This is to certify that the

dissertation entitled

Object-Oriented Modeling and Simulation for the Superconducting Super Collider

presented by

Jiasheng Jason Zhou

has been accepted towards fulfillment of the requirements for

Ph. D. degree in Computer Science

Major professor

Moon-Jung Chung

Date 8/16/98

DATE DUE	DATE DUE	DATE DUE

OBJECT-ORIENTED MODELING AND SIMULATION FOR THE SUPERCONDUCTING SUPER COLLIDER

By

Jiasheng Jason Zhou

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1998

ABSTRACT

OBJECT-ORIENTED MODELLING AND SIMULATION FOR THE SUPERCONDUCTING SUPER COLLIDER

By

Jiasheng Jason Zhou

The object-oriented paradigm is being widely exploited in computer modeling and simulation. It has the advantage of effectively capturing the semantics in real-word objects with reusability and extensibility; thus it has great intuitive appeal in the area of simulation.

This thesis presents techniques in the area of simulation framework, simulation object modeling, and configuration management and version control (CMVC) of simulation data. Those techniques are based on a composite object model, which effectively captures diverse semantics of domain objects in a simulation process. The techniques are developed against a background of simulations in the design of the Superconducting Super Collider (SSC). A layered framework is proposed to achieve design reuse in modeling, simulation construction, data management, and simulation visualization. The Actor model is introduced to model dynamic behavior of the simulation system. The Actor model is further extended to support CMVC using the concept of modeling object. Dependency between objects is utilized to control change notification and propagation in a simulation system. Constraints are specified to validate configuration bindings and to limit the scope of change propagations in simulation. The concept of

generic object is used to allow dynamic configuration binding in dynamic simulations. A workspace model is presented to select and record simulation context in a multi-user environment.

This research is one of the first to apply object-oriented simulation and modeling techniques to a large scale real project. It enriches our understanding of dynamic simulation, deepens our knowledge in the design of a simulation framework, and extends CMVC to the area of simulation.

To my parents and family

ACKNOWLEDGMENTS

I wish to express my gratitude to my thesis advisor Dr. Moon Jung Chung for his consistent guidance and encouragement throughout my Ph. D. program. I appreciate our many discussions and the invaluable suggestions he provided.

I extend my personal thanks to my committee members, Dr. Lionel M. Ni, Dr. Richard Reid, Dr. Richard York, Dr. Jerry Nolen, and Dr. Richard Talman for their help and guidance. My personal appreciation goes to Dr. Karen Tanino for her extensive review of the draft of this thesis.

Finally, special thanks go to my wife Weiwen Guo and my son Andrew, for their patience and love during many long days. Especially, I must express my heartful thanks to my parents who give me life, strength, and confidence.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

(continued)

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Simulation is a process of representing the behavior of one system by the behavior of another system. In computer science, simulation refers to the use of computation programs to implement a model of some system or phenomenon. The purpose of simulation is usually to make experimental measurements or predict behavior, thus moving the laboratory to the computer environment. The prime justification of employing simulation is that the economics or logistics of experimenting with the actual system may be expensive or prohibitive. Simulation thus provides a cost effective and time efficient prototype system with which to answer questions of a "what if" nature for engineering design of large complex systems [49, 50]. The problem to be solved by simulation can be considered as the identification of the behavior of some dynamic system. "Dynamic" and "simulation" are interrelated terms [59, 65, 70, 87]. Systems that are dynamic — those whose states or structures change in response to progressive stimulations — are customarily defined as simulatable.

Simulation of complex systems such as the Superconducting Super Collider (SSC) is particularly useful. It is not only an effective way to assist and verify the design before

the machine is ever built, but also a cost-saving and time-efficient approach for a large project. However, the simulation of such complex systems is not easily accomplished. A complex system usually consists more smaller sub-systems. In object-oriented term, such a complex system can be regarded as a *composite object* that could contain other objects (including other composite objects). The contained objects are usually called *components*. Relationships between composite object and its components as well as relationships among components themselves can be complicated when an object's composition is deeply nested, one contains the others in a composite hierarchy. The *configuration* of a composite object defines its components, their relationships with the composite object, and relationships among themselves. The configuration is regarded as a specification of how a composite object is formed and what the relationships should be. The execution of such a specification is called a *configuration binding*. When those relationships change, the configuration changes to redefine the composite object, which will derive a new *version* of this composite object from its previous version. In this thesis, *dynamic simulation* focuses on action and reaction a composite object should take when its configuration undergoes changes. Such a configuration change may be considered as an object state change in a broad sense. The state here is not measured by values but by the stages in which an object's configuration evolves in simulation, thus making configuration a key element in modeling dynamic systems. The action and reaction demonstrated during the simulation exhibit the *dynamic behavior* of the composite object. The composite object model has its advantage of being hierarchical. It makes configuration binding possible at different levels of the hierarchy, therefore provides an effective way to simulate dynamic behavior of a composite object under various versions of component bindings. The difficulties in dynamic simulation arise from four different aspects:

- Modeling technique: analysis of experiment requirements, design and construction of simulation models that bring the real world to the computing environment;

- Simulation design: design and development of simulation engines to reflect the dynamic behaviors of the system to be simulated;
- Configuration management and version control (CMVC) of simulation system: controlling the changes in a simulation system due to corrections, extensions, and adaptations that are applied to the system and record the history of the system evolution over its lifetime;
- Simulation visualization: representation of the behavior of the system being simulated through a graphical user interface (GUI).

Although great progress has been made in the past decade in computer simulation, many problems still remain to be solved. These problems include the lack of integrated frameworks to link all above aspects in simulation, weak semantic models to represent dynamic behavior due to configuration change, and poor CMVC support for managing simulation system evolution. Solving these problems becomes more and more imperative to deal with the complexity of modern computer simulations. The research of this thesis provides a solution to these problems with an integrated simulation framework that emphasizes composite object modeling. By changing configuration of a composite object, the behavior can be captured through binding between a composite object and its components. Together with an integrated CMVC support tailored to the composite object model, this framework presents a powerful set of tools for building simulation environment. The framework is the first attempt in engineering simulation that modeling, simulation, data management, and visualization are brought into a single framework with a unified object model working seamlessly in simulation application.

1.1 Object-Oriented Simulation and Framework

Simulation has always appeared to be one of the natural uses of object-oriented systems [56, 73, 87]. The appeal of object-oriented simulation is that it conforms to the

notion that the world is composed of *objects*. It is not difficult to view the real world around us as a set of objects that interact with each other. In the case of a simulation, the real intent is to represent these objects from the real world with a computational model. Therefore nothing could be more natural than to organize the model structure around the objects being simulated. This aspect of the object-oriented paradigm is perhaps most significant to simulation.

An object comprises a collection of data representing a particular entity, together with operations that provide the means for manipulating and accessing these data. An object-oriented approach treats functions and data as invisible aspects (*encapsulation*) of entities in the simulation and, consequently, the structure of this approach tends to mirror the application domain structure. A simulation typically models an application as a collection of real-world interrelated entities. Each entity in the application is represented as one object in the model. The mapping of an object from the problem domain (the view of the real-world entity in analysis) to the solution domain (the data representation in design) is straightforward because it directly interprets its real world counterpart as an object in the simulation model [87]. Changes to the state of the application are typically expressed as events passed via messaging among objects. An object captures the identity, structure, and behavior of the application that it models [26].

An object model aims to be well suited to support the representation and manipulation of complex objects in the simulation of engineering design [19, 49, 55, 61, 67]. It offers a natural conceptual model to the engineers and facilitates the efficient structural clustering of information that tends to be obstructed by normalization in the relational model [7, 33]. A fundamental property of objects is that they are abstract and encapsulated to facilitate reuse. The object's data and operations are packed together in a single module, and the object's internal data representation is de-coupled from its external operation interface, which makes the reuse much easier. Objects can also be extended, and the new object's behavior can inherit from the behavior of the existing objects. Simulation

is evolutionary in nature. An object-oriented approach will be much less resistive to such evolution because of its abstraction and encapsulation. Reusability, extensibility, and maintainability form the central features that make object-oriented simulation different from traditional simulation techniques.

The object-oriented technique originated in the early 60's when Ole-Johan Dahl and Kristen Nygaard created Simula [21], a simulation language introducing the concepts of encapsulation and class to the programming world. These concepts were further exploited in the 70's at Xerox PARC, and subsequently publicized in Smalltalk-80 [30]. Up to now, object-oriented approaches have been broadly adopted by all aspects of software development including simulation [4, 20, 34, 41, 56, 62, 65, 70, 87]. The essence of object-oriented technology is the object-oriented decomposition of the user's needs into executable language constructs [3]. When object-oriented analysis is used, a system can be decomposed into its logical or physical components. Such decomposition will yield three structures: *composite hierarchy*, *class hierarchy*, and *messaging protocol*. Object composite hierarchy decomposes a system to a component level where the necessary granularity is reached in terms of the design requirements. By examining and grouping objects at the component level by their functionality, class hierarchies can be formed, in which common features are shared and differences are derived, in a generalization-specialization process. The top level of a generalization is called a subsystem or a *layer* [8, 84, 87], which stands for a set of cohesive and self-contained services or functionalities in a particular domain such as model, simulation engine, data management, and user interface. Messaging protocol defines a set of message abstractions between layers. Dynamic behavior of the system is created by object interaction based on composite hierarchies and messaging protocols. Those layers of classes with pre-defined messaging protocols among them form a framework. Simulation applications can be instantiated from the framework for a specific experiment. This concept will greatly simplify the simulation design process through design reuse and component level integration.

1.2 Current Research on Modeling and Simulation

The techniques presented in this thesis are related to three fields in computer simulation: simulation framework, modeling techniques for dynamic simulation, and configuration management and version control in simulation process.

1.2.1 Simulation framework

A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software such as simulation [28]. For example, a framework can be geared toward building a simulation system for a different domain like the high-energy physics simulation, weapon simulation, and mechanical simulation. Framework can be customized to a particular application by creating application-specific subclasses of abstract classes from the framework. The framework dictates the architecture of the application. It will define the overall structure, its partitioning into layers, classes, and objects, the key responsibilities thereof, and the collaboration of classes and objects (*protocols*). A simulation framework captures the design decisions that are common to the domain of simulations such as modeling, simulation constructing, data management, and graphic visualization; all those are important aspects of a simulation environment that will be described in this thesis. Simulation frameworks thus emphasize design reuse over code reuse in building simulation applications. By using simulation frameworks, not only are applications built faster, but the applications have similar structures. They are easy to maintain, and they seen more consistent to their users.

A simulation framework is different from a simulation application or a simulation class library. A simulation framework is application independent and can be easily extended to build various applications. Classes in a simulation framework are loosely coupled via protocols to coordinate their requests and services, so reuse can be achieved at the system design level, as well as at the class level. A simulation framework also imposes

a data model, an infrastructure for building applications. Such a model as the composite object model described in this thesis lays a foundation for modeling behaviors in simulation.

In early 1994, Reznik [60] developed a general-purpose framework for dynamic behavior modeling using ScriptX language from Kaleida Lab. The framework creates a domain-independent model for character-based simulation for computer games. By creating an abstract model de-coupled from the presentation, the model can be used in other contexts. The framework contains the simulation-environment component that is divided into the presentation space and the model space. The presentation space contains all visible objects, as well as objects that manipulate the visual aspects of the environment. The model space contains character component, character behavior, and playground. At each clock tick, characters sense, process, and react to their surroundings. Their reaction depends both on the state of the playground and their own internal state. But the framework does not support versioning of characters. The character-based simulation only represents behaviors of an individual character component interacting with the environment (playground) rather than behaviors of a composite object with versioned components and the interaction among its components under different configurations, which is the main focus of this thesis.

Calhoun and Lewandowski's object oriented framework [9] is implemented using Smalltalk and C++. It emphasizes dynamic simulation that allows the user to define hierarchical models for dynamic systems. Using inheritance hierarchy, the user can build specific models from general ones, and using the part-whole relationship, the user can build models consisting of submodels. However, their dynamic simulation is inheritance-based implementation binding. That is different from the component-based configuration binding proposed in this thesis. The framework does not handle changes inside a composite model (contains submodels) very well, especially changes to the part-whole relationship in a composite model at simulation run time. It does not provide a change response mechanism to adjust the model during simulation. The framework also lacks version control capability,

therefore its part-whole relationship is much simpler than those which support versioned component.

Huh and Rosenberg focus their work on the change management framework [34] that manages dependency relationships between shared objects, as well as dependent user views in a collaborative simulation system. The framework provides mechanisms to coordinate changes and propagation activities between the shared objects and their dependent user views. First, it provides a set of abstract object classes that constitute the core constructs of the change management framework. Composite objects are aggregations of shared objects with dependency relationships defined among them. Second, it extends the framework in two directions: persistent shared objects and distributed simulation. At the highest level of their framework, change manager classes are provided to encapsulate all the complex structures and dynamic behavioral schemes of the mechanisms. The framework is developed under a commercial ODBMS called ObjectStore using C++. The dependency-based configuration management works well in this framework to handle changes of relationships in simulation system. But version control is quite limited for persistent shared objects, especially for composite objects. The relationship is very difficult to be versioned along with shared objects.

Recently, Vaishnavi and his group developed the Smart Object paradigm [76], which provide a new concept for the modeling, design, and simulation of the Operations Support Systems (OSS). OSS is an interactive simulation system for the management of large, complex operations environments, such as manufacturing plants, military operations, and large power generation facilities. The framework is a combination of object-oriented model and AI knowledge representation and active inferencing model. The model assists in representing simulation domain data and knowledge with its structure, in addition to modeling multilevel control and inter-object coordination. But configuration changes in OSS are prohibited during a simulation process. Dynamic behaviors are generated by OSS's control knowledge based on simulation input. Configuration of a composite object

with versioned components does not have a dynamic effect on its behavior in simulation since changes to configuration cannot be applied at run time. Such a weakness affects OSS's modeling power when system's component structure is under frequent adjustment during simulation.

The most recent advance in simulation framework comes from Cubert and Fishwick's work in late 1997. They developed a Multimodeling Object-Oriented Simulation Environment (MOOSE) [20], a framework for modeling and developing simulation software. MOOSE supports model refinement and abstraction, allows creation of heterogeneous hierarchical models including composite object models. Dynamic models comprising multi-models include Finite State Machines, Functional Block Models, Equation Constraint Models, and Rule Based Models. MOOSE emphasizes visualization and effective use of object-oriented metaphors to connect the conceptual model to the program and to capture model structure and behavior. The MOOSE human-computer interface has two GUI's: Modeler, for model design, and Scenario, for model execution control and visualization. The back end of MOOSE generates a model description in a target language such as C++, then translates and adds run-time support to form an Engine. Model execution consists of Engine running synchronously with Scenario. The MOOSE approach facilitates model development with greater intuitive appeal. MOOSE's descriptive modeling approach is less-extensible when dealing with already translated Engine. The off-line model translation also adds difficulties in directly observing dynamic behavior caused by configuration change of composite object. Our approach proposed in this thesis is more powerful. It adjusts configuration changes during the simulation run without the interruption for model re-generation. Therefore it provides a dynamic feedback to effectively steer simulation to the right direction.

1.2.2 Modeling techniques for dynamic simulation

Modeling plays a very important role in simulation. Modeling creates the way of mapping between real world entities and their counterparts in the computer environment, therefore controlling the effectiveness of the simulation.

In the last decade, many object-oriented modeling techniques have been developed and used to improve simulation [23, 40, 43, 63, 65, 70, 72, 75, 77, 87]. Some of them are capable of modeling dynamic behaviors by using constrained relationships, configurations, and control flow diagrams of a composite object.

Using constrained relationship or dependency to model dynamic behavior is one of the approaches. OBJECTModeler [23] is used in the simulation of dynamic behaviors of shopping networks. It implements an object-oriented modeling methodology supporting the definition of various types of semantic objects and their interrelationships. Its semantic object type include *simple*, *composite*, *association*, and *subtype*. Simple objects are single-valued properties. Composite objects contains one or more multi-valued properties. Associations represent relationship between objects. OBJECTModeler incorporates the mechanism to extract functional and multi-valued dependencies as well as relational constraints from the defined objects such as *customer*, *payment*, *order*, and validate the dependencies for consistency when changes occur in the simulation. Other semantic integrity constraints can be specified when objects and properties are defined using OBJECTModeler's dictionary functions that are implemented using a graphical user interface. OBJECTModeler currently does not support object encapsulation. Methods and objects have to be attached at simulation run-time, rather than treated as an integrated entity. This greatly restricts the model's extensibility in simulation. OBJECTModeler also does not support object versioning as well as configuration binding for composite object.

Another approach is based on abstraction through hierarchical modeling to create multiple views of the system to represent its dynamic behaviors in simulation. The implementation view of the abstraction can be switched at run time based on simulation

context, therefore causes the system to behave differently. Kiczales [40] describes a model of abstraction, which he calls the "two-view" approach for simulation. The first view is the traditional one; it provides the functionality of the abstraction in simulation. The second view allows the user to participate in some implementation decisions in terms of modeling. Kiczales' hierarchical modeling is inheritance based rather than composition based suggested in this thesis. It works well to dynamically bind implementations based on object's run time type, but it causes difficulties to change configuration of a composite object. HSL [65] is a Hierarchical Simulation Language for process-oriented simulation of discrete systems. It supports object-oriented programming by user-defined entity classes. It allows a parent class to be specified in an entity class definition, with the child class inheriting all data members and processes defined for its parent class. However, dynamic binding in HSL is quite weak. The model lacks support for design evolution and version management. PRISM from MCC [77] is a system-modeling environment developed for Electronic Computer-Aided Design (E-CAD). In PRISM, models manage simulation units, and because they are simulation units themselves, hierarchical models may be constructed for composite object. The simulation paradigm supported best by PRISM is one in which packets of information called transactions are passed through a fixed network of simulation primitives or submodels. PRISM supports dynamic connection between components by run time type checking on a pointer assignment extracted from the NIH (National Institute of Health) Class Library. But PRISM does not provide object versioning, therefore lacks the capability to represent possible behavior changes caused by its dynamic connections among versioned primitives and submodels.

MORE [75] improves the above approaches by supporting object structure changes in simulation. MORE is an object-oriented data model defined by BNF in which the user can specify an object with various structures. The structure of an object is changed by passing messages. The same as the above approaches, each object in MORE can take several different views. MORE also provides a class type, a user-defined type for classes

with similar structure in order to simplify the specification of class descriptions. Objects in MORE are divided into two groups: *primitive* and *complex*. The structure of an object is represented as a hierarchy of existing objects, named an object hierarchy. An object can take on several object structures, one of which can be chosen by sending a structure message according to the scenario in simulation. The change message is handled by a daemon method *Trigger* in an object. The object's structure is then evaluated as its behavior changes. But MORE does not support object versioning, therefore it cannot effectively capture behavior changes caused by configuration binding of versioned components in a composite object.

DEVS model [87] is the approach using control flow diagrams to model dynamic behavior. DEVS model provides an approach to object-oriented Discrete EVent Simulation. Its formalism provides a means of specifying a mathematical object called *system*. Basically, a system has a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs. Such an abstraction provides a formal representation of event systems capable of mathematical manipulation which includes behavioral analysis whereby properties of the behavior of a system are deduced by examining its structure. But DEVS only emphasizes the control flow of a simulation system and ignores the dynamic behavior caused by change of configuration of a composite object. Therefore it is difficult to support configuration binding in a versioned environment.

1.2.3 Simulation configuration management and version control

The iterative and exploratory nature of the simulation process leads to two aspects of the complexities. First, the configuration of a composite object (an instantiation of a composite object from its components) could be complex. Components of a composite object can be changed during simulation time as well as their inter-relationships. Configuration management is needed to response promptly and adapt to these changes without interrupting the simulation process. Second, there could be several alternative

versions of the composite object at the same time due to bindings of different versions of its components. Version control is needed to systematically document these alternatives and to easily retrieve versioned object in simulation. Configuration management and version control (CMVC) thus become a challenging task in modern simulation. CMVC must provide mechanisms to generate new configurations dynamically and consistently to respond to changes and alternatives in simulation.

CMVC has received increased attention in three application areas: object-oriented database management systems [1, 5, 24, 36, 39], Computer-Aided Software Engineering (CASE) systems [6, 27, 51, 81, 82], and CAD repositories [2, 14, 38, 46]. An object-oriented database provides object versioning. But configuration management (CM) support is limited because it does not provide a CM framework for any application including simulation. The design of a configuration management mechanism totally depends on the individual application. CASE systems provide the support necessary to allow a user to configure the modules of a program consistently [66]. They are typically language tools developed specifically for software repositories. CASE systems cannot handle semantic changes efficiently since they are not based on an explicit data model for engineering design. They do not allow complex and fine-grain simulation objects to be modeled, and as a consequence, it is impossible to define an adequate and customized simulation target model. The versioning problem is reduced to the evolution of a single file, regardless of its content, and little control is possible. CAD systems support the design of engineering objects. They focus on understanding the system-level requirements, but not on the dynamic behavior of complex objects in simulation. It is difficult to integrate application models into an existing CAD system.

The CMVC approaches can be grouped into four models. The check-out/check-in model [38] offers version management of individual system components. The long transaction model [36] emphasizes the evolution of systems as a series of configuration versions and the coordination of concurrent team activity. The change set model [88]

promotes a view of configuration management focused on logical changes. The composition model [2] focuses on improving the construction of system configurations through selection of alternative versions.

1.3 Contributions

Although great progress has been made in the past decade in simulation framework, modeling, and CMVC, many issues and weaknesses still need to be addressed. Most approaches discussed so far lack either the modeling strength to represent the dynamic behaviors of composite object and its configuration or CMVC capabilities tailored to simulation model. Their frameworks usually emphasize one aspect but neglect the other. To overcome these weaknesses in simulations, this research focuses on design issues among different aspects of the simulation framework and modeling techniques that reflect dynamic behavior through configuration changes. The main contributions of this thesis to simulations can be summarized as follows:

1. This thesis research developed a layered framework for building a simulation environment. Layers for modeling, simulation, data representation and graphical user interface are relatively independent and can be extended towards domain specific applications in parallel. The proposed responsibility-driven approach offers loosely coupled protocols to coordinate communications between objects in different layers with great flexibility. At the same time, through these protocols, the framework achieves functional integration among layers in simulation.

2. This thesis introduces a mechanism to model dynamic behavior of a composite object through its configuration change and configuration binding in simulation. The proposed Actor model is quite unique comparing to traditional control flow or inheritance based approach in previous research. Its hierarchical composite model makes configuration binding easy to manage with versioned components during

simulation, therefore makes the modeling of dynamic behavior more effective.

3. This research integrates CMVC into the underlying composite object model for the first in dynamic simulation. This native CMVC support provides a well-balanced complexity distribution into the composite hierarchy, and makes the composite object more intelligent and effective to manage its components in constraint checking, change propagation, and versioning.

Most frameworks in previous research did not cover all aspects of the simulation process. Some focused on modeling change propagation [34]; some emphasized modeling dynamic behaviors [9]; some provided a framework only for modeling or simulation [76], but not the whole spectrum that will include CMVC for simulation data and graphic user interface for user interaction. This research marks the first time in engineering simulation that an integrated framework is provided to cover and coordinate all aspects of the simulation process: from modeling techniques to simulation design, from graphical user interface to configuration management and version control of simulation data. The proposed protocol-based responsibility-driven design approach makes such an coordination seamless but also flexible. Such a framework greatly facilitates the construction of simulation applications.

The framework, InterSim, described in this thesis provides four conceptual layers of classes in the design of a simulation environment. The INTERFACE layer is an object-oriented GUI framework. It applies the model-view-controller [28] (MVC) architecture to the design of user interface for dynamic simulation. It decouples the view from the model and leaves application specific logic to the controller. So the view can be reused with different models. The MODELER layer centers on the composite object model and extends to CMVC. A composite object is an aggregation of component objects with relationships defined among them. The bindings between the composite and its components can be changed when new versions of components are created, thus cause a new version of the composite. Relationships can also change in response to component changes. These

changes result in a new version of the composite object that may demonstrate different behaviors. Dynamic behavior caused by these changes is the focus of this thesis research and is rather unique observation compared to the previous approaches. The SIMULATOR layer provides a set of simulation engines that can be extended to meet special simulation requirements. Different simulation engines can be attached to the same model to generate dynamic behavior. The DATA layer provides a uniform approach for data accessing on different platforms. The design of each layer can proceed independently, based on a set of predefined responsibilities and protocols between each layer. These four layers in the framework provide a rich set of functionality for building a simulation application.

The uniqueness of the CMVC model proposed in this thesis is its integration with the underlying composite object model in simulation. In this extended model for CMVC, configuration is represented by the composite graph through the relationship of *is-a-part-of*. Configuration changes are managed by the composite object using the dependency graph through the relationship of *depend-on*. The concept of a *modeling object* is introduced as an aggregation of modeling primitives treated as a coherent unit for versioning. Attributes of a modeling object are classified as variant or invariant in order to control the scope of versioning and change propagation. Version is recorded by *version graph* through the relationship of *is-derived-from*. *Workspace* is used to control the evolution of versions in a multi-user environment. Few workspace models are known that can properly reflect on features of team design in a cooperative environment. The workspace model proposed in this thesis organizes workspaces hierarchically to capture the process of simulation design. It supports group check-in and check-out based on object dependencies, an important feature of this model. Several approaches are proposed to control change notification, and to limit the scope of change propagation by using dependency graph. A constraint-driven approach is presented, which uses user-defined constraints to control the change propagation.

This thesis research is originally targeted to the simulation of the SSC, the largest

high-energy physics accelerator ever designed. The complexity of the design requires highly sophisticated simulation tools. The proposed system [89, 90, 92] has been implemented and the modeling strength and CMVC functions in the framework have been demonstrated. Many concepts resulting from this research can be applied or extended to other simulation application domains.

1.4 Overview of the Thesis

The thesis is organized into several chapters:

Chapter two presents an overview of the Superconducting Super Collider (SSC) and its component structure for simulation. Several issues are discussed in the context of SSC simulation requirements.

Chapter three introduces the framework InterSim for constructing a simulation system. The design of the framework applies object-oriented technology to data management, visualization, behavior modeling, and dynamic simulation. The goal is to accumulatively create complete functionality within each layer of the framework for reuse in future software development. The design and implementation of an object-oriented simulation environment, OZ, for the SSC by using the framework is described in this chapter. 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 composite hierarchy before it is dynamically bound into the system for simulation. Inheritance is used to derive new behavior of the system or subsystem from the existing one.

Chapter four proposes an object-oriented data model for dynamic simulation. The Actor model supports hierarchical structured decomposition of a system. Higher level abstraction is able to hide lower level implementations. The Actor model is used to represent a component in a system. Attributes are treated as resources, which can be

created, destroyed, or shared dynamically in the simulation. Constraints are used to impose restriction and propagate events between actors at a different level of the hierarchy. Relationship is used to coordinate activities and measure connections between actors in the system. The model allows the system to be configured by adding or deleting components to its high level composition. Configuration is bound in simulation at execution time. Inheritance is used in the model to create a different behavior while still keeping components interchangeable. Dynamic binding is used to provide a consistent interface to the abstraction, while still having a different implementation for different behaviors.

