SSCL-N-813

Object-Oriented Simulation for the Superconducting Super Collider

J. Zhou

Superconducting Super Collider Laboratory*
2550 Beckleymeade Ave.
Dallas, TX 75237

M-J. Chung

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

March 1993

*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@cps.msu.edu

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

ABSTRACT

The design and implementation of an object-oriented simulation environment, OZ, for the Superconducting Super Collider (SSC) Laboratory is described in this paper. The design applies object-oriented technology to data management and visualization, behavior modeling, and dynamic simulation. A Meta Class Model (MCM) is proposed to model different types of objects in large systems by their functionality. Our MCM support encapsulation, code reuse, and a loosely coupled development approach. A meta class is a complete set of domain-specific classes that are cohesive and self-contained to fulfill particular responsibilities in a specific domain. It provides four conceptual layers in the design of a simulation environment. The design of each meta class can proceed independently, targeting the responsibilities and protocols of each meta class. Our goal is to accumulatively create a complete functionality for each layer for reuse in future software development.

OZ provides a graphical user interface that allows the user to visualize the design data as objects in the database and to interactively model system components through direct manipulation. Modeling can be exercised at different levels of the system decomposition hierarchy before it is dynamically bound into a system for simulation. Inheritance is used to derive new behavior of the system or subsystem from the existing one.

The implementation uses C++, GLISTK library, InterViews 2.6, ISTK library, GNU C++ library, and the ObjectStore database management system.

Keywords: Meta Class, aggregation hierarchy, generalization-specialization hierarchy, object-oriented decomposition.

1. INTRODUCTION

This paper describes the mechanisms used to build an integrated environment for dynamic modeling and simulation of large complex systems using object-oriented approaches. This mechanism has been applied to the development of OZ, a project for dynamic simulation at the Superconducting Super Collider (SSC) Laboratory. The goal of this project is to build an environment that enables visualization of design data, aids interactive modeling and simulation to exercise the SSC before it is actually built. We developed an object-oriented data model for SSC simulation. A dynamic simulation paradigm is proposed and implemented based on our data model.

The SSC is an accelerator built to perform high-energy physics experiments. It mainly consists of magnets with various attributes. Each machine design has a configuration based on structured data residing in databases. 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 a top collider ring (TC) and a bottom collider ring (BC) (See Figure1). These 20-GeV beams finally collide in the interaction region (IR).

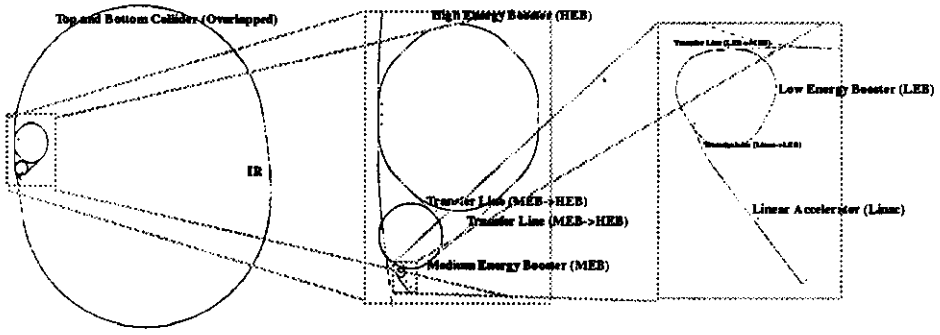


Figure 1. The Configuration of the SSC

Simulation at the SSC Laboratory uses both static and dynamic data. Static data created in the accelerator *lattice* (the layout structure of magnets in an accelerator) design are stored in the database. These data can be manipulated using a particular simulation model to create simulation results. Dynamic data is the footprint of such results subject to a particular configuration of the accelerator lattice. So simulation is a process of manipulating static data based on a simulation

model to create dynamic data. The OZ project at the SSC has three goals:

1. A graphical browser for visualizing the accelerator lattice database. This browser includes: (1) a geometric view of the accelerator complex in three dimensions, (2) a symbolic representation of the lattice structure and configuration, (3) a beamline locator, which locates a section of an accelerator in the selected lattice with a name and expands it into its components, and (4) a plotter for examining various lattice optics functions.

2. A dynamic optics function simulator. Users can change some 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 effect of these changes.

3. A particle tracking simulator. The simulator simulates a bunch of particles distributed in a predefined pattern passing through each accelerator for several turns. It can also simulate particles passing through transfer lines between accelerators with different energy levels. The simulator aids research in beam synchronization, timing, and transfer of a trajectory within a given aperture in the accelerator.

Currently, most simulation and modeling tools are designed for either 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 by a persistent object management system with a dynamic model. ABLE [Round89] is a knowledge-based simulator for particle accelerator control developed at Stanford Linear Accelerator Center (SLAC). ABLE does not support interactive modeling and simulation. Its simulation capability is limited to beam trajectory fitting. It is difficult to change the lattice configuration of an accelerator at a component level for simulation. DIMAD [SerBr85] is another accelerator lattice development tool created at TRIUMF National Laboratory 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 an on-line database. It also does not support direct behavior modeling through the object itself. In both ABLE and DIMAD, data model and application are tightly coupled because of less encapsulation. System decomposition based on procedure rather than object makes dynamic modeling difficult.

In project OZ, we have developed an object-oriented paradigm for data modeling and simulation. In our paradigm, the *system* (SSC) is logically decomposed into its *components*. These components are modeled as objects that can be manipulated through graphical user interfaces. The objects can be decomposed repeatedly until the necessary granularity is reached in terms of the design requirements, yielding an object aggregation hierarchy. By examining and grouping objects at the component level, we can get class hierarchies, in which common features are shared and differences are derived in a generalization specialization process among classes.

The concept of *meta class* is introduced to tie a group of cohesive classes which share certain common aspects in the modeling, such as data source, magnet and simulator. Meta Class Model (MCM) imposes a set of protocols between meta classes. MCM facilitates incremental development for simulation software, and promotes encapsulation and code reuse. Our MCM allows not only to build the OZ project with maintainability and extensibility but also to bring reusable software constructs to be shared by successive projects.

The object-oriented approach has been used for simulation since the programming language Simula [DahNy66]. Zeigler developed a formal system called DEVS-Scheme [Zeigl90] for modeling discrete event simulation. In DEVS-Scheme, model and processors, the main subclasses of the universal class entities, provide the basic constructs needed for modeling and simulation. Models and processors are abstract classes that serve to provide basic slots needed by their specializations. Atomic- and coupled-model are two major subclasses of model for realizing the atomic-level model and embodying the hierarchical model. At the SSC, we obtain data from a heterogenous source and we manipulate and view the data in a fully interactive and distributed environment. In such an environment, object-oriented tools have to be available for multi-domain development such as data analysis, graphic-based editing, and rapid prototyping [Niels91].

The Meta Class Model proposed in this paper is influenced by the work of Wirfs-Brock's [WirWi91]. Her Responsibility-Driven Design approach stresses focusing on functional decomposition of complexities in object-oriented modeling. The concept of contract, a cohesive set of related responsibilities defined by a class, is introduced to model objects and their relationships. We expand the idea of contract to a cohesive group of classes (meta classes) and call these contracts protocols. The horizontal layer concept from Coad and Yourdon [CoaYo91] also

helped us to divide a system into problem domain layer, a meta class.

The remainder of the paper will discuss our Meta Class Model and its implementation in the context of the OZ project. Section 2 will discuss the conceptual design of the MCM. Section 3 will describe the data model in MCM. Sections 4 and 5 illustrate the modeling of dynamic behavior and simulation in OZ. Section 6 will discuss graphical data visualization. We reach our conclusion at section 7.

