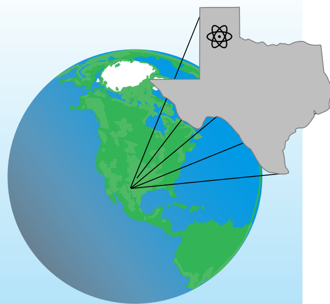

ANRCP-1999-17
April 1999

Amarillo National Resource Center for Plutonium

A Higher Education Consortium of The Texas A&M University System, Texas Tech University, and The University of Texas System

User Manual for Storage Simulation Construction Set

Anil Sehgal and Richard A. Volz
Computer Science Department
Texas A&M University

Edited by

Angela L. Woods
Technical Editor

600 South Tyler • Suite 800 • Amarillo, TX 79101
(806) 376-5533 • Fax: (806) 376-5561
http://www.pu.org

This report was prepared with the support of the U.S. Department of Energy (DOE) Cooperative Agreement No. DE-FC04-95AL85832. However, any opinions, findings, conclusions, or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of DOE. This work was conducted through the Amarillo National Resource Center for Plutonium.

This page intentionally left blank.

ANRCP-1999-17

AMARILLO NATIONAL RESOURCE CENTER FOR PLUTONIUM/
A HIGHER EDUCATION CONSORTIUM

A Report

User Manual for Storage Simulation Construction Set

Anil Sehgal and Dr. Richard Volz
Department of Computer Science
Texas A&M University
College Station, TX 77843

Submitted for publication to

ANRC Nuclear Program

April 1999

This page intentionally left blank.

User Manual for Storage Simulation Construction Set

Anil Sehgal and Dr. Richard Volz
Department of Computer Science

Abstract

The Storage Simulation Set (SSCS) is a tool for composing storage system models using Telegrip. It is an application written in C++ and motif. With this system, the models of a storage system can be composed rapidly and accurately. There are very aspects of the SSCS and are described within this report.

This page intentionally left blank.

TABLE OF CONTENTS

1. INTRODUCTION..1
1.1 SSCS Overview..1

2. FILE ORGANIZATION..3

3. SSCS USER-INTERFACE GUIDE..5

This page intentionally left blank.

LIST OF FIGURES

Figure 1: Directory Structure 3

Figure 2: SSCS Main Menu 5

Figure 3: New Vault Option Menu 6

Figure 4: Open Vault Dialog Box 7

Figure 5: Object Menu 8

Figure 6: New Object Type Dialog Box 8

Figure 7: View/Delete Object Type Menu 9

Figure 8: Create Object Dialog Menu 10

Figure 9: User Input Dialog Box 11

Figure 10: View/Delete Object Dialog Box 11

Figure 11: Constraint Menu 12

Figure 12: Creation of a Constraint 13

Figure 13: Subclass/Constraint Dialog Box 14

Figure 14: Delete Constraint Dialog Box 14

Figure 15: Constraints Present in a Currently Opened Vault 15

Figure 16: Dialog Box for Modifying or Viewing a Constraint 15

Figure 17: Warning Message Dialog Box 16

Figure 18: TGRIP Menu 16

Figure 19: CLI Dialog Box 17

1. INTRODUCTION

The Storage Simulation Construction Set (SSCS) is a tool for composing storage system models using Telegrip. This document describes how to use the tool. The user should be familiar with the SSCS design document, which explains the terms and concepts in SSCS.

1.1 SSCS Overview

The SSCS is an application program written in C++ and motif. With SSCS the models of the storage system can be composed rapidly and accurately. Some of the highlights of SSCS are as follows:

- In SSCS, there is a library of graphical models of storage system components.
- Each graphical model is represented by a C++ class in SSCS.
- Each of the geometric models has some common attributes, such as location and orientation.
- Additional attributes or properties can be added to these geometric models with SSCS.
- Relationships between different geometric models can be defined.
- The relationships are expressed as a set of constraints, each of which is also a C++ class.
- New kinds of graphical models can be added to the library of the graphical models and these can be incorporated into SSCS.
- New constraints can be defined for any model.

The source code of SSCS consists of a number of files, which are organized in a directory structure. The file organization and the directory structure of SSCS are explained in Section 2 of the report. Section 3 is a guide for using the SSCS interface. Appendix – A has the templates for adding the object and constraint sub-classes.

This page intentionally left blank.

2. FILE ORGANIZATION

The file organization in SSCS is as follows. The top directory contains the source code for C++ files (.C) and the header files (.h) for object and constraint class definition. It also contains `sscs.so`, which is the shared object file (similar to a library) formed by linking all the object files. For starting Telegrip and SSCS the required environment variables are set by the script `astest`.

There are 2 sub-directories named library and sub-class. The library sub-directory is the library of Telegrip geometric models. The sub-class directory contains the user-defined sub-classes for objects (in object sub-directory) and constraints (in constraint) sub-directory. Figure 1 shows the directory structure:

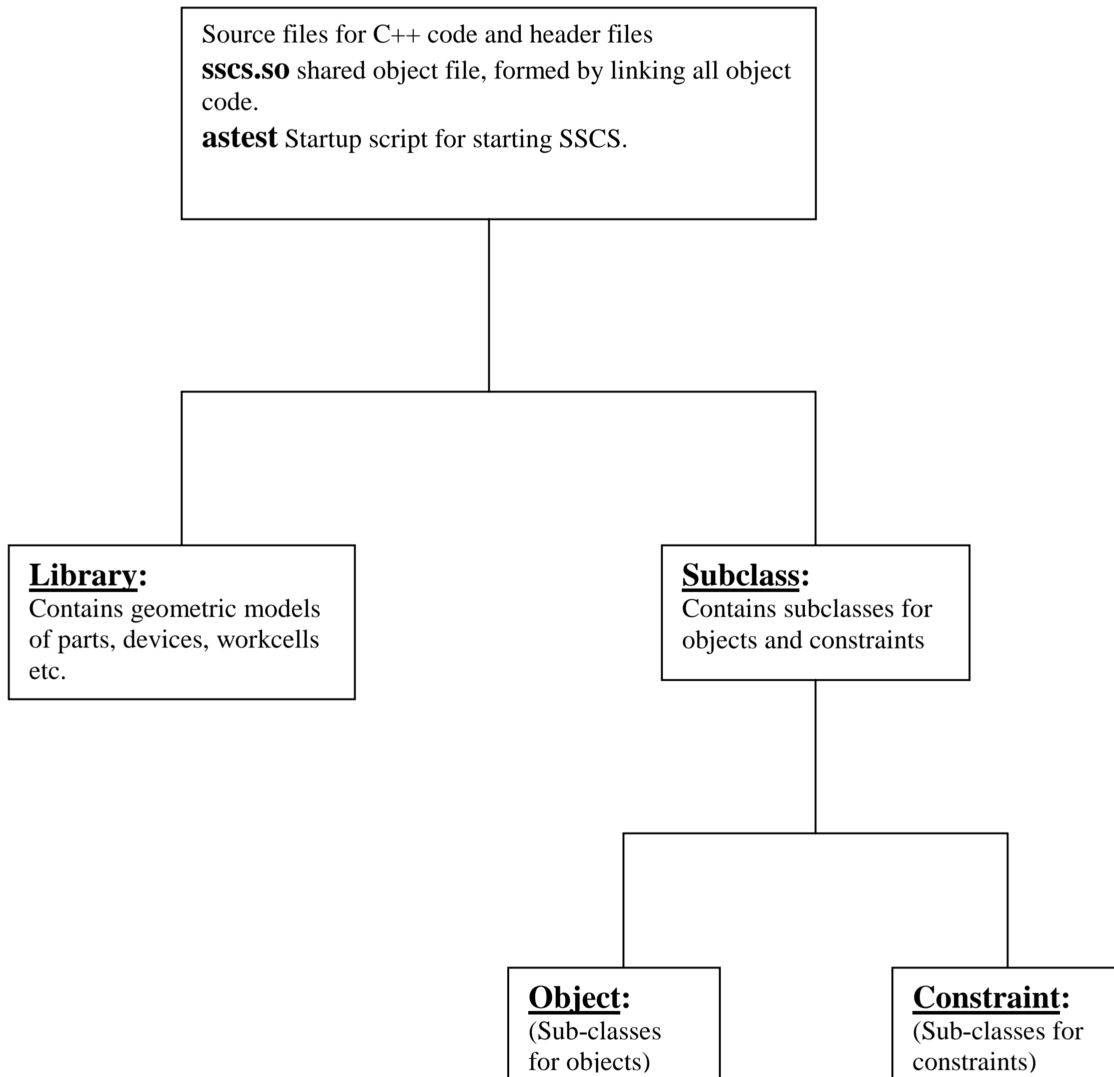

Figure 1: Directory Structure

This page intentionally left blank.

3. SSCS USER-INTERFACE GUIDE

The SSCS can be started by running the script ‘astest’. This script does the following:

1. Sets up the required paths for Telegrip library routines.
2. Starts the TELEGRIP.
3. Starts the SSCS.

The SSCS main menu is shown in Figure 2. This interface mainly consists of a menu-bar and a text area. The menu bar consists of the following menus:

(a) File Menu
(b) Object Menu
(c) Constraint Menu
(d) Tgrip Menu (Telegrip Menu)
(e) Help Menu

File Menu: This is a pull-down menu. When this menu is invoked by Left Mouse button, it displays the following four options:

- New Vault
- Open Vault
- Save Vault
- Quit SSCS

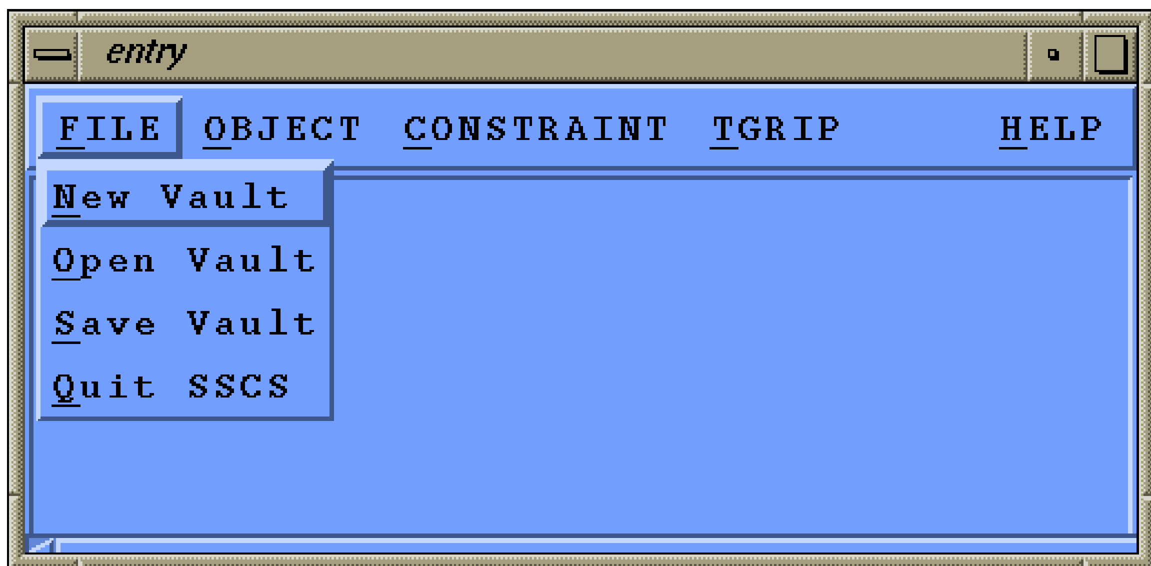

Figure 2: SSCS Main Menu

New Vault: This is for creating a new vault. When this button is selected the following dialog box appears for taking the user's input for the name of the vault being created and the path of the associated Telegrip workcell.