Chapter five extends the Actor model for configuration management and version control in an object-oriented simulation with a background of the SSC simulation. In this approach, version control is based on object attribute classification. Versions are grouped into version sets associated with a generic object and version relationships are modeled with version graphs. Workspace and context are introduced to facilitate configuration management. Dependency graph is used to manage the scope of change propagation. Several approaches are proposed to control change notifications.

Chapter six concludes this thesis with a list of further research issues.

CHAPTER 2

THE SUPERCONDUCTING SUPER COLLIDER SIMULATION

2.1 An Overview of the Superconducting Super Collider

The Superconducting Super Collider (SSC) is an accelerator complex built to perform high-energy physics experiments. Figure 2.1 shows the main collider rings and two zoomed-in views for different accelerators. Proton beams are fed into the main Collider synchrotrons from a chain of injector accelerators, which consist of a linear accelerator (Linac pictured at the bottom right), two resistive synchrotrons, and one superconducting synchrotron. During the normal collider filling scenario, the Linac will accelerate negative hydrogen ions (H^-) to a kinetic energy of 600 MeV (million electron volts). These ions will be charge-exchange injected into the next accelerator, the Low Energy Booster (LEB pictured at the right), and bunched into RF buckets. The LEB will then accelerate the proton beam to a momentum of 12 GeV (billion electron volts). The 12 GeV protons will then be fast extracted from the LEB in a single turn, and transferred bucket-to-bucket into the Medium Energy Booster (MEB pictured at the bottom middle). Six cycles of the LEB will

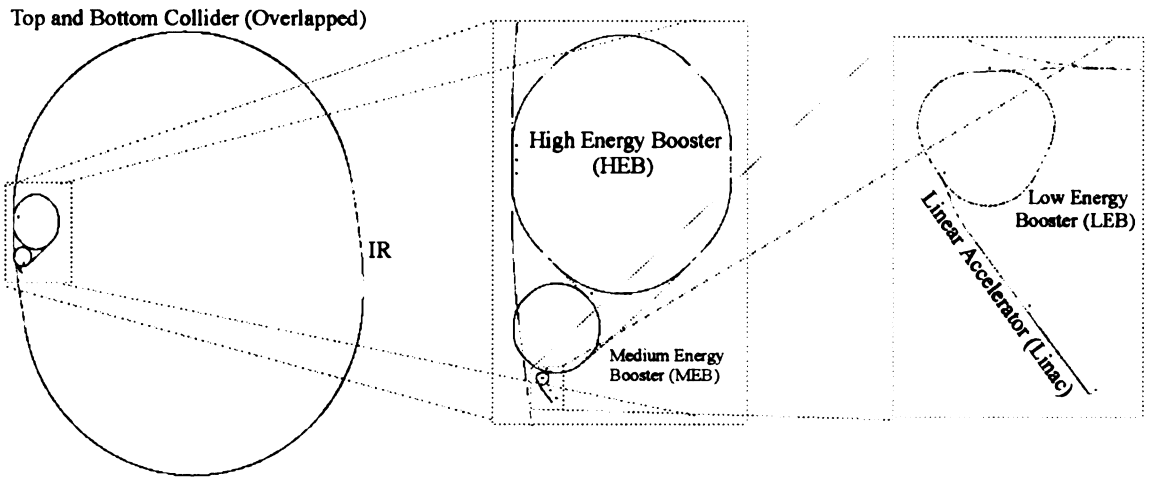


Figure 2.1: The composition structure of the SSC

be accumulated in the MEB before the beam is accelerated to 200 GeV momentum. The 200 GeV beam will then be fast extracted from the MEB and injected into the High Energy Booster (HEB pictured at the middle). Three cycles will be accumulated in the HEB before acceleration to 2000 GeV (2 Tev, trillion electron volts). The protons will then be fast extracted from the HEB and injected into one of the Colliders (pictured at the left). There will be a change in polarity and beam direction in the HEB on alternate cycles. The process is repeated 16 times until both Collider rings are filled with counter-rotating proton beams.

The HEB uses 1000 superconducting magnets in a bipolar mode, providing protons to each of the two collider rings at an energy of 2 TeV. Eventually, more than 10,000 superconducting magnets will be assembled into two rings of accelerator components within the Collider tunnel. Once the Collider is filled, the proton beam in each ring will be accelerated by the RF (radio frequency) system to an energy of 20 TeV. At this energy, the beams can be brought into collision and focused to a small spot size at four interaction regions (IR), where the physics experiments will be performed.

2.2 An Overview of the SSC Simulation

2.2.1 The component structure of the SSC

The SSC is a big composite object. The entire system consists of accelerators and beam transfer lines that connect the accelerators, each of which is called a *machine*. A machine mainly consists of different kinds of physical elements positioned one after another along the machine's trajectory. These elements can be magnets, beam adjusters, beam detectors, or RF cavities. They are used to observe, control, transfer, and accelerate the particle beam in the experiment. A machine is composed of beamlines, a logical sector of the machine. Large beamlines can be composed of small beamlines. The smallest beamline is the element. By decomposing a machine into logical beamlines, we can obtain a beamline composite hierarchy with the machine at the root, beamlines in the middle, and

elements at the bottom as leaves.

2.2.2 The SSC simulation: motivation and requirements

The basic goal of the SSC simulation is to obtain an optimal design of the accelerator *lattice* (the layout structure of magnets in an accelerator, also referred to as the configuration of the machine) in order to deliver high luminosity particle beams at a designated energy level at each stage of the acceleration for experiments. High-energy physics experiments usually take a long time to prepare and have a high operating cost. In particular, no machine yet exists to run any experiment in the initial design stage of the SSC. Simulation is a cost-effective approach to help accelerator physicists identify design flaws through visualization and improve productivity through directly manipulating design objects without the real machine. To achieve that goal, a project called OZ [89] was launched and further extended to the BumpView [90] project for dynamic simulation at the SSC using the composite object model proposed in this thesis. OZ is a dynamic simulation environment for SSC. OZ provides various types of simulations as well as data and structure browsing through a set of graphical user interfaces. As an extension to OZ, BumpView is used to adjust the beamline configuration for large particle throughput (luminosity) using the composite object model. Finally, a structured accelerator lattice editor called OBSERVER (Object-Based Structure EditoR for Visual EnviRonment) [89, 92] is built for direct manipulation of machine configuration with CMVC support using the InterSim framework. The SSC is displayed in OBSERVER as a composite hierarchy using the Actor model. Node in the hierarchy represents an object (Actor) such as a beamline or a magnet. Users can directly edit an object's attributes, constraints, and even its composite structure to create a new configuration and version it into the database.

Simulation at the SSC uses both *static* and *dynamic* data. Static data created in the accelerator lattice design are stored in the database. These data can be manipulated using a particular simulation model to create simulation results. Dynamic data are 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 three requirements of the OZ project are:

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 magnets, and injection position of the particles. A feedback can be obtained from the dynamic optics function simulator that tells the user the effect of these changes.
3. A particle-tracking simulator, which simulates a bunch of particles distributed in a predefined pattern passing through each accelerator for several turns. The simulator can also simulate particles passing through transfer lines between accelerators with different energy levels. The simulation aids research in beam synchronization, timing, and transfer of a beam within a given aperture in the accelerator.

A very important quality measurement of accelerator tuning is its beam luminosity, that is the number of particles in the beam. To increase beam luminosity, we must keep the beam inside the trajectory of a machine, i.e., avoid losing the beam in the process of acceleration. In order to achieve that, an accelerator must provide an extremely well focused and well-positioned beam to a target that could be miles away from the source. The beam is guided by a magnetic force which focuses, de-focuses, or bends the beam along the machine trajectory during the acceleration. When such a force is too strong or too weak, particles in the beam will start to deviate from their trajectories and finally get lost. Beams

are most vulnerable to being lost from stable orbits at the injection energy when transferring from one machine to another, before acceleration begins. This is due to the effect of imperfections in the dipole field of the machine bending magnets arising from errors in wire placement during magnet construction and other effects. The magnet imperfections, if large enough, could lead to unstable orbits and unacceptable particle loss. Most magnets placed in a machine have fixed magnetic strength and are designed to bend the beam with a certain angle at specified location. To correct dynamic errors that may affect the beam trajectory, hundreds of adjusting magnets (kickers) are placed among the built-in magnets (with fixed-strength). There are also hundreds of detectors (beam position monitors, or BPMs) interspersed near those kickers to monitor the results of corrections and to locate the beam positions as a result of the adjustments. For a particular BPM reading which indicates the amount of impact needed to adjust the beam, the adjusting value will be calculated dynamically for each kicker, especially those kickers near the BPM being monitored to generate such a impact in simulation. The activated kickers will change the configuration of the accelerator machine, therefore change the behavior of particle acceleration

Simulation of the SSC starts with an initial design in the design database. As the beam luminosity is continuously increased, the accelerator will start to lose beam at a certain point. A stronger binding or focusing force is needed to keep those lost beams inside the accelerator tunnel. In an accelerator, two major system parameters affect the behavior of the particle: dynamic aperture and deviation. Dynamic aperture depends on the magnetic strength of the magnets the particle passes through. Deviation relates to the accelerating pattern decided on by the structure of the machine. Therefore two approaches are used to correct the beams. One is to fine-tune the magnetic strength setting at a particular point to force the beams inside their trajectories using BumpView. But sometimes such an adjustment will either be constrained by technical limitations or have an uneven effect on particles. So another approach is to change the configuration of related beamlines using OBSERVER. Configuration changes basically replace, insert, or delete components from a

composition in the Actor model, therefore changing the behavior of the composition. A top-down step-by-step approach is used in order to achieve an ideal configuration. Simulation is first run on some small beamlines to generate a good acceleration pattern by continuously reconfiguring the beamline. These small beamlines are delivered as an abstraction to the next level of the simulation run. They are combined to form some large beamlines for further simulation in the next phase. Such a simulation is run, level by level, up and down within the composite hierarchy to reach an ideal result. The composite object model developed in this thesis research makes such an up-and-down process very effective. The concept of configuration change and binding presents great flexibility to adjust the model at different levels of the composite hierarchy, therefore provides a quicker path to find the best configuration in simulation. Further information about the simulation of the SSC can be found in [89-92].

CHAPTER 3

SIMULATION FRAMEWORK

This chapter describes the mechanisms used to build an integrated environment for modeling and simulation of large complex systems using object-oriented approaches. This mechanism has been applied to the development of dynamic simulations at the Superconducting Super Collider (SSC) Laboratory. Our goal is to build an environment that enables visualization of design data, and aids interactive modeling and simulation to exercise the SSC before it is actually built. To achieve our goal, we propose an object-oriented framework called *InterSim* for interactive modeling and simulation. In our framework, the system (SSC) is logically composed from its *components*. These components are modeled as objects that can be manipulated through graphical user interfaces. The dynamic behavior of the system is modeled using these objects in a particular configuration. Simulation results are generated by applying a specified version of a model to the simulator under certain initial conditions.

3.1 Dynamic Simulation and Framework

Dynamic simulation is different from regular simulations in the sense that in a dynamic simulation the composite object to be simulated can be reconfigured with different bindings to its components, and the user can interact with the system in the simulation process to improve the simulation model. Dynamic simulation requires an adaptable modeling technique, a model that can adjust itself to manage changes. It requires a set of extensible simulation engines that can be loosely coupled with the transitional model. It also requires a set of reusable GUI components for building different applications. The framework InterSim described in this chapter meets those requirements for dynamic simulations. It provides an architecture to seamlessly link simulation data, simulation model, simulation engine, and user interface in a dynamic simulation environment.

Our framework also improves the simulation by design reuse. It helps the user to identify key sub-systems in a simulation application and guides the user to build the application by extending or reusing classes in our framework. For the simulation itself, framework provides knowledge reuse in modeling techniques. For applications, framework can provide a fast prototype as well as easy maintenance. For simulation end-users, framework provides a more consistent operating environment in different applications.

The remainder of the chapter discusses InterSim and its implementation issues in the context of the OZ and BumpView project. Section 3.2 presents a conceptual design of the framework that contains a group of classes needed to build a system for a dynamic simulation environment. The framework contains four layers: DATA layer, MODELER layer, SIMULATOR layer, and INTERFACE layer. Each layer of the framework, its functionality, and relationship is described. The chapter explains how this layered architecture can be used to achieve design reuse in dynamic simulation. Section 3.3 will describe the DATA layer with examples. Section 3.4 will describe the MODELER layer, its attributes, and the process for modeling dynamic behavior. Section 3.5 includes several

examples of constructing dynamic simulations in an interactive environment. Section 3.6 will discuss some implementation issues in the INTERFACE layer. Section 3.7 summarizes the chapter.

3.2 Conceptual Framework Design

Before complexity reaches an unmanageable level in a software system, a new level of abstraction must be introduced to allow further increases in functionality [78]. From a simple type data to an abstract class, we can obtain one level of abstraction. From a class and a class inheritance hierarchy to a toolkit or a framework, we can achieve another level of abstraction. For design reuse in software development, a general division of abstraction beyond the class level is necessary and useful. Such a division may be termed a layered framework.

The framework proposed in this chapter contains a group of classes needed to build a system in an interactive simulation environment. A layer is quite similar to Wirfs-Brock's subsystem [84, 85], except that it is domain-specific (rather than task-specific). For example, database, model, simulator are possible layers in our framework. Classes in a framework can collaborate to achieve their responsibilities through a set of protocols. All classes that fulfill a given responsibility (probably in one layer) should respond to the same message with their own behavior. For example, in a drawing editor (such as MacDraw) several classes perform the conceptual operation of *displaying*. Class `Drawing` displays its elements, and class `DrawingElement` displays itself. The drawing editor therefore defines a message named `Display` in the class `Drawing`, in the class `DrawingElement` and its subclasses. These classes support the concept of displaying, so their displaying messages should have the same name and even the same argument list (signature), even though the methods that implement these displaying operations are very different. This mechanism is called *polymorphism* [86]. If we can design a set of protocols

for each layer, software development can be conducted by using constructs much higher than those at the code level through the use of shared protocols with homomorphism. Furthermore, code development and class derivation in each layer can be independent and targeted to the *contract* [85] advertised through the protocols.

In our design process, a system is decomposed into a group of objects (tangible or conceptual). The same objects can be grouped into classes, and similar classes can be combined into an inheritance hierarchy where the common features are shared through generalization, and the individual characters are derived through specialization. The design ends up with several class hierarchies, each of which will be designed to fulfill a particular function in the system. Classes can be further grouped into layers by their modeling and application domain. Layers are themselves independent and cohesive in terms of their functionality. But they may have relations, collaborations, interactions, and communications in the process of simulation. In InterSim, each layer defines a set of generic messaging protocols that can be used on class instances of that layer, such as the message `retrieve` that is generic to both a relational database management system (RDBMS) and an object-oriented database management system (ODBMS). Within each layer, these protocols can be interpreted differently based on class and simulation context through dynamic binding. Protocol is defined in the design model and is language independent. It 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 a request, and descriptions about the service or responsibility [85] the server offers. When a protocol is no longer adequate to a subclass, either a high-level abstraction is needed or a new protocol should be introduced to that layer. 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 interfaces among classes and extracts their abstractions to their base class. In InterSim, objects in a simulation system can be classified into four layers, each of which is implemented by a set of classes:

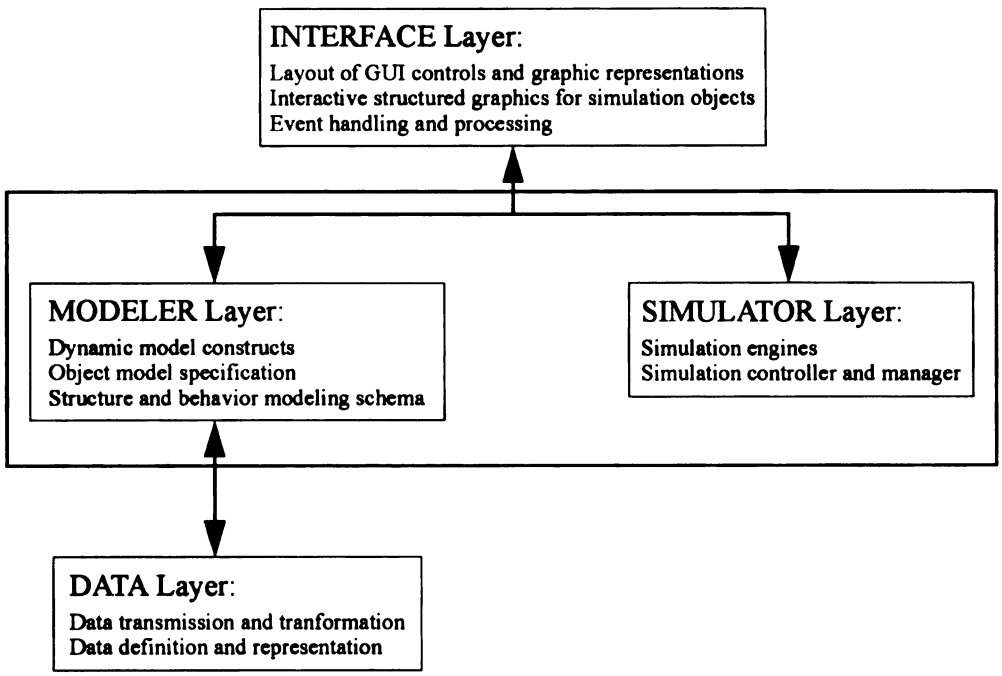


Figure 3.1: Relationship between layers

1. DATA layer comprises classes handling data transmission and transformation. It provides services for modeling and simulation. In simulation, data may come from different sources with different data formats: binary data from sensors and ports, ASCII from flat files, structural data in SDS (Self-Describing Standard) [64] files, tables in RDBMS, and objects in ODBMS. Although various data may represent the same real-world object, their data model is restricted by the feature of the repository wherein they reside. Data provider and consumer are probably loosely coupled. The DATA layer isolates the impact of the data management schema, whether flat file, relational, or object-oriented. It makes the details of data transmission and transformation transparent to its clients, and narrows the semantic gap between restricted data models in various repositories and the object models in simulation.

At the SSC, data describing the structure, identities, and attributes of the accelerator are stored in a relational database management system, SyBase [74]. SDS is used as a vehicle to move data structures between the application and the database. The DATA object encapsulates the semantic gap caused by different data models and maps data into an object model for other parts of the system, such as a simulator or a graphic user interface. As a result, high-level abstractions (DATA) will bring low-level flexibility in applications.

2. MODELER layer comprises classes organizing the information to represent the essence of real-world objects based on interrelations and interactions. MODELER defines the structure and the inter-relationships of the system in terms of the simulation to be conducted. 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 the effect of high-order magnets in the lattice. Each class in MODELER also provides a data context in which protocols get interpreted in DATA and SIMULATOR (explained below) layers. A model can be derived or composed by existing models.

3. SIMULATOR layer comprises 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. SIMULATOR layer provides a set of simulation classes used by different models. These classes have methods for special algorithms used in the simulation and control mechanism to conduct the simulation. A simulator (instance from SIMULATOR layer classes) can be built by deriving it from, or composed using existing ones.

4. INTERFACE layer comprises classes for developing a graphical user interface. It provides varieties of graphical representations to the process of modeling and

simulation. Through class derivation, INTERFACE layer can be shared among systems with few modifications. A well-established INTERFACE framework can make interface prototyping easier and faster. A consistent look-and-feel is also important to help the user learn new applications. An INTERFACE layer class can be built independently from its applications. The domain-specific editor in InterViews' Unidraw [78] is an example.

The relationship between the four layers mentioned above is illustrated in Figure 3.1, where arrows point in the direction of the simulation dataflow. Each node represents a layer with one or possibly several class hierarchies to fulfill its functionality. MODELER constructs a model using information from DATA. The model in MODELER can be manipulated 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 either direct use or derive their own domain-specific classes from high-level abstract classes in our framework. A simulation application can be built by using classes from four layers of the InterSim framework.

The layered architecture of our framework has three major advantages:

First, it promotes the loosely coupled design and development of classes (hierarchies) or frameworks for different knowledge domains. An 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 tested 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 layer protocols. So design and implementation of each layer can be relatively independent.

Second, the layered framework increases code reuse and domain knowledge encapsulation. A well-encapsulated class can be instantiated to build a more complex object, while the implementation of the original object need not be understood or modified. 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 extend the existing protocols as needed.

Third, once interfaces between the layers are clearly specified by protocols, development can proceed in parallel among class hierarchies. Independent development also makes software testing and maintenance easier and more efficient.

3.3 DATA Layer

We differentiate data modeling from system modeling (discussed in section 3.4) 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 is identical except for its color and position. In system modeling, a picture is a collection of objects with behaviors. A 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 model 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 heterogeneous environment. At the SSC, static data for each lattice are stored in several database tables such as GEO, OPTICS, and TWISS. 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. Table GEO records all the geometrical information of all the magnets in the lattice. Each magnet has an entry through the table GEO. Attributes can be referred to by pointing to other tables, such as OPTICS and TWISS, that contain detailed information about that magnet such as length, strength, and optical functions. This information is stored in SyBase, a RDBMS, as well as in the SDS files. Lattice structural information is stored in ObjectStore [35, 53], an ODBMS, in an object composite hierarchy.

A simulation model can be selected and exercised against the structural information stored in ObjectStore. Both SyBase and ObjectStore are working in a client-server environment. Data can be shipped between 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 a UNIX file system. Any abstract data type can be stored in an SDS file directly.

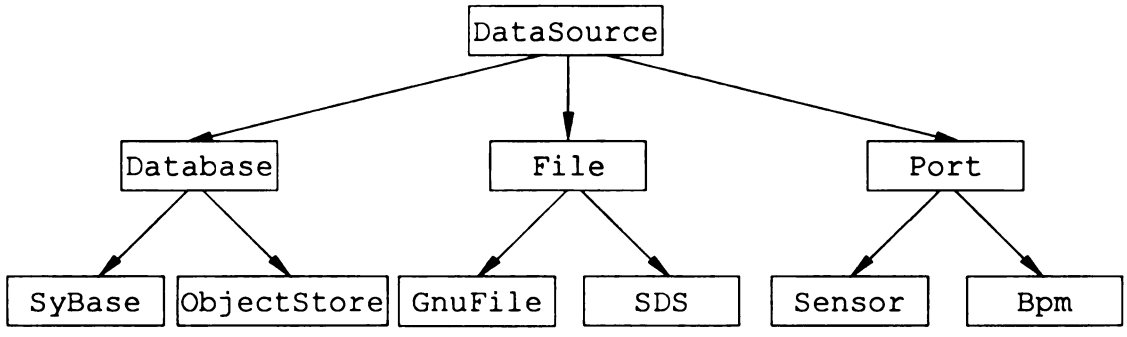


Figure 3.2: DataSource class hierarchy in DATA

`DataSource` class hierarchy in DATA layer is shown in Figure 3.2. `DataSource` is an abstract class in DATA. It represents any kind of data information used in simulation. `Database`, `File`, and `Port` are three subclasses derived from `DataSource`. A set of protocols is declared as virtual functions (when implemented in C++) in `DataSource` and can be shared or re-defined in subclasses derived from it. For example, all `DataSource` objects have operation `Open` that opens that data source with a name and a mode. Opening different databases, files, or physical ports could be done in very different ways. But the `Open` operation is generic, standing for a request from a client

to make that data source accessible from the server. So `Open` is a protocol defined at the highest level in DATA. All subclasses should support the `Open` operation. But how to support it depends on that particular derived class. There are three subclasses derived from `DataSource`.

a) Class `Database` is a base class for database operations. Class `Database` has two derived classes: `SyBase` and `ObjectStore`. `Database` supports a set of protocols such as `Open`, `Load`, `Close`, `Transaction`, `Update`, and `Retrieve`, which is generic to all of its derived classes. These protocols provide common interfaces and contracts to clients, regardless of what kind of database is used. Figure 3.3 gives a code example of how protocol such as `load` can be used with different databases. The operation of loading a database to memory (virtual or cache memory) is generic to all databases. The operation needs a `source` (indicating how to find the database) and a `mode` (read/write or read only). `Source` and `mode` are designed as classes to encapsulate various representations of `file`, `database`, and `port` and their accessing approaches. The two databases, `objectstore` and `sybase`, are constructed as an instance of class `ObjectStore` and `SyBase` with different sources. They are loaded into memory by calling the appropriate `Load` method with different modes, and the transaction is opened for accessing. The `Load` and `TransactionOpen` operations for `ObjectStore` and `SyBase` can be quite different. Even the representation of `source`, `mode`, and parameters needed for the operation can be different. The problem is how an object knows which method should be called to respond to a generic protocol such as `Load`. 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 invoked 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 among them can be encapsulated in the protocol and

```
Source* source1 = new Source(), source2 = new Source("fremont:/survey/heb");
Mode* mode = new Mode("read/write");
source1->Server("Banyan");
source1->Directory("/ssc/lattice");
source1->Name("leb");

//Create an ObjectStore object.
ObjectStore* objectstore = new ObjectStore(source1);

//Create a SyBase object.
SyBase* sybase = new SyBase(source2);
objectstore->Load(mode); //Load a database of ObjectStore.
mode->Access("read-only");
sybase->Load(mode); //Load a database of SyBase.

objectstore->TransactionOpen();
Dipole* dipole23 = objectstore->Locate("dipole", 23);

sybase->TransactionOpen();
Table* t = sybase->Locate("Element_Table");
Strength* s =t->Retrieve(23);
```

Figure 3.3: Example of using database operation protocol to Load and Retrieve data

recovered later through the process of method resolution. In C++, the preceding three approaches can be implemented using virtual function dynamic binding and skin-body class structure [18]. ObjectStore's Meta Object Protocol (MOP) [42, 52] also provides a run-time type-checking capability.