2. CONCEPTUAL DESIGN

The target of object-oriented software development is the object-oriented decomposition of user's needs into executable language constructs [AkBer92]. In object-oriented decomposition, same objects can be grouped into classes and similar classes can be combined into a class inheritance hierarchy where the common features are shared and individual characters are derived through generalization-specialization hierarchy. The process ends up with several class hierarchies, each of which will be designed to fulfill a particular task in the system. In MCM, these classes are grouped again as meta classes depending on their domain and functionalities. Meta classes are themselves independent from each other with less clustering. But each meta class is cohesive (self-contained) in terms of its designated functionalities. Objects in different meta classes may have relations, collaborations, interactions, and communications in the process of simulation. Meta class defines a set of generic operations that can be performed on these meta classes, such as operation `retrieve` to a database is generic to both Relational Database Management System (RDBMS) and Object-oriented Database Management System (ODBMS). We call these generic operations *protocol* because they stand for a general agreement or contract between meta classes. Within each meta class, these protocols can be interpreted differently based on class and simulation context through dynamic binding [Oscar89]. A protocol includes a list of requests that a client can make of a server, a list of rules that a client has to obey when making the request, and descriptions about the service or responsibility [WirWi89]. When a protocol is no longer adequate to a subclass, either a high-level abstraction is needed or a new protocol should be introduced for that meta class.

In MCM, the protocol design process is both top-down and bottom-up. The top-down process

specifies a set of virtual protocols in the base class and defines them in individual subclasses when needed. The bottom-up process seeks similar responsibilities among classes and extracts their abstraction to their base class. In OZ, there are four meta classes, each of which is implemented by a framework of classes or class hierarchies:

1. DATA: classes handling data transmission and transformation, and providing services for modeling and simulation. In simulation, data may come from different sources with different data models, binary data from sensors and ports, flat ASCII files, structures in SDS (Self-Describing Standard) [Saltm91] files, tables in relational databases, and objects in object-oriented databases. Although various data may represent the same real-world entity, their data model is restricted by the feature of the repository wherein they reside. Data provider and consumer are probably loosely coupled. The meta class DATA isolates the impact of data management schema, whether flat file, relational, or object-oriented. It makes the details of data transmission and transformation transparent to its clients, and it narrows the semantic gap between restricted data models in various repositories and the object models in analysis and design. At the SSC Laboratory, data describing the structure, identities, and attributes of the accelerator (called lattice structure for each accelerator) are stored in a RDBMS, SyBase [TraZh91]. SDS is used as a vehicle to move data structures between the application and the database. The DATA object maps data from different data models into an object model for other parts of the simulation system, such as a simulator and a graphic plotter. As a result, high-level abstractions through DATA will bring flexibility in applications.
2. MODELER: classes organizing the information to represent the essence of real-world entities based on interrelations and interactions in the model used for the simulation. MODELER defines the data structure and its external view in terms of the simulation to be conducted. It creates meta data that specify the structure and configuration of objects. An application model is defined or derived from an existing model in MODELER. For example, in an accelerator particle-tracking model, a non-linear model is derived from a linear model by considering high-order magnets in the lattice. Each class in MODELER also provides a context in which protocols get interpreted in DATA and SIMULATOR (explained below). By using object-oriented techniques, class hierarchy can be used to decompose a large model by two inter-

component relationships: *is-a*—an generalization-specialization hierarchy, and *part-of* —an aggregation hierarchy. Delegation can be used to represent a complex model by its component structures. Class hierarchy facilitates inheritance and makes dynamic binding possible. A model can be derived or composed by existing models.

3. 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 that 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.

4. INTERFACE: classes providing a man-machine graphical interface. INTERFACE provides windows to graphically present the process of modeling and simulation to the user. Through class derivation, classes in INTERFACE can be shared among systems with few modifications. A well-established INTERFACE class library or framework can make interface prototyping easier and faster. A predefined look-and-feel is also important to help the user learn new applications. An INTERFACE class can be built independently from its applications such as the domain-specific editor in InterViews' Unidraw [Vliss90].

Figure 2 illustrates the relations among the four meta classes, where arrows point in the direction of the dataflow. MODELER constructs a model using information from DATA. The model in MODELER can be viewed through INTERFACE. SIMULATOR is run based on the model (in MODELER) it uses, and the result is conveyed to the user through INTERFACE. Application users can derive their own domain-specific classes from high-level abstract classes in our Meta Class Model. A simulation application can be built by using classes from the four meta classes.

Our Meta Class Model has three major advantages. First, it promotes independent design and development of different classes (hierarchies) or frameworks for different knowledge domains. An

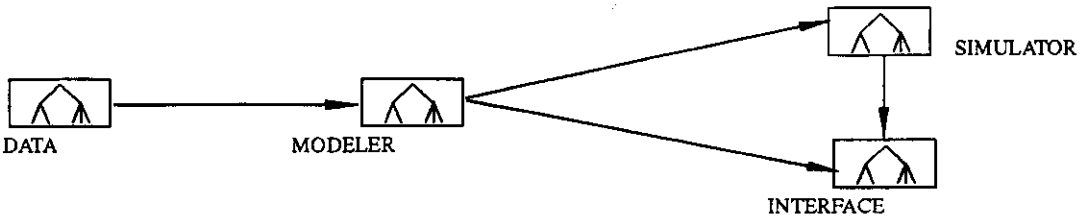


Figure 2. A Meta Class structure.

accelerator physicist builds a magnet class hierarchy; a mathematician builds a number class hierarchy. In a large simulation system, classes of various kinds will likely be designed, developed, and debugged in different environments by different people in their knowledge domains. Each type of class has its own inheritance hierarchy. The relations between these hierarchies are described by the meta class protocols. So design and implementation of each meta class can be relatively independent. Secondly, the MCM increases code reuse and domain knowledge encapsulation. A well-encapsulated class can be instantiated to build a more complex object, while the original object need not be modified or understood. Different applications may use similar objects to save coding effort. Newly derived classes can still share the protocols defined at higher levels in their base class. Derived classes can take advantage of inheritance and dynamic binding to use or redefine the existing protocols as needed. Thirdly, once interfaces between the nodes are clearly specified by protocols, development can proceed in parallel among class hierarchies. Independent development also makes software testing and debugging much easier and more efficient.

3. OBJECT-ORIENTED DATA MODELING

We differentiate between data modeling and system modeling (discussed in section 4) in the sense that data modeling emphasizes the syntax of the data, while system modeling focuses on the semantics of the data in a particular model. In data modeling, for example, a picture is just a bitmap. Each bit has no difference except its color and position. In system modeling, a picture is a collection of objects with behaviors. User can move objects around and change their shape. Data modeling is concerned with how the data in the repository will be presented to the structured frame in the MODELER. A data model is a set of classes that can be used to describe the structure of, and operations on a data source in a heterogenous environment. There are several types of data model the simulation deals with at the SSC: a relational model in SyBase (RDBMS); an object-oriented

model in ObjectStore (an ODBMS); a file model in Unix file system and a hardware device model in all detectors and adjusters in the accelerator. Meta class DATA encapsulates the differences between various data models and provides a unified operating interface by a set of protocols.