After entering the name and the full path of the workcell, when 'ok' is pressed, a new workcell with the given name appears in Telegrip and a file called 'name.val' is created in the current directory, where 'name' is the name of the vault user the has created. Each SSCS vaults is represented by a file with extension 'val'.

The 'Clear' button is for clearing the text fields for the name and the Telegrip path of this dialog box.

The 'Cancel' button is for closing the dialog box without performing any operation.

Open Vault: This option is for opening an existing vault for modification or viewing. When this option is selected, a file selection dialog box is displayed. The user has to select the appropriate '.val' file, which represents the vault. When 'ok' is pressed, the vault gets loaded in SSCS and the corresponding workcell appears in Telegrip. The file selection dialog box is shown in Figure 4.

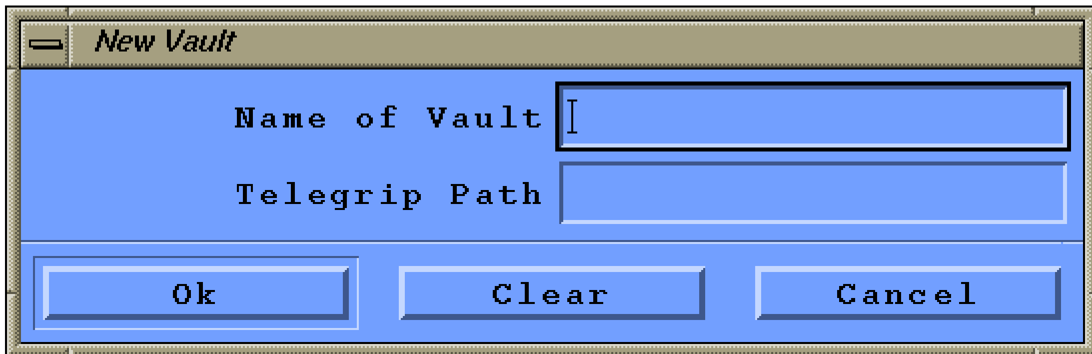

Figure 3: New Vault Option Menu

Figure 4: Open Vault Dialog Box

The 'cancel' button is for closing the dialog box without making any selection.

Save Vault: This button is for saving the opened vault. When pressed the current configuration of the vault is saved, and the corresponding workcell in Telegrip is also saved.

Quit SSCS: This button is for exiting from SSCS. When invoked, it closes both Telegrip and SSCS.

Object Menu: This menu is for carrying out operations related to objects and object types. This menu has the following options in the pull down menu:

- New Object Type
- Delete Object Type
- Create Object
- Delete Object

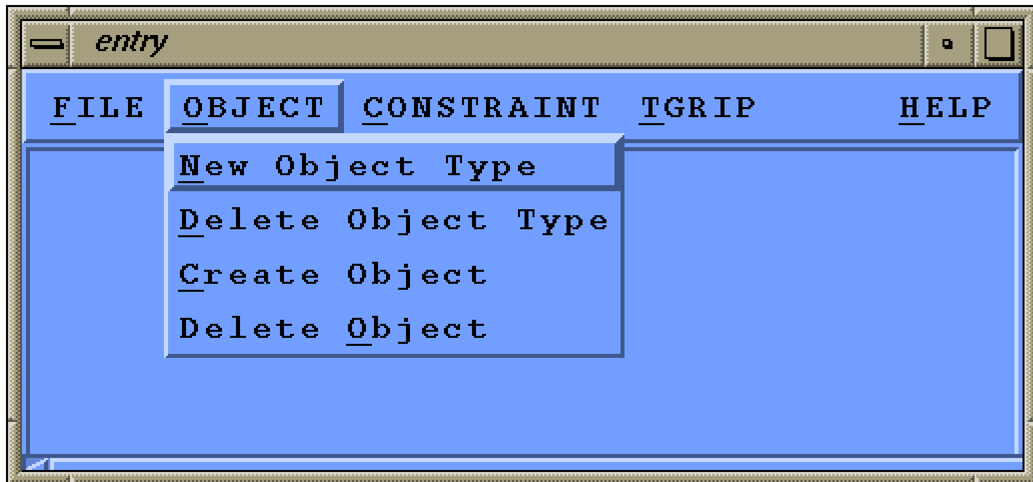

Figure 5: Object Menu

New Object Type: This is for populating the Telegrip's library with new object types. For adding a new object type, the geometric model of the object has to be first created in Telegrip and saved in the library. When this button is pressed the dialog box shown in Figure 6 appears.

The user has to enter the name of the object type and the path of the associated geometric model in Telegrip's library. The name of the object type should be same as the name of the file, which represents the geometric model of the object in Telegrip. When a new object type is created the object super class is instantiated and the instance is appended to the list of other object types, which are already present.