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

c) Class `Port` is a base class to model physical devices. `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, Column` in `SyBase,` and `TimeStamp` in `Port`. These classes are not subclasses of `DataSource` but are utility classes used as data members (instance variables) of it.

```
class SyBase : public DataSource {...
   char databaseName[32];... ..
   Table* T_o;
   Column* A_o; ... ...
   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 T_o. This object is dynamically created when a table is loaded and pointed to by a member variable in class `SyBase`. In SDS, the `Table` is an array of C++ structures.

Class `Column` models an attribute A_o (corresponding to a column in SyBase). A_o

is able to extract a particular field from an array of structures. Usually only some of the attributes are involved in a particular simulation at any given time. Loading the entire database table into memory usually takes lots of time and space. So making an attribute an object and loading it individually as needed is more efficient for such simulations that only use few attributes in a large table.

`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 devices. DATA layer 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 layer and other parts of the system will keep the data implementation details transparent to the user, no matter what kind of data repository or source is used.

3.4 MODELER Layer

A *model* is an abstraction (possibly a mathematical abstraction) of a real-world entity for the purpose of understanding it before building it [74]. It is natural to represent real world 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 a set 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:

1. Structure modeling (only for composite object). This step models the configuration structure of the object, the coupling pattern of its components.
2. 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 layer in our framework contains a set of model classes, each of which 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 layer 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 layer objects drive the MODELER layer objects. The MODELER layer objects generate behaviors based on DATA layer objects via its understanding and interpretation.

3.4.1 Structure modeling

Structure modeling models the configuration structure of a composite object and the

coupling scheme of its components. In structure modeling, an accelerator can be decomposed into *beamline*, which is a set of magnets placed in a specific order. The structure of an accelerator can be modeled by using configuration binding techniques. A *configuration* is an instantiation of all components in a structured system. And the same structured system, usually described in a composite hierarchy, can be configured using different components or coupling schemes, which cause different configurations of the same system (in terms of its external functionality). Accelerator is the root of this

Member Variables and Member Functions	Function Illustration
Beamline* bmLnElmnt;	bmLnElmnt point to the current component (smaller beamline or magnet)
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 realize polymorphism on all kinds of beamlines and magnets.

Table 3.1: Beamline class

composite hierarchy. It is decomposed into major beamlines; these major beamlines are further decomposed into smaller beamlines, which are in turn decomposed all the way to the magnet level. An instance of such a composite hierarchy is called a *lattice configuration* for an accelerator. The class `Beamline` is derived from `Magnet`. `Beamline` holds a collection of its components, 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` has its own methods to specify its structure. Members and methods of `Beamline` are listed in Table 3.1.

A new lattice configuration can be created by replacing an existing beamline with a new beamline or by changing its attributes (such as magnetic strength). In Figure 3.4, a

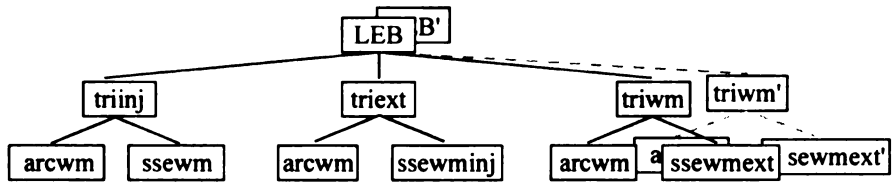


Figure 3.4: Two different lattice configurations for LEB

new design for the beamline *triwm'* creates a new configuration for the Low Energy Booster (LEB). Configuration binding is deferred at the simulation stage, and the binding actually occurs at the bottom level of the hierarchy, i. e., at the magnet level. The major advantage of this hierarchical model is its flexibility for reuse. A beamline's implementation (*triwm'*) can vary as long as its functionality in its containing beamline (LEB) remains the same. Therefore, the structure of a beamline (*triwm'*) is encapsulated from its containing beamline. A user can pick up the right beamline at some level of the hierarchy with the right granularity of encapsulations. 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 being modeled. We will elaborate on this composite model later in Chapters 4 and 5.

3.4.2 Behavior modeling

Behavior modeling seeks a set of methods governing the object's control logic and state transitions based on domain knowledge. At the SSC, three kinds of objects are modeled: 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 the distribution of magnet-field strength around it. Particle momentum and magnet strength distribution are decided by the accelerator through which a particle is passing.

Particle distribution hierarchy (PDH) is used to record a beam model. The root class `Beam` has only one particle, and it is placed at the origin. Particles with some standard statistical distributions are subclasses of `Beam`. Beam has five instance variables listed in Table 3.2. Vector $D = [d, d', \delta]$ is called the principal 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 by using a beam class library with a graphical user interface. A beam object can be created in three ways: instantiating from a beam class;

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

Table 3.2: Instance variables in Beam class

copying an existing beam from the beam object library and changing the particle distribution or amount of the particles (Figure 3.5); or as a result generated by the beam-tracking simulation.

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.

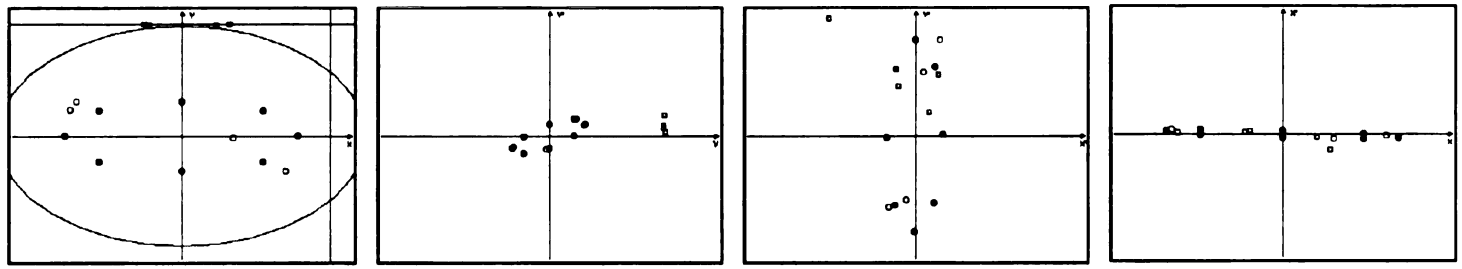


Figure 3.5: Beam objects created from Particle Distribution Hierarchy

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 principal magnet hierarchy (PMH) is shown in Figure 3.6. A prototype of the magnet attributes modeling system is shown to the right of Figure 3.6. Magnet instances are graphically represented by a collection of icons (See Figures 3.8 and 3.14.) 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.

The behavior of the magnet can be modeled by a 3 by 3 transformation matrix $M(t, s, l, o, m)$ based on Steffen's theory [69]. 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 principal vectors of a particle at position i and $i+1$, respectively. And we have $D_{i+1} = M \bullet D_i$, i.e.:

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

A set of methods is defined in class `Magnet` to support operations on the transformation matrix. The `Magnet` class definition is partially given in Table 3.3.

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 principal vector (PV) from the previous one

Table 3.3: Magnet class definition

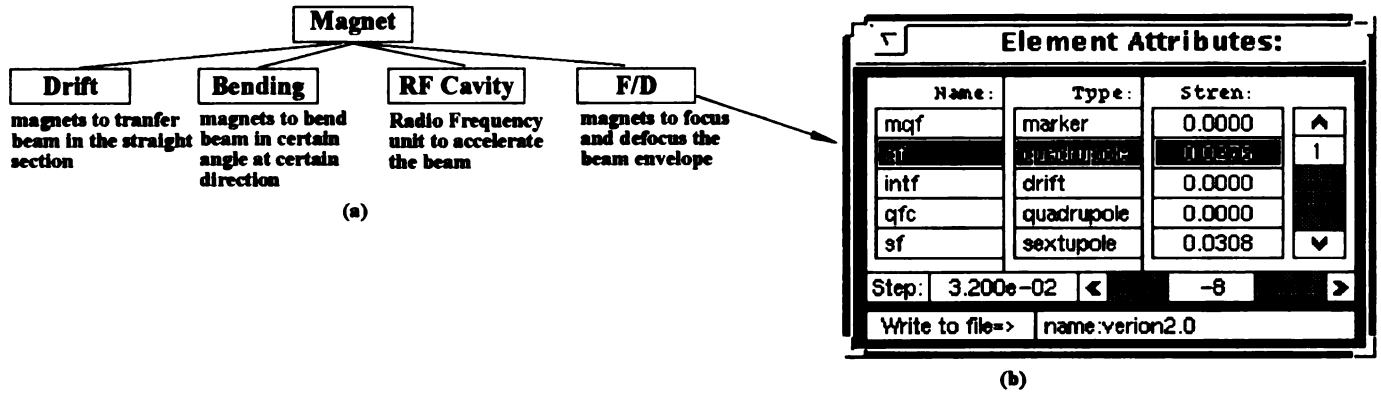


Figure 3.6: Principal Magnet class hierarchy (a) and its interface (b)

All principal 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 methods 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 the C++ virtual function. New C++ code has to be recompiled and linked into the system; 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. A virtual function dynamically binds the rule number (set by the user through interface) with a pointer to a member function to construct behavior. Interactive modeling basically becomes rule-picking and function-binding. Rules can also be added off-line using C++.

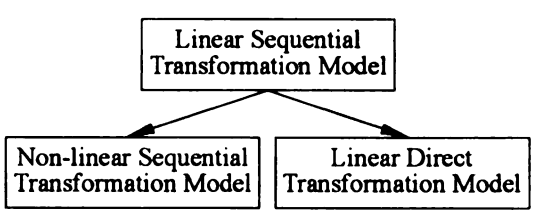


Figure 3.7: A beam tracking model class hierarchy in MODELER

A beam tracking model class hierarchy is shown in Figure 3.7. A *Linear Sequential Transformation* (LST) Model uses a composite hierarchy of the SSC. All nodes in the

hierarchy are modeled as objects of a class called Actor ([90]. The Actor model is discussed in detail in chapter 4) 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 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 from its trajectory. A car on the assembly line is also an LST. After each assembly stage (element), more parts will be assembled into the car, therefore changing the car's internal structure and state. The structure of the LST could be a tree, list, queue, or any type of generic class (also called a class template in C++). 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 tracking result and has impact on the next step. In LDT, several transformations can be linearly combined into one. Transformation is independent of the object to be transferred.

3.5 Constructing Dynamic Simulation with SIMULATOR

Dynamic simulation provides an interactive environment between the user and the simulator. It allows the user to select an appropriate model for the MODELER, to pinpoint objects (component in SSC lattice) to modify their attributes and structure, and to re-run the simulation to see the impact upon the result system. Most configuration adjustments caused

by individual modification will be propagated automatically by the simulator (by calling the proper method of the model).

SIMULATOR is a layer 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. The relationship between SIMULATOR and MODELER is quite similar to the relationship between algorithm and data structure. But MODELER and SIMULATOR are loosely coupled. Certain models in the MODELER will create certain behaviors for a specific simulation required by SIMULATOR. MODELER provides a base for creating behaviors.

The SSC simulation environment OZ [74, 89] deals with three types of models: beam model, magnet model, and lattice configuration model. The behavior of each object observed in the interactive simulation process is used to adjust the model for a better design. BumpView [90] simulates the beam bending effect caused by adjusters at certain positions. It provides a simulated beamline aperture for adjusting particle trajectory. A bigger trajectory will have a larger particle throughput, therefore increase the chance of collision in the experiment. A display panel always prompts attributes of the magnet engaged with a pointing device. A particle object can be constructed at run time using a graphical user interface object in INTERFACE layer and track through the beamline with the requested bending force set by the adjusters.

Figure 3.8 gives an example of the BumpView simulation interface. The trajectory of the LEB is displayed. The bottom part of the window is an icon representation of magnet objects in the LEB lattice. Above it are the positions of the detectors and adjusters along the LEB. All objects in the icon representation are active (sensible and associated with actions). The middle of the window (shaded part) displays the dynamic aperture of the LEB, which basically depends on the attributes of the magnet at each point. The middle part

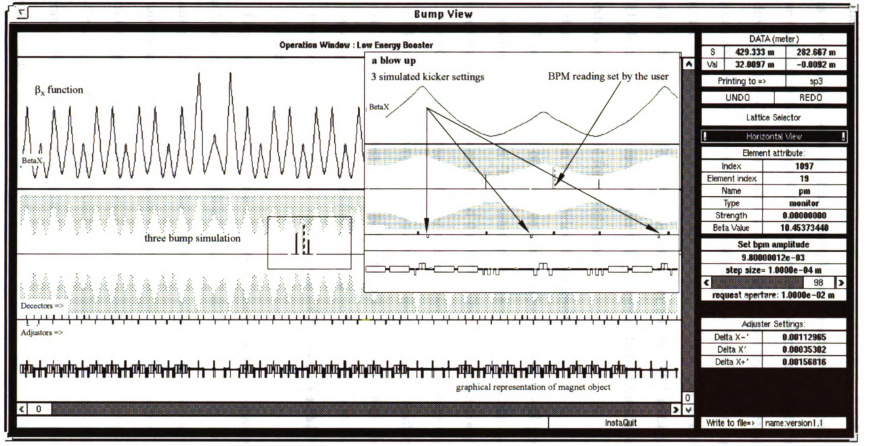


Figure 3.8: 3-bump simulation using BumpView simulator

is expanded at the upper right corner. The dashed vertical bar is the BPM reading set by the user as the expected bending effect. BumpView simulation uses only magnet model because particle tracking deals solely with leaf-level components. 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, the Detector model (for BPM) is bound to provide special behavior. Actually Detector model is derived from the magnet model for the BPM setting. So a model in MODELER can be directly used or inherited by a model derived for a specific simulation. Such extensibility is essential for model reuse.

BumpView simulation will provide 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 three green bars) 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 verifies the model is correct. In the figure, there are only three solid bars in the middle. The upper part of the window is the β-tron oscillation along the LEB.

Figures 3.9 and 3.10 present more examples of dynamic simulations using OZ. Figure 3.9a is the α function of LEB generated by simulator `Twiss`, which is capable of calculating various types of optics functions of an accelerator. These optics functions are important measurements for the design. Figures 3.9b and 3.9c are dynamic particle tracking by turns through the acceleration orbit or by every magnet along the longitudinal trajectory using `Track`, a simulator that provides a single-particle tracking profile in different accelerators. Figure 3.9d is dynamic tracking of a beam created from beam class hierarchy by using simulator `Emit`, a multi-particle tracking profiler. `Emit` can also be used to aid

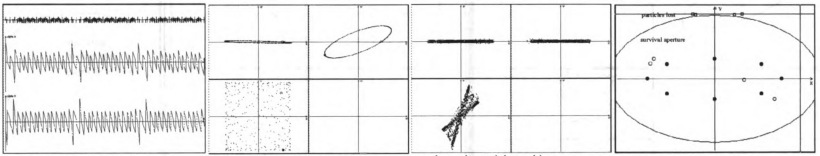
the research of relations between beam particle distribution and its particle survivability during the tracking process. All those simulators are objects from the SIMULATOR layer.

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 3.10 provides such an example. Users can change the dynamic aperture and particle emit position in the process of running the tracking simulation to see under which circumstances the particle will be wiped out. Figure 3.10a shows a particle passing through LEB and wiping out. By zooming in to the picture, the user can find the exact wiping out position. Figure 3.10b gives another example of simulating a beam passing through the LEB.

3.6 INTERFACE Layer

This section describes the functionality of data visualization in OZ and related implementation issues in details. Data visualization allows a user to directly manipulate an object and to access information through a graphical user interface for modeling and simulation. Data visualization makes modeling and simulation efficient, informative, and much easier to handle [47]. In OZ, the whole SSC complex can be visualized through a graphical window with zooming and scrolling capability. Various physics functions can be dynamically plotted through different windows, and configuration of the beamline can be edited using a graphical user interface that supports direct object manipulation.

After a particular lattice has been loaded for simulation, the position and size of each object can be extracted from a `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. Class `VisualData` is derived from `DataSource` to interface with `ViewPlot` by redefining (not re-declaring) 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. The whole plot can be zoomed in. When



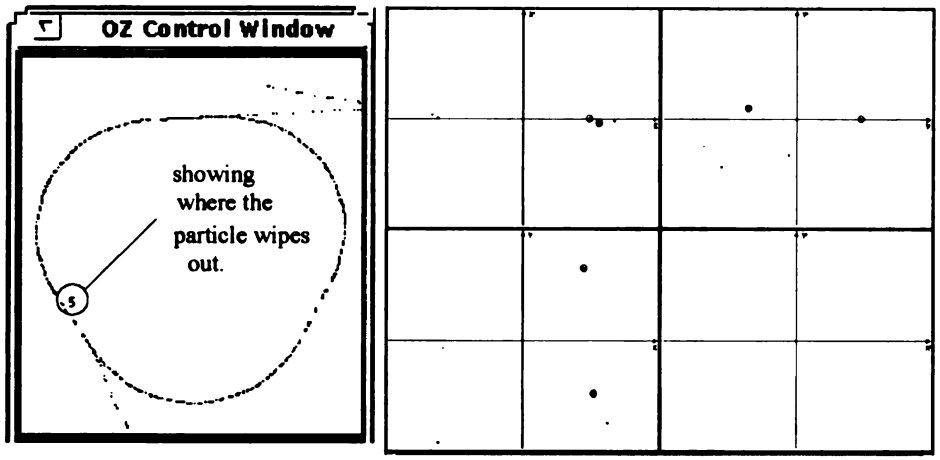
a. dynamic optics function creation of the MEB. Here is α function

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

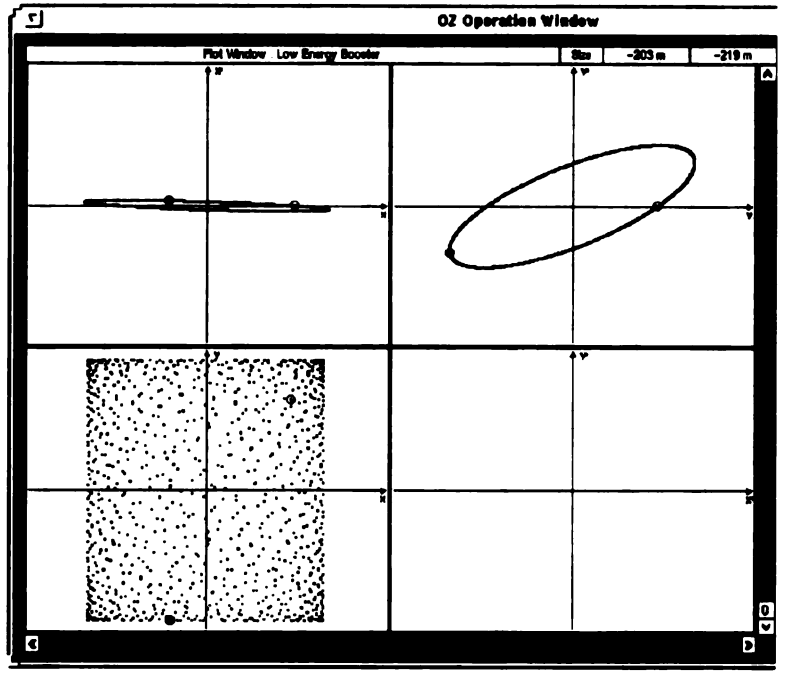
c. dynamic particle tracking for one turn and recording its d and d' after each magnet.

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 3.9: 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 relationship between lattice configuration and magnet attributes.

Figure 3.10: Beam survivability research using OZ

a plot is zoomed in, the plot boundaries are pushed onto the stack. Zooming boundaries become new boundaries for the new plot. `ViewPlot` re-scales the new plot based on the new boundaries. The undo operation is equivalent to a zoom out. The previous boundaries will be popped out from the stack and become the current boundary. Incremental drawing is also supported by `ViewPlot` for accumulatively displaying simulation results without refreshing the whole picture.

Memory is dynamically allocated for all plotting data. In order to keep all information available for redraw, memory is deleted only when new data are loaded in from a `DataSource` object. By taking advantage of the dynamic binding of C++ virtual functions, 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 factor, 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 beamline graphically and expands its components into a tree structure. Figure 3.11 is a graphical user interface for the lattice configuration browser (Observer) that provides an interactive modeling environment to the MODELER. The composite hierarchy shown in the figure can be cut and pasted using an existing component in the tree. A new component (graphically represented by `Node` object) can also be created on the fly by the user. Configuration change in a subtree will be informed to its parent component. And the parent component will update the corresponding structural model in the database.

INTERFACE layer provides classes needed to construct GUI for simulation in OZ. INTERFACE classes are developed using the Graphical Library for the Integrated Scientific Tool Kit (GLISTK) [37] and InterViews [44, 45] class libraries. Two important issues should be addressed in building a graphic interface:

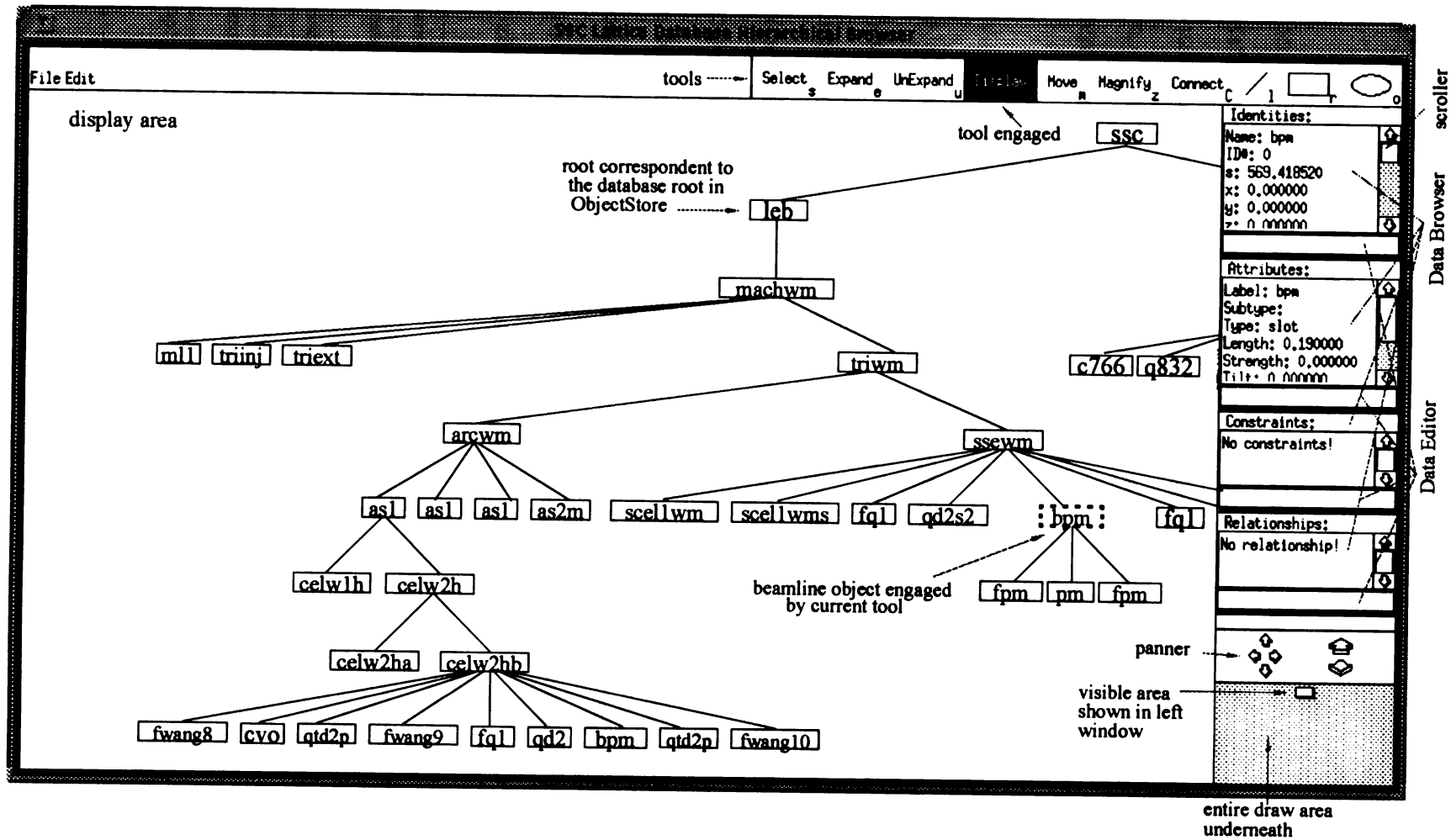


Figure 3.11: Lattice containment hierarchy browser

1. The *layout* of the interface and the connection among *control elements*, such as buttons and menus;

2. The interactive graphics (*view*).

`ControlLayout` is a base class to lay out control elements (such as button and menu) and define their behaviors. It has two subclasses: `Layout` and `ControlElement`. `Layout` is an invisible object primarily used for arranging `ControlElements` on the screen. `ControlElement` is a base class for all control elements, such as button, menu, and scroll bar etc.

`Layout` is derived from class `Gorgan` in GLISTK, which is, in turn, derived from InterViews' `Scene` (Figure 3.12). `ControlElement` is derived from class `LabelGlistk`, and class `Glistk` in GLISTK, which is, in turn, derived from InterViews' `Interactor`. Control element has a label, size, position, state, and associated control action. A label can be text or can be attached to a bitmap. 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.

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.
StopInput()	Some actions need a start event to enter its mode and wait for an 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().

Table 3.4: Protocols in class ControlLayout

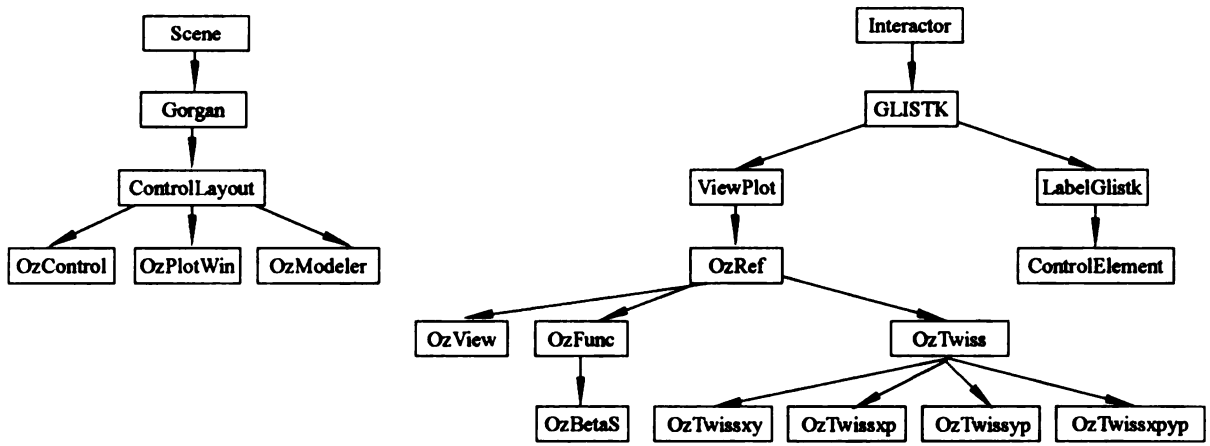


Figure 3.12: ControlLayout and Viewplot class hierarchy

Usually an instance of `ControlLayout` only instantiates its control elements (instances of `ControlElement`) through delegation. It only keeps a pointer to its control element. Control elements are created not inside the constructor but by another virtual method called `CreateAndInsert()`. Different applications may override `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 code reuse. Table 3.4 summarizes a few protocols provided by `ControlLayout`.

`ViewPlot` is a class to plot dynamic structured graphics [79] controlled by `ControlLayout`. `ViewPlot` is a GLISTK which is derived from InterViews' `Interactor` (Figure 3.12). 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, `OzRef` is a subclass of `ViewPlot`. `OzRef` keeps a list of objects drawn inside the window and encapsulates the functionality of zooming, scrolling, resizing, and refreshing. It has three subclasses: `OzView`, `OzFunc`, and `OzTwiss`. They provide the graphical representation of objects to be drawn. `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 require incremental drawing. The result of several simulations can be superimposed 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 "cached" so that when the window is opened later, the previous image can be restored as before. 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 3.13). `Refresh` 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 them on the screen. If the window is closed and then opened again, `Refresh` will be called. `Refresh` will in turn call `Draw` to cumulatively draw whatever is in the `IDQ`. Figure 3.11 illustrates how the incremental drawing queue works.

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

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

Figure 3.13: Methods for incremental drawing

Because thousands of magnets need to be drawn on the screen, making each magnet a structured graphic object in InterViews is not realistic. On the other hand, 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, a set of methods must be designed to do the mapping among objects on the screen, their ID# in `ViewPlot`, and their data in a `DataSource` object. Figure 3.14 shows such a 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 filtered out. An object caught by the mouse 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

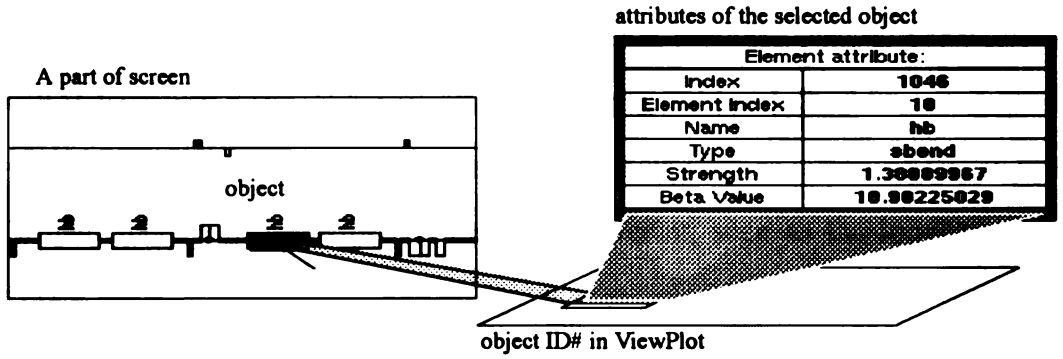


Figure 3.14: Object mapping

through MODELER. Some protocols of `ViewPlot` are listed in Table 3.5.

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 scrolling
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.
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).

Table 3.5: Protocols in class ViewPlot

As a good example, let's consider drawing a beamline on the screen. A list of graphic objects is created for a pictorial representation of the magnet. The corresponding magnet object is either referenced 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 composed 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 via a class called Communistk. State is a variable with a valid C++ type in ControlElement. State can be attached to 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. Communistk is an abstraction of a state variable. It encapsulates the notification and update mechanism of a state variable. Every Communistk has a CommuList, a list of ControlLayout, which is informed any time the value of state on which Communistk is focused is changed by Communistk::SetValue. Every object that attaches to a Communistk is automatically added to that Communistk's CommuList. 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. Such notification can also be put on hold by calling Communistk::SetValueNoHit. A message cannot only be sent back and forth between ControlElement and ControlLayout, but also can be sent out to another application using the GLISH event sequencer [57]. 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 [44, 45]. 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`.

3.7 Summary

In this chapter, we have described the design and implementation of object-oriented simulation environments OZ and BumpView. The issues of building a reusable simulation framework InterSim have been discussed. Our framework decomposes a simulation system into four types of classes (layers). Each layer handles data management, user interface, modeling, and simulation, respectively. Protocols are pre-defined among layers. These protocols are abstract and generic to achieve maximum reusability. Each layer in InterSim is relatively independent and focused on its own problem domain. Services can be requested by classes in one layer to classes of another layer based on protocols. Such a loosely coupled, responsibility-driven design approach not only promotes productivity but also simplifies the development and maintenance process.

CHAPTER 4

OBJECT-ORIENTED MODELING FOR DYNAMIC SIMULATION

4.1 Introduction

This chapter will focus on the object-oriented modeling techniques for dynamic simulation. Object-oriented means that the decomposition of a system is based on the classes of objects which the simulation manipulate, rather than on the functions which the simulation performs. The target of object-oriented modeling is the object-oriented decomposition of user's needs into executable language constructs [3]. The object-oriented decomposition process can be sub-divided into *analysis*, *design*, and *implementation* phases. The important characteristic of object-oriented development is that these three phases adopt similar models, an object model. The analysis phase consists of mapping between the *real world* entities and the entities in the analysis model: object. The mapping is also called *domain analysis*. It concentrates on two specific types of relations: aggregation and classification. Dynamic behavior of the system is realized by object interaction after the relations between objects have been determined. The design phase

consists of mapping the analysis model to a design environment. Aggregation defines the configuration of the system. Classification structures manifest themselves as class inheritance hierarchies. The implementation phase consists of mapping between the design model and the language model.

In dynamic simulation, a system is viewed as a collection of interacting objects bound with certain relationships and constraints. Each object models an entity or event in an application domain. Objects and classes allow the modeling of application concepts and their relationships in a natural way. They support not only structural definition, but also the modeling of dynamic behavior. Applying object-oriented concepts to the modeling technique has advantages over the traditional approach. It promises an improvement in modeling capabilities, since the modeling concepts of classes and class hierarchies bridge the semantic gap between the problem domain in the real world and the solution domain in applications.

Our model is based on the concept of class inheritance and relationship. In our model, class *Actor* is used as a base class to model any object in a system. An actor (an instance of Actor) has *identities* to identify itself, *attributes* to represent itself, and *constraints* to adjust itself. An Actor can be structurally decomposed into its children. An instance of such a decomposition is called a *configuration*. In Actor, identity, attribute, and configuration are bound with constraints. In dynamic simulation, the configuration of an Actor can be modified not only between each simulation run but also at run time. It is similar to an interactive programming debugger. The user can set break point, initiate the run, stop in the middle of the run, check or change some variables and resume the run. But a debugger does not have any knowledge of the logic inside the program being debugged. It just passively takes over the new value and continues to run even if this value is not logical to its semantics. Our model provides more dynamic capability. A new configuration introduced by the simulation at run time will be judged by our dynamic model based on constraints and relationships.

In our model, relation (also called association in [63]) between two objects is modeled by the class Relationship. Relationship contains information that is not subordinate to a single object, but depends on two or more objects. For example, the connectivity between two Actors is a sort of relationship. Relationship defines participants and the nature of the connection. Relationship also has the knowledge about how to build and manage the connection. Relationship is important for dynamic simulation. It provides a way to record interconnectivities between objects and a platform to adjust them based on constraints.

The organization of the chapter is as follows:

- Section 4.2 describes the basic modeling concept based on the Actor model. Some modeling techniques are introduced including virtual constructor and skin-body design pattern for dynamic simulation.
- Section 4.3 presents an example from the SSC simulation using our dynamic modeling techniques.
- Section 4.4 discuses implementation issues including how data are stored in and accessed from the database (navigational, query, transaction management), and how dynamic simulation is run using the Actor model with detailed examples.
- This chapter is summarized in section 4.5.

4.2 Object-Oriented Modeling for Dynamic Simulation

4.2.1 Functional requirements

Simulation is the process of designing a mathematical-logical model of a real system and experimenting with this model on a computer [58]. Simulation modeling assumes that a system can be described in those terms acceptable to a computing system. In object-oriented simulation, a system can be characterized by a set of interacting objects with states. A simulation involves observing the dynamic behavior of a model by moving

from state to state in accordance with well-defined operating rules designed into the model. Dynamic simulation is a type of simulation characterized by continuous change, activity, or progress of object interaction and system configuration. It not only allows the user to obtain continuous feedback from the system through automated analysis and adjustment procedures during the simulation, but also provides direct configuration manipulation to objects composing the system. It supports both dynamic behavior binding and configuration binding. In dynamic simulation, the type (class) of an object can be dynamically changed; therefore, its run time behavior can be dynamically bound based on its run time type. The definition and semantics of a class also can be evolved during the process of simulation without any reinterpretation of the data. The configuration of an object (in terms of its object-oriented decomposition) can be altered without losing its logical consistency in the dynamic model. A model for dynamic simulation should be able to:

1. guarantee that the configuration evolution is logically consistent to the rest of the system, compatible with the previous version;
2. adjust and manage the new configuration to fit in the system it belongs;

Our modeling technique supports the concepts of decomposition (i.e., how a system is hierarchically broken down into components) and coupling (i.e., how these components may be interconnected to reconstitute the original system). A configuration change includes: modifying the object's composition structure, changing the object's representation (identity, attribute), and semantics (method definition). Operations that could change an object's configuration include *inserting* or *deleting* components from the configuration, *setting* values to the identity or attribute of an object, *changing* definitions of a method, and *adding* or *removing* members from class definition (schema). By attaching constraints to those operations, we are able to build a change propagation hierarchy that is a reverse-ordered composite hierarchy. Such propagation can be either bottom-up triggered by a particular dynamic reconfiguration, or top-down explicitly called by the simulation

program when the semantics or schema are affected.

4.2.2 Basic modeling concept for dynamic simulation

A model is a mathematical and logical abstraction that is used by a computer to simulate the real world. Simulation vs. modeling is parallel to analysis vs. design. Simulation is done in a problem domain, and it deals with real objects such as a magnet, particle, and engine. Modeling is done in the solution domain. It deals with a class, object, relationship, message, etc. Behavior is driven by attributes and identities. By assigning attributes and identities to each object, the user can do dynamic simulation, but that is not all. Configuration manipulation can also create dynamic behaviors. A good example is the simulation of a power supply. The failure of a power supply to a particular set of beamlines in SSC will actually reconfigure the structure of it and therefore change the behavior of the beam and the accelerator.

This section provides definitions of the modeling concepts in a dynamic simulation environment. These definitions are presented to establish a basis for further discussion. Actor and relationship are described as tuples to model different aspects of the object. On the one hand, Actor is a base class for further derivation in the application domain (SSC is an example). On the other hand, Actor supports hierarchical structural decomposition that is a backbone for dynamic configuration manipulation. The following concepts are used as basic constructs in our model to support those functional requirements stated in section 4.2.1.

- *Object* is a structured persistent data with methods that have the knowledge to operate on the data.

An object may have a transient extension to a particular application. Such an extension depends on the interpretation of the data by different applications. For example, a digital switch is an object. It uses structured data for representation and methods for behavior description. Such an abstraction is persistent and can be retrieved from a design

database for different applications. A graphical switch is its transient extension to a particular application. It defines its look-and-feel that is not essential to its abstraction, and may vary from different applications.

• *Actor* is a base class in our model that is defined as a 6-tuple:

Actor ::= {Identity, Attribute, Constraint, Parent, Children, Method}

Actor represents the basic component in our model. We use it to differentiate the general concept of object (above). Actor is also used as a skin (envelope) class [18, 22] to support dynamic typing; run time type checking; and encapsulate type casting and message dispatching. The skin/body pattern is a technique using a pair of classes that act as one: an outer ("skin" class) that is the visible part, and the inner ("body" class) where implementation details are encapsulated. The skin is said to forward requests made of it to the body inside it through a technique called delegation. The skin/body class idiom is used when the skin class and the body class share the same behaviors but when the body class behaviors specialize or augment the behaviors of the skin class. The two classes are related in much the same way that a base class relates to a derived class in our Actor model, but with more run-time flexibility than found from inheritance alone. The important design guideline of our object-oriented dynamic model is that each class should know how to take care of its own business. "Much of the power of object-oriented programming is in the polymorphism that comes from combining inheritance and virtual functions" [18]. Inheritance and virtual functions used together enable a user to communicate with an object exclusively through the interface defined by its base class. Messages are dispatched to the appropriate derived class member function at run time. Now the question is: where is the object made, and what information was at the disposal of the creator to indicate which derived class to use? How are data instantiated as objects and in which class do they belong? Our model provides Actor as a skin class, a third party abstraction as an agent to do business with a derived body class (elements in the simulation) on behalf of the client creating the object. The Actor class now appears to be able to build different kinds of