At the SSC Laboratory, static design data for each lattice are stored in SyBase, or Self-Describing Standard (SDS files) with several tables such as GEO, OPTICS, and TWISS. Each table consists of rows and columns. An index number (`ID#`) is associated with each row (also called an entry, or a *record*) and each column corresponds to a particular attribute. Table GEO records geometrical information of all magnets in the lattice. Each magnet has an entry through GEO. Attributes could be pointers referencing other tables, such as OPTICS and TWISS, that contain detailed information about magnet such as its length, strength, and optical functions. Objects in MODELER are instantiated with information in these tables (through DATA objects) and stored in ObjectStore [ObDes92], an ODBMS, with the structure defined in MODELER. Model can be imposed on the structure in ObjectStore to directly support simulation. Data can be shipped among databases, beam position monitors, sensors, and applications on 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 the attributes into an object, and load the object to an SDS file. Thus, a database table will correspond to an array of persistent structures in the SDS file. Generally speaking, SDS provides a structured file in the UNIX file system. Any abstract data type can be stored in an SDS file directly.

`DataSource` class hierarchy in meta class DATA is shown in Figure 3. `DataSource` is an abstract class in DATA which represents any kind of data information used in simulation.

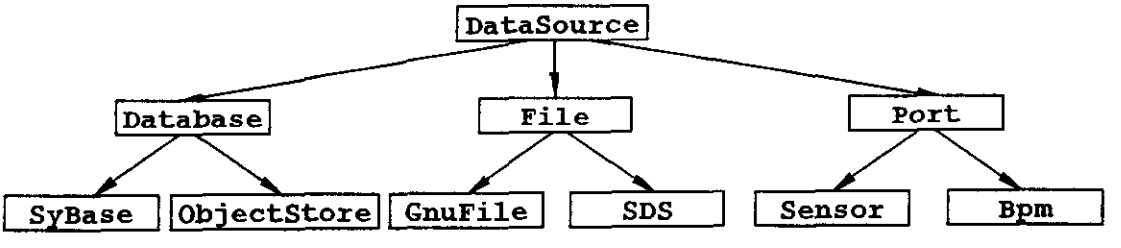


Figure 3. DataSource Class Hierarchy in DATA Meta Class

`Database`, `File`, and `Port` are three subclasses derived from `DataSource`. A set of protocols is declared as virtual functions in `DataSource` and can be shared or defined in its subclasses. The

question that needs to be resolved here is how an object knows which method should be called to respond to a generic protocol. There are three ways to bind a protocol to a method: First is the run-time type of an object, which is the key for dynamic binding. Second is the signature of the parameter list of protocols; different signatures will result in different methods to be selected to fulfill the contract toward a particular protocol. Third is the run-time type of argument passed to the protocol, such as source or mode. Although the identity of database, file, or port will all be represented by class `Source`, the difference between them can be encapsulated in the protocol and recovered later in the process of method resolution. In C++, the preceding three approaches can be implemented using virtual function dynamic binding and skin-body class structure [Copli92]. ObjectStore's Meta Object Protocol (MOP) also gives us a run-time type-checking capability through database schemata. The three subclasses derived from `DataSource` are discussed below.

Class `Database`: a base class for database operations. `Database` supports a set of protocols which is generic to all of its derived classes. These protocols can be `Open`, `Load`, `Close`, `Transaction`, `Update`, and `Retrieve`. The protocols provide common interfaces and contracts to clients, regardless what kind of database used. Class `Database` has two derived classes: `SyBase` and `ObjectStore`.

Class `File`: a base class for file operations. Class `GnuFile` and `SDS` are derived from `File`. `GnuFile` are object-oriented wrappers of GNU's SFile class and `SDS` are SDS C++ class library. `GnuFile` supports simple-type-based sequential files. An integer, floating number can be directly written to a file. `SDS` supports structured files. A C-language structure can be directly read from or written to an SDS file. Structure in SDS is self-describing with meta data that can be retrieved together with data.

Class `Port`: a base class to model physical equipment. `Port` has two subclasses: `Sensor` and `BPM`. `Sensor` is a class for real-time data acquisition. Data from `Sensor` is time-stamped. `BPM` is a data pool located at certain positions of the accelerator. Data from `BPM` is read-only.

Other classes are designed to be embedded in subclasses of `DataSource` to provide data abstraction and implementation encapsulation, such as `Table` in `SyBase` and `TimeStamp` in `Port`. These classes are not subclasses of `DataSource` but are data members (instance variables) of it. A part of class SyBase declaration is given as follows:

```
class SyBase : public DataSource {...
char databaseName[32];... ..
Table* T0;
Column* C0; ... ...
Status* Load(mode*);
}
```

Class **Table** is used as a data member in **SyBase** and **SDS**. If necessary, a particular table can be loaded as a **Table** object. This object is dynamically created when a table is loaded and pointed by a member variable in class **SyBase T0**. In SDS, the **Table** is an array of C++ structures.

Class **Column** models an attribute **A0** (corresponding to a column in SyBase). This attribute is pointed by a member variable of **T0**. **A0** is able to extract a particular field from an array of structures (table). 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 it is not efficient for such simulations, so making an attribute as an object is very useful.

TimeStamp is used for real-time data acquisition. It can be embedded into any DATA object to support real-time operation.

The **DataSource** 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 through different data sources: files, RDBMS, ODBMS, or physical equipment. DATA provides a set of classes and protocols that can keep its clients from the details of particular data models and repositories. 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 data repository or source is used.

4. MODELING DYNAMIC BEHAVIOR

A *model* is an abstraction (possibly a mathematical abstraction) of a real-world entity for the purpose of understanding it before building it [RumBl91]. It is natural in simulation to represent entities in an application domain as objects that respond to a set of well-defined messages. For example, in an accelerator system model, domain objects might be magnets, particles, and accelerators (a composite object). In our approach, a model is represented as a set of methods for generating dynamic data for the observables in the real system. New types of models may be

created by specializing existing ones. Complex systems can be modeled with composite objects (also called submodels) and can be used in other models like a built-in type in programming language. A model as a whole is itself a composite object that responds to a set of messages. The tolerant threshold toward certain attributes is called *constraint*, which is defined as a function f_c of some attributes A_o for a particular object, $C_o = f_c(A_o)$. *Behavior* of the object is modeled as a set of methods M_b, which is a function of attributes A_o and constraints C_o based on algorithms developed with domain knowledge. Dynamic behavior describes those aspects of the object concerned with time, sequencing of operation, and its configuration. These aspects include events that mark changes, sequences of events, states that define the context of events, and the configuration of the system where the object is placed. Modeling dynamic behavior can be divided into a two-step process:

- Structure modeling (only for composite object). This step defines the configuration structure of the object, the coupling pattern of its components.
- Behavior modeling. This step requires the user to design a set of methods to create dynamic behaviors based on an object's attributes, constraints, and configuration structure.

The MODELER in MCM is a library that contains a set of models and model class hierarchies where each model emphasizes different aspects or represents different levels of the real-world entities. Different models of the same real-world entity provide different abstractions interested in simulations for different purposes. It is the responsibility of MODELER to provide a structured frame or representation schema that interprets the data from the DATA object in terms of the simulation to be conducted. It is also the responsibility of the MODELER to provide all necessary methods to demonstrate behaviors to meet particular simulation requirements. The DATA object drives the MODELER object. The MODELER object generates behaviors based on DATA via its understanding and interpretation.

