have a name, which will become a GLISH event name. An event name must be registered though GorganMaster, which is a GLISTK class derived from InterViews' World. Any change to the Communistk's focus will trigger the GorganMaster to build an event frame and message body and give it to a GLISH executive. An incoming event will be checked against registered Communistks and the indicated change, if any, will be presented to the Communistk to accept or reject and to notify its attached control element.

There are two ways to issue an action: one is deriving a specific GLISTK, for example, QuitButton, with its own *PerformAction* method. The other way is associating its Communistk with a particular ID and adding its ControlLayout to its CommuList. ControlLayout's CommuHit will be called when the Communistk value is changed. CommuHit can control the action based on CommuID. Table 1 summarizes methods provided by ControlLayout.

Table 1: Method in class ControlLayout

method name	function description
CreateAndInsert()	create control elements and insert them in a proper form using alignment variables. ControlLayout records these alignments in a table and possible reposition and resize.
RaiseAndLower()	element popup control, such as popup menu and popup message, dialogue, etc.
LockAndUnlock()	provides availability control to control elements.
CommuHit()	provides communication among objects, including between ControlLayouts and between its elements.
StopInput()	Some actions need a start event to enter its mode and wait for end event to exit its mode. Such action should be registered with ControlLayout. Then the end event can be directed to its target by StopInput()

ViewPlot is derived from LabelGlistk. It provides a dynamic graphic view of objects from the database.

After a particular lattice has been loaded, the position and size of each object can be extracted from DATA object. ViewPlot scales these data based on current plotting size and displays the object on the screen.

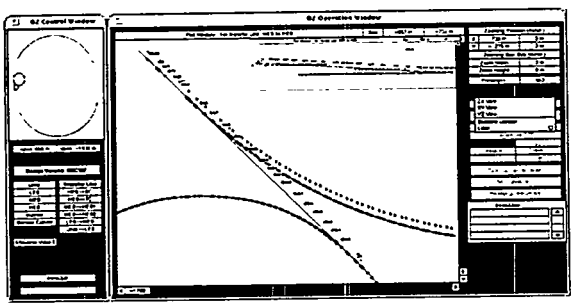
When resize occurs, ViewPlot will rescale the position and size of all objects and replot them. The whole plot can be zoomed in. When a plot is zoomed in, the plot boundaries are pushed on the stack. Zooming boundaries become new boundaries for the new plot. ViewPlot rescales the new plot based on new boundaries. Undo operation is equivalent to a zoom out. The previous boundaries will be popped up from the stack and become the current boundary.

Space is dynamically allocated for all plotting data used in the preceding method. In order to keep all information available for redraw, space is only deleted when new data are loaded in from DATA object. Figure 2 shows some examples.

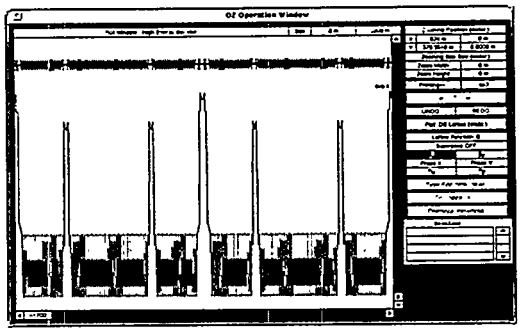
By taking advantage of dynamic binding of C++ virtual function, all methods for graphic manipulation are virtual. A zooming operation on an optics function plotting will cause a one-dimensional zoom in. The same operation on a geometrical representation of an accelerator will cause a two-dimensional zoom in. If several plots have to be zoomed in simultaneously with the same scaling, a virtual function call of zoom operation on all these plots will work polymorphically.

Most dynamic graphics requires incremental drawing. The result of several simulations can be superposed or plotted in different areas of the screen one-by-one at different times. But what will happen if the window is closed and opened later? The current image on the screen should be "remembered" so when the window is opened later, the previous image can be restored as is. It is not realistic to repeat all the simulation again to recreate these images.