objects at run time based on identities determined by context provided to the constructor in the dynamic simulation. This mechanism gives classes the run time flexibility that virtual functions give objects, a capability that has come to be known as *virtual constructors*.

Identity is used as a property for an instance of the Actor to distinguish it from all other instances. For example, a social security number is an identity of a particular person (an instance of class People). Identity must be unique among all instances of the class and meaningful to serve assigned purposes. *Attribute* is a metric for, a characteristic of, or a piece of information about an instance of the Actor. Different from identity, attributes do not belong to a particular instance of the Actor. It is treated as a resource shared among Actors. Different instances can share the same attributes. Attributes are dynamically created as needed and can be changed or deleted at run time. Attribute uses an internal reference counter to track its subscriber, manages its own memory and destroys itself when no longer needed. For example, the magnetic field strength is an attribute of a magnet object. It can be shared because magnets of the same type will have the same strength. However a particular magnet can change its strength during the simulation process, thereby changing its original attribute. The previous attribute strength can no longer be shared. A new attribute strength has to be dynamically created in the simulation process and bound with that particular magnet object. Such an attribute model directly supports dynamic simulation by allowing attaching, detaching, and picking attributes for an object at run time.

Constraint of an Actor specifies a restriction or criteria to adjust inconsistency and exception on an Actor to a given course of action or inaction. A constraint can be considered as a sort of attribute except that it is used to impose restriction, and adjust the relationship between objects. For example, the aperture of a magnet is a constraint that decides whether a particle can pass through it. Constraint either raises exceptions in terms of restriction (particle will wipe out) or adjusts the magnet aperture by using some predefined criteria. The concept of constraint is critical to dynamic simulation that allows configuration change. Such a change will not only alter the structure of the system, but also

affect the relationship among objects that compose the system. The adjustment of relationships and the resolution of inconsistency in a dynamic simulation is driven by constraints. Another good example is from [80] about the symmetric house. The house in our model is the Actor. A house can be decomposed into windows, doors, walls, and so on. in certain layouts that form a configuration. Suppose we want to enlarge the right-hand side window. Under the space constraint, we have to also change the configuration of the wall to make extra room for the enlarged window. The symmetric constraint may also want us to enlarge the left-hand side window for consistency.

Parent and *Children* are used to establish a *1-n* relationship among Actors to support their composite hierarchies. The parent-child relationship directly supports hierarchical abstraction in dynamic simulation. A reconfiguration at a certain level will only affect all the levels below, but not those levels above. High-level components encapsulate their low-level implementations that will greatly simplify the system configuration at the functional level. Parent-child relationship also ensures data consistency and proper communication between the parent and the children using constraints, when events happen on either of them. For example, when a child is deleted from its parent, the child object may be informed to disconnect its relation with the parent, and vice versa. By creating objects at run time using virtual constructor, a new aggregation of objects can be instantiated and inserted as a child at run time. Therefore, the configuration of a system is no longer a static concept. *Method* of an Actor stores the knowledge about the Actor and generates its behaviors. These methods are virtual and are always bound dynamically based on run time type. Such a mechanism is called polymorphism, which is useful in dynamic simulation to apply the same message to different configurations. Method can be external or internal. The external (public) method, also called operation, exists in the interface to an Actor and advertises a capability. The internal (protected or private) method defines the actual mechanism (algorithm) by which an operation is accomplished.

- *Relationship* is an object defined as a 4-tuple:

Relationship ::= {Participants, Relation, Constraint, Method}

Relationship is a physical or conceptual connection between Actors. The relationship concept was first inspired by the concept of association in [63]. We further developed the idea based on constraint-driven approach.

Participants are Actors involved in the relationship. Such a relationship can be *1-1, 1-n, m-1, n-m*. For example, the distance between two objects can be defined as a *1-1* relationship. It is neither a property of either object, nor a simple pointer to each other. It has value and the knowledge to interpret the value. When an object changes its position, it informs the distance relationship, which recalculates the distance as its new value. *Relation* provides a quantitative measurement to represent the relationship. Relationship is implemented as a template (generic class), so relation could be represented as a floating number, a structure, or even a class. For example, a float number may be used as a representation for the distance relationship. *Constraint* of a relationship provides criteria to create an event or trigger when relationships change. For example, the distance between two particles will decide the dominating force acting on them; different implementations of a beamline should have the same length. *Method* of a relationship stores the knowledge about the relationship. How to calculate the distance between two actors is an example. The major difference between our relationship model and traditional relationship models is that our relationship is defined for each object rather than on its type (class). Relationships can be dynamically changed through each particular object participating in the simulation.

• An *object-oriented model* is defined as a set of Actors and a set of Relationships.

4.2.3 Modeling technique

Our model provides a virtual constructor for all derived classes from Actor (the skin class). Actor in our model is both a base class and a skin class. External methods (public) are defined as virtual in class Actor to provide run time dynamic binding. Figure 4.1 is a simple example. A department has four types of persons: Engineer (E), Scientist (Sc),

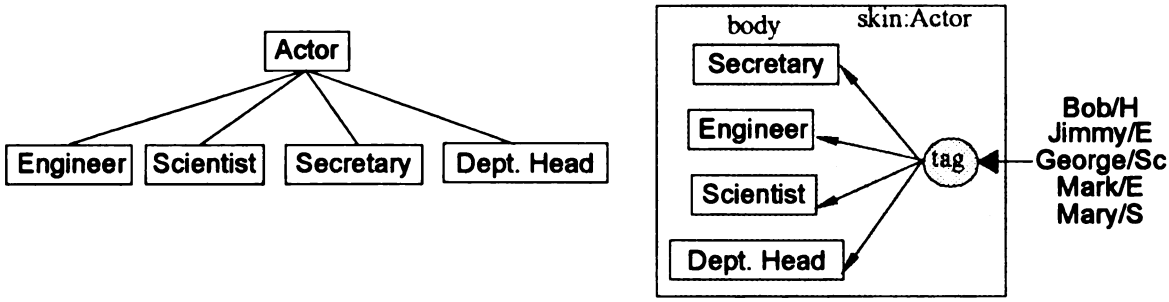


Figure 4.1: An example of virtual constructor in class Actor

Secretary (S) and Dept. Head (H). They are all derived from class Actor. They all have their own member functions called Work() that is defined also in Actor as a virtual function. By applying our modeling technique, everyone in the department is an instance of an Actor. Actor provides a virtual constructor that takes any person in the department and turns it into a particular type (H,E, Sc, S) based on some tag value such as employee identification number (an identity). But such a representation is internal to class Actor. From the user's point of view, everyone is just an Actor. When a message Work is sent to an Actor, it passes the message to its internal representation, the body class, which could be an Engineer. Because method Work() is defined in Actor as virtual, it will be dynamically bound to Engineer::Work() at run time. Actor is not only able to construct an object at run time, but also provides encapsulated run time behavior. The skin/body pattern also provides exception handling for the dynamic simulation. An encapsulated representation will allow run time type checking. By making a derived class called Default from Actor, Actor will turn someone like Technician to a Default type and raise exceptions. An exception handler in the constraint may pick this exception and do something like turning a Technician to an Engineer with less experience.

A configuration change in our model includes: modifying the object's composition

structure, changing the object's representation (identity, attribute), and semantics (method definition). Figure 4.2. shows an example of an engine configuration. Engine' changes

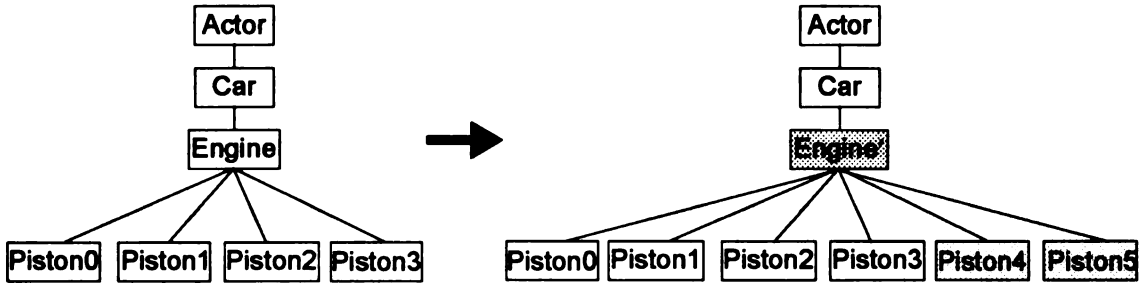


Figure 4.2: Dynamic configuration binding

Engine's configuration by adding two Pistons to the Engine. Engine::Insert(Piston4) will first check its bound constraint to see if such an insertion is acceptable. If accepted, relationship Ignition Sequence will changed to include Piston 4. Finally, this adjustment is propagated to the up level, Engine, to adjust engine size (due to extra pistons), acceleration power and fuel consumption. Then a new engine, Engine', is created and can be used by the same Car. Usually, such a change will evolve the system to a new design. The engine will always be bound to the newest configuration. The user can keep old configurations by versioning them. But the application has to point out what version is currently engaged with the simulation to provide information for run time configuration binding. Another good example is to tune up an engine. The machinist will first start the engine and then tune each part of the engine one by one. He can change parts or adjust settings. The engine is always bound to its newest version. From the logical point of view, different versions are different objects; they may have different behaviors. But physically they are instances from the same class; they are interchangeable.

Different versions of Engines are stored in a design library built on an object-

oriented database management system, ObjectStore [35]. In order to take advantage of ObjectStore's version facilities, the user can derive his Engine class from both Actor and ObjectStore's Configuration class to gain versioning support. Our experience shows that there is too much overhead at run time by making everything versionable through ObjectStore's Configuration. We leave the decision to the application developers to control the version granularity. Chapter 5 discusses this further by introducing the concept of modeling object as a coherent versioning unit in version control.

4.3 Object-Oriented Modeling for Dynamic Simulation Applied at the SSC

4.3.1 A composite object model for SSC simulation

SSC is a complex system comprised of several particle accelerators (a partly zoomed picture is shown in Figure 2.1). The entire system consists of 12 accelerators and beam transfer lines, each of which can be called a *lattice*. A lattice can be decomposed into a beamline hierarchy. At the bottom of the hierarchy, magnets, adjusters, detectors, and beam pipes are used to control, transfer, or accelerate the particle beam. These are called *elements*, and they are physically installed in the machine. Elements arranged in a pattern form a *beamline*. Small beamlines can be used to form a larger beamline. The whole lattice is composed of beamlines in the hierarchy. To some extent, element and lattice is a specialization of beamline. The behavior of the particle such as its aperture and deviation depends on the magnetic strength of the magnet it is passing through and the accelerating pattern — the structure of the lattice. Dynamic simulations at the SSC are both discrete and continuous type simulations. For example, beam bump simulation is a continuous simulation. It takes a BPM reading from somewhere of the lattice and predicts its surrounding adjuster (kicker) settings by solving 3 differential equations [71]. Events are

initiated from that BPM and handled by methods locally in the BPM or globally in the lattice the BPM belongs to. Events are treated as messages passing from one element to another through the lattice. The state transition of each element and lattice is continuous. But in terms of particle tracking, some attributes can only be changed to certain values. The internal state of the attribute is discrete. Lattice simulation starts with an initial lattice design from the physicist in the design database. The design database records the design based on our Actor model. When the beam luminosity is continuously increased, the accelerator will start to lose beam at a certain point. A stronger binding or focusing force may be needed to keep those lost beams inside the accelerator tunnel. Two approaches are used to correct these lost beams. One is to fine-tune the magnetic strength setting at a particular point to force the beams inside their trajectories. But sometimes such a strength adjustment will be either limited by technical availability or have an uneven effect on particles. So another approach is to change the configuration of related beamlines. Configuration changes basically replace, insert or delete components from a composition, and therefore change the behavior of the composition. Top-down step-by-step approach is used in the lattice structure design. Simulation is first run on some small beamlines to generate a good acceleration pattern by dynamically reconfiguring the beamline. These small beamlines are delivered as an abstraction to the next upper level (their parent). They are combined to form some large beamlines for further simulation in the next phase. Such a dynamic simulation is run, level by level, up and down within the lattice hierarchy to accomplish a better design. The next stage is to fine-tune the attribute setting of individual magnets (i.e., beam adjusters). Attributes can be assigned to a magnet for a particular simulation run. Since changing attributes of a particular magnet may affect local or global attributes of the beamline, constraint should be imposed, and relationships between beamlines should be updated. Change of configuration will change the sequence of iteration in particle tracking and semantics of message passing between magnets. Change of attributes will affect the parameter of these differential equations governing the

behaviors of particles and accelerators. These changes are propagated by relationships defined between magnets and constraints defined on magnets. Relationships will be updated to change the sequence (timing) of the messaging (event). Constraints will adjust the relationship to make change consistent through the system.

Generally speaking, a beamline can be mathematically defined as a linear composition function of other beamlines: $beamline = F_{linear}(beamline_1, beamline_2, ..., beamline_n)$, $n \in$ integer.

For example, consider a beamline α, which is composed of beamline β followed by three instances of beamline ε; thus we can have $\alpha = \beta + 3*\varepsilon$. Note that $\beta + 3*\varepsilon$ is not equal to $3*\varepsilon + \beta$. Such a function F_{linear} defines a configuration of α and can be represented as a hierarchical graph as shown in Figure 4.3.

Figure 4.3: Beamline representation

We have three instances of ε in α. They are exactly the same object except for their different positions. We extract their differences out and call them *identities*, ε', ε'', ε''' in our Actor model. The remaining sharable parts are called *attributes*, $\underline{\varepsilon}$. Such a model provides the flexibility to replace or modify the attributes of objects. α is an abstraction of $\beta + 3*\varepsilon$, and $\beta + 3*\varepsilon$ is an implementation of α. The dynamic behavior of α depends on the implementation of its configuration and the behaviors of its components. In lattice beamline hierarchy, each node represents an object or entity with identities, attributes and behaviors. Each node has relationships with other nodes. Our model supports this

hierarchical abstraction, maintains relationships, and facilitates dynamic simulation for the SSC.

As an example, let's look at the hierarchical composition structure of the LEB in the SSC shown in a graphical browser (Figure 4.4). LEB consists of one beamline (child): *machwm* (LEB = *machwm*). *machwm* consists of four children: *mll*, *triinj*, *triext*, *triwm* (*machwm* = *mll* + *triinj* + *triext* + *triwm*). And *triwm* consists of *arcwm* and *ssewm* (*triwm* = *arcwm* + *ssewm*). *arcwm* consists of three instances of *as1* and one instance of *as2m* (*arcwm* = 3**as1* + *as2m*). Such a decomposition procedure can be continued until we reach the leaf level of the hierarchy (level 0), where every object is decomposed into an element. This decomposition process creates a hierarchy where every node is implemented as an instance of Actor in our model.

An Actor has children, a parameterized array (collection) of Actors. An Actor also has a parent (that is also an Actor). In Figure 4.4, *celw2h* is a child of *as1*, and *arcwm* is the parent of *as1*. The parent-child relationship is one type of relationships (1-to-m) supported in our model. The participants can be the parent plus all of its children. The constraints are defined as a set of rules to keep parent and its children logically consistent. For example, if *celw2h* changes its length, *as1* will also adjust its length accordingly. On the right side of the browser in Figure 4.4, identities, attributes, constraints and relationships are presented with selected Actor (*bpm*). Because of limited space, the entire expanded hierarchy of LEB cannot be shown above. But if each *as1 is expanded* with Display Tool engaged, three instances of *as1* will be displayed that have different identities such as ID#, longitudinal coordinate *s*, and position *x*, *y*, *z*, etc., and the same attributes such as *length* and *strength*, etc. If the hierarchy is expanded under these three instances of *as1* as was partially done to the first one, three identical subtrees will appear. These subtrees under *as1* internally refer to the same configuration. So in terms of identity, the composite hierarchy is a tree. But in terms of attribute and configuration, it is a directed acyclic graph.

By using our hierarchical model in SSC lattice simulation, an object (such as *as1*)

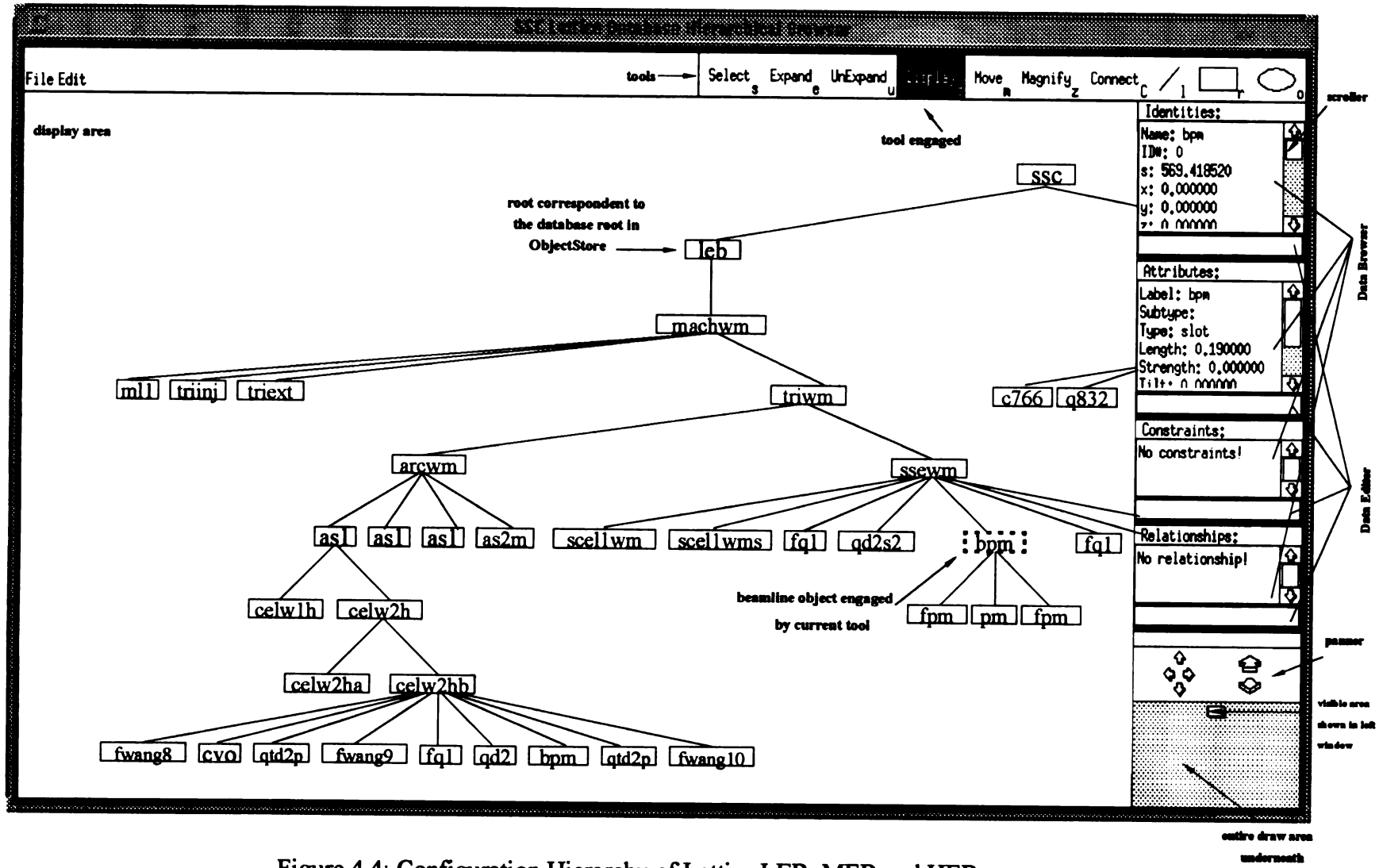


Figure 4.4: Configuration Hierarchy of Lattice LEB, MEB and HEB

in the composite hierarchy can be practically redefined in terms of its configuration without modifying the simulation program.

4.3.2 Class inheritance hierarchy in SSC simulation

In this section, we will examine all classes in the SSC simulation. Figure 4.5 is a

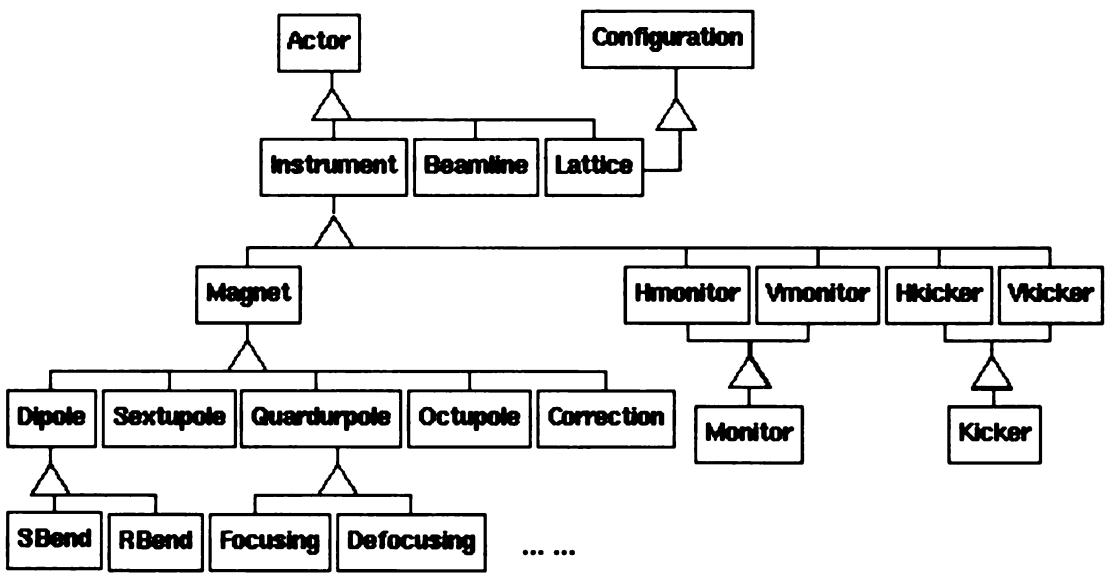


Figure 4.5: Class inheritance hierarchy in Super Collider

class inheritance hierarchy of simulation components of the SSC represented in OMT[1] notation. Root Actor is a base class that provides all the concepts we discussed in the previous sections. These concepts deliver the support to the dynamic simulation which can be reached through inheritance. Non-leaf level classes are used for further derivation and functional sharing. Leaf level classes are used for object instantiation. `Lattice` and `Beamline` are composite objects built from all types of `Instruments`, which are the

1. The notations we used here and the rest of this paper for modeling object classes and their relationships are defined in James Rumbaugh's *OBJECT-ORIENTED MODELING AND DESIGN*, 1991 by Prentice-Hall, Inc. ISBN 0-13-629841-9

elements in the accelerator lattice. `Lattice`, `Beamline`, and all types of `Instruments` are derived from `Actor`. Class `Lattice` is also derived from class `Configuration`, which implements the concept of versioning in ObjectStore. `Instrument` can be further classified into `Monitor` (horizontal or vertical) which detects the particle's position, or `Kicker` which adjusts the particle's position, or into different kinds of `Magnet` which affects the particle's behavior. Multiple inheritance is used to functionally combine horizontal and vertical monitors into a monitor. The same case applies to kickers. `Magnet` can be further subclassed to different types based on their functionality. A `Dipole` basically bends the particle in the horizontal or vertical direction. So the path of a particle passing through a dipole looks like a curve. But the `Dipole` itself may be manufactured in the shape of a curve (`SBend`) or rectangle (`RBend`). `SBend` usually is put in the circular portion of the accelerator, while `RBend` is put in the linear portion. `Quardrupole` is used to focus or de-focus a bunch of particles (beam) along the trajectory of the accelerator lattice. `Sextupole` and `Octupole` are high-order magnets used to obtain fine-grade control over the particle movement in multiple directions. Unlike other magnets, a `Correction` magnet is not installed permanently. It is used on an as needed basis when a small correction is required in certain locations of the accelerator. It can be plugged in and pulled out at any time.

4.3.3 Attribute, constraint, and relationship in SSC simulation

The classes `Attribute` and `Constraints` are derived from `Resource` class (Figure 4.6), which provides reference counting and change-event dispatching mechanism using the *observer* pattern [28]. `Resource` also provides type identification for its derived classes. `Resource` holds a reference counter for resource management. As an example, an accelerator uses multiple copies of bending magnets. Instead of duplicating its attributes `BendAttri` (a resource) in each copy, each instance of the bending magnet only holds a reference to the attributes. Only one copy is made the first time, and subsequent

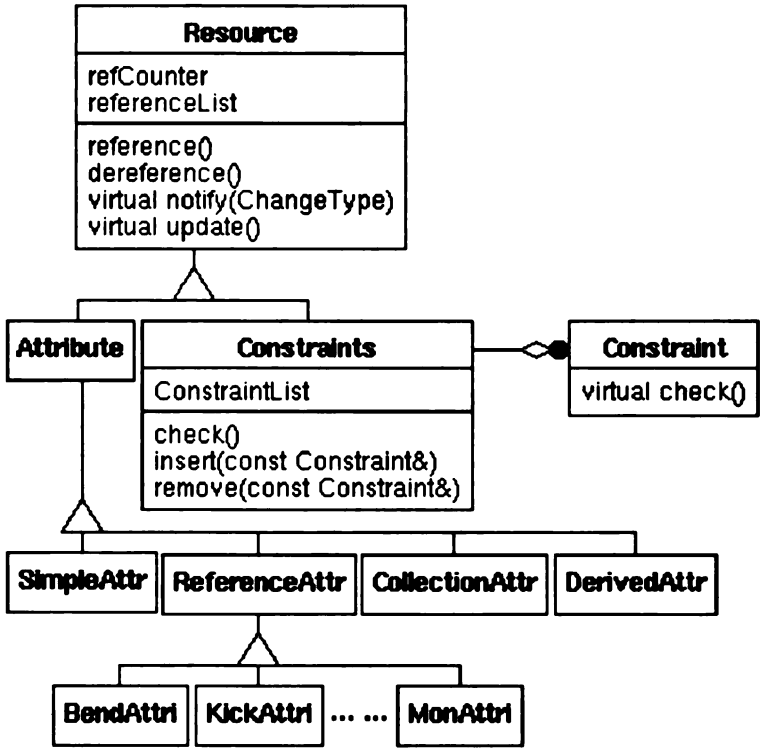


Figure 4.6: Attribute and Constraint

requests of creation will get the same copy and the requesting magnet will be added to the resource's reference list. A resource will be deleted automatically when its reference counter decreases to zero. `Constraints` is a collection of `Constraint`. `Constraints` has a member function `Check()` which basically goes through each `Constraint` object in the collection and calls their member function `Check()` one by one. Class `Constraints` is a slot for user defined `Constraint`. `Constraint` is an abstract class that encapsulates the representation of a constraint. It requires a virtual function `Check()` to be defined in the user derived classes.

`Constraint` serves two functions in our simulation:

1. Guard function: informs abnormal situation, raises exceptions to a constrained object. In such a case, `Constraint` has no responsibility to serve the constrained object.

2. Recover or rectify function: make things correct for the constrained object. In such a case, `Constraint` usually has to be a *friend* of the constrained object in order to operate on its internal representation.

`Constraint` is pluggable. A `Constraint` is enabled when a user plugs it into the `Constraints`. `Constraint` can be deactivated if user removes it from `Constraints`.

Virtual function `Check()` of a `Constraint` provides a run time binding between `Constraint` and its `Check` function. `Constraints` does not know each `Constraint`. But by using polymorphism, it is able to invoke its `Constraint` at run time without type checking.

The advantage of this constraint model is that the users can create their own constraint, design their own algorithm and insert it to the `Constraints`. The working protocol between `Constraints` and `Constraint` is independent from the definition that may vary from different users.

Currently, `Constraint` is implemented by a structure that contains a threshold value and a pointer to the callback function. Exceptions are handled by its handler which is defined in `Constraint`.

`Relationship` is an object that describes the physical and conceptual connection between objects and passes messages among them when an event happens. For example, in order to get the T-vector of a Detector, we need to calculate the so-called *influence* function [71] between each pair of adjusters and detectors in the lattice. The influence function is defined as follows:

$$InferenceFunction\,(a,d) \;=\; \frac{\cos\,(\mu/2-\phi_{da})}{2\sin\,(\mu/2)}\sqrt{\beta_a\beta_d}$$

Where, *a* and *d* stand for the position of adjuster and detector, respectively; μ stands for the tune for the entire lattice, a function of the phase advance of every Actor in the lattice, defined as a member variable in class Lattice; ϕ_{ad} is the summation of phase advance from *a* to *d*; and β_a and β_d are the β-tron oscillations at *a* and *d*. We define a relationship called *influence* in Table 4.1.

Participants:	adjuster: *a*, detector *d*.	
Relation:	floating number.	
Constraints:	1. β–oscillation at *a* or *d* changes its value.	1. identity and attribute.
		2. β-oscillation in front of them
	2. Phase advance between *a* and *d* changes.	1. β-oscillation at *a* or *d*
		2. Transformation matrix at *a* or *d*
	3. μ changes.	1. Phase advance at end point
Method:	$\frac{\cos\,(\mu/2-\phi_{da})}{2\sin\,(\mu/2)}\sqrt{\beta_a\beta_d}$	

Table 4.1: Relationship: *influence*

Constraints in relationship define a set of rules invoked when events happen on related data (called dependents). The granularity of these dependents is critical. The object that actually updates the relationship, called *executor*, should be carefully chosen. In the SSC lattice model, we just choose class Lattice to fulfill such a task for simplicity.

A modification to a basic attribute can affect many things such as composite attributes and relationships to other objects. Who should coordinate such an updating propagation? Let *s(a)* be a set of objects whose model (including its identity, attributes, constraints, and relationships) could be affected by changing *a*, the model of one particular object. Then the propagating executor *e* is the Actor with a minimum number of

descendants that cover *s(a)*. We called *e* a ***minimum cover*** of the propagation. So the callback function should point to the minimum cover of current modification.

In some circumstances, the timing of update is also important. If a relationship is not finalized (in the process of changing), propagation should be held until it reaches a stable stage. In our model, propagation can be queued and fired at the end of the data transaction between simulation and design database. This improves the efficiency of managing data consistencies in the database. Users have more control over when and where the propagation should start and the transaction should be committed to the design database.

4.4 Implementation Issues in SSC Simulation Using Actor Model

4.4.1 Design database access

Design database is used to store different configurations of the system for dynamic simulation. Database is abstracted in our framework (described in Chapter 3) as a database object. In this particular case we use ObjectStore, an object-oriented database management system as the design database, but generally speaking, our model encapsulates the underlying implementation of a particular database by providing a set of generic operations for database access.

In an object-oriented database, object access is achieved in two ways: (1) the database provides a set of *global* objects that are always accessible to the applications. These global objects are called database *roots*, through which all objects in the database can be reached; (2) the public interfaces of these global objects provide access to objects that are referred to by the global objects. Such an access is "navigational" [26]. In our model, only the top level Actor in the composite hierarchy (such as the SSC that stands for the

entire SSC complex) is made global (root). The rest of the SSC complex can be reached by navigation through the root. When a design database is *open*ed by an application and a transaction is in progress, the root is automatically mapped into the database. This is quite similar to the situation when a procedure is entered. The variables in the parameter list are mapped to its calling value from the caller of the procedure. The variable in our case is defined not in the application but in the model.

The transaction concept in database management allows:

1. global objects to be realized in the database;
2. an application session to proceed with making data changes, i.e., altering objects that it had reached, but without immediately affecting the data stored in the database.

Dynamic simulation usually creates a new configuration, with new data, as results. In a transaction, such a new configuration and result data are only stored at local disk. The user has the control of whether or when to commit these changes to the database. A database transaction can be viewed as a sequence of alterations to objects and configuration manipulation. All changes to database objects are made permanently; i.e., the changes are reflected by the new state of the database (if simulation results in a good design), or none of the changes are made (if bad changes are made to the design). In the first case, a transaction completes successfully by "committing" all changes to the database. In the second case, the transaction "aborts," and none of the changes take effect. The database is still in the same state as if the transactions were never started.

4.4.2 Dynamic simulation at the SSC

One of the most important issues in dynamic simulation is how to deal with system reconfiguration and integrate system behavior from its components to help synthesize the model. The composite hierarchy we built is subject to change in the process of dynamic simulation. Such changes include: modifying the composition structure of the system by

adding, deleting, or replacing its components (Actors); changing attributes and behaviors of the components; breaking the old relationship and establishing a new relationship among Actors in the system while its configuration changes. Objects (Actors) in the model may be related by relationships so that when one object is changed others must be amended to retain consistency.

Our model provides the simplicity to reconfigure a complex system. There are only a few database roots available to the simulation application. The entire system is well encapsulated as a functional abstraction. Information accessing is navigational, which not only is a structured way to unveil the abstraction but also provides a conceptual boundary for the design. Recursive query can be issued at any level to locate a particular Actor. So any Actors in the hierarchy can be reached from the root without digging down to the hierarchy.

Any configuration change to a composite object can be localized to its descendants down to the hierarchy. The change is not visible from above. Figure 4.4 shows *machwm* = *ml1* + *triinj* + *triext* + *triwm*, and *triwm* has a child *arcwm* and *ssewm*. Consider the case that *arcwm* is reconfigured as *arcwm'* by adding another *as1* to the beginning of the *arcwm*. A *particle* object is passed to the new *leb* to be transferred. From the level of *leb*, the new configuration of *arcwm'* will not be visible until it binds to its configuration in the simulation. The Actor *leb* will take over the *particle* and invoke virtual member function `transfer()`. Member function `transfer()` will in turn invoke member function `transfer()` of its children one by one. Such a process is propagated until it reaches a level where `transfer()` is bound to a particular operation (*arcwm'* may have different implementation of method `transfer()`). There are two important observations from the above:

1. In a dynamic simulation, the component is treated as an abstraction and the configuration is bound at run time. The advantage of abstraction is that its implementations are interchangeable. As long as it is an Actor, it can transfer a

particle no matter what kind of Actor (in terms of its composition) it is. Run time configuration binding provides a strong support for dynamic simulation. It supports simulation that runs on an abstraction, such as *leb* at a higher level, without knowing its configuration at the low-level.

2. Member function `transfer()` can be defined at a higher level as well as at a lower level while a configuration is bound later. For example, when the `transfer()` function of *arcwm* is called, the transfer operation can be realized at the composite object (*arcwm*) level without an implementation at a lower component level. This allows a functional implementation at a higher level while a low level configuration is still undecided.

Using our model, a lattice can be dynamically configured without stopping the simulation process. An Actor can be created at run time by using operator `new(db)`, and can be assembled interactively through a graphical user interface called *Observer* (Figure 4.4). Then it can be inserted into the proper place through navigation or local database query. The local and global adjustment will be automatically taken over by the higher-level Actors (for example, the `Lattice` object). It will conduct an up/down adjustment through the hierarchy and notify actors that may be affected by the reconfiguration.

Our model provides both static and dynamic behavior binding. Behavior depends on the algorithm in the methods and data (attributes) of the object (Actor). Static binding is data-driven that binds only the data at run time with a generic algorithm. Dynamic binding is type-driven that binds both data and algorithm at run time.

In our model, static binding is implemented by a transformation matrix as the state of an object. The transformation matrix is a composite attribute tied to our attribute model and monitored by the constraints. When a certain attribute changes, its corresponding constraint is checked, and the transformation matrix is recomputed. Such adjustments occur before the simulation. The actual particle-tracking simulation is implemented by transformation matrix multiplication [69]. Static binding provides greater run time

efficiency. The data-driven approach is also feasible to change the behavior of the Actor without modifying the operation code (method).

Dynamic binding uses polymorphism (a virtual function call), and the method is bound at execution time based on run time type. `Transfer()` is defined as a virtual function (shown below) that describes the transfer behavior of the magnet. New classes are derived from class Actor to represent the difference between magnets. Each derived class