4.1 Structure Modeling

Structure modeling decomposes the complexity of a system into several sub-systems. The principle of such decomposition is based on domain analysis of inter-relationships within the system. In structure modeling, an accelerator can be decomposed into *beamline*, a set of magnets placed in a specific order as design components. The structure of an accelerator can be modeled by

using configuration binding techniques. Accelerator is on the top of this configuration hierarchy. It is decomposed into major beamlines, such as lattice LEB is decomposed into three major beamlines, *triinj, triext, triwm* as shown in Figure 4; these major beamlines are further decomposed

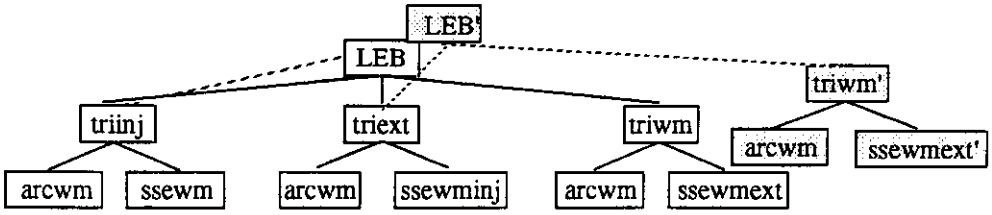


Figure 4. Lattice Configuration Hierarchy.

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 **`Magnet`**. **`Beamline`** holds a pointer to its component, which may be smaller beamlines or magnets. **`Beamline`** class inherits certain behaviors from **`Magnet`** class, such as transferring particles. It is also easy to insert or replace beamline's component with another beamline or magnet.

`Beamline` inherits all members and methods from **`Magnet`**, but **`Beamline`** has its own methods to specify its structure. Members and methods of **`Beamline`** are listed in Table 1.

Table 1 Beamline Class.

Member Variables and Member Functions	Function Illustration
Beamline* bmLnElmnt;	bmLnElmnt point to the current components (smaller beamlines or magnets)
Insert(WhichSide);	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 a 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 beamlines and magnets.

A new lattice configuration can be created by replacing an existing beamline with a new beamline or by changing existing beamline's attributes (such as strength). In Figure 4, a new design for the beamline *triwm'* creates a new configuration for its parent LEB, LEB'. LEB and LEB' are referred as the same object logically with different configuration. Configuration binding is deferred

at the simulation stage by setting the proper configuration name and the binding actually occurs from the bottom level of this hierarchy, i. e. at the magnet level. Further discuss about configuration management is beyond the scope of this paper and reader can refer to [Zhou92]. The major advantage of this hierarchical model is its reusability. Beamline *triinj* and *triext* can be shared by two different configurations. In terms of modeling itself, any system (especially a complex system) can be decomposed hierarchically. Hierarchical decomposition distributes complexity into different layers of abstractions. It provides the flexibility to adjust modeling focus between abstraction and specification. In terms of simulation, the same model can be used differently by attaching different attributes for various types of simulations. A submodel can also be derived from an existing model to change the behavior of the object modeled.

4.2 Behavior Modeling

Behavior modeling seeks a set of methods governing the object's control logic based on domain knowledge. At the SSC Laboratory, there are three kinds of objects to be modeled: the particle beam, the magnet in the accelerator, and the accelerator itself. The behavior of a particle depends on its momentum, its position, and the distribution of magnet-field strength around it. Particle momentum and magnet strength distribution are determined by the accelerator through which a particle is passing. In simulation the behavior of a bunch of particles (beam) will be more interesting statistically. *Particle distribution hierarchy* (PDH) is used to record such a beam model.

The root class **Beam** has only one particle, and it is placed at the origin. Particles with standard statistical distributions, such as normal and average, are subclasses of **Beam**. Beam has five instance variables listed in Table 2. Vector $D = [d, d', \delta]$ is called the principle vector (PV), where d is the displacement, d' is the angular deflection, and δ is the momentum deviation of the particle. A new beam class can be derived from a beam class library with a graphical user interface. A beam object can be created in three ways: instantiating from a beam class; copying an existing beam from the beam class library and changing the particle distribution or amount of the particles (Figure 5); or as a result of beam-tracking simulation.

Table 2 Instance Variables in Beam Class

Instance Variable Name	Illustration
num	number of particles in the beam
Position *pos[num]	position of those particles, displacement

Table 2 Instance Variables in Beam Class

Instance Variable Name	Illustration
Deflection *dp[num]	angular deflection of the particle
Deviation *delta[num]	momentum deviation of the particle
distribution form	statistical distribution of those particles
void Generate(seed)	generate a particle distribution

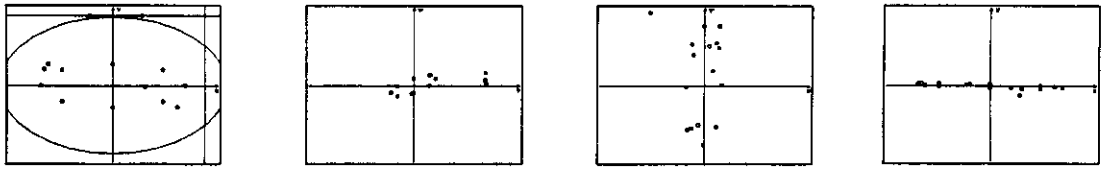


Figure 5. Beam Objects Created from PDH.

After a 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), optics functions (such as β function), phase advance (Φ), and other attributes. The principle magnet hierarchy(PMH) is shown in Figure 6. A prototype of the magnet attributes modeling system is shown on the right of Figure 6. Magnet instances are graphically represented by a collection of icons (See Figures 8 and 14.) A magnet class is represented by a list of its attributes. Magnets are constructed from their own class using this interface. After a derived magnet type is created from the hierarchy, it is added back to the list as a part of the new hierarchy.

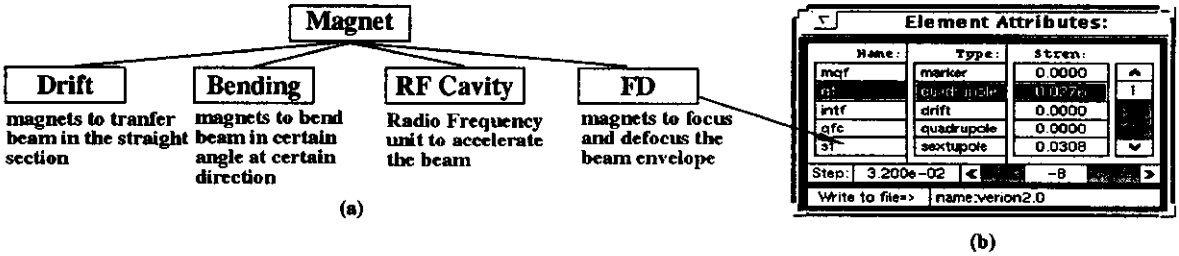


Figure 6. Principle Magnet Class Hierarchy (a), and its interface (b)

The behavior of the magnet can be modeled as a 3 by 3 transformation matrix M based on Steffen's theory [Steff85]. M is defined as a function of t, s, l, o, and m for a particular magnet, where t, s, l, o, and m are defined as above. D_i and D_{i+1} are the principle vectors of a particle at position i and $i+1$ respectively. The relationship between D_{i+1} and D_i are: $D_{i+1} = M \cdot D_i$, i.e.:

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

The **Magnet** class provides methods to support operations on the transformation matrix. The **Magnet** class definition is partially given in Table 3.