Delete Object Type: This option is for deleting an object type. When this button is selected the selection dialog box shown in Figure 7 appears.

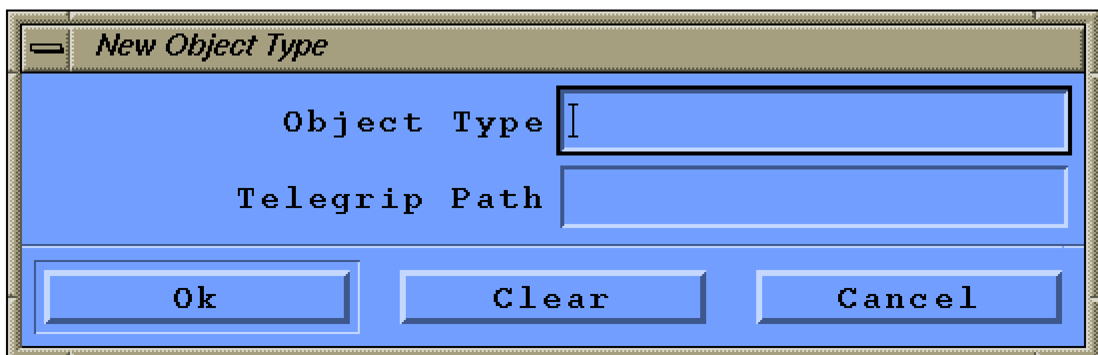

Figure 6: New Object Type Dialog Box

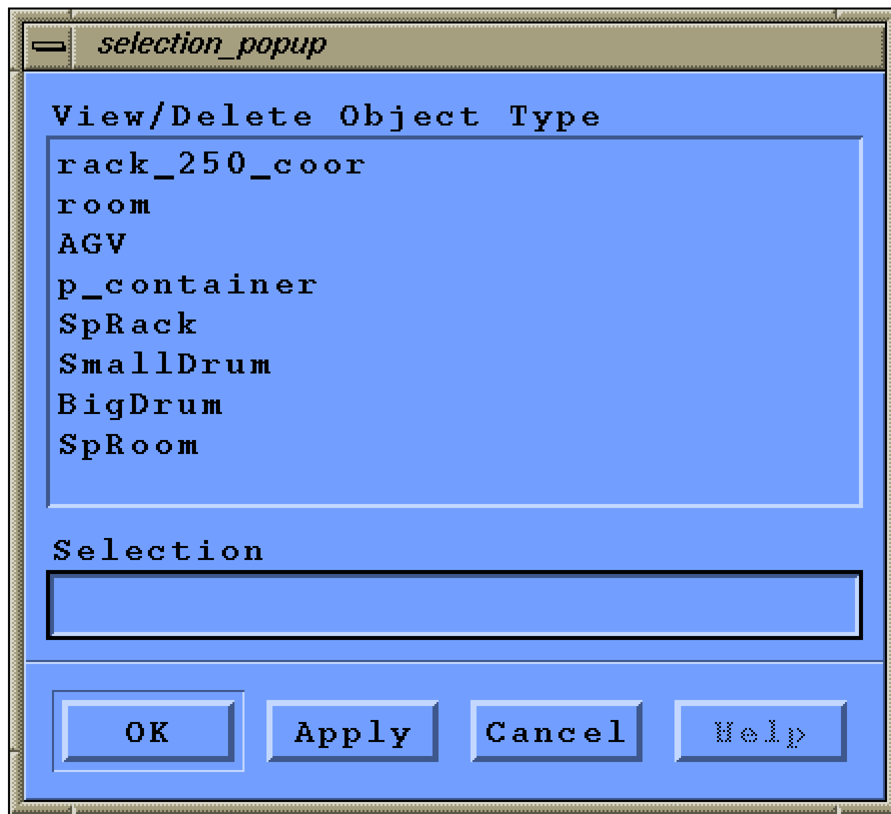

Figure 7: View/Delete Object Type Menu

On selecting the object type from the list and pressing 'ok', the object type is deleted.

Create Object: This option is for creating an object in a vault (a vault should be open before user can create an object). When this button is invoked the following dialog box appears which displays the list of object types available to the user.