A feasible solution is to create an incremental drawing queue IDQ to record incremental drawing data. Two methods are used for drawing. *Refresh* handles initial drawing such as legend, measurement, symbolic representation and marks. We call these graphics static graphics and they should be always on the screen. To draw something dynamic on the screen, call *Draw*



A zoom in at the transfer line region between MEB and HEB. A side view is in the up right corner.



A static β function plot for the entire top collider (TC). A central region blowup is given to its right

Figure 2. Examples of data visualizations

and push data into the IDQ. Draw will pick up the data from the top of the IDQ and draw it on the screen. If the window is closed and then opened again, then Refresh will get called. Refresh will in turn call Draw to accumulative draw everything in the IDQ, if any. Figure 3 illustrates how incremental drawing queue works.

```
ViewPlot::Refresh(){
  MapRawDataToDrawableData();
  DrawStaticData();
  if (SizeOfIDQ>0)
     Draw();
}
ViewPlot::Draw(){
  If (AnythingNewInIDQ())
     DrawViewAtTheTopOfTheStack();
}
```

```
ViewPlot::CreateDynamicData(){
  CreateSimulationResults();
  SizeOfIDQ++;
  RegisterToIDQ();
  Push(IDQ, CurrentDrawingData);
  Draw();
}
```

Figure 3. How to do incremental drawing

Because of a large number of magnets (thousands) may need to be drawn on the screen, making each magnet as a structured graphics object in InterViews is not realistic. If we make the whole accelerator as an object, then it is difficult to pinpoint an individual magnet object. A feasible solution is to make the whole accelerator as an object. At the same time, design a set of methods to do the mapping among objects on the screen, their ID# in ViewPlot, and their data in DATA. Figure 4 shows such a mapping:

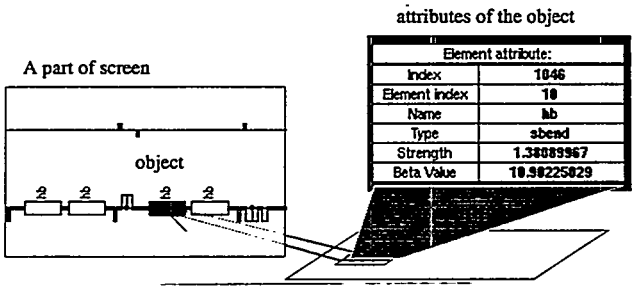


Figure 4. Object mapping.

In ViewPlot, the screen position of each object gets registered when it is drawn. A mouse down event catches an object if it occurs within the sensitive boundary of that object on the screen. ViewPlot keeps a list of all types of mouse-sensitive objects, such as magnet, adjuster, and detector. Sensitivity can also be screened out. A caught object is called a focusing object O_f. ViewPlot will do a binary search within the current plotting boundaries to find ID#(O_f). Then all information of that O_f can be found through DATA. Some member functions of ViewPlot are listed in Table 2:

Table 2: Members and methods in class ViewPlot

name	function
myModel	pointer pointing to MODELER object
realBoundary	the real dimension of visual target. For example, optics function of HEB

Table 2: Members and methods in class ViewPlot

name	function
visualBoundary	current dimension of the visual target. This is used by zooming and strolling
StretchAndFit()	stretch the view and fit it to the size of the window.
CatchAndZoom()	handle zooming base on size of the rubber box created by a mouse down.
Scroll(currentPosition)	handle scrolling from current position to a new position.
Redo(), Undo()	handle unzooming
EventHandler(*event*)	for *event*, there is a event handler.
Refresh()	Refresh() handles initial drawings. It keeps a pointer an object called IncrementalDrawingQueue. Refresh will call Draw if there is anything in the queue. Dynamic drawing is handled by Draw().
Draw()	Handle add on (or called incremental) drawing.
ID# Find(position)	return object ID# based on its current registered position
ShowValue(ID#)	show attributes of the object with ID# (focusing object O_f)

3. MODELLING DYNAMIC BEHAVIOR

A *model* is an abstraction (possibly a mathematical abstraction) of something for the purpose of understanding it before building it [Rumba91]. Because a model usually focuses on some essential issues of the simulation entity, it is easier to manipulate and simulate than the original entity.