Table 3 Magnet Class

Member Variables and Member Functions	Function Illustration
Class category	There are four categories: *drift, bending magnet, RF cavities* and *focus/defocusing magnets*.category is used for dynamic type checking, which is not supported by C++.
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. Each **Magnet** instance has a pointer to a **Magnet** class in the PMH. When a new **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 instance as its first instance. The new class inherits all methods from its parent, such as **CreateTM** and **BehaviorMap.**

Behavior modeling through **Magnet**'s method is supported by two approaches:

1. Each subclass of **Magnet** has its so-called behavior file that is included by the virtual function **BehaviorMap()**. A edit 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; then the whole process needs to be restarted. As an example, **CurrentDipoleBehavior** is the behavior file for the dipole, a

```
PV* Dipole::BehaviorMap() {
 include "CurrentDipoleBehavior"
}
```

subclass of **FD** (Focusing and Defocusing magnet). The new behavior file created through edit window replaces the previous behavior file and the old one is renamed as **CurrentDipoleBehavior.~1~**. The advantage of this kind of approach is that code in other

simulation programs can be cut and pasted into our behavior modeling system with little modifications. The restriction is that user have to know C++ programming and the context of that virtual function.

2. Several models (such as linear and nonlinear method) can be predefined based on knowledge and domain specific rules. Virtual function dynamically binds the rule number (set by the user through interface) with a pointer to the member function to construct behavior. Interactive modeling basically becomes rule-picking and function-binding.

Beam tracking model class hierarchy is shown in Figure 7. A *Linear Sequential Transformation*

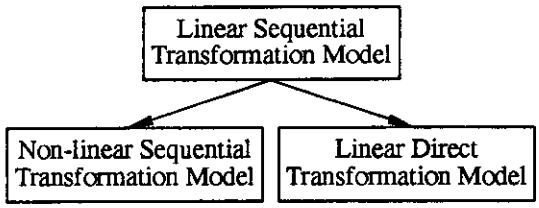


Figure 7. A Beam Tracking Model Class Hierarchy in MODELLER

(LST) Model uses an aggregation hierarchy of a system. All nodes in the hierarchy are modeled as objects of class Actor [Zhou92] with internal data representation and a set of methods for creating behaviors. The result of interactions between objects is created using mathematical transformation on an object's internal data representations. To simplify the problem, such a transformation could be considered as linear (one step) and sequential (no concurrency). For example, an accelerator can be decomposed as several sectors (also called beamlines). Beamline can finally be decomposed as elements such as magnets and RF cavities. A particle launched at a certain place passing through the accelerator can be considered as an LST. Whenever the particle passes through an element, its position, deviation and momentum may change; it therefore changes its behavior, such as wiping out of the trajectory. A *Non-linear Sequential Transformation* (NST) Model and a *Linear Direct Transformation* (LDT) Model can be derived from the LST Model by overriding the transfer behavior of the model. In NST, transformation contains several steps based on the type of the element. Such a transformation is no longer context-free. It may rely on the previous result and has impact on the next step. In LDT, several transformations can be aggregated into one. Transformation is independent of the object to be transferred.

5. CONSTRUCTING DYNAMIC SIMULATION

This section includes several examples of constructing a dynamic simulator in an interactive environment. Dynamic simulation provides an interactive environment between the user and the simulator. It allows the user to select an appropriate model from the MODELER, to pinpoint objects (component in SSC lattice), to modify their attributes and structure, and to rerun the simulation to see the impact of the result system. Most configuration adjustments caused by individual modification will be propagated automatically by the simulator (by calling the proper model method).

SIMULATOR is a meta class that exercises models to actually generate dynamic behaviors to meet simulation requirements. Simulator (an instance of SIMULATOR) is the manager of the entire simulation. It decides when and which method should be invoked in terms of the model used during the simulation process. Objects are controlled under simulator to interact with each other and create dynamic behavior. We stress that "modeling" and "simulation" are two different tasks and we will attempt not to use them interchangeably; one model may be simulated using several different simulation algorithms. The relationship between SIMULATOR and MODELER is quite similar to the relationship between algorithm and data structure. MODELER provides a base for creating behavior. Certain models in the MODELER will create certain behaviors for specific simulation in SIMULATOR.

As explained earlier, OZ deals with three types of models: beam model, magnet model and lattice configuration model. The behavior of each object in the interactive simulation process is used to adjust the model for better design. As an example, let's illustrate how BumpView simulation works using the SIMULATOR. A basic problem in accelerator physics is how to keep the beam inside the correct trajectory, i.e., to avoid losing the beam. The beam is basically guided by magnets. Most magnets have fixed strength and are designed to bend the beam at a certain angle at specified locations. To correct dynamic errors that 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, or BPMs) near those kickers to monitor the results of corrections and to locate the beam position. For a particular BPM reading, simulation should be able to predict the adjusting value for each kicker, especially those kickers near the BPM being

monitored. BumpView simulator is built to achieve such a goal. It provides a simulated beamline aperture for adjusting particle trajectory and simulates the effectiveness of bending force created by adjusters to particles at the requested position. BumpView uses a linear model, a submodel derived from structured hierarchical model we mentioned before. Actually a method is added to the linear model for sending recursive query to the parent hierarchical model to pull these leaf-level magnet out. The objects involved in the simulation are just of type **Magnet** and a particular lattice configuration. A *influence* function is defined as a relationship between each pair of monitor and adjuster in the derived model. These influence functions are used as coefficients of a set of differential equations to be solved for the simulation.

Figure 8 gives the BumpView simulation interface. The bottom part of the window is a symbolic

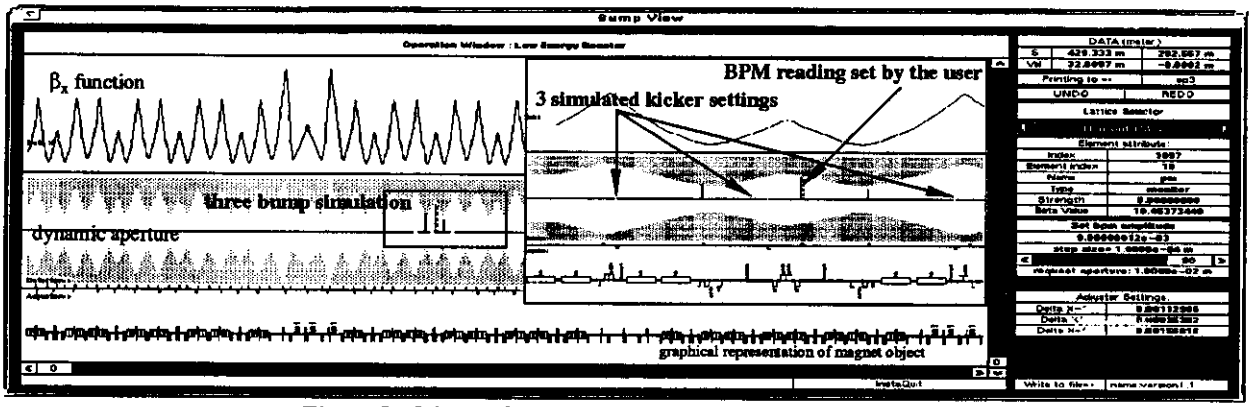


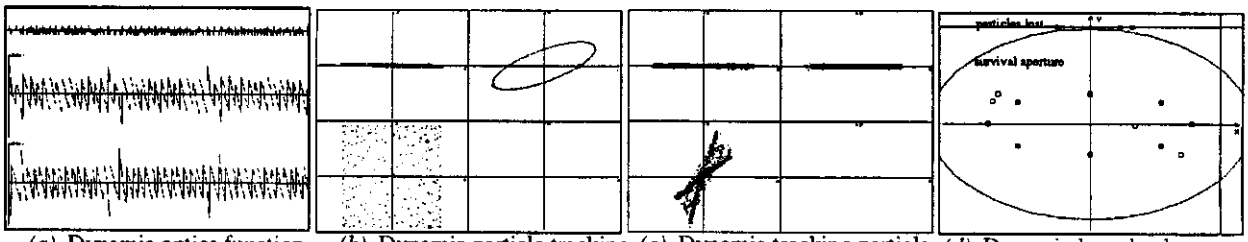
Figure 8. 3-bump Simulation Using Bumpview Simulator.