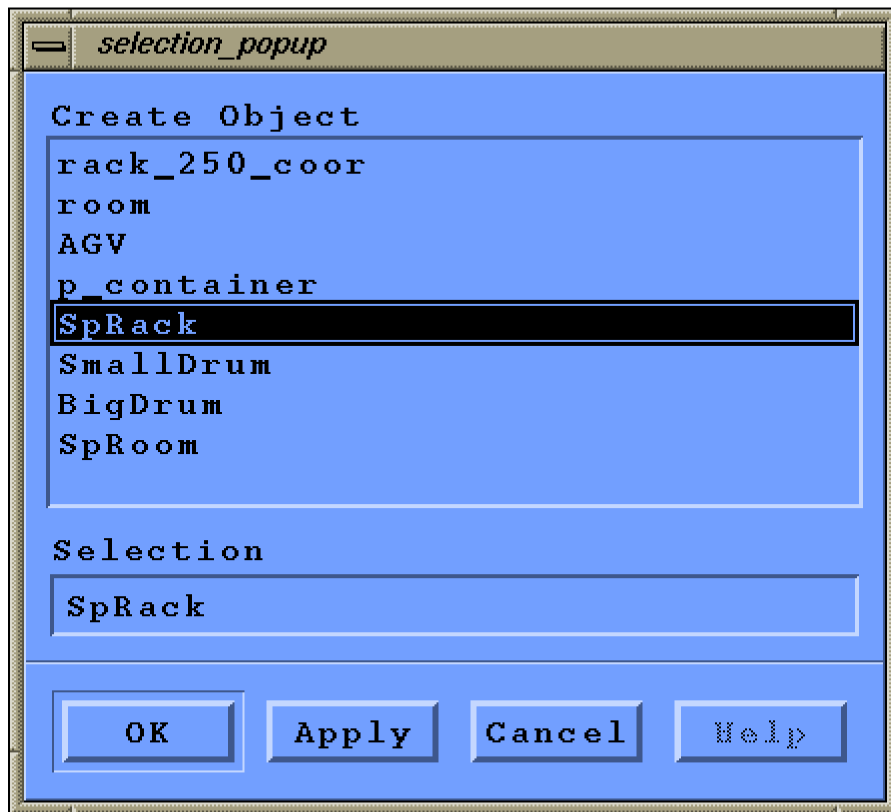

Figure 8: Create Object Dialog Box

When a particular object type is selected and 'ok' is pressed, the dialog box shown in Figure 9 appears for getting the user's input. After entering the name of the object, its location and orientation, when 'ok' is pressed, the object is created in the Telegrip's workcell.

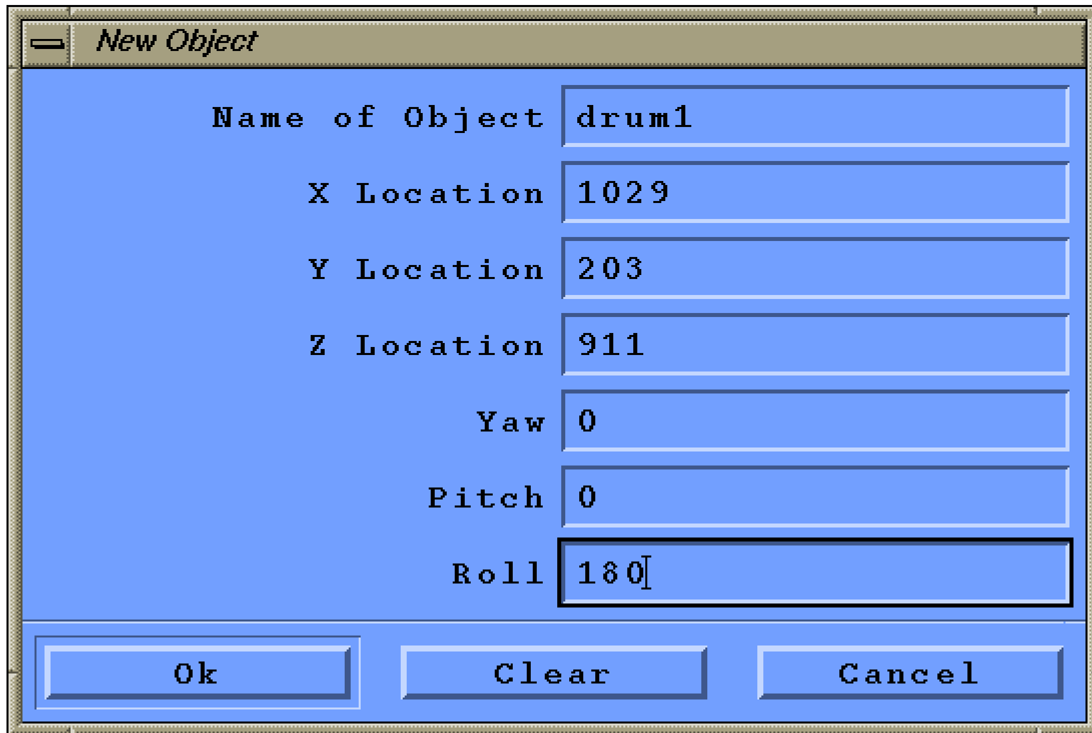

Figure 9: User Input Dialog Box

Delete Object: This option is for deleting an object from a vault. When this button is pressed, the selection dialog box shown in Figure 10 appears which shows all the objects present in the particular vault.

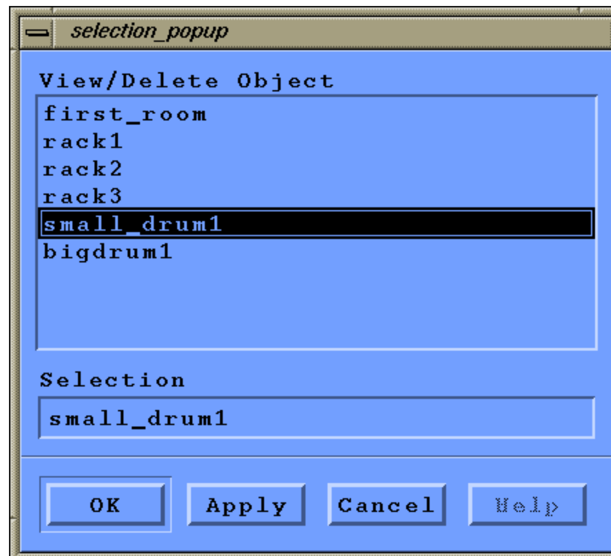

Figure 10: View/Delete Object Dialog Box

When a particular object is selected and 'ok' is pressed, the object is deleted from the vault and also from the Telegrip's workcell.

Constraint Menu: This menu is for carrying out operations related with constraints. The user can create, delete, modify and check constraints. It has the following options:

New Simple Constraint
- New Specific Constraint
- Modify Constraint
- Delete Constraint
- Check Constraint

New Simple Constraint: This is for creating a new type of simple constraint. On selecting this option the dialog box in Figure 12 appears for taking the user's input.