```
class Actor
{
   ... ...
   Particle& virtual Transfer(Particle&){};//a virtual function doing nothing
   ... ... ...
}

class Detector : public Actor
{
   Particle& virtual Transfer(Particle&)
   {
     #include "detector_behavior_description"
   }
}
```

overrides the virtual function `Transfer()`. The algorithm can be either physically contained in the method or included from a description file elsewhere. The former has to be in C++ programming language; the latter, which is more flexible, can be a higher-level behavior description language or even a lattice file created by some simulation package. But the latter needs a translator to read the file and turn it into a C++ program.

The trade-off of dynamic binding is the run-time efficiency. It is usually slower than static binding. But it gives more flexibility to model dynamic behaviors.

Finally, we give two examples to demonstrate a single particle tracking simulation using our Actor model. We assume that the lattice database is already stored in ObjectStore with name "ssc_lattice" and root `ssc`. The program (shown in Figure 4.7) illustrates the tracking procedure. It opens the database, enters a transaction, and locates the root `ssc` in the database. It first locates a lattice `leb` in the `ssc` using a query. Then it creates a particle

```
database *sscLatticeDB = database::open("ssc_lattice");
do_transaction(){
    persistent<sscLatticeDB> Actor* ssc; //locate database root ssc
    Lattice *leb = ssc->children[strcmp(Name(), "leb") == 0]; //query pick leb
    //create a particle:
    Particle *myParticle = new Particle(POSITION, DEVIATION, OFFSET);
    foreach (Actor *actor, zeroLevelActor) //iterating over collection array
      actor->Transfer(myParticle); // polymorphism through dynamic binding
    myParticle->PrintYourself(); //print the tracking result
} //end of transaction
```

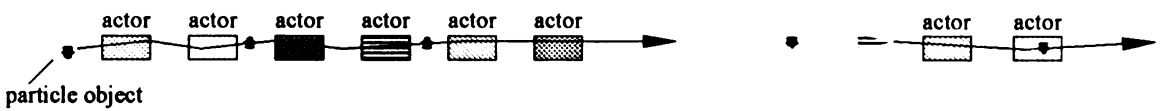



Figure 4.7: Single particle-tracking simulation

object (myParticle) with initial position, deviation, and offset. The foreach loop iterates sequentially through all elements (zero level Actors) in the leb and sends the message "Transfer" one after another with the particle object just created. The last statement prints the new position, deviation, and offset of the particle after one-turn of tracking.

Figure 4.7 also shows a picture of a particle object passing through elements in the lattice with different attributes (shown with different filling pattern) as a tracking simulation. The simulation results will identify where the largest particle deviation occurs, and where the particle exits the magnet if it is lost in the middle of the tracking simulation. These results are useful as a feedback to adjust the design. The above program is so simple that we only need to deal with the lattice *leb*. In the program, an Actor can be an element (in this case) or a beamline, but the internal structure is irrelevant or transparent to the simulation. Simulation sends a "Transfer" message to each Actor in the lattice one by one to track the particle through the lattice. The tracking simulation process will not change when some beamlines or magnets in the *leb* are reconfigured, since such a change is encapsulated and bound dynamically at run time. This is the strength of the dynamic simulation.

Now we try to change the configuration of lattice *leb* and redo the tracking simulation on a part (beamline *triinj*) of the leb (Figure 4.8). First, a query is issued to *leb*'s zero-level descendants to find those whose name are "hb" and attribute type are "sbend." The colons on both sides of the boolean equation mean to find all the Actors that meet the query condition. The query result is a collection of Actors. Then their magnetic strength is increased by 0.0011. Another query is issued to find *leb*'s zero-level descendants whose name is "fpm" and attribute type is "drift". This time only the first one that meets the condition (without colons) is chosen. The *drift* is then removed from lattice *leb* to create a new *leb* configuration. When those *sbend*'s strength are changed, their constraint on attribute *strength* is checked. A strength change will affect other attributes such as β

```
os_List<Actor*> *hb = leb->Descendents(0)[:strcmp(Name(), "hb") &&
strcmp(attributes->Type(), "sbend"):]; //query on a sub collection of 0-level actors
foreach (Actor *actor, hb) // change strength of a group of actors
    actor->attributes->strength(actor->attributes->strength() + 0.0011);
Actor *fpm = leb->Descendents(0)[strcmp(Name(), "fpm") &&
 strcmp(attributes->Type(), "drift")];
//change configuration, which will invoke constraint checking, replace-
ment should have the same length.
fpm->parent->children.remove(fpm); //change configuration
```

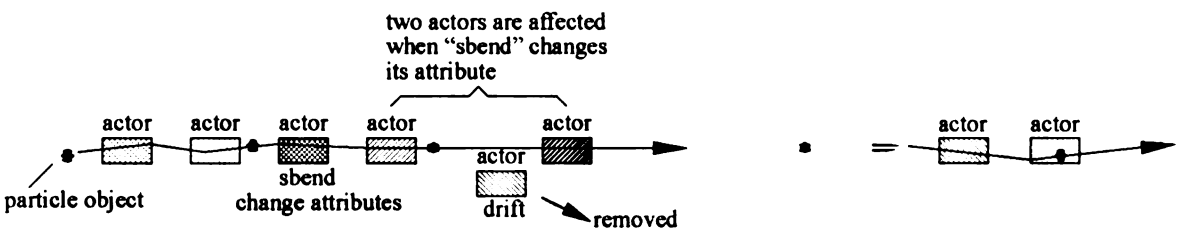


Figure 4.8: The impact of configuration change in dynamic simulation

function, phase advance, and transformation matrices. Local propagation starts by invoking callback functions registered by the constraints. When drift *fpm* is removed from lattice *leb*, it will also be removed from its up-level beamline (parent). Its neighbors on both sides will become neighbors by adjusting their *relationship*. These adjustments are automatically accomplished by implicitly invoking appropriate methods of the relationship. Some changes may have global effects. When *sbend* changes it strength, it also affects the β function of all magnets lined after it in the lattice. So the β function propagation cannot be handled locally by the next up-level beamline. It has to be coordinated by the object *leb*. We call `computeAll` on *leb* to start a global adjustment.