representation of the LEB lattice structure. Above it are the positions of the detectors and adjusters along the LEB. All objects in the representation are active (sensible and associated with actions). A display panel always prompts attributes of the magnet engaged with a pointing device. A particle object can be constructed on the fly using graphical user interface and can track through the beamline under the requested bending force. In the middle of the window (shaded part) 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 upper right corner. The dashed bar is the BPM reading set by the user. Bumpview simulation only uses magnet model because particle tracking solely deals with leaf-level *component*. But when a user wants to modify attributes of a magnet or the structure of the beamline, the beamline model kicks in and controls the propagation of the modification. When

a BPM is engaged for value setting, Detector model (for BPM) is bound to provide special behavior. Actually Detector model is derived from magnet model for BPM setting. So a model in MODELER can be directly used or inherited for specific simulation. Such extensibility is useful for model reuse. Bumpview simulation will give the following:

1. The setting value of the three nearest adjusters, which will generate the BPM reading set by the user. Three white points (actually a three green bar) stand for the settings of three kickers around that BPM. The actual values are given as deltaX$_-$', deltaX', deltaX$_+$' in the "Adjuster settings" box at the bottom of the control panel.
2. To make things simpler, we assume that the adjusting will affect only the three BPM readings nearest to the BPM selected. All other BPMs should have zero readings. The simulation proves the model is correct. From the picture, there are only three solid bars in the middle. The up part of the window is the β-tron oscillation along the LEB.

Figures 9 and 10 give more examples of dynamic simulations. In Figure 9, *(a)* is the optics function of LEB created by object **`Twiss`** in SIMULATOR; *(b)* and *(c)* are dynamic particle tracking by turns or by every magnet using Track from SIMULATOR; and *(d)* is dynamic tracking of a beam created from beam class hierarchy by using **`Emit`**, which is also a simulator object from the SIMULATOR class hierarchy. **`Emit`** can also be used to aid the research of relations between particle distribution in the beam and beam survivability.



(a). Dynamic optics function creation of the MEB. Here is α function.

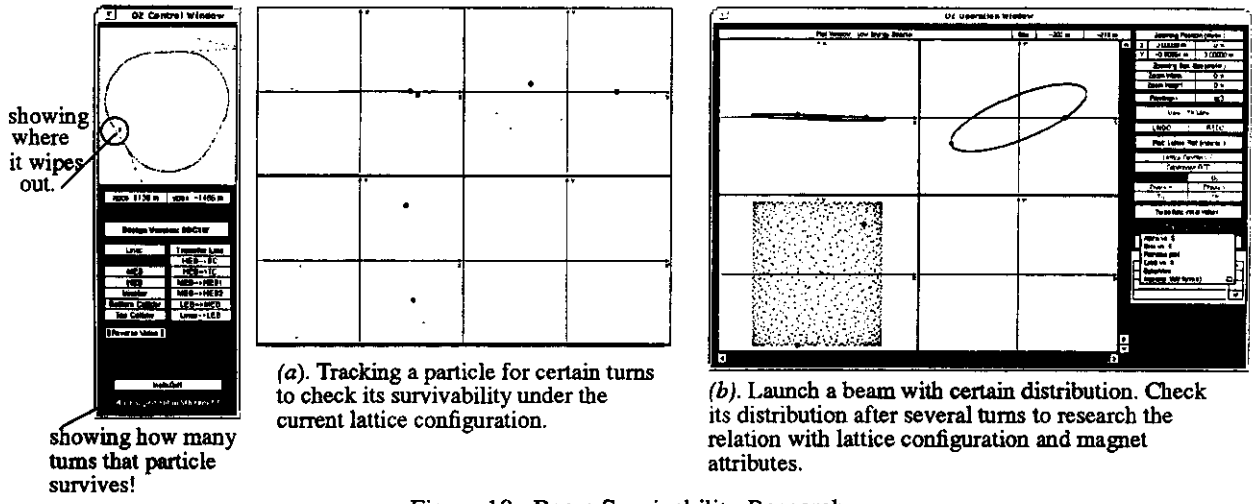
(b). Dynamic particle tracking in LEB for 100 turns, and recording its d and d' after each turn.

(c). Dynamic tracking particle for one turn 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 9. Dynamic Simulation.

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 the BumpView simulator. Figure 10 gives such an example. User can change the dynamic aperture and particle emit position in the process of tracking simulation to see under which circumstance the particle will wipe out. *a* shows a particle passing through LEB



(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 10. Beam Survivability Research.

and wiping out. By zooming in to the picture, user can find the exact wiping out position. *b* gives another example of simulating a beam passing through the LEB.

6. OBJECT-ORIENTED DATA VISUALIZATION

In this section, we will describe the functionality of data visualization in OZ and related implementation issues. Data visualization allows a user to directly manipulate an object and to access information through the graphical user interface to conduct modeling and simulation. It makes modeling and simulation efficient, informative, and much easier to handle.

In OZ the whole SSC complex can be visualized through a window with zooming and scrolling capability. Various physics functions can be dynamically plotted through different windows, and configuration of the beamline component can be edited using graphical interface that supports direct object manipulation. After a particular lattice has been loaded, the position and size of each object can be extracted from `DataSource` object. The plotting window (an object of class `ViewPlot`) scales these data based on current plotting size and displays the object on the screen. A `VisualData` subclass is derived from `DataSource` to interface with `ViewPlot` by redefining (not redeclaring) the protocols dealing with domain-specific operations such as scaling and color-coding. When resize occurs, `ViewPlot` will rescale the position and size of all objects and replot them. Incremental drawing is supported by `ViewPlot` for accumulatively displaying simulation without refreshing the whole window. By taking advantage of dynamic binding of the C++ virtual

function, all methods for graphic manipulation are virtual. For example, 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.

Lattice configuration editing is supported by direct graphical object manipulation. Class **Node** represents a component graphically and expands its subcomponent into a tree structure. Figure 11 is a graphical interface for the lattice configuration editor that provides an interactive modeling