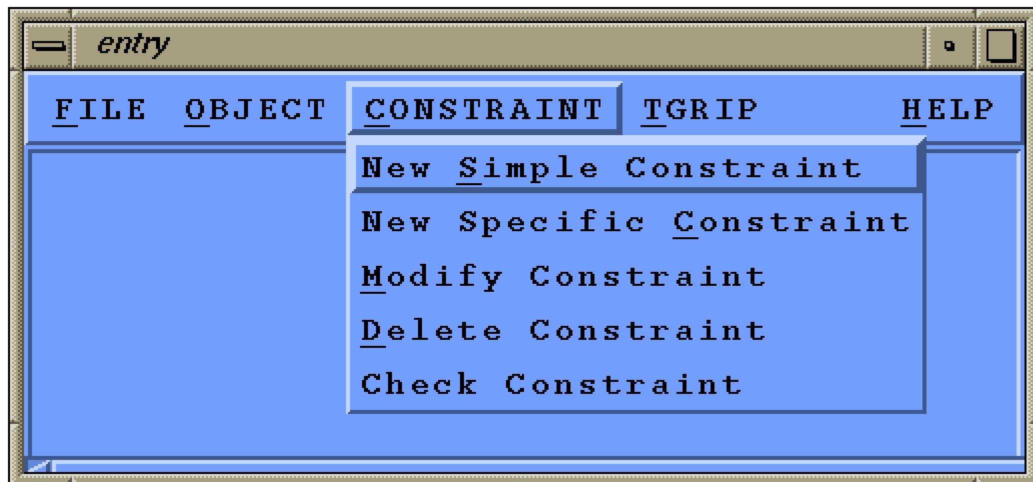

Figure 11: Constraint Menu

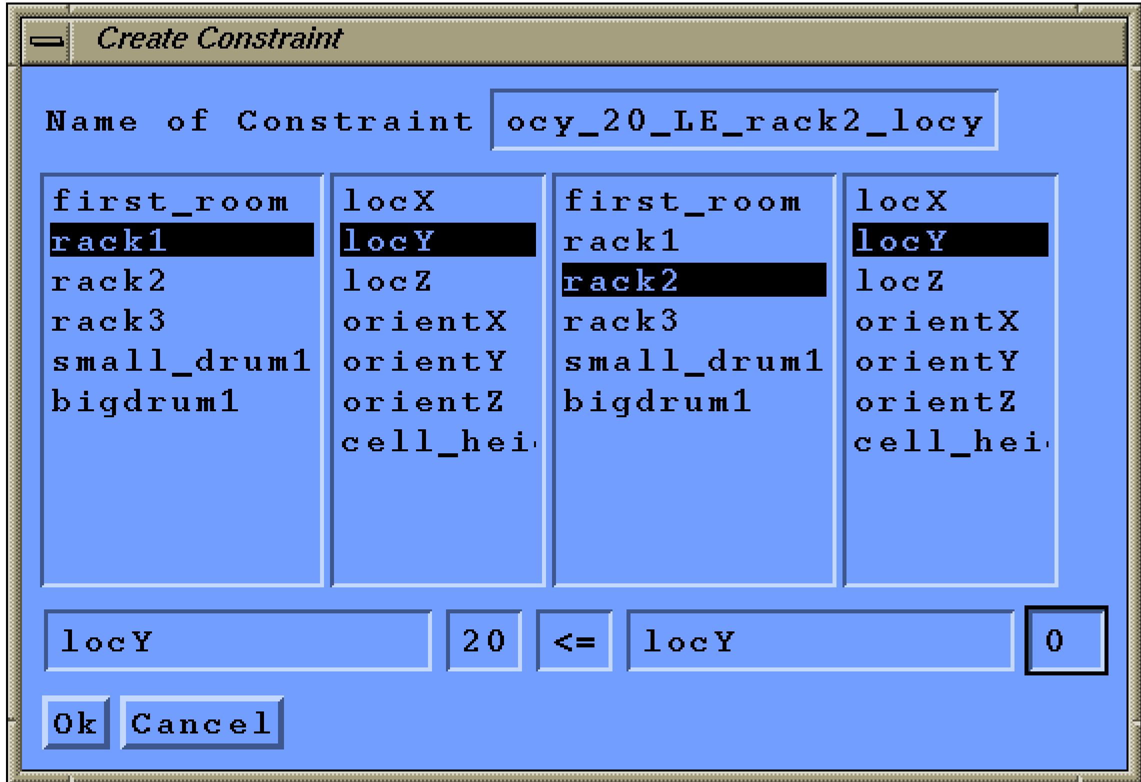

Figure 12: Creation of a Constraint

Figure 12 shows the creation of a constraint, which represents the relation that the minimum distance between `rack1` and `rack2` is 20. For creating this, the following steps have to be followed.

- Enter the name of the constraint.
- Double click `rack1` (all the attributes of rack1 will appear in the second window).
- Select locY attribute for `rack1` (it will appear in the text box below the window).
- Double click on `rack2` in the third window (all the attributes of `rack2` will appear in the fourth window).
- Select locY attribute of `rack2` (this will appear in the text box below the window).
- Enter the values of tolerances and the relation operator as '<=' and 20.
- Press 'ok' button.

New Specific Constraint: This option is for creating a specific constraint (complex constraints). All such constraints are sub – classes of the super class 'Constraint'. The C++ sub-class corresponding to the constraint should be written by the user as per the template provided in the Appendix – A. All sub-classes pertaining to constraints should be added in the directory 'subclass/constraint'. When the user selects this option, the dialog box in Figure 13 appears.