```
leb->ComputeAll(); // global adjustment
```

Now the new configuration of *leb* is obtained. This new configuration is not yet committed to the database. A new particle is created and tracked through beamline *triinj* to

```
// redo the particle tracking
 myParticle = new Particle(POSITION, DEVIATION, OFFSET);
//tracking particle through triinj, a beamline of leb
Actor *triinj = leb->children[strcmp(Name(), "triinj")];
 foreach (Actor *actor, triinj)          //iterating over collection array
    {    actor->Transfer(myParticle);    // polymorphism through dynamic binding
         Draw(actor->Beta()); }          //draw β function
 myParticle->PrintYourself();            //print the tracking result
```

see the change effect on the beam (as shown with code above). The β functions drawn at each magnet the particle is passing through show the change occured after the first *sbend* magnet on the beam trajectory. The simulation result can then be committed to the database at the end of transaction.

4.5 Summary

This chapter described an object-oriented model for dynamic simulation. The Actor model presented here provides an object-oriented approach for dynamic simulation. By

using this model, the user is able to directly modify the system configuration during the simulation process, assigning attributes to objects, binding constraints, and altering relationships. Become attribute is made a separate object, attribute binding becomes more flexible. The constraint implemented by using exception handling gives more control to the user in the dynamic simulation. The concept of relationship allows more complex connection between objects. Two ways of behavior binding in dynamic simulation are discussed: data-driven is quick but less flexible; type-driven is more dynamic and expandable. An Object-oriented database, ObjectStore, is used in our implementation. This model is currently used to model SSC lattice structure and to dynamically simulate behavior of the accelerator. The next chapter integrates configuration management and version control into our model to provide more capabilities to the simulation.

CHAPTER 5

CONFIGURATION MANAGEMENT AND VERSION CONTROL FOR SIMULATION

5.1 Introduction

This chapter describes issues and approaches for configuration management and version control in simulation. In simulation processes, the model of the target system usually changes in order to exploit optimal configurations or reflect real world alternatives. For example, in high-energy physics, simulation of a particle accelerator usually requires fine-tuning of the magnetic trajectory to provide an extremely well focused and well-positioned beam. The direct effect of such a tuning process causes complicated changes in one or more of the component objects (the magnet) comprising a system (the accelerator, a composite object) in simulation. The tuning process results in versioning of component objects and a different system configuration. These changes affect all aspects of the system, and must respect many constraints and dependencies between the components to avoid damaging the integrity of the system. *Versions* are different implementations of the same

interface. *Configuration* is a generic description of components from which the system is composed and the actual instantiation (implementation) of the system generated from the description. The problem of managing a simulation model is compounded by the fact that composite objects can have multiple alternative configurations, if versions of components are taken into consideration. The complexity of the modern simulation models and the scope and frequency of the changes they typically undergo in simulation makes effective control of configuration and version imperative.

During most simulation life-cycles, a growing number of versions of a single component have to be dealt with, and a complete system has to be reassembled for simulation in different ways using these components. The basic goals of version control in the simulation are: (1) to effectively record, retrieve, and keep track of different versions or revisions of the same component and their relationships; and (2) to enforce restrictions on the evolution of a component so that such an evolution is observable and controllable. The basic goals of configuration management in the simulation are to facilitate the description and instantiation of a system from versioned components (*configuration binding*) and to ensure consistency and integrity of the resulting system.

The complexities of CM and VC in the simulation environment can be summarized as follows:

1. Multiple abstraction levels due to object hierarchical decomposition and de-coupling of interface and multiple implementations of a simulation object.
2. Multiple representation, since the simulation iteration and step-wise refinement cause versioning of object, its input parameter, and the simulation result.
3. Dynamic binding between object and its versioned components, interface and its particular implementation.

This chapter is organized as follows. Section 5.2 introduces the concept of *modeling object* and object relationship in CMVC. It identifies the issues of CMVC and emphasizes the importance of CMVC in object-oriented simulation. Section 5.3 proposes a composite

object model of CMVC for simulation. It shows how CMVC is done using modeling objects, generic reference, and workspace. It discusses several issues in change notification and propagation. Section 5.4 summarizes this chapter.

5.2 CM and VC in Object-Oriented Simulation

The iterative and exploratory nature of the simulation process leads to two aspects of simulation objects that must be dealt with. First, they are usually complex; that is, they are assemblies of components that themselves may be constructed hierarchically from component objects. Second, there can be several alternative descriptions, called *versions*, of a simulation object, its input and output parameters, as well as simulation results. Most of the current simulation systems suffer from the fact that they normally hold only one *valid* view of the model. In reality, however, a user or a group of users might be interested in pursuing a particular simulation design along several possible paths or directions simultaneously, and may decide later on one of the alternatives, or a merger of alternatives, as the final design. Therefore at any given time, several valid representations of the model may exist simultaneously. Management of simulation data in the presence of versioning and configuration changes thus becomes a challenging task.

In SSC, lattice simulation usually involves a large amount of data. The simulation will also generate new configurations and result measurement data. These new configurations and data need to be matched and recorded as a group for later analysis. The traditional file system management tools are usually error-prone with a large number of versions, and are less organized and space-inefficient. They need to be monitored manually sometimes to ensure system consistency. CMVC provides a systematic way to group configuration with related data as versions and to record or retrieve them automatically, therefore greatly enhancing the productivity of simulation. CMVC is also critical for filtering out invalid configurations and for guaranteeing system consistencies in new

configurations.

5.2.1 An object-oriented model for CMVC

The object model aims to be well-suited to support the representation and manipulation of complex objects in simulation, offering a natural conceptual model to the designer and facilitating the efficient structural clustering of information which tends to be obstructed by normalization in the relational model [5]. An *object* is defined as an instance of a given *class*. A class defines a set of *attributes* for each of its instances, and a set of *operations* on these instances. The set of attributes and operations is called the *scheme* of the class. Each attribute of an object may contain either a *value* or a *reference* to another object. The set of classes is organized into a class inheritance hierarchy. A sub-class inherits the scheme of its super-class.

Modeling objects in the simulation are aggregates of design primitives treated as a coherent unit in simulation version control. Modeling objects feature a set of versions with a single distinguished *current version*. A modeling object can reference other modeling objects to form a hierarchical aggregation and it is called *composite* modeling object. Composite aggregation of a modeling object identifies its immediate component objects. The relationship between a component object and the composite object that "contains" or "uses" it has been called *is-a-part-of*. The composite object mechanism has the advantage of being hierarchical, that is, the internal binding of its components (which in-turn can be composites) can be defined within an environment associated with that particular composite object. Therefore it enables a layer of data abstraction and information hiding, one of the most useful aspects of an object-oriented model. *Primitive objects* reside at the leaves of a component hierarchy, whereas a composite modeling object is the root of a subgraph. We can define a composite object as a directed acyclic graph (DAG) where the nodes with an in-degree of zero are the types of the highest level composite objects; the nodes with an out-degree of zero are the types of primitive objects. The directed edges in

the graph represent the *is-a-part-of* relationship. We call this graph the *composite graph* (Figure 5.1). Multiple component objects may use a given component simultaneously, yielding a DAG structure.

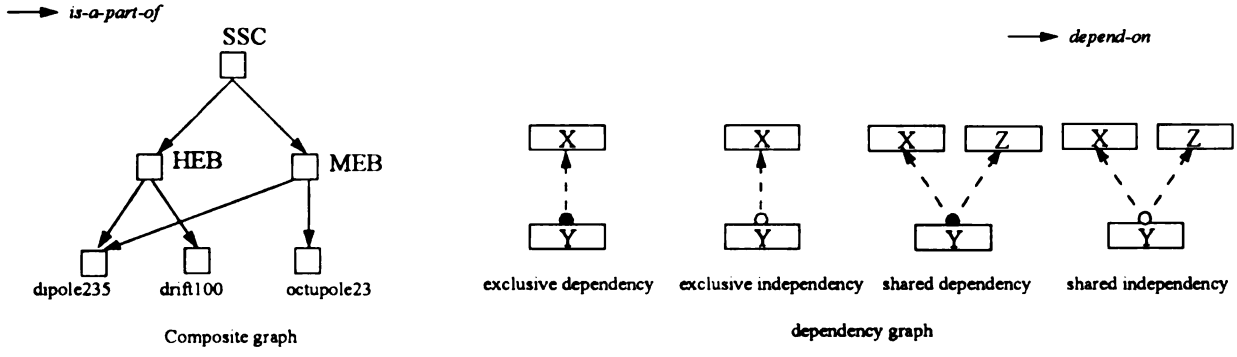


Figure 5.1: Composite graph and dependency

For example, let us consider an accelerator in the SSC. An accelerator complex usually contains several types of accelerators (called *machines*) with different energy levels. These machines can be either a circular accelerator or a linear accelerator. Machines are connected with transfer lines that transfer beams from a low energy machine to a high energy machine. A machine contains different types of magnets. These magnets bend, focus, de-focus, or accelerate the particle beam along its trajectory. A single magnet is a primitive object and is a-part-of an accelerator, while an accelerator is a composite object at the root of the composition graph. An accelerator is also a modeling object since it is treated as a single unit when versioned. Composition by itself is not completely adequate for simulation modeling.

Dependencies among a composite object and its components must also be explicitly represented [39]. "*A depends on B*" means that component *A* is hierarchically subordinated in some way (physical or logical) with respect to component *B*. The relationship *depend-*

on implicates that when *B* changes, *A* should get notified and change accordingly. This relation can also be represented by a DAG, called *dependency graph* (Figure 5.1), in which one node depends on another node if there is a directed path joining them. Dependencies can be classified into four categories [54]:

1. Exclusive dependency: if Y is a component of X, then the existence of Y depends on X.

2. Exclusive independency: Y can still exist even if Y is a component of X and X no longer exists.

3. Shared dependency: Y is a component of X and Y could be a component of Z at the same time. But the existence of Y depends on both X and Z.

4. Shared independency: Y is a shared component of X and Z but the existence of Y does not depend on either X or Z.

Here X, Y and Z are all objects. Dependencies are relationships between objects, not versions. Let us consider the previous example. An accelerator complex depends on a couple of accelerator machines, and accelerator machines depend on their magnet components. The bending magnet is exclusively used in certain types of circular accelerators to force the particle turning its direction, so it causes exclusive dependency. The focusing magnet is widely used in all accelerators and causes exclusive independency. The control and power supply units are usually shared among magnets, and they form a shared dependency.

Dependencies are used extensively in change notification and propagation. While changing a single bending magnet is simple, changing a widely used focusing magnet may cause unforeseen problems throughout the accelerator that uses it. By taking advantage from the underlying structure of a composite hierarchy, dependency constraints can be used along with the dependency graph to help make change notification and propagation conceivable and controllable. We will show later the important role that dependency graph plays in the configuration management for simulation.

5.2.2 Version control in object-oriented simulation

Versions are distinct snapshots of a modeling object in different states. Version control is used to manage configuration changes in object-oriented simulation. These changes are usually caused by iterative simulation processes in a cooperative multi-user environment. Changes usually involve simulation-input data, system parameters, the simulation data model, and simulation results.

Simulation is itself a substitute for real experimentation, and usually is conducted in the absence of the complete set of real data. To state the validity of a result, one must establish an accuracy measurement related to an understood criterion. This is important because simulation itself is an approximate process, frequently employing hypothetical or statistically varying data and system parameters. Establishing the final accuracy (hence, validity of the model) involves an iterative process whereby successive runs and adjustments converge and satisfy some accepted criterion. As a simulation iterates, the version control system preserves the simulation data as they change at particular points of time so the date can be retrieved later. Each simulation run will generate a version. Previous versions are retained as new versions are created. This is referred to as *linear* versioning and represents a series of sequential successive changes caused by iterative simulation runs.

Versioning systems are also used to support cooperative multi-phase multi-user simulations. First, a simulation process can usually be divided into several phases. For example, a circular accelerator can logically break into several sections called beamlines. Simultaneous simulations can be run first on individual beamlines to obtain local optimization, and then be run at the whole scale level. On the whole scale level, simulations can also be conducted in several phases. Each phase has a specific target energy or beam luminosity that the accelerator should achieve. Version control should be able to track and retrieve versions of simulation runs in each phase (input scenario, parameter settings, model used, and result generated) so that the results can be used to advance the simulation for the next run. Second, it is very common for people to work on the same simulation in

parallel. Simulation models can also be multifaceted, so the base model can be exploited with different interesting behaviors to meet new simulation objectives. The extended simulation model can be run simultaneously by people with different requirement targets. In this scenario, multiple states of simulation data and their associated operations can exist in any point of time. The versioning process branches from the single preceding version into multiple versions to keep the footprint of individual simulation activities. At some point in time, these multiple versions are merged to unite the work of all contributing parties into a new and final version of the original. Version control should support such cooperative work.

Version control in the simulation should offer mechanisms to effectively record, retrieve, and keep track of different versions of the same simulation data, as well as enforce restrictions on the evolution of the simulation system so that such an evolution is consistent and controllable.

5.2.3 Configuration management in object-oriented simulation

Configuration is a binding between a specific version of a composite object and a version of each of its components; it is version control extended to a composite object, not something intrinsically different. Configuration management in the simulation comprises three distinct tasks: defining acceptable configurations, monitoring configurations for departures from the defined limits (constraints) during the simulation, and correcting configurations for which such a departure has occurred.

First, configuration management in the simulation should support constraints to define acceptable configurations. On the one hand, when a simulation model (a modeling object) is assembled from a library of more primitive component objects, certain criteria have to be met before the selected version of a component can be "plugged" into the composite object. Such criteria select a component's external features required by the composite object, as well as impose restrictions that the components have to meet to avoid

breaking the integrity of the composite object. On the other hand, when an existing model undergoes changes between simulation runs, interconnection among component objects needs to be constrained so that such changes will not break the validity of the model. Most importantly, the criteria for selecting and the mechanism for managing input data for the next simulation run based on the previous one is critical to the effectiveness of the simulations.

Second, configuration management in the simulation should provide a means to monitor and effectively propagate configuration changes so that such changes are observable and controllable. Configuration of a composite object can be changed in many ways. Adding, deleting, or replacing components causes configuration changes. Changing attributes of a component, including user-input data and system parameters, also causes configuration changes. The component initiating the changes should be updated, and notifications should be sent to objects along the dependency hierarchy. The configuration management system must identify the dynamic scope of the changes, decide the scale of the change propagation, and propagate the changes in a systematic way.

Third, configuration management in a simulation should provide mechanisms to correct configuration changes when a violation has been identified. Usually certain restrictions are needed to limit changes to some degree due to available resources or design specifications. When changes happen, especially when changes proceed during the propagation, the integrity of the model can be broken in the chain-effect of propagation. After the violation has been identified in one branch of the propagation, the configuration management system has to decide whether to abort the entire propagation, repair the branch or component causing the violation, or correct the problem by adjusting another part of the system to accommodate the changes.

5.3 Proposed Approach for CM and VC in the Simulation

This section presents a composite object model especially suited for CMVC in the simulation. The types of versions in the model are shown and their creations are explained. Based on the object model, the CMVC schemes are developed using version graphs and generic objects. The concepts of workspace and context are introduced to facilitate the CMVC. Finally, approaches of managing change notification and propagation in CM are proposed.

5.3.1 Composite object model

Our approach is centered at the composite object model that provides a seamless integration of configuration management into the underlying versioned system. A configuration of a composite object is specified by binding each of its versioned components with a particular version. Figure 5.2 describes the model using a two-level component binding example. The composite object at the top is defined when each of its versioned components has a binding. Dotted lines are used for binding since each binding represents just one possible choice. A version of a composite object is generated when it adopts a new configuration. *Version1* and *version2* are different versions since they have different component configurations. But the binding of *versionX* at a component level (we call it internal binding) is encapsulated from the binding of *version1* as a composite object. Therefore, a configuration of a composite object only needs to know which versions of its components need to be bound, not how the versions of its components are generated. This implies the uniqueness of our composite object model.

First, internal bindings of versioned components can be encapsulated within the composite object to provide an extra layer of abstraction. Such an encapsulation will limit the change propagation to the minimum in a simulation because versions are usually

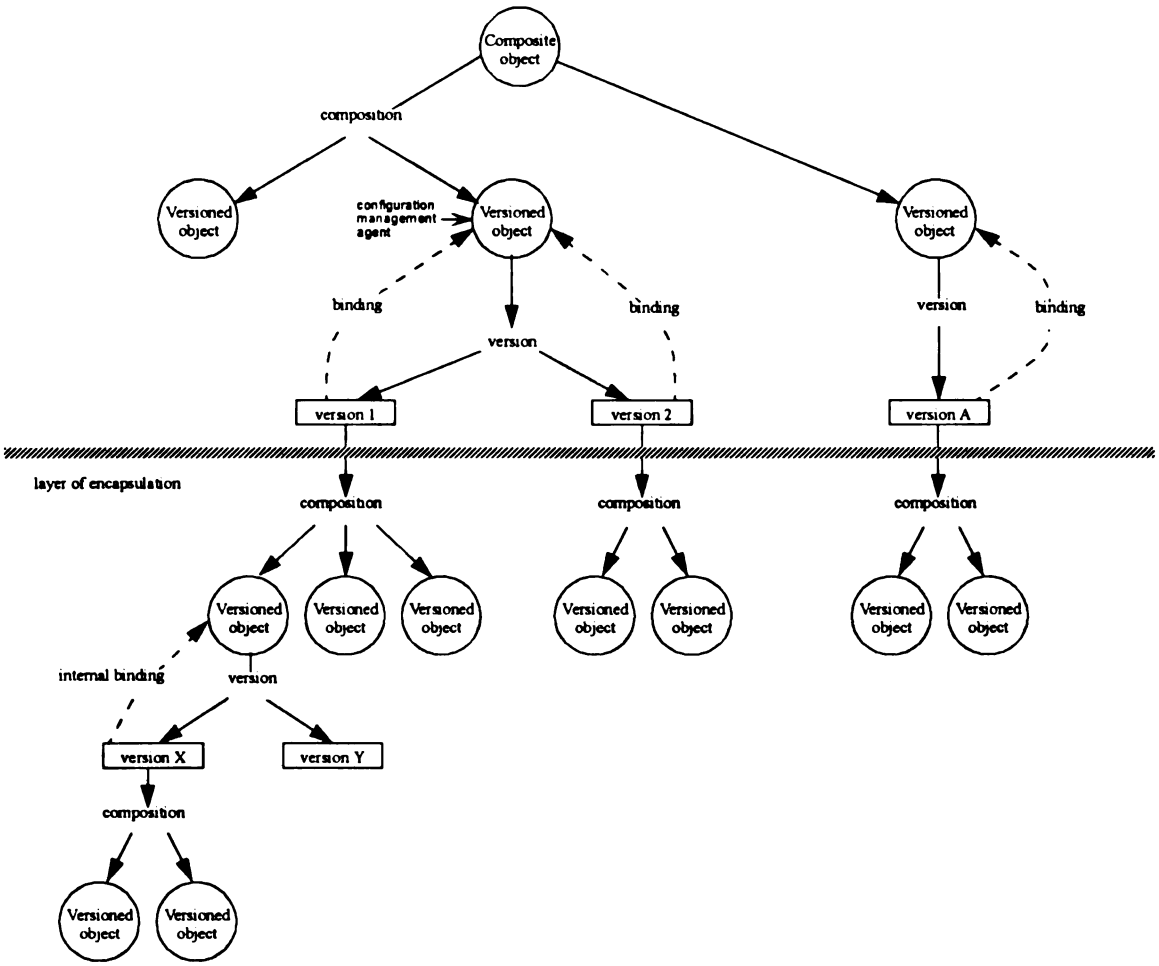


Figure 5.2: Composite object model

specified without exposing the configuration of its components. A simulation commonly uses many components designed by some one else, and such an encapsulation will promote component-level reuse. Second, a composite object is used as a special purpose configuration management agent whose task is to monitor and repair a given set of components when its configuration changes. Each composite object implements a set of operations specifically tailored to the task required for managing the components which are part of the composite, allowing them to be reasonably intelligent and effective in managing changes. Such a built-in CM functionality is especially important when it comes with dynamic simulation, since at run time the immediate parent composite object has the most knowledge of how to choose and reject a binding of its child component object, rather than the simulation environment. Third, the composite object model distributes the complexity of CM into individual components in composite hierarchy; therefore, changes become more modular and easier to manage in a simulation.

In our composite object model using OMT notation (Figure 5.3), class `Actor` [90] represents a composite object. It contains a collection of components called `Children`. Each component has its composing object called `Parent`. A set of methods is provided for configuration management and version control, for example, check in, check out, change notification and update, dependency handling, etc. These methods can be overridden in derived classes for particular needs. A good example is the calculation of particle transformation. The coordinates of a particle entering and leaving an accelerating magnet are determined by a 3 by 3 transformation matrix [68]. But different types of magnets have different ways to calculate each element of its transformation matrix. Therefore a virtual function `getTransformationMatrix()` is defined in the base class `Actor` and overridden in derived magnet classes. An important class derived from `Actor` is the `ModelingObject`, a composite object treated as a coherent unit in terms of CM and VC. Although an individual `Actor` instance can be versioned in our model, it is usually versioned in the context of a logically more self-contained unit in the simulation,

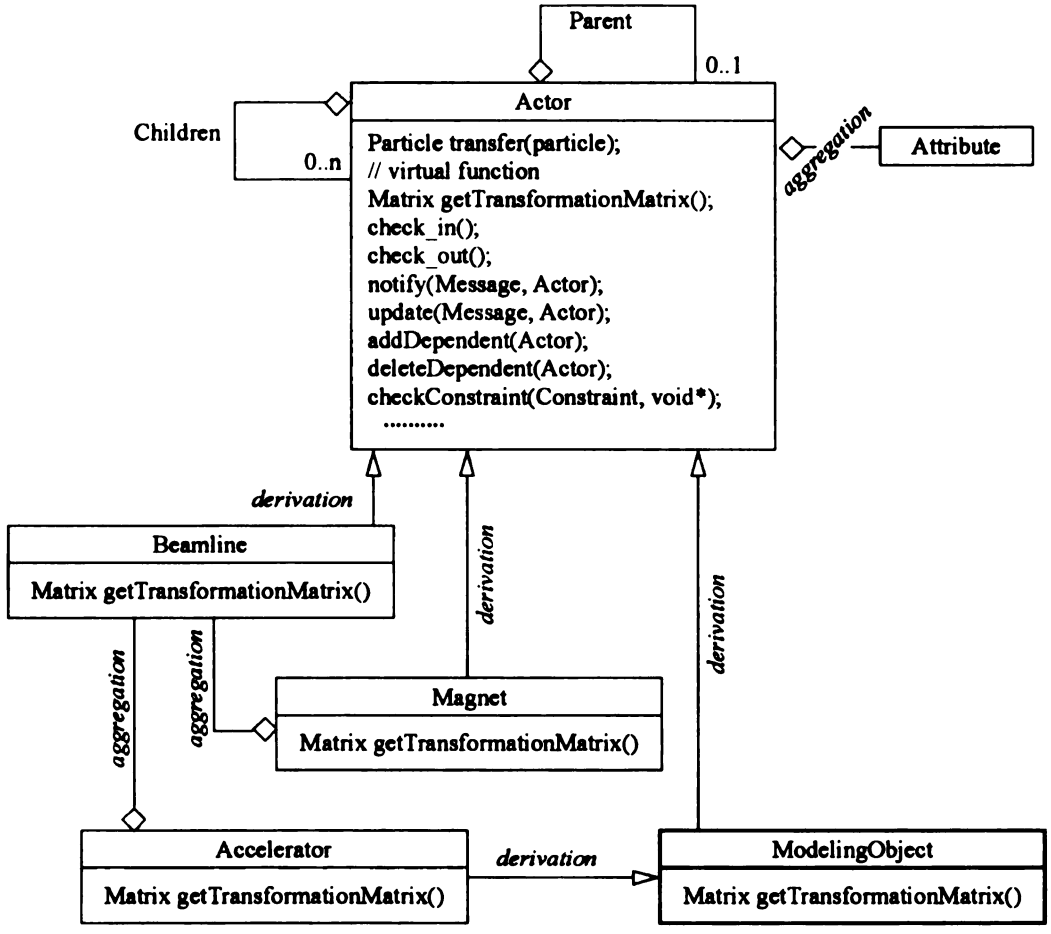
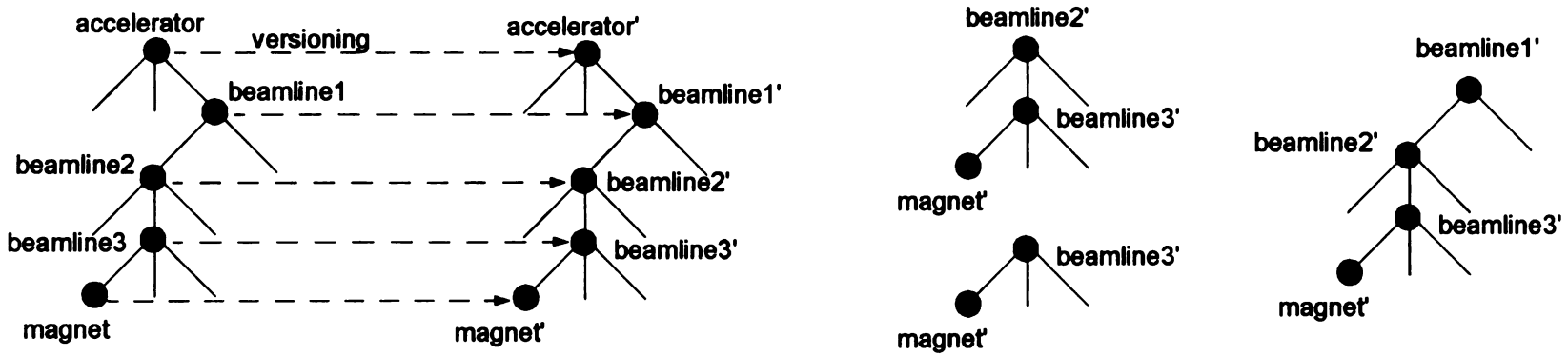


Figure 5.3: Actor model extended for CMVC

that is the `ModelingObject`. Applying version control on an individual `Actor` object is not particularly useful without a composition context. An `Actor` object is usually versioned in the context of a `ModelingObject` in which it resides. All magnet classes are derived from class `Actor`, and accelerator from class `ModelingObject`. As an example, consider Figure 5.4. An accelerator is composed of beamline1, which in turn is composed of beamline2, etc. (Figure 5.4a). When the design parameters of a magnet are changed in an accelerator, one common approach is creating a new version of the



a. Composite hierarchy of an accelerator.

b. New version of the accelerator caused by changes in magnet.

c. New versions of composite objects which *contain* the changed magnet

Figure 5.4: Modeling object is a coherent version unit

accelerator (Figure 5.4b) as well as a new version of each composite object (such as beamline1, 2, 3) along the composite hierarchy (Figure 5.4c). In our approach, only one new version of the accelerator is created (Figure 5.4b). The new version of the magnet, magnet', is created in the context of a new accelerator, a `ModelingObject`. The new version is recorded and retrieved as a unit of an accelerator rather than an individual magnet or beamline. Identifying a modeling object is important since it will greatly reduce the number of objects needed to be tracked by the version control system, therefore improving efficiency and performance in the simulation.

During the versioning process, some attributes of objects can be changed while others cannot. Attributes that will not be changed are called version invariant. Invariant attributes act as constraints that define the scope or pattern of the model evolution in version control. It is important to classify attributes in order to identify which attributes are version variants and which are version invariant. This classification serves as a base to control the versioning scope. It also provides a mechanism to restrict the change propagation in configuration management.

Our classification of attributes is similar to [2], which is based on the abstract view of composite object. There are two types of attributes: External and Internal. *External attributes* are non-structural attributes that are visible to the external world. They include interfaces through which an object interacts with the external world, as well as the specification (including constraints) of modeling objects. In our model, external attributes are version invariant. They are used as constraints to dictate the equivalence of different versions of the same modeling object. Different versions of the same modeling object must have the same external attributes. The versioning process of modeling objects should always respect the integrity of its external attributes during the entire process of evolution. Objects with different external attributes are different modeling objects. *Internal attributes* of a composite modeling object include description attributes (such as weight, height), and structural attributes that describe the bindings with its components as well as their

interrelationships. Internal attributes are version variant attributes, which spawn versions for the same modeling object.

Again, let us consider the accelerator example. For each type of magnet used as a primitive object, its public programming interfaces, i.e., the signature of all public member functions if implemented in C++, are external attributes. We do not want to change those interfaces that could cause ripple effects through the program. The descriptions of behaviors generated by calling these interfaces under specified circumstances are also external attributes. A focusing magnet should focus the beam no matter what kind of wiring pattern it uses. A de-focusing magnet should not be a result of versioning a focusing magnet. External attributes are version invariant across the entire versioning process. By restricting the versioning scope, we can understand and define the versioning process more clearly and make the version control more manageable. For an accelerator, on the one hand, the distribution of magnetic field in each of its component magnets is usually controllable. The magnet structure is also changeable to obtain optimal acceleration patterns. These are internal attributes and different alternatives can be exploited as simulation proceeds in iterations. On the other hand, injecting and transferring energy of an accelerator are external attributes which are version invariant.

5.3.2 Version history, generic object and context

Versions of a modeling object are created in two ways, either explicitly through version derivation, called *mutation,* or implicitly as a result of changes to the object or the propagation of changes made to other objects, called *propagation*. Changes to a pre-specified "significant" property trigger a mutation. The current version is frozen, and a new version is created before the change is made. In the second case, the creation of a new version of a component triggers the creation of a new version of its containing object. For example, when changing the wiring of a magnet to enhance its magnetic field strength, a new version of the magnet design is created through mutation. But when this new version

of the magnet is used to replace its original in a beamline which uses it, a new version of the beamline is generated because it triggers a broader change in magnetic field distribution in the entire beamline, which may force changes to settings of other parameters. Version history explicitly records the ancestor/descendent relationship among versions through distinguished *is-derived-from* relationships in version graph (Figure 6).

In our model, each object belongs to one of three pre-defined version types: *Generic*, *versioned*, and *unversionable*. The common characteristics that are used to relate all versions of a modeling object are its invariant attributes. A *generic* object is an object where the external attributes are defined, but the internal attributes are not bound. Since all versions of the same object have the same invariant attributes, they can be characterized by its invariant external attributes through the generic object. Each versioned object has a single associated generic object and zero or more associated versioned objects.

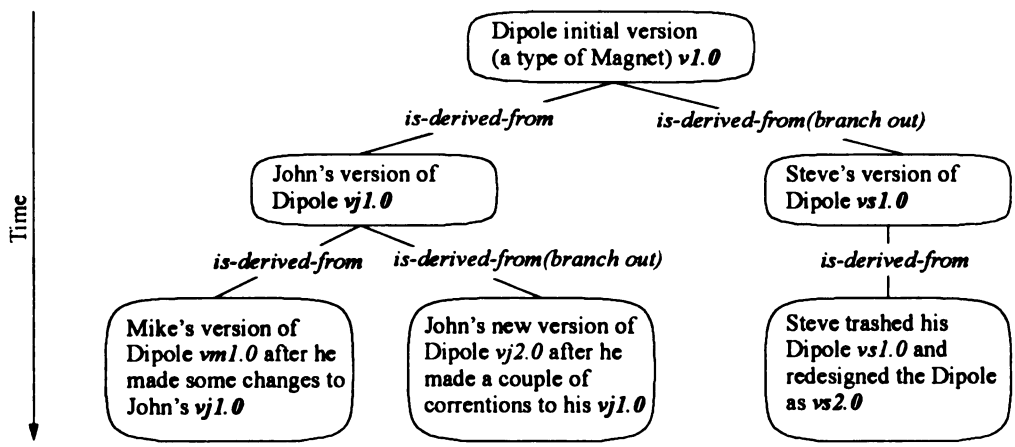


Figure 5.5: Version graph

Let us again use Figure 5.5 as an example. Dipole is a type of magnet used to focus or de-focus the beam. The initial design of the dipole has to meet certain minimum requirements; for example, the length of the dipole has to be exactly 2 meters to fit the

installation slot. Those minimum requirements will form the invariant attributes of our generic object: dipole 2 meters long. John comes in, and he tries to increase the beam luminosity by enlarging the dipole's aperture. A new version *vj1.0* is derived from the initial version. After John puts his new dipole into simulation, he finds out that his new dipole may run the risk of losing beam after he changed the aperture. So he corrects the wiring which will generate a stronger magnetic field to keep the beam inside the trajectory. The new version is called *vj2.0*. At the same time that John is working, Steve is working on the power supply of the dipole. He notices that the dipole just consumes too much electricity, so he starts to use superconducting material and designs a new version called *vs1.0*. Tom is designing a beamline that uses a couple pieces of dipole each 2 meters in length. But it is just too early to put it in simulation. So he uses generic objects (2-meter long dipoles) wherever the dipole is required in his beamline. At this moment, Tom thinks that any versions of dipole (*vj1.0, vj2.0, vs1.0*) are equivalent as long as it meets his requirements. With this purpose, a generic object is appropriate to use here. When he finishes the beamline design later, he has to specify a particular version from those equivalent sets of versions (called version set) before the simulation starts. Versions of a generic object all have the same scheme, so they differ only in the values for their variant attributes. These different attributes reflect the different *choices* that caused the version to be created. Version instances are organized into *version sets* associated with a single generic instance. Each version in the version set is *a-kind-of* its generic object sharing the same external attributes but differs in some different internal attributes. Values for versioned attributes are stored with their versioned object. Changes to any of these attributes cause a new version to be created (Figure 5.6).

Any references to a versioned object can be either a *specific reference* to a particular version of the referenced object, or a *generic reference* to the generic object, but not to any particular version of it. The usefulness of generic references is clear. Suppose a simulation uses a particular modeling object. As long as the external attributes of the object remain the

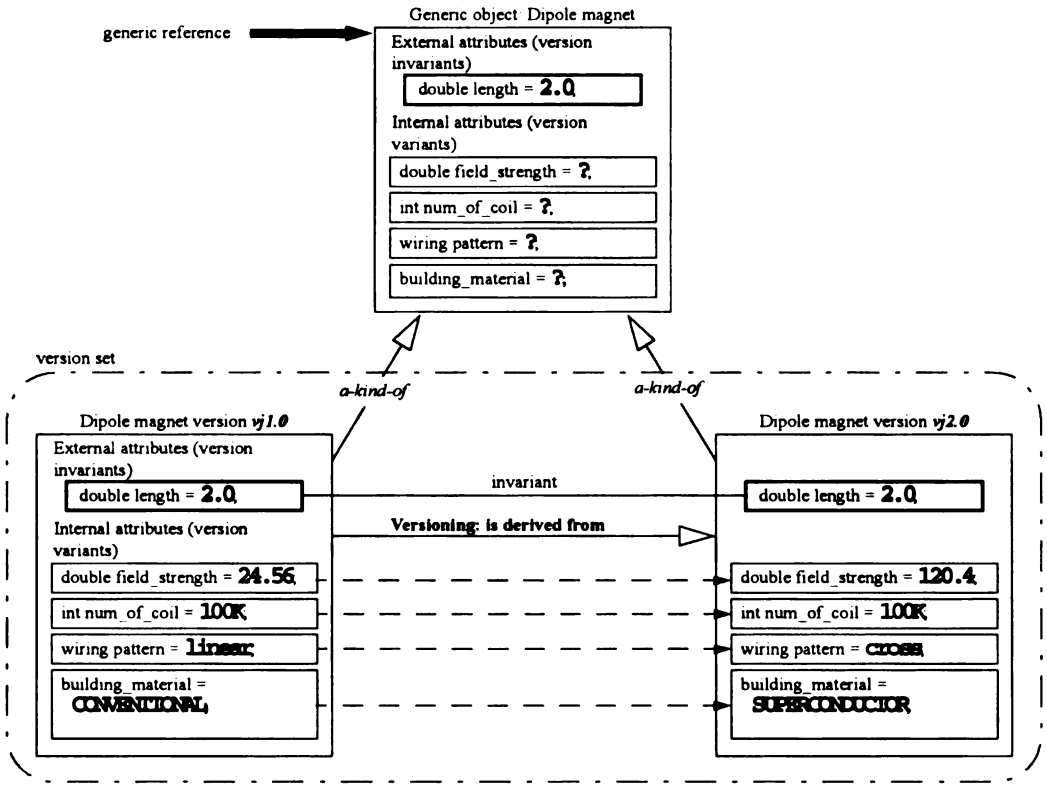


Figure 5.6: Versions in the version set is *a-kind-of* its generic object

same, the simulation can take advantage of improvements made to the object by keeping a generic reference to the object rather than a specific reference to a particular version (static binding). The resolution (binding) of a generic reference can be delayed to the latest version of the object when the simulation is actually run . Generic references also make dynamic configuration possible: coerced to refer to specific versions of objects according to the specified criteria at run time. A generic reference binds a generic component in composition space with a specific version in the version space (Figure 5.7). Such a binding is through version graph using pre-defined selection criteria called *context*. The most commonly used context is the *default* context. The default context will simply select the *latest* version in the version graph based on a time-stamp to bind to the generic reference.

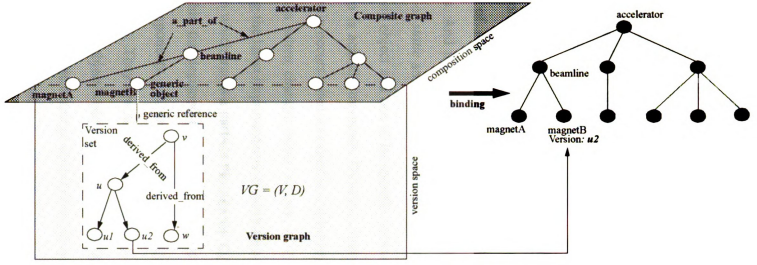


Figure 5.7: Generic object and its binding

The user-defined contexts usually select the version based on the values of certain attributes of the modeling object.

Our model defines a version graph of a modeling object as a directed acyclic graph (VG = (V, D) in Figure 5.7). The nodes V in the graph are the versions that belong to a generic object. The edge D of this graph represents the successor/predecessor version relationship called *derived-from*. If $(v, w) \in D$ and $w, v \in V$, then we say w is derived from v. If $(v, u) \in D$ and $u \in V$, we say that u is a branch of v, and w, u are *alternatives*. All versions in a version graph constitute a version set. Versions in a version set are considered equivalent in terms of their external features, as they have identical invariant attributes specified in their corresponding generic object.

A version of a composite object is composed of specific versions from its component objects. And the binding between composites and components leads to the important concept of *static* and *dynamic configurations* which allow the binding between a component generic object and one of its specific versions to be deferred until the *is-a-part-of* relationship is actually traversed. Dynamic configuration is more time-efficient since the binding only occurs when it is used in a simulation, therefore eliminating all the unnecessary updates when changes occur in one of its components. It also has the advantage of taking the latest improvements of its components. Let us look at the particle tracking simulation which tracks a particle through the accelerator to see how many turns it can survive within the designed trajectory. As shown in Figure 5.7, `magnetA` and `magnetB` are components of the `beamline` object, which is a part of the `accelerator` object. They are all versioned objects under design, so there will be multiple versions in each object's version set. The a-part-of relationship is referred through generic references, so no static bindings exist before the actual simulation begins. When a new simulation starts, the particle to be simulated passes through each component of the `accelerator` while the simulation actually traverses the composite hierarchy in pre-order. When the particle hits a `beamline` referred by a generic reference, the simulation

will ask the `beamline`'s version graphic to return a version with the latest time stamp according to its context (we assume default context is used here). The `beamline` could experience a couple of versions after the last simulation run. Each version may cause change propagation in the `accelerator`. Because of delayed binding, the `accelerator` will be only updated once when it is really passed by a particle in the simulation. Static binding will cause multiple changes in between, which in this case, is totally unnecessary. We find the latest `beamline` consists of two magnets: `magnetA` and `magnetB`, both referred to by generic references. When we calculate the transformation matrix of `magnetA`, `magnetB` has a new version just checked in with a much better design by a different engineer. We do not know how much improvement has been done since the last simulation run. Thanks to dynamic configuration, the latest version of `magnetB` will be used in the current simulation to reflect the most recent improvements.

5.3.3 Workspace in simulation

Workspaces are named repositories where applications can access simulation objects, change models, select input data, set system parameters and collect simulation results.

A workspace in our model is different from those in an object-oriented database system (OODB). Workspaces in OODB are repositories of versioned objects for the purpose of version control and configuration management. A workspace in our model actually defines the entire simulation environment. It specifies the model to be simulated and a set of pre-selected stimuli. It has to record the new configuration taking shape during the simulation life cycle since it has not yet been checked into the version control system. It also keeps transient data such as simulation inputs (stimuli), system parameters, and simulation results while workspaces in OODB usually only deal with persistent objects. In a simulation, users can check out modeling objects from the version control system and modify them independently in their own workspace, without disturbing or being disturbed

by other developers. Figure 5.8 shows a beamline object in two independent workspaces, *A* and *B*. Adjustments are made to different magnets (`octupole857` and `dipole238`) and cause the configuration of the beamline to evolve in different directions independently in *A* and *B*. Two workspaces actually share all components (default version from the

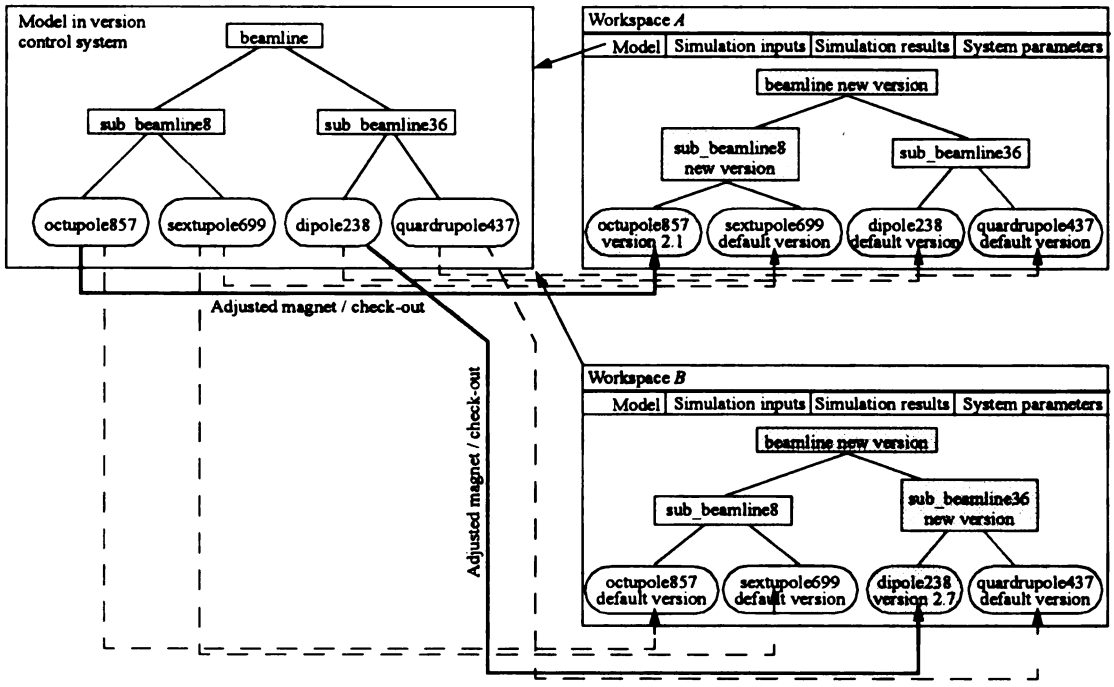


Figure 5.8: Versioning in different workspaces

version control system) except those versioned due to individual adjustments. Workspace is not a separate piece of storage; it is an extension of the repository for a user to work on versioned objects within simulation applications. In our model, the child workspace is used as the current workspace that contains an individual's private "work in progress," and the parent workspace contains public, shared data. To work on a versioned object, it must be locked by checking out to a current workspace. This creates a new version of the object in

the current workspace. The new version is not visible from other workspaces until it is checked into the parent workspace. *Check-in* freezes the version in the parent workspace so that no more changes can be made to it. Further work on it either involves just read access or requires a check out for new version. The flexibility to create alternative versions of a modeling object allows a user to carry on his work on a versioned object even if it is already checked out by someone else. This is achieved by checking out an object on an alternative branch of design. Thus, different workspaces can be used to work on different components (`octupole857` **and** `dipole238` in Figure 5.8) or alternatives of the design simultaneously.

Consider a simulation using our workspace. Engineers at the SSC write a simulation application to find out the initial acceleration pattern using different magnets in a beamline. The application creates a workspace and associates it with the specified beamline model to be simulated. It also sets the initial energy level, beam launch position, starting trajectory and stores them in the workspace. The objective is to tune the beamline to its optimal condition, i.e., achieving the highest beam luminosity (the density of the particles in the beam). The simulation accesses the existing simulation model in the version control system, starts the simulation as specified, collects the output data if any, and finally adjusts the input and system parameters based on the output using the analysis algorithm provided by the simulation analyzer. The workspace records the version of components (here magnets) used in this first round of the simulation. Although nothing has been changed in the simulation data model, the workspace is needed to access the model, as well as bookkeeping the simulation inputs (initial energy level, beam launch position, starting trajectory), system parameters (how many turns the beam should be accelerated through the accelerator), and simulation results (beam luminosity). Usually these simulation data are not part of the model. With the help of workspace, these initial settings and results can be stored in the workspace with a reference to the corresponding model used in the simulation. So next time when the simulation program runs, the workspace brings in the previous

"image" after the last run, which includes the right version of the model, as well as all the settings. Before the second round of simulation is begun, usually not only the input data and system parameters need to be changed based on the result analysis, but the simulation model also needs to be adjusted when certain magnet strengths need to be changed to enhance the performance. When changes involve objects in the model, those objects, such as the magnet, need to be checked out from the version control system into the workspace (like `dipole238` in workspace *B*, Figure 5.8). While changes happen to the object in the workspace, other simulation applications using the same model will not be affected because the experimental model is only recorded in the private workspace (`dipole238` does not change in workspace *A* in Figure 5.8). Now the simulation runs with a new set of input data and system parameters plus an adjusted model only visible in this workspace. The workspace only maintains the delta between the original version of the model and the adjusted one. The new model in the workspace can be checked in to the version control system to become a part of the version history.

Using dependencies for checking out objects in workspaces is a unique feature of our approach. If no dependency exists between two modeling objects, they can be checked out at the same time into different workspaces for parallel development. If dependencies exist between two modeling objects, the object is automatically locked when its dependency is checked out. A branching approach will be used if a locked object needs to be checked out into another workspace. In Figure 5.8, there is a dependency between `sub_beamline8` and `octupole587`. When `octupole587` is checked out to workspace *A* and evolved to a new version `octupole587`:version2.1, `sub_beamline8` is also locked and checked out to evolve to a new version. The same happens to the `beamline`. When workspace *B* requires checking out the beamline while it is locked by *A*, *B* has to check it out on a branch. The evolution of the model actually happened in the workspace while a new configuration binding causes a new version of the composite object (`beamline`) as a result of change propagation through its dependencies.