Object-oriented modelling abstracts an entity in the real system as an object. It is natural to represent entities in an application domain as objects which respond to a well defined set of messages. For example, in an accelerator system model, domain objects might be magnets, particles, and accelerators. New types of objects may be created by specializing existing ones. Complex systems can be modelled with composite objects (also called submodels) and can be used in other models. A model as a whole is itself a composite object which responds to a set of messages. Object's *identity* is modelled as member variables (also called *attributes*). The tolerant threshold towards certain attributes is called *constraints*, which is defined as a function f_c of some attributes for a particular object, $C_o = f_c(A_o)$. *Behavior* of the object is modelled as a set of methods M_b, which is a function of attributes A_o and constraints C_o based on algorithms developed domain specifically. Dynamic behavior describes those aspects of the object concerned with time and sequencing of operations, such as events that mark changes, sequences of events, states that define the context of events, and the configuration of the system where object is placed.

In this paper, we emphasize the difference among the following concepts [Zeigl90]:

- the *real system*, in existence or proposed, which is regarded as fundamentally a source of data, which in OZ is provided by DATA.
- the *model*, which is a set of methods for generating dynamic data to that

observable in the real system. The structure of the model is its set of methods. The behavior of the model is the set of all possible data that can be generated by faithfully executing the model methods.

- the *simulator* which exercises the model's methods to actually generate its behavior.

At the SSC, there are three objects to be modelled. The particle beam, the magnet in the accelerator, and the accelerator itself.

The behavior of a particle (proton at the SSC) depends on its momentum, position and distribution of magnet field strength around it. Particle momentum and magnet strength distribution are decided by the accelerator a particle is passing through. So the behavior of a bunch of particles (beam) will be more interesting. *Particle distribution hierarchy* (PDH) is used to record such modelling.

The root class Beam has only one particle and it is placed at the origin. Some standard statistical distribution with a certain number of particles are its subclass, such as normal and average distributions. Beam has 5 instance variables listed in Table 3:

Table 3: Instance variables in beam class

instance variable name	illustration
num	number of particle in the beam
Position *pos[num]	position of those particle, displacement *d*
Deflection *dp[num]	angular deflection of the particle *d'*
Deviation *delta[num]	momentum deviation of the particle δ
distribution form	statistic distribution of those particles.

Vector D = [d, d', δ]is called principle vector (PV). Beam object can be created by using beam class library with a graphic interface. Either by picking up an existing beam from the library or creating one by rearranging the particle distribution or changing the amount of the particles (Figure 5), a new beam configuration can be created. A new beam can also be created as a result of beam tracking simulation.

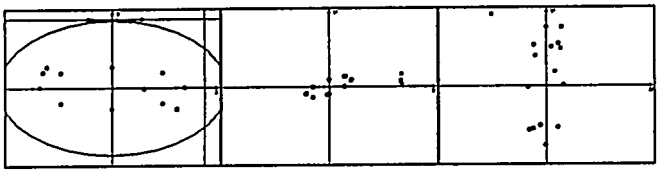


Figure 5 beam objects created from PDH

After beam is created, it is sent to an acceleration pattern (which is the logical path from its launch position to its end observing position through accelerators) for simulation. The momentum will be dynamically bound to the particle when passing though the corresponding accelerator.

The behavior of the magnet depends on its magnet type(*t*), magnet strength(*s*), length(*l*), tilt(*o*), linearity(*m*), its optics functions (such as β function), its phase advantage (Φ) and other attributes. Magnet attribute hierarchy is used to record such modelling.

The principle magnet hierarchy(PMH) is shown in Figure 6.

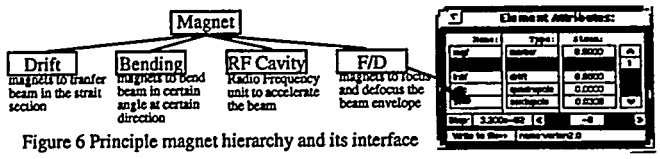


Figure 6 Principle magnet hierarchy and its interface

A prototype of the magnet attributes modelling system is shown in Figure 4, 6. Magnet instances are represented by a collection of icons (Figure 4). A magnet class is represented by a list of attributes (Figure 6). Different models of magnets are constructed from their own class category using this interface. After a sub-model is created in the hierarchy, it is added back to the list as a part of the new hierarchy.