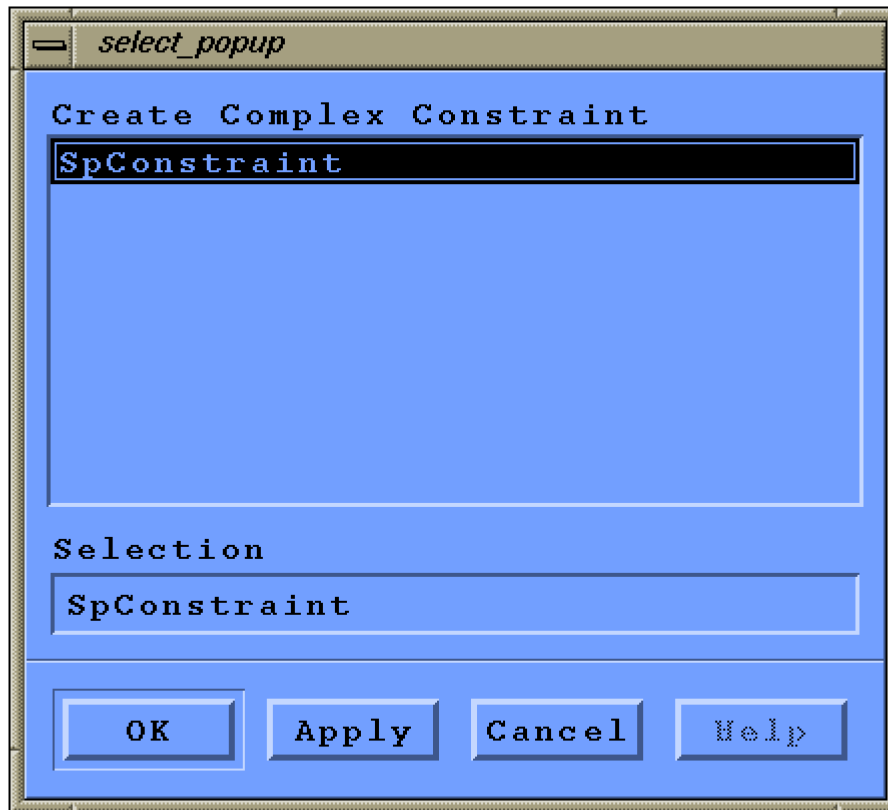

Figure 13: Subclass/Constraint Diaglog Box

The user has to select the constraint and press 'ok' to create the constraint.

Delete Constraint: This option is for deleting a particular constraint in the currently open vault. On selecting this option the following dialog box appears.

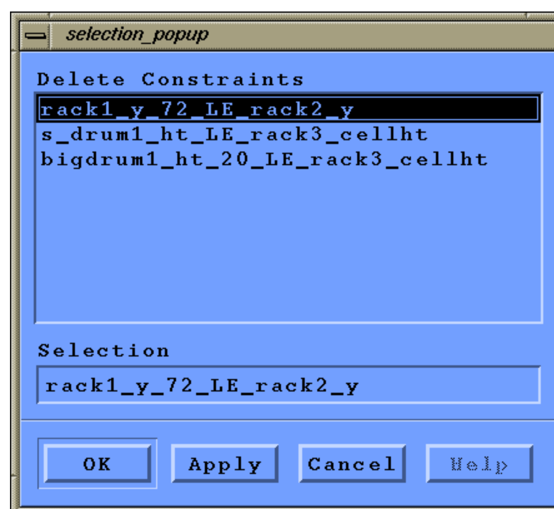

Figure 14: Delete Constraint Dialog Box

When a particular constraint is selected and 'ok' is pressed, the constraint is deleted from the vault.

Modify Constraint: This option is for modifying/viewing a particular constraint. When this option is selected the dialog box shown in Figure 15 appears, which shows the list of constraints present in the currently opened vault.

When a particular constraint is selected and 'ok' is depressed, the dialog box in Figure 16 appears for modifying or viewing the constraint.

After making the required changes to the constraint, when ok is pressed, the constraint gets modified.