```
int Dipole238::checkConstraint(Constraint& aConstraint, void *result)
{
// check the type of constraint sent to this magnet object
// if this is a request to check its β function impact
if (aConstraint.instanceOf(BetaFunctionConstraint)) {
 // calculate the current β function which has x and y values
 value betaValue = computeBetaValue();
 // pass the β value to the constraint for checking and get the result back
 result = ((BetaFunctionConstraint)aConstraint).check(betaValue);
 return 1; // indicate success to the caller
}
... ...
// similar code to other constraint types such as DimensionConstraint
... ...
// if aConstraint does not apply to this type of object, pass to base class
return Magnet::checkConstraint(aConstraint, result);
};
```

Figure 5.9: Sample C++ code for constraint checking

Another important feature of our approach is the introduction of constraints that can be attached to a composite object. These constraints are user-defined policies that enforce mutual consistency between components in a binding. For example, the magnetic strength of a bound magnet should not cause a sharp rise of the value of β-tron oscillation function, etc., as illustrated in Figure 5.9. Without such constraints, a binding could result in an inconsistent system in the workspace.

5.3.4 Change notification and propagation in configuration management

The relationship of dependency implicates that when one object changes, all its dependents should get notified and changed accordingly. One of the most important aspects of configuration management is managing change notifications and propagation. Chou and Kim proposed two approaches in [14] for change notification: message-based and flag-based. The message-based approach is further divided into immediate or deferred, depending on whether the affected objects are notified immediately after the changes to a version are committed or at some later time specified by the user.

We define *change propagation* as the process that automatically instantiates new configurations to incorporate a newly created version of one of its components. The basic premise of our approach is that different versions of a modeling object share the same external attributes but may have different internal attributes. There are three key issues to be addressed in the change propagation mechanism:

- Controlling the scope of change propagation using constraints.
- Postponing the time of propagation until it is absolutely necessary.
- Disambiguating the path for change propagation.

Controlling the scope of propagation using constraints

A number of different mechanisms can be used to limit the scope of propagation [39]. The most effective way is to place constraints on the configuration so that change

propagation will halt if creating a new version of the configuration would cause the constraint to be violated. For example, in order to increase beam luminosity, we often try to maximize the strength of the magnetic field that will cause a larger aperture of trajectory to the beam throughput. But this also runs into the risk of losing beams due to the non-linear effect around the edge of the trajectory. Here constraints are used to halt the propagation of β function re-calculation when the accelerator starts to lose the particles. Change is rejected since the new configuration violates the constraint. In our model, constraints are usually attached with the composite object functioned as a special purpose configuration management agent. So constraints can be used on an individual basis tailored to particular requirements, and get enabled during change propagation, which is rather unique compared to other approaches mentioned in [39].

Postponing the time of propagation

Control of the timing of change notification and path of change propagation is very important for effective configuration management. Flag-based deferred change notification is used in our model whenever it is possible. When a set of changes is planned to go to one version of an object, notification should be deferred until all changes in the set have been applied. An effective way to do that is using dynamic configuration binding described in section 4.2, which only responds to the propagation when the affected object is loaded into the simulation environment. However, in certain circumstances, changes are accumulative rather than preemptive. The current change has to be applied to the previous change rather than nullify it. In such a case, message-based change notification is adopted. The propagation path is defined in a composite object used as configuration control node. An optimal path should be defined on each individual composite object based on its dependent graph and type of dependency. Circular propagation should be avoided. Reconsider the four types of dependencies introduced previously:

For exclusive dependency, notification should be immediate since the dependency

is a one-to-one correspondence, and the propagation path is well understood. Exclusive independency is different. We select flag-based notification because dependency may no longer exist when a change occurs. For shared dependency, flag-based notification is recommended since the dependency is a one-to-many relationship; immediate notification may not be efficient here. For shared independency, we also recommend flag-based notification. It is more efficient to trigger the propagation when dependency is explicitly accessed through one of its references.

The dependency graph is established when an attribute is added to the reference list. The propagation using dependency graph is implemented using the Observer pattern [28]. A composite attribute can be regarded as a function of identities, attributes, and constraints. Such a function is implemented as a method invoked when an attribute is created or when events (changes) happen to its "dependent," which may cause an update. For example, when an Actor's attribute strength is modified, its corresponding constraint will be checked. If that constraint is broken, the dedicated method will be called to propagate such a change, i.e., to recalculate the transformation matrices (composite attributes). Transformation matrix object may further check its bound constraint and determine whether a further propagation is needed. In this case, it will recalculate the β-tron oscillation function and phase advance function. Change propagation is depicted in Figure 5.10 through a dependency graph.

One of the dependencies reflected in composition is through reference `Parent` and `Children` in class `Actor`. When a magnet (child) is inserted into a beamline (parent), the dependency is established. Any change to a dependency has its propagation scope. The scope includes not only the version variant attributes of the object to be changed but also those objects that depend on it. The dependency graph specifies this scope and the order to propagate changes within the scope, i.e. the propagation path.

We use a *guard* statement to create a trigger for change propagation. For example, a guard statement:

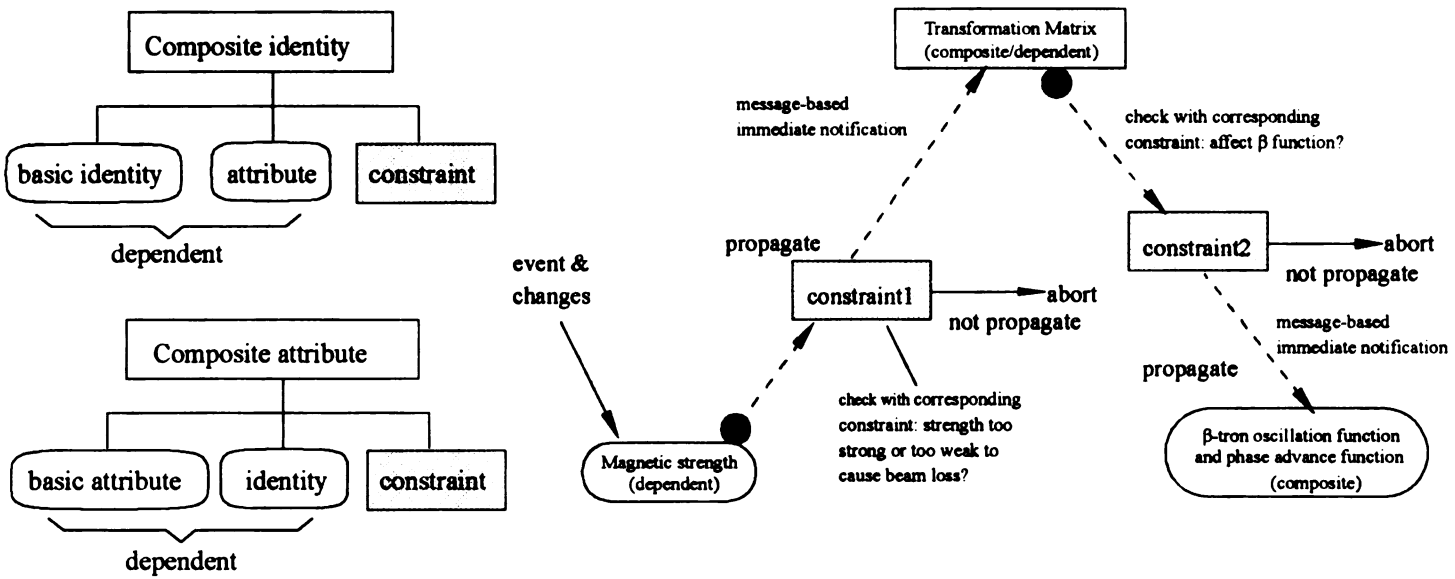


Figure 5.10: Dependent graph and change propagation in composite