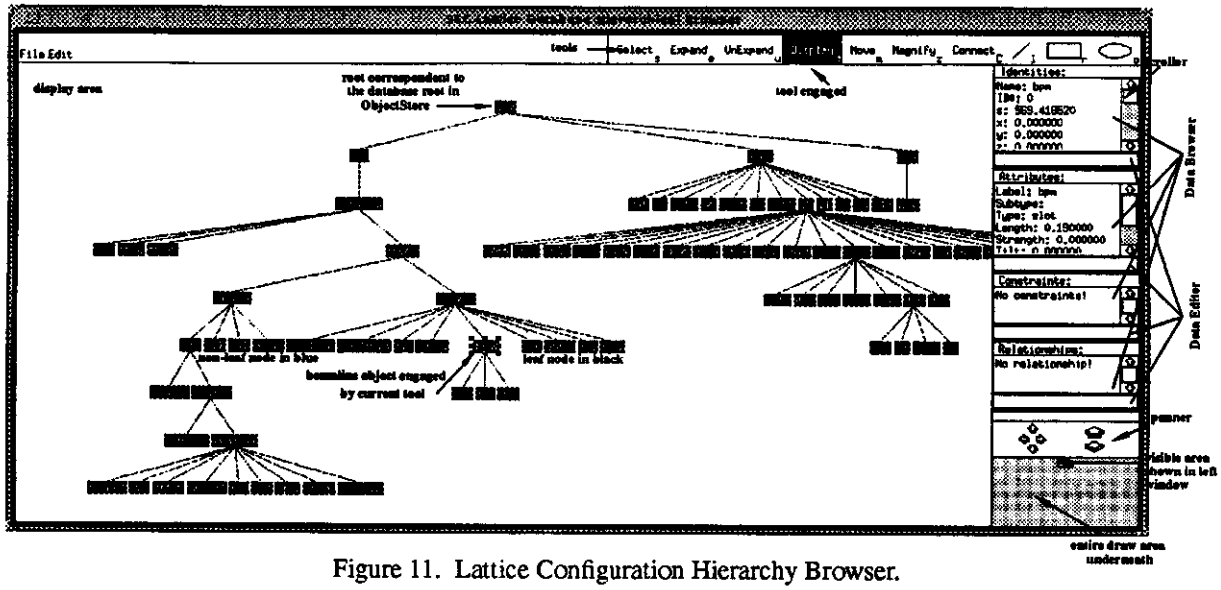


Figure 11. Lattice Configuration Hierarchy Browser.

environment to the MODELER. The configuration tree shown in the figure can be cut and pasted using an existing component in the tree or a new component (graphically represented by **Node** object) created on the fly by the user. Configuration change in a subtree will be informed to its parent component (its aggregation). And the parent component will update the corresponding structure model of MODELER in the database.

Meta class INTERFACE is implemented to provide all the classes necessary to construct visualization software for simulation in OZ. INTERFACE classes are developed using GLISTK [Kan91] and InterViews [Linto90] libraries. Two important classes should be addressed in building a graphic interface: (1) the layout of the interactive interface and connections among *control elements,* such as buttons and menus, and (2) the interactive graphics (*view*).

`ControlLayout` is a class to lay out control elements (such as button and menu) and specify their behaviors. It has two subclasses: `Layout` and `ControlElement`. `Layout` is an invisible object which is primarily used for arranging `ControlElements` on the screen. `ControlElement` is a base class for all graphical interface building components, such as button, menu, and scroll bar etc. Control elements are created not inside the constructor of `Layout` but by another virtual method called `CreateAndInsert`(). Different applications may override `CreateAndInsert`() to create and insert their own control elements.

`Layout` is derived from class `Gorgan` in GLISTK, a subclass of InterViews' `Scene` (Figure 12). `ControlElement` is derived from class `LabelGlistk` in GLISTK, a subclass of

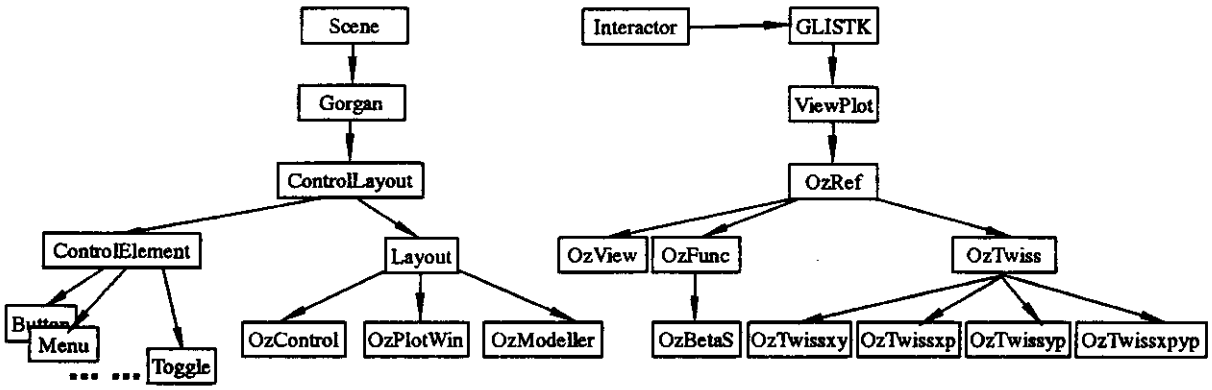


Figure 12. Controllayout and Viewplot Class Hierarchy.

InterViews' `Interactor`. A callback function defined in SIMULATOR object can be attached as a control action of a `ControlElement` (such as a button or menu item) to respond to a button clicking or menu selecting. In OZ, three subclasses are derived from `ControlLayout`. They are `OzControl` for the layout of the control panel to switch among different lattices, `OzPlotWin` for the layout of various `ViewPlots` and the connection of their controls, and `OzModeler` for the layout of the magnet attributes editor. Table 4 summarizes a few protocols provided by `ControlLayout`.

Table 4 Protocols in Class ControlLayout

Member Functions	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.

Table 4 Protocols in Class ControlLayout

Member Functions	Function Description
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 a class to plot dynamic structured graphics [VliLi88] controlled by `ControlLayout`. `ViewPlot` is derived from class `glistk` in GLISTK, a subclass of InterViews' `Interactor` (Figure 14). It provides a dynamic graphic view of objects from MODELER and SIMULATOR. Subclasses can be derived from `ViewPlot` such as bar chart, two-dimensional multi-function plotter, object browser, and accelerator view.

In OZ, a subclass of `ViewPlot` `OzRef` keeps a list of objects drawn inside the window and encapsulates the functionalities of zooming, scrolling, resizing, and refreshing. It has three subclasses, `OzView`, `OzFunc`, and `OzTwiss` to provide the graphical representation of objects. `OzView` is used to display a geometrical view of the SSC complex with magnets. `OzFunc` is used to plot various optical functions with sample points. `OzTwiss` is used for dynamic particle tracking with magnet trajectory and particle as the drawing objects.

Most dynamic graphics in `ViewPlot` require 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 that when the window is opened later, the previous image can be restored as is. It is not realistic to repeat the entire simulation to recreate these images. A feasible solution is to create an incremental drawing queue (`IDQ`) inside the `ViewPlot` to record incremental drawing data dynamically. Two methods are used for drawing (Figure 13). `Refresh`

```
ViewPlot::Refresh(){                    ViewPlot::CreateDynamicData(){
MapRawDataToDrawableData();             CreateSimulationResults();
DrawStaticData();                       SizeOfIDQ++;
if (SizeOfIDQ>0)                        RegisterToIDQ();
Draw();                                 Push(IDQ, CurrentDrawingData);
}                                       Draw();
                                        }
```

Figure 13. How To Do Incremental Drawing.

handles initial drawing such as legend, measurement, symbolic representation, and marks. We call these static graphics, and they should be always on the screen. To draw something dynamic on the

screen, call `Draw` 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, `Refresh` will get called. `Refresh` will in turn call `Draw` to accumulatively draw whatever is in the `IDQ`. Figure 13 illustrates how the incremental drawing queue works.

Sometimes because thousands of magnets (thousands) need to be drawn in `ViewPlot`, making each magnet as a structured graphic object in InterViews is not realistic. If we make the whole accelerator an object, then it is difficult to pinpoint an individual magnet object. A feasible solution is to make the whole accelerator a composite 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 `DataSource` object. Figure 14 shows such a mapping.