Based on Steffens' theory[Steff85], the behavior of the magnet can be modelled as a 3 by 3 transformation matrix M(t, s, l, o, m). M is defined as a function of *t, s, l, o,* and *m* for a particular magnet. D_i and D_{i+1} are the principle vectors of a particle at position *i* and *i+1* respectively. And we have $D_{i+1} = M \cdot D_i$, i.e.:

$$\begin{bmatrix} d_{i+1} \\ d'_{i+1} \\ \delta_{i+1} \end{bmatrix} = \begin{bmatrix} C & S & D \\ C' & S' & D' \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} d_i \\ d'_i \\ \delta_i \end{bmatrix}$$

$$\text{Vector } D, \begin{bmatrix} d_i \\ d'_i \\ \delta_i \end{bmatrix} is...called...principle...vector\,(PV) \begin{bmatrix} d_i, displacement \\ d'_i, deflection \\ \delta_i, deviation \end{bmatrix}$$

Magnet class definition is partially given in Table 4:

Table 4: Magnet class

members	function illustration
Class category	There are four categories: *drift, bending magnet, RF cavities* and *focus/defocusing magnets.*
Attributes *myAttributes	Attributes is a C++ class with all attributes: basic and composite
virtual Matrix* CreateTM()	Create transformation matrix for that magnet
virtual PV* BehaviorMap()	create a result principle vector (PV) from the previous one.

All principle magnet classes are predefined and created from DATA. Each Magnet instance has a pointer pointing to a Magnet class in the PMH. When new a Magnet class has to be created, a particular Magnet instance will be selected. By changing the proper attributes, a new class will be created with that magnet as its first instance. The new class will inherits all methods from its parent, such as CreateTM and BehaviorMap.

Behavior modelling is supported by two approaches:

1) A text window is provided for examining and overriding the previous behavior model (such as method BehaviorMap) by using C++ code. Behavior binding is implemented by taking advantage of dynamic binding of C++ virtual function. New C++ code has to be recompiled and linked into the system and then the whole process needs to be restarted.

2) Several models (such as linear and nonlinear method) can be predefined based on knowledge and domain specific rules. Virtual function dynamically bind the rule number (set by the user through interface) with a pointer of a member function to construct behavior. Interactive modelling basically becomes rule-picking and function binding. Rules can also be added off-line by using C++ code.

Modelling accelerator uses configuration binding techniques. An accelerator can be decomposed into *beamline*, which is a set of magnets placed in a specific order as a design component. Accelerator is on the top of this configuration hierarchy. It is decomposed into major beamlines. And these major beamlines are further decomposed into smaller beamlines, which are in turn decomposed all the way to the magnet level. Such a structure hierarchy is called a lattice configuration for an accelerator.

The class Beamline is derived from the base class DList, which implements a doubly linked list (NIHCL and GNU all have that type of class). Beamline holds a pointer pointing to its component, which may be smaller beamlines or magnets. Beamline class is also derived from Magnet class that makes it easy to insert in, replace by another beamline or magnet. Multiple inheritance (Figure 7) is used to make such a class possible.

Beamline inherits all members and methods from Magnet. But Beamline has its own methods to specify its structure. Members and methods of

Figure 7 Beamline: multiple inheritance from Magnet and DList

`Beamline` are listed in Table 5.

Table 5: Beamline class

members	function illustration
Beamline* bmLnElmnt;	bmLnElmnt point to the current component (smaller beamline or magnet)
Insert(WhichSide);	All these methods are inherited from DList class. Insert() inserts a beamline before/after (depends on the value of WhichSide) the current beamline. Replace() and Delete() replaces and deletes the current beamline. Get() moves the bmLnElmnt to another beamline.
Replace(Position, Beamline*);	
Delete(Position);	
Get(Position);	
virtual Tracking(Particle*)	Beamline's own method, which accepts a particle (or beam that is derived from particle) object as its argument, does straightforward, magnet-by-magnet tracking at the bottom of the configuration hierarchy through the beamline. The keyword "virtual" means that each beamline or magnet object must implement such a method. One of the extraordinarily useful features of the virtual method is that it allows us to perform polymorphism on all kinds of beamline and magnet which is a component.