```
guard constraint
    target description
        handler function_pointer(ChangeType)
```

is added before the operation of changing the magnetic strength of a magnet. Because such a change will cause propagation to those magnets in the same accelerator (its dependent graph). When a guard statement is added, a trigger for propagation is created. In object-oriented programming, instance variables are kept private and can only be accessed through public member functions, where a trigger can be added and invoked implicitly.

A guard statement specifies the constraints used to test changes. A *handler* is used if propagation is required. The handler contains a pointer to an object that will handle and relay the propagation. A `ChangeType` is provided with the handler to identify the nature of the change. The propagation is controlled in the virtual function `ChangeHandler` in the derived class of `Actor` based on `ChangeType`. This model provides both the dynamic and static binding for change-oriented propagation. The dependent graph under this model can be dynamically updated when a component is inserted or deleted from the object's composition. By default, the handler usually points to the object's current parent. But since handler is just a pointer to the base class `Actor`, it can be redirected. As a matter of fact, a dependent graph is woven while the composition is formed. The task (what should be done) of the propagation is also dynamically bound to the virtual function `ChangeHandler`. In Super Collider, for example, different types of beamline (an aggregation of magnets) may act differently to propagation due to the magnetic strength change of the same type of magnet component. The implementation of `ChangeHandler` is statically bound with the type of derived class from `Actor`. Each composite object has a predictable response to changes of its components.

Disambiguating the path for change propagation to curtail the proliferation of change notification

There are several choices of handling ambiguities, which are inherent in attempting

to propagate changes in configuration using a dependent graph. The first is simply to disallow any change propagation, but this is clearly too restrictive. The second is to create the cross product of all resulting configurations but to represent these as alternative versions of the root of the configuration. But the number of alternative versions could be enormous, few of which are meaningful and of interest to users. The third is to propagate whenever the choices are unambiguous but to abort propagation as soon as an ambiguous situation is encountered. We propose *group check-in* and *group check-out* using our modeling object. Propagation is handled through a logically clustered group of objects. It is desirable that the group is a self-contained composite modeling object. The result of group check-in is to guarantee the unambiguous creation of a single new configuration. In group check-ins, a configuration node spawns a new version at most once, no matter how many times a change occurs. If multiple objects are checked back individually, a new configuration will be created for each, and cause ambiguities and proliferation of change notifications. On the other hand, when objects are checked back as a group, a single new configuration is created. The general rule is: in propagating changes through the lattice of relationships, no change should cause an object to evolve through more than one new version [39]. The modeling object concept provides the vehicle to achieve that.

5.4 Summary

This chapter presented an object-oriented configuration management and version control model for simulation. The composite object model makes simulation modeling more straightforward in terms of engineering design at the SSC. With version history, SSC simulation data can be well tracked and easily compared for engineering analysis. The concept of workspace and context makes the cooperative working environment more manageable. By using generic objects, configuration can be dynamically bound and change propagation can be effectively accomplished using a well-defined dependency graph.

Three simulation applications are built using the `Actor` model. OZ [74, 89] is an object-oriented simulation environment. BumpView [90] is exclusively used for tuning the accelerator to achieve high beam luminosity. OBSERVER [89, 92] (Object-Based Structure EditoR in a Version EnviRonment) is a model driven accelerator designer that extensively uses our CMVC model to obtain optimal design. The model is implemented on top of ObjectStore, an object-oriented database management system, using C++ programming language.

We make no claim that the model proposed here is either optimal or exhaustive. The model will be improved as we obtain more experience and feedback from the field. In particular, the issue of how to handle version merge effectively is far from solved. Future directions will emphasize version merge management and a more efficient change propagation model.

CHAPTER 6

CONCLUSION

With the SSC simulation as the background, the issues of building a reusable simulation framework InterSim have been discussed in this thesis. Our framework decomposes a simulation system into four types of classes (layers). Each layer handles data management, user interface, modeling, and simulation, respectively. Protocols are pre-defined among layers. These protocols are abstract and generic so that they can achieve maximum reusability. Each layer in InterSim is relatively independent and focuses on its own problem domain. Services can be requested by classes in one layer to classes of another layer based on protocols. Such a loosely coupled, responsibility-driven design approach not only promotes productivity but also simplifies the development and maintenance process.

The Actor model we presented in this thesis provides an object-oriented approach for dynamic simulation. By using this model, the user is able to directly modify the system configuration during the simulation process, assigning attributes to objects, binding constraints, and altering relationships. By making attribute a separate object, attribute binding becomes more flexible. Constraint implemented by using exception handling gives

more control to the user in dynamic simulation. The concept of relationship allows more complex connection between objects. Two ways of behavior binding in dynamic simulation are discussed: data-driven is quick but less flexible; type-driven is more dynamic and expandable. An Object-oriented database, ObjectStore, is used in our implementation. This model was used to model SSC lattice structure and to dynamically simulate behavior of the accelerator.

We integrate configuration management and version control into our model by introducing the concept of modeling object, a coherent design unit for versioning. With version history, simulation data can be well tracked and easily compared for engineering analysis. The concept of workspace and context makes cooperative working environment more manageable. By using generic objects, configuration can be dynamically bound, and change propagation can be effectively accomplished using a well-defined dependency graph.

Three simulation applications are built using the Actor model. OZ [74, 89] is an object-oriented simulation environment. BumpView [90] is exclusively used for tuning the accelerator to achieve high beam luminosity. OBSERVER [89, 92] (Object-Based Structure EditoR in a Version EnviRonment) is a model driven accelerator designer that extensively uses our CMVC model to obtain optimal configuration. Those applications were used by the Lattice Database Group, Lattice Simulation Group, and Instrumentation and Diagnostics Group at the SSC lab. Physicists can visualize the entire SSC accelerator complex and zoom in to different levels of details using OZ. OZ provides a vivid feedback to their designs without operating on a real machine. BumpView is the first tool in SSC to support visual simulation and modeling. With BumpView, physicists can see the accelerator and its correction effect on a beam passing through it. BumpView brings the accelerator lab to their desktop computers. OBSERVER is the first modeling tool that allows direct object manipulation in accelerator design. Physicists can see their beamline decomposed level-by-level into a hierarchy through a GUI. The composite can be directly

edited and then versioned through our CMVC facility, thus bring the accelerator design from piles of paper and hours of calculation to their fingertips. The general comments are very positive, especially to the object-oriented GUI front end, the modeling capability, the model extensibility to other simulation applications at the SSC, and the integrated CMVC facility. Performance is an issue because most of our simulation engines run at the client side on a UNIX workstation. But most of the simulations require hours of time running on super computers or parallel machines. How to enable our simulation engines to run on those machines and communicate with the rest of the system is a challenging job. Version merge also still needs improvement.

OZ and BumpView are completely finished and deployed in the field. OBSERVER as a browser is completed. Some work remains to fully support lattice-editing capability with CMVC support.

All simulation applications, as well as our framework, are written in C++ on a UNIX platform. We use Motif and InterViews library [44, 45] as a base for GUI implementation. ObjectStore [42] is used as a database for object persistency. The entire project consists of more then one hundred classes and thousands of lines of code. The code archives can be obtained from the Fermi National Accelerator Laboratory in Chicago.

To further this research, it is recommended that following topics be investigated in the future:

- Extension to support version merge capability in the CMVC model that provides a mechanism to evaluate two versions in terms of their differences and strategies to combine them into one version.
- A constraint framework that provides a set of reusable constraint checking algorithms for common configurations in simulation applications.
- Design methodology support such as the Unified Modeling Language (UML) in the design process.
- Parallel-processing support in the SIMULATOR layer that provides simulation

engines run on parallel machines. Since most of the accelerator simulation engine deals with a tremendous amount of data and often uses parallel processing, such support is important.

- Web support in the INTERFACE layer to provide a web-enabled graphical user interface for simulation.

BIBLIOGRAPHY

[1] Agrawal, R.; Buroff, S.; Gehani, N.; Shasha, D.; "*Object Versioning in Ode*," Proceedings of Seventh International Conference on Data Engineering, 1991.

[2] Ahmed, Rafi and Navathe, Shamkant B.: "*Version management of composite objects in CAD databases*," Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, pp. 218-227, Denver, Colorado, May 1991.

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

[4] Baba, Marietta L.; Mejabi, Olugabenga: "*Advances in sociotechnical systems integration: object-oriented simulation model for joint optimization of social and technical subsystems*," International Journal of Human Factors in Manufacturing, Vol. 7, No. 1, winter of 1997, pp. 37-61.

[5] Beech, David and Mahbod, Brom: "*Generalized version control in an object-oriented database*," Proceedings of the 1988 IEEE Fourth International Conference on Data Engineering, pp. 14-22, Los Angeles, California, February 1988.

[6] Bersoff, Edward H.: "*Elements of software configuration management*," IEEE Transactions on Software Engineering, SE-10(1): pp. 79-87, January 1984.

[7] Blakey, Adrian; "*Beyond the RDBMS: object database management systems: Data models, data definition language, and data access*," Object Magazine, March/April 1992.

[8] Booch, Grady; *Object-Oriented Design with Application*, The Benjamin/Cummings Publishing Company, Inc., 1991

[9] Calhoun, D.; Lewandowski, A.: "*Object oriented framework for dynamical systems modeling: implementation in C plus plus*," Proceedings of the IEEE Annual Simulation Symposium 1994, pp. 70-77.

[10] Cattell, R. G. G.: *Object Data Management: object-oriented and extended relational database system*, Addision-Wesley Publishing Company, Inc. 1991.

[11] Chakravarty, Amiya K.; Jain, Hemant H.; Liu, John J.; Nazareth, Derek L.: "*Object-oriented domain analysis for flexible manufacturing systems*," Integrated Computer-Aided Engineering, Vol. 4, No. 4, 1997, pp. 290-309.

[12] Champeaux, D. de; "*Object-Oriented Analysis and Top-Down Software Development*," European Conference on Object-Oriented Programming, pp. 360-375, July 1991.

[13] Choi, Byoung K.; Han, Kwan H.; Park, Tea Y.: "*Object-oriented graphical modeling of FMSs*," International Journal of Flexible Manufacturing Systems, Vol. 8, No. 2, Apr. 1996. pp. 159-182.

[14] Chou, Hong-Tai and Kim., Won: "*A unifying framework for version control in a CAD environment*," Proceedings of the Twelfth International Conference on Very Large Data Bases, pp. 336-344, Kyoto, Japan, August 1986.

[15] Coad, P.; Yourdon, E.: *Object-Oriented Analysis*, 2nd edition, Yourdon Press Computing Series, Prentice Hall, 1991.

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

[17] Coatta, Terry; Neufeld, Gerald: "*Distributed configuration management using composite objects and constraints*," Distributed System Engineering (UK): Vol. 1, No. 5 Sept. of 1994, pp. 294-303.

[18] Coplien, James O.; *Advanced C++ Programming Style and Idioms*, pp. 133f, AT&T Bell Laboratories. Addison-Wesley Publishing Company, 1992.

[19] Cornelio, A. ; Navathe, Shamkant B. ; Doty, Keith L.; "*Extending Object-Oriented Concepts to Support Engineering Applications*," Proceedings of Sixth International Conference on Data Engineering, 1990.

[20] Cubert, Robert M.; Fishwick, Paul A.: "*Framework for distributed object-oriented multimodeling and simulation*," 1997 IEEE Winter Simulation Conference Proceedings, pp. 1315-1322.

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

[22] Davis, Stephen R.; "*C++ objects that change their type*," Journal of Object-Oriented Programming," July/August 1992, Vol.5, No. 4.

[23] Decorte, G.; Eiger, A.; Kroenke, D.; Kyte, T.: "*An Object-Oriented Model for Capturing Data Semantics*," Eighth International Conference on Data Engineering, pp. 126-135, Feb. 1992, Tempe, Arizona.

[24] Dittrich, Klaus R. and Lorie, Raymond A.: "*Version support for engineering database systems*," IEEE Transactions on Software Engineering, 14(4): pp. 429-437, April 1988.

[25] Doherty, Michael; Hull, Richard; Rupawalla, Mohammed: "*Structures for Manipulating Proposed Updates In Object-Oriented Databases*," Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, June 4-6, 1996, pp. 306-317.

[26] Ege, Raimund K.; *Programming in an Object-Oriented Environment*, Academic Press, Inc., 1992

[27] Faltenbacher, W.: "*Computer aided software configuration management*," Proceedings of the 12th International Computer Software and Applications Conference, pp. 18-25, Chicago, Illinois, October 1988.

[28] Gamma, Helm, Johnson, Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[29] Gengler, Barbara: "*CACI Leverages Simobject. (Comnet III) (Distributed Applications)*," LAN Computing, n6, v5, June, 1994 p50(1).

[30] Goldberg, Adele; *Smalltalk-80: The Interactive Programming Environment*, Xerox PARC, Addison-Wesley Publishing Company, 1894.

[31] Harter, Richard; "*Object-Oriented Software Configuration Management*," Dr. Dobb's Journal, #181 Oct., 1991.

[32] Hong, Shuguang; Maryanski, Fred; "*Using a Meta Model to Represent Object-Oriented Data Models*," Proceedings of Sixth International Conference on Data Engineering, 1990.

[33] Hughes, John G.; *Object-Oriented Databases*, Prentice Hall, 1991.

[34] Huh, Soon-Young; Rosenberg, David A.: "*Change management framework: dependency maintenance and change notification*," Journal of Systems and Software, Vol. 34, No. 3, Sep. 1996. pp. 231-246.

[35] *Introduction To ObjectStore*, 1992 Object Design, Inc.

[36] Jordan, Mick; Vanter, M. L.; Van De: "*Software Configuration Management in an Object Oriented Database*," USENIX, COOTS, June 26-29, 1995.

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

[38] Katz, Randy H.; Chang, Ellis; and Bhateja, Rajiv: "*Version modeling concepts for computer-aided design databases*," Proceedings of ACM SIGMOD 86, pp. 379-386, Washington, D.C., May 1986.

[39] Katz, Randy H.: "*Toward a unified framework for version modeling in engineering databases*," ACM Computing Surveys, 22(4): pp. 375-408, December 1990.

[40] Kiczales, Gregor: "*Towards a New Model of Abstraction in Software engineering*," Proceedings of 1991 International Workshop on Object Orientation in Operating System, pp.127-128, Palo Alto, California.

[41] Kiviat, Philip J.; "*Simulation, Technology, and the Decision Process*," ACM Transaction on Modeling and Computer Simulation, Vol. 1 Num. 2, April, 1991

[42] Lamb, Charles; Landis, Gordon; Orenstein, Jack; Weinreb, Dan; "*The ObjectStore Database System*," Communications of the ACM, Vol 34, No. 10, Oct. 1991.

[43] Lieberherr, K.; Holland, I.; "*Assuring Good Style for Object-Oriented Program*," IEEE Software, pp. 38-48, Sept. 1989.

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

[45] Linton, Mark; "*InterViews Reference Manual 3.1 Beta*," June 1992, Silicon Graphics.

[46] Mahler, Axel; Lampen, Andreas: "*Integrating configuration management into a*

generic environment," Proceedings of ACM SIG-SOFT '90: Fourth Symposium on Software Development Environments (SDE4), pp. 229-237, Irvine, California, December 1990. Published as ACM SIG-SOFT Software Engineering Notes, vol. 15, no. 6, December 1990.

[47] Mamou, Jean-Claude; Medeiros, Claudia Bauzer; "*Interactive Manipulation of Object-oriented Views*," Proceedings of Seventh International Conference on Data Engineering, 1991.

[48] Monarchi, David E.; Puhr, Gretchen I.; "*A Research Topology for Object-oriented Analysis and Design*," Communications of the ACM, Sept. 1992, p. 35.

[49] Mujtaba, M. Shahid: "*Enterprise Modeling And Simulation: Complex Dynamic Behavior Simple Model Of Manufacturing*," Hewlett-Packard Journal, n6, v45, Dec, 1994 pp. 80-113.

[50] Nielsen, Norman R.: "*Application of Artificial Intelligence Techniques to Simulation, Knowledge-Based Simulation, Methodology and Application*," pp. 1-19, Advances in Simulation, Vol. 4, Springer-Verlag, 1991.

[51] Noro, Masami; Harada, Kenichi: "*Version control of software systems in the STEP environment*," In Proceedings of COMPSAC 87, pp. 110-116, 1987.

[52] Object Design, Inc.: *ObjectStore User Guide*, Release 2.0, Oct, 1992

[53] ObjectStore Reference, *ObjectStore User Guide*, 1991 ObjectDesign, Inc.

[54] Odell, James J.: "*Specifying structural constraints*," OBJECT Magazine, 6(6), pp. 12-16, Oct. 1993.

[55] Parashar, Manish; Wheeler, John A.; Pope, Gary; Wang, Kefei; Wang, Peng: "*New generation EOS compositional reservoir simulator: Part II - framework and multiprocessing*," Proceedings of the SPE Symposium on Reservoir Simulation 1997, Society of Petroleum Engineers, Richardson, TX. pp. 31-38.

[56] Pavicic, Mark; "*Making the Transition to an Object-Oriented Simulator*," Proceedings of the SCS Multiconference on Object-Oriented Simulation, pp. 65-71, Jan. of 1991.

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

[58] Pritsker, A. Alan B.: *Introduction to Simulation and SLAM II*, Third Edition, Halsted Press Book, 1986.

[59] Ralston, Anthony; Reilly, Edwin D.: *Encyclopedia of Computer Science*, Third Edition, Vain Nostrand Reinhold, 1993.

[60] Reznik, Assaf: "*Character Simulation With ScriptX; A General-Purpose Framework for Dynamic Behavior*," Dr. Dobb's Journal, n13, v19, Nov, 1994, pp. 76-82.

[61] Robinson, J. T.; Kisner, R. A.: "*An Intelligent Dynamic Simulation Environment: An Object-Oriented Approach*," Proceedings of the IEEE International Symposium on Intelligent Control, 1988, pp. 688-692.

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

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

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

[65] Sanderson, D.P.; Sharma, R; Rozin, R.; Treu, S.; "*The Hierarchical Simulation Language HSL: A Versatile Tool for Process-Oriented Simulation*," ACM Transaction on Modeling and Computer Simulation, Vol. 1 Num. 2, April, 1991

[66] Sciore, Edward: "*Versioning and Configuration Management in an Object-Oriented Data Model*," VLDB Journal, 3, pp. 77-106, 1994.

[67] Sengupta, Soumitra; Dupuy, A.; Schwartz J.; Yemini, Y.; "*An Object-Oriented Model for Network Management*," Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, pp.283-295, Prentice-Hall, 1991.

[68] 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

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

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

[71] Talman, Richard; "*A Universal Algorithm for Accelerator Correction*," Laboratory of Nuclear Studies, Cornell University, Ithaca, NY 14853, Sept. 26, 1991.

[72] Tan, P. L.; Dillon, T. S.: "*The Conceptual Design of OSEA: An Object-Oriented Semantic Data Model*," Proceedings of the Fourteenth Annual International Computer Software and Application Conference, pp. 221-230, Oct. 1990, Chicago, Illinois.

[73] Tello, Ernest R.: "*Object-Oriented Programming for Artificial Intelligence*," 1989 by Addison-Wesley Publishing Company, Inc.

[74] 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.

[75] Tsuda, Kazuyuki; Yamamoto, Kensaku; Hirakawa, Masahito; Tanaka, Minoru; Ichikawa, Tadao: "*MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures*," IEEE Transaction on Knowledge and Data Engineering, Vol. 3, No. 4 December 1991.

[76] Vaishnavi, V.K.; Buchanan, G.C.: "*Data/Knowledge paradigm for the modeling and design of operations support systems*," IEEE Transaction on Knowledge and Data Engineering, Vol. 9, NO. 2, Mar-Apr 1997, pp. 275-291.

[77] Vaughal, Paul. W.; Newton, David. E.; Johns, Rich P.; "*PRISM: An Object-Oriented System Modeling Environment in C++*," Proceedings of the SCS Multiconference on Object-Oriented Simulation, pp. 65-71, Jan. of 1991.

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

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

[80] Wilk, Micheal R.; "*An Object-Oriented Constraint Solver*," OOPSLA '91 Conference Proceedings, SIGPLAN Notice, Vol.26, No. 11, Nov. 1991.

[81] Williams, Tom: "*New CASE Tools Aimed At Coordinating Complex Projects. (Special Report: CASE Tools For Embedded And Real-time Applications)*," Computer Design, n5, v33, April, 1994, pp. 65-72.

[82] Williamson, Ronald; Stucky, Jack; "*An Object-Oriented Geographical Information System*," Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, pp. 296-323, Prentice-Hall, 1991.

[83] Winston, Howard; Tulpule, Sharayu: "*Fault model analysis based on compositional modeling*," Proceedings of the 1997 International Gas Turbine & Aeroengine Congress and Exhibition

[84] Wirfs-Brock, R.: *Responsibility-Driven Design*, Prentice Hall, 1991.

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

[86] Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.: *Designing Object-Oriented Software*, pp. 33-36, pp. 161-176, Prentice Hall, 1991.

[87] Zeigler, Bernard P.: *Object-Oriented Simulation with Hierarchical, Modular Model*, Academic Press, 1990.

[88] Zeller, Andreas: "*A unified version model for configuration management*," Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, volume 20 (4) of ACM Software Engineering Notes, pp. 151-160. ACM Press, October 1995.

[89] Zhou, Jiasheng; Chung, M.J.: "*Object-Oriented Simulation for the Superconducting Super Collider*," Transaction of Society for Computer Simulation, Vol. 12, No. 1, pp. 1-26, March 1995.

[90] Zhou, Jiasheng; Chung, M.J.: "*Object-Oriented Modeling for Dynamic simulation*," 1993 International Simulation Technology Multiconference, San Francisco, Nov. 7-10, 1993.

[91] Zhou, Jiasheng; Chung, M.J.: "*Object-Oriented Simulation for the Superconducting Super Collider*," Object-Oriented Simulation Conference (OOS '93), San Diego, CA. Jan. 17-20, 1993.

[92] Zhou, Jiasheng; Bhogavalli, Rao: "*A Graphical Hierarchy Browser for the SSC Lattice Configuration Database*," SSC Internal Report: SSCL-N-804, December of 1992.

[93] Zhou, Jiasheng; Chung, M.J.: "*Configuration Management and Version Control in Object-Oriented Simulation for the Superconducting Super Collider*," being submitted to the ACM SIGMOD.

APPENDICES

Glossary

A

Actor - A composite object model for dynamic simulation described in this thesis (64, 68)
aggregation - A collection of objects that can be treated as a whole (63)
attribute - A qualitative or quantitative measurement of characters of an object (64, 69)

B

behavior modeling - Modeling the action and reaction of a system in response to changes (42)
BumpView - BumpView is a simulation application built with the framework developed in this thesis research to adjust the beamline configuration for large particle throughput in an accelerator (22)

C

change notification - A mechanism to notify dependents when change occurs to an object (120)
classification - The systematic grouping of objects based on their common characteristics in object-oriented design (63)
CMVC - Configuration management and version control (3)
component - An object that can be a part of the other object (2, 26)
composite graph - A directed acyclic graph to represent the object composite hierarchy (98)
configuration - A specification of how a composite object is constructed from other objects and their inter-relationships (64, 95, 101)
configuration change - A change of the way a particular composite object is formed, including changes to its components and relationships among them (2, 72)
constraint - An object used to monitor another object against certain thresholds of limitations (64, 69)
context - The circumstances in which a configuration binding happens (112)
contract - An agreement between two objects (29)
coupling - The degree to which objects depend on each other (66)

D

DAG - Directed Acyclic Graph (97)
DATA layer - A layer in the framework developed in this thesis research to provide a uniform object-oriented data representation for heterogenous data sources (30)
data modeling - The modeling of data representation to the application (33)
decomposition - Breaking a large system (object) into smaller sub-systems (objects) (5, 66)
delegation - An implementation mechanism in which an object forwards a request to another object. The delegate carries out the request on behalf of the original object (58, 68)
dependency - One object is hierarchically subordinated in some way (physical or logical) with respect to another object (98)
dependency graph - A graph used to model dependencies among objects in configuration management. An edge between two nodes represents the dependency relationship

(98)

domain analysis - A requirement analysis in a particular knowledge domain (63)

dynamic behavior - The action and reaction demonstrated during the simulation exhibit the *dynamic behavior* of the composite object (2)

dynamic binding - A run-time association between a composite object and particular versions of its components (29)

dynamic configuration - Change the configuration of a composite object during the simulation process (114)

dynamic data - Data generated as the footprints of simulation process as it goes (22)

dynamic simulation - In this thesis, *dynamic simulation* focuses on action and reaction of a composite object when its configuration undergoes changes in a particular scenario. (2, 27, 66, 142)

E

element - An object that cannot contain any other object (21)

encapsulation - A technique to hide some of the internal implementation details from the object's external interface (4)

external method - A method to response message defined in object's external interface or protocol (70)

F

framework - A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software such as simulation (5, 6, 142)

G

generic object - A *generic* object is an object where the external attributes are defined, but the internal attributes are not bound (110)

generic reference - A reference to a generic object (111)

H

I

identity - An attribute used to distinguish one object from the others (64, 69)

INTERFACE - A layer in the framework developed in this thesis research to provide GUI support for simulation (31)

internal method - A method only used by internal implementation of an object (70)

L

layer - Layer stands for a set of cohesive and self-contained services or functionalities in a particular domain such as model, simulation engine, data management, and user interface (5, 28)

M

minimum cover - Let *s(a)* be a set of objects whose model (including its identity, attributes, constraints, and relationships) could be affected by changing *a*, the model of one particular object. Then the propagating executor *e* is the Actor with a minimum

W

INDEX

INDEX

INDEX