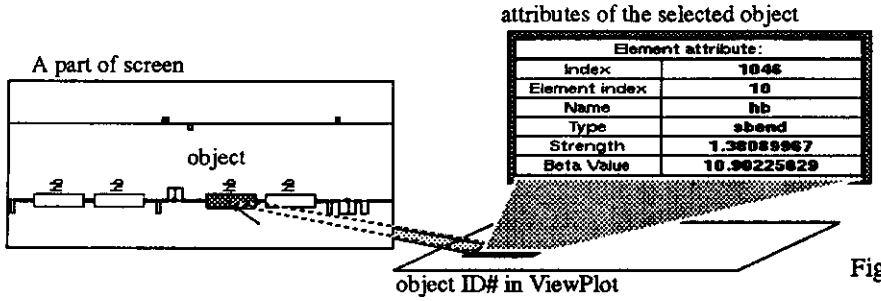


Figure 14. Object Mapping.

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 MODELER. Some protocols of `ViewPlot` are listed in Table 5.

Table 5 Protocols in Class ViewPlot.

Name	Function
myModel	Pointer to MODELER object
realBoundary	The real dimension of visual target. For example, optics function of HEB
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 an event handler.

Table 5 Protocols in Class ViewPlot.

Name	Function
Refresh()	Refresh() handles initial drawings. It keeps a pointer to 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).

As a good example, let's consider drawing a beamline on the screen. A list of graphic objects is created for pictorial representation of the magnet. The corresponding magnet object is either pointed by or embedded in the graphic object. These graphic objects are inserted into the `IDQ` in the drawing process. Each graphic object knows how to draw itself by calling its member function `Draw`(). Drawing is recursive in a composite `ViewPlot`. `IDQ` is a parameterized collection that provides general behaviors at the collection level and different behaviors at the component level. So different magnets can be aggregated into a collection using a set of protocols such as insert, delete, and iterate, while their individual behavior can be much different, such as the implementation of `Draw`().

The communication between `ControlElement` and `ControlLayout` is handled via a class called `Communistk`, an object which focuses on a value and has a list of other objects to notify when this value changes. `Communistk` enables and simplifies communication between objects in a program. It encapsulates the notification and update mechanism of a state variable it focuses on. `State` is a variable with a valid C++ type to represent current status of a `ControlElement`. `State` can be attached to a `Communistk`. `Communistk` focuses on the value of the state and has a list of `ControlLayout` to notify when this value, or the focus itself, changes. Every `Communistk` has a `CommuList`, which is a list of `ControlLayout`, which is informed any time the value of state on which `Communistk` is focused on is changed by `Communistk::SetValue`. When the value on which a `Communistk` is focused is changed by `Communistk::SetValue`, the `Communistk` calls its `HitCommuList` method, which informs every `ControlLayout` in its `CommuList` by calling their `CommuHit` method. A message not only can be sent back and forth between `ControlElement` and `ControlLayout`, but also can be sent out to another application using the GLISH event sequencer [Paxso91]. 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 Communistk 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 to derive a specific `glistk`, for example, `QuitButton`, with its own `PerformAction` method; the other is to associate its `Communistk` with a particular ID and add its `ControlLayout` to its `CommuList`. `ControlLayout`'s `CommuHit`() method will be called when the `Communistk` value is changed automatically. `CommuHit`() can control the action based on `CommuID`.

7. CONCLUSION

In this paper, we describe our experience in 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 model that decomposes a design into four types of classes (meta class) that handle data management, user interface, modeling, and simulation, respectively. We design classes in each of the meta classes not only for the OZ project but also for reuse with other projects. We set the protocol between each of the meta classes before we started to build them. We kept the protocol generic and elementary so that it can achieve maximum reusability. We built each meta class independently and focused on the problem itself rather than struggling with the interface between other meta classes. Such a responsibility-driven approach not only achieves productivity but also simplifies the testing process with a more loosely-coupled system.

In our object-oriented data modeling, data, meta data, and procedures that handle data accessing and manipulation are combined as an object. Data as an object is able to describe itself and provide information to the modeling and simulation. Data object has its view which can be directly manipulated through a graphical user interface. A system can be decomposed into aggregation hierarchy with dynamic behaviors. Attributes and constraints are used to model dynamic behavior of the object. Attributes and constraints can be dynamically bound to an object in an inheritance

hierarchy. Different configurations 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 modeling system.

OZ has been implemented and is 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 the three bump effect in the accelerator. With the experience we had and with classes already available in developing OZ, it took us only one month to finish the prototyping. The results achieved with our current effort have been encouraging, leading us to believe that the object-oriented approach will provides us more flexibility and extensibility in future software development. We plan to extend our effort to build a more general and complete framework for simulation at the SSC Lab.

ACKNOWLEDGMENTS

We greatly appreciate the valuable contributions and advice provided by Dr. Richard Talman and Dr. Garry Trahern at the Project Management Organization of the SSC Laboratory. We also appreciate the help from Dr. Chris Saltmarsh, Matt Fryer at Lawrence Berkeley Laboratory, and Matthew Kan at Carnegie Mellon University.

References

[AkBer92] Aksit, Mehmet; Bergmans, Lodewijk: "Obstacles in Object-Oriented Software Development," *OOPSLA '92 Conference Proceedings*, Oct. 18-22, 1992, Vancouver, Canada.

[Copli92] Coplien, James O.: "Advanced C++: Programming Styles and Idioms," pp133, Addison Wesley Publishing Co. 1992.

[CoaYo91] Coad, P.; Yourdon, E.: "Object-Oriented Design," Yourdon Press Computing Series, Prentice-Hall, 1991.

[DahNy66] Dahl, O.J.; Nygaard, K.: "Simula - an algol-based simulation language", *Communication of the ACM*, 9(9), pp 671-678, September, 1966.

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

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

[Niels91] Nielsen, Norman R.: "Application of Artificial Intelligence Techniques to Simulation," "Knowledge-Based Simulation, Methodology and Application," pp1-19, *Advances in Simulation*, Vol. 4, Springer-Verlag, 1991.

[ObDes92] Object Design, Inc.: "ObjectStore User Guide," Release 2.0, Oct, 1992

[Oscar89] Oscar, Nierstrasz: "A Survey of Object-Oriented Concepts", *Object-Oriented Concepts, Databases and Applications*, pp 3-21, ACM Press and Addison-Wesley, 1989.

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

[Round89] Round, Alfred: "Knowledge-based Simulation", *The Handbook of Artificial Intelligence*, Volume IV, Chapter XXII, Addison-Wesley Publishing Company, 1989.

[RumBl91] Rumbaugh, James; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. General Electric Co.: "Object-Oriented Modeling and Design," Prentice Hall, 1991

[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 Elektronen-Synchrotron DESY*, Hamburg, March, 1985.

[SerBr85] 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

[TraZh91] 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.

[VliLi88] Vlissides, John M.; Linton, Mark: "Applying Object-Oriented Design to Structured Graphics," *Proceedings of the USENIX C++ Conference*, Denver, Colorado, Oct. 1988.

[Vliss90] Vlissides, John M.: "Generalized Graphical Object Editing," *Technical Report*: CSL-TR-90-427, Stanford University, June 1990.

[WirWi89] Wirfs-Brock, R.; Wilkerson, B.:"Object-Oriented Design: A Responsibility-Driven Approach. *Proceedings of OOPSLA* '89, pp 71-76, Oct. 1989.

[WirWi91] Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.:"Designing Object-Oriented Software," pp33-36, pp161-176, Prentice Hall, 1991.

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

[Zhou92] Zhou, Jiasheng: "Object-Oriented Modeling for Dynamic Simulation", SSC internal report, Oct., 1992.