A new lattice configuration can be created by replacing a beamline by a new design. In Figure 8, a new design for the beamline *triwm'* creates a new configuration for the LEB. Configuration binding is deferred at the simulation stage and the binding actually occurs at the bottom level of this hierarchy, i. e. magnet level. Configuration binding will also be discussed in version control later.

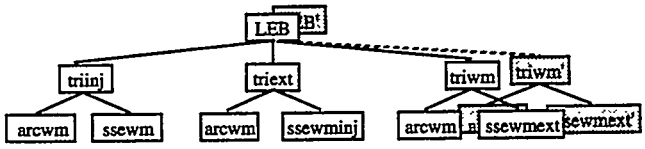


Figure 8. Lattice configuration hierarchy

4. DYNAMIC SIMULATION

SIMULATOR is a class which exercises the model's methods to actually generate its dynamic behavior. In OZ, simulation might deal with several kinds of models, such as beam model, magnet model and lattices configuration model. The behavior of each object in the interactive simulation process is important to adjust the model for better performance. When DATA and MODELER are created, simulator (an instance of SIMULATOR) will be triggered to launch the simulation. Simulator is the manager of the entire simulation. Objects are controlled under simulator to interact with each other to create dynamic behavior. Figure 9. gives an example for "BumpView" simulation, which meets the fourth goal of the OZ system.

The bottom part of the window is a graphical representation of the lattice structure of the LEB. Above it is the graphical representation of the detectors and adjustors along the LEB. All objects in the representation are active (sensible and associated with actions). In the middle of the window is the dynamic aperture of the LEB which basically depends on the attributes of the magnet at each point. The middle part is expanded at the up right corner. The dashed bar is the BPM reading set by the user. A bumpview simulation will give the following:

- Three white points (actually a three green bar) stands for the settings of three kickers around that BPM. The actually values are given as deltaX_', deltaX', deltaX_+' in the "Adjuster settings" box at the bottom of the control panel.
- To make modelling simpler, we assume that the adjusting will only affect 3 BPM readings nearest to the BPM selected. All other BPM should have zero readings. The simulation proves the model is correct. From the picture, there are only 3 solid bars in the middle.

The up part of the window is the β-tron oscillation along the LEB.

Dynamic simulation allows user to pinpoint objects (magnets) in the lattice, modify their attributes and rerun the simulator interactively.

Figure 10 and 11 gives more examples of dynamic simulations. In Figure 10. *a* is the optics function of LEB created by `Twiss` derived from SIMULATOR. *b* and *c* are dynamic particle tracking by turns or by every magnet using `Track`. *d* is dynamic tracking of a beam created from beam class hierarchy by using `Emit`, which is also a simulator from SIMULATOR class hierarchy. `Emit` can also be used to aid the research of relations between particle distribution in the beam and beam survivability.

A particle could be lost during the acceleration. It is important to know where it is lost in order to make the correction by using BumpView simulator. Figure 11 gives such an example.

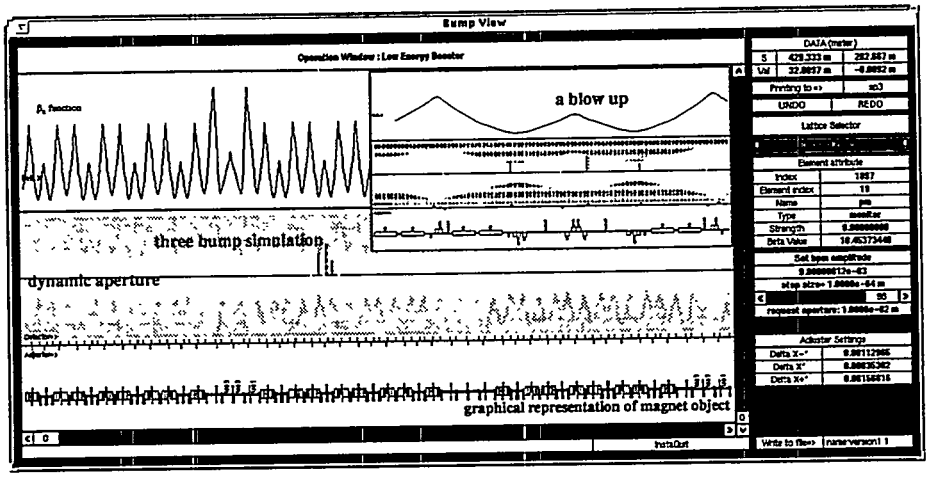
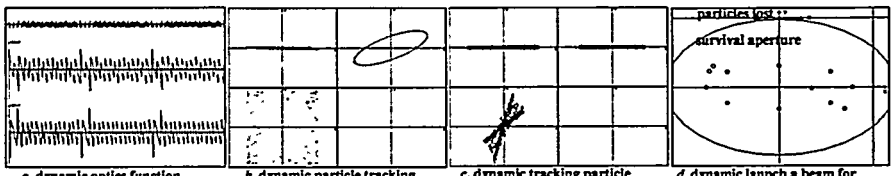


Figure 9. 3-bump simulation using Bump View simulator



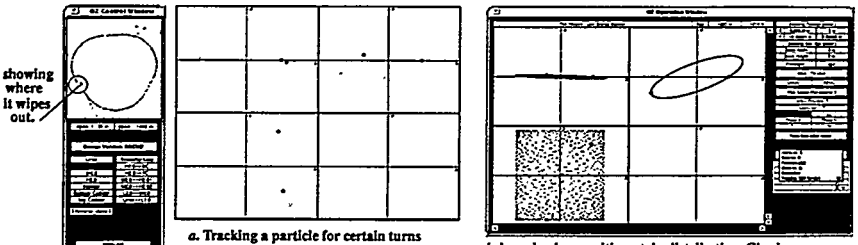
a. dynamic optics function creation of the MEB. Here is α function

b. dynamic particle tracking in LEB for 100 turns and recording its d and d' after each turn.

c. dynamic tracking particle for one ture and recording its d and d' after each magnet.

d. dynamic launch a beam for 10 turns to see how many particles are still in the survival aperture. Beam is in solid dot. Survival particle is is small circle.

Figure 10. Dynamic simulation



a. Tracking a particle for certain turns to check its survivability under the current lattice configuration.

b. launch a beam with certain distribution. Check its distribution after several turns to research the relation with lattice configuration and magnet attributes.

Figure 11. Beam survivability research

5. VERSION MANAGEMENT AND RELEASE CONTROL

As the design complexity increases, OZ needs to support cooperative work by a number of physicists on the same design. As a result, a requirement has emerged for version management mechanisms that record every stage of a design, merge individual designs to a complete design and make alternative designs available. Version management and release control needs to deal with the following issues:

- how to document an alternative design?
- how to document the design evolution?
- how to merge individual design into a complete design?

In previous sections, magnet attribute hierarchy and lattice configuration hierarchy have been mentioned for dynamic behavior modelling. This section will discuss how to manage these hierarchies.

OZ version management system is a version hierarchy (Figure 12). *VersionManager* sits at the top of this hierarchy to manage the version control.

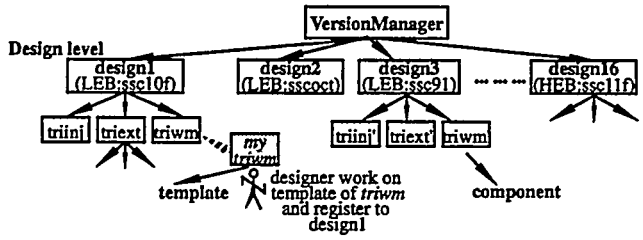


Figure 12. Version management hierarchy

A *design* in this paper is a generic configuration shared by designers. it is implemented as a top level Version Management Unit (VMU) for the purposes of versioning. For example, the LEB of the SSC has three different designs: SSC10F, SSCOCT and SSC91. Each designer has to *register* to a design. Registration makes the current work tied with a specific design. *Component* is also a VMU which sits below the design and it is used to built up to a design. *Configuration* is the internal structure or attributes (if at the bottom level of the version hierarchy) of a component. *Template* is a working space for *private* development on a particular component to yield a alternative configuration.

Each designer can create a template at each component node. The component in the node is frozen as "official" configuration(OC). When a template is created, a copy of the OC is checked out into the template. Template can be made public to form an alternative configuration library. A configuration also can be checked into the node and frozen to be official. Currently check-in is controlled by the VersionManager through the design the designer registered.

By using version management hierarchy, alternative designs (configurations) by different designers are kept in their own templates. A design is originally created in the MODELER and then put into the version management hierarchy. VersionManager records its creator, and check-in date. The individual designer will be assigned to each of its component to exercise the configuration using modelling and simulation in its own template. Release control is through dynamic configuration binding. The component will be bound to its OC by default if there is no template created by the designer at that node, otherwise it will be bound to the current configuration the designer is working on. As a matter of fact, that configuration is derived from OC. When a new configuration is created in the template, the pointer (which is inside its parent configuration) pointing to the OC or previous configuration will "float" to the new one. Then all messages sent to the configuration pointed by that pointer will be dynamically bound to the new configuration. The new configuration can be checked into the node as OC, made public as an alternative of OC along with a performance testing report for reference, or just left in the template for further development.

A configuration is usually created from the bottom using magnets available. A configuration may instantiate sub-configurations as its components by picking up candidates from the component library consisting of OC (default) and all its public alternatives, or just from the scratch (magnet). A complete design is created in such a bottom-up fashion. A new configuration can also be created by inheritance. A special version of a component can be naturally derived from existing one by using a template.

Our version management mechanism supports configuration hierarchy and multiple configurations in the template. Cross referencing between different designs are also available. It also allows deriving classes directly from version management hierarchy and sharing its capability through multiple inheritance. Therefore, version management can be handled automatically.

Version management and release control are still in prototyping stage. Future research includes managing concurrent check-out configurations, and a graphic interface for version management.

6. CONCLUSION

In this paper, we have described our experience with designing and implementing an object-oriented simulation environment OZ. The issues of building a generalized simulation system have been addressed by proposing a meta class structure which decomposes a system into four types of classes that handles data acquisition, user interface, modelling and simulation respectively.

In our object-oriented data modelling, data, meta data, and procedures that handle data accessing and manipulation are combined together as an object. Data as an object is able to describe itself and provide information to the modelling and simulation. Data object has its view which can be directly manipulated through a graphic user interface. A dynamic system can be decomposed into objects with dynamic behaviors. Attributes and constraints are used to model dynamic behavior of the object with a class hierarchy. Attributes and constraints can be dynamically bound to an object in an inheritance hierarchy. Different configuration can also be dynamically bound to an object through configuration hierarchy. Simulation can be exercised using a particular configuration with data objects as parameters in our modelling system. Version management and release control, which are important aspects in dynamic modelling, are implemented using configuration hierarchies and persistent objects.

OZ has been implemented and currently available on a local network of Unix and X based workstations at the SSC Laboratory. We used the same approach presented to prototype the BumpView, which is an extension to OZ for dynamic simulation. With the experience we had in developing OZ, it took us only one month to finish the prototyping of BumpView. The results achieved with our current effort have been encouraging and lead us to believe that the object-oriented approach will provides us more flexibility and extensibility. We plan to extend our effort to build a generalized framework for building more simulation tools for the SSC.

Because of a user friendly interactive graphic interface, most of the physicists in the lab who used to FORTRAN have started to use OZ to design and perform the simulation. An integrated object-oriented simulation environment such as OZ will help the design and simulation of the SSC.

7. ACKNOWLEDGMENTS

We greatly appreciate the valuable contributions and advice provided by Dr. Richard Talman, Dr. Garry Trahern, Dr. George Bourianoff and Ellen Syphers at the Accelerator System Division of the SSC Laboratory. We also appreciate the help from Dr. Chris Saltmarsh, and Matt Fryer at Laurence-Berkeley Laboratory and Matthew Kan at Carnegie Mellon University.

References

[Beaum90] Beaumariage, T.; Mize, J. H.: "Object-oriented modeling: concepts and ongoing research", Proceedings of the SCS Multiconference on Object-Oriented Simulation, pp. 7-12, San Diego, CA, 1990.

[Butler91] Butler, G.F.; Corbin, M.J.: "Introduction to Object-oriented Simulation", IEE Colloquium on "object-oriented Simulation and Control", Digest No. 057, pp. 1/1-3, London, UK, 1991

[Gobbe91] Gobbetti, Enrico; Turner, Russell, "Object-oriented Design of Dynamic Graphics Application", Computer Graphics Laboratory, Swiss Federal Institute of Technology, Lausanne, New Trends in Animation and Visualization, John Wiley & Sons, 1991

[Gorle91] Gorlen, Keith; Orlow, Sanford; Plexico, Perry: "Data Abstraction and Object-Oriented Programming in C++", March, 1991, John Wiley & Sons

[Ege88] Ege, Raimund K.: "Constraint-based user interfaces for simulation", Proceedings of 1988 Winter Simulation Conference, pp263-271.

[Kan91] Kan, Matthew: "GLISTK: Graphic Library for the Integrated Scientific Tool Kit", Laurence-Berkeley Laboratory, March, 1991

[Klahr86] Klahr, P.: "Expressibility in Ross: an object-oriented simulation system", AI Applied to Simulation, Proceeding of the European Conference, SCS, San Diego, CA. pp. 136-139

[Linto90] Linton, Mark: "InterViews Reference Manual", Version 2.6, Computer Systems Laboratory, Stanford University, Feb., 1990

[Objec91] Object Design, Inc.: "ObjectStore User Guide", Release 1.1, March, 1991

[Page89] Page, Thomas W.; Berson, Steven: "An Object-oriented Modeling Environment", Proceedings of 1989 OOPSLA Conference. pp. 287-296

[Paxso91] Paxson, Vern: "Reference Manual for the Glish Sequencing Language", Laurence-Berkeley Laboratory, April. 16, 1991.

[Rober88] Robert, S. D.; Heim, J.: "A perspective on object-oriented simulation", 1988 Winter Simulation Conference Proceedings, pp. 277-281, San Diego, CA, 1988.

[Robin89] Robinson, J. T.; Kisner, R. A.: "An intelligent dynamic simulation environment: an object-oriented approach", Proceedings IEEE International Symposium on Intelligent Control, 1988, Washington, D. C., pp 687-692

[Rou89] Round, Alfred: Knowledge-based Simulation, "The Handbook of Artificial Intelligence", Volume IV, Chapter XXII, Addison-Wesley Publishing Company, 1989.

[Rumba91] Rumbaugh, James; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. General Electric Co.: "Object-Oriented Modeling and Design", Prentice Hall, 1991

[Stall90] Stallman, Richard: "Using and Porting GNU CC", version 1.37.1, Feb. 21, 1990

[Saltm91] Saltmarsh, Chris: "The SDS Document: A Conceptual Basic Towards Understanding the Self-Describing Data Standard", Laurence-Berkeley Laboratory, Dec. 1, 1991.

[Steff85] Steffen, K.: "Basic Course on Accelerator Optics", DESY HERA 85/10, Deutsches Ele, ktronen-Synchrotron DESY, Hamburg, March, 1985.

[Steph91] Stephanie, J. C., Burdorf, Christopher: "PSE: an object-oriented simulation environment support persistence", Journal of Object-Oriented Programming, Oct., 1991, pp 30-40.

[Servr85] Servranckx, Roger; Brown, Karl; Schachinger, Lindsay; Douglas, David: "User Guide to the Program DIMAD", Stanford Linear Accelerator Center, Report 285 UC-28(A) May, 1985

[Tang91] Tang, Ming Xi; Smithers, T: "Towards object-oriented simulation", IEE Colloquium on "object-oriented Simulation and Control", Digest No. 057, pp. 7/1-4, London, UK, 1991

[Trahe91] Trahern, Garry; Zhou, Jiasheng: "SSC Lattice Database and Graphical Interface", 1991 International Conference on Accelerator and Large Experimental Physics Control Systems, KEK, Japan, Nov, 1991.

[Zeigl90] Zeigler, Bernard P. : "Object-Oriented Simulation with Hierarchical, Modular Models", Academic Press, 1990.

Superconducting Super Collider Laboratory is an equal opportunity employer.