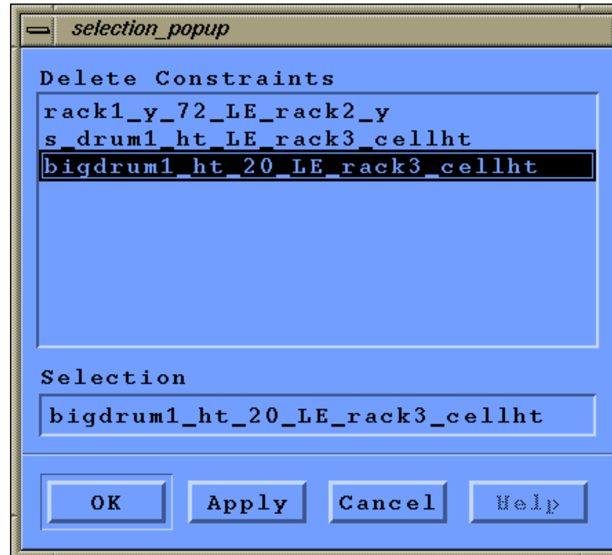

Figure 15: Constraints Present in a Currently Opened Vault

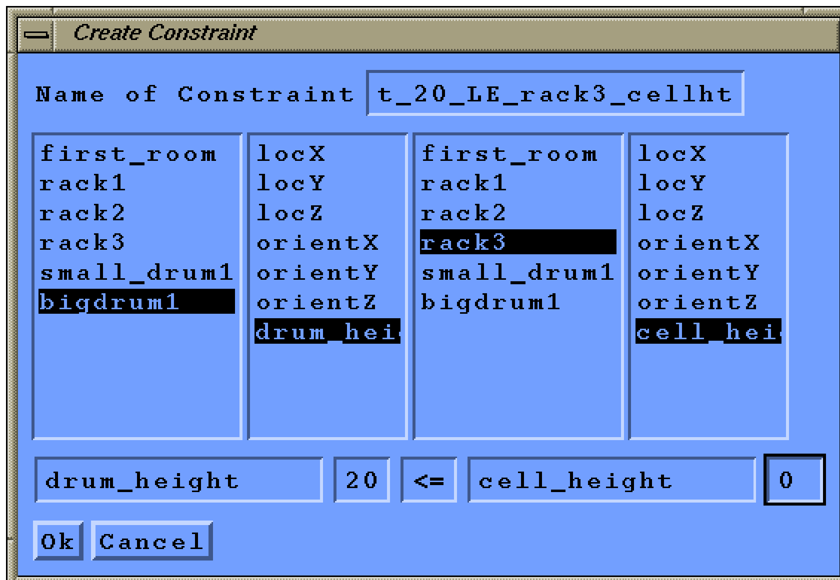

Figure 16: Dialog Box for Modifying or Viewing a Constraint

Check Constraint: Constraints can be checked at any time while composing a model, by invoking 'Check Constraint'. When invoked, the user is displayed the warning messages regarding the constraints which are not satisfied. On pressing 'ok' the next constraint is checked and a warning message is displayed if the constraint is not satisfied.
The dialog box in Figure 17 shows a warning message being displayed to user.

Tgrip Menu: The Telegrip menu shown in Figure 18 lists available options. **CLI**: For sending CLI commands to Telegrip.

Transfer to Tgrip: For transferring the control to Telegrip.

CLI: This is for sending a cli command to the Telegrip. This may be helpful to user for providing some information about the details of model such as how many objects are present and their position/orientation. When this option is selected, the following dialog box appears.

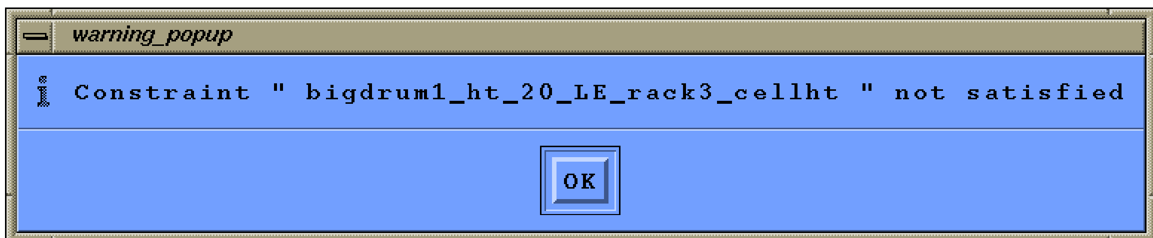

Figure 17: Warning Message Dialog Box

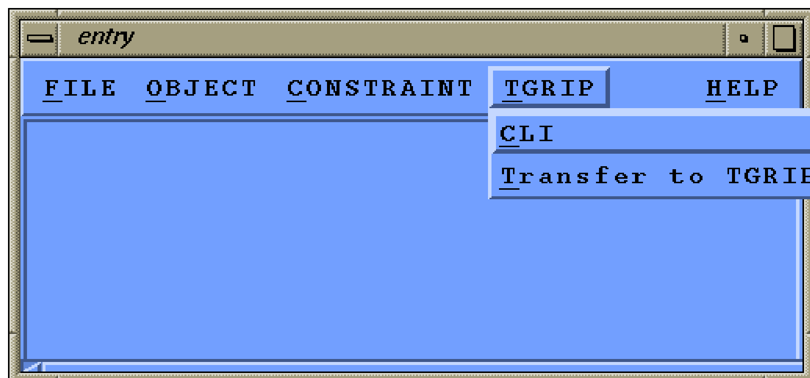

Figure 18: Tgrip Menu

Figure 19: CLI Dialog Box

When a command is entered and 'ok' is pressed, the command is executed and the Telegrip's reply is displayed in the Text area of the main window.

Transfer to TGRIP: While composing a model, if there is a need to change views or position/orientation of any object, the control can be transferred to Telegrip's menu. After making the required changes, the control can be transferred back to SSCS menu by exiting from Telegrip's menu.

This page intentionally left blank.

Appendix A

Procedure for Adding a Sub-Class for Objects and Constraints

This page intentionally left blank.

Appendix – A

This section explains the procedure for adding a sub-class for objects and constraints. All sub-classes for objects should be added in the directory 'subclass/object.' The template for writing the C++ sub-class (named SomeClass.C) has been provided in the following text. The sub-class can be created by making changes to the template. The necessary instructions are provided as comments in the respective templates. After the sub-class has been created, it should be compiled by typing 'make' in 'subclass/object' directory.

The sub-classes for constraints should be added in the directory 'subclass/constraint'. The template for writing the constraint sub-class 'SomeConstraint.C' has been provided in the following text. Again the sub-class should be compiled by typing 'make' in 'subclass/constraint' directory.

```
// Filename : SomeClass.C


// Template for the object subclass
// We will refer to the subclass as "SomeClass"
// For writing a class from this template "SomeClass" has to be replaced
// by the actual class name. The file name should also be changed
// to the class name. Any class deriving from this class should be added
// in the directory /subclass/object


#include <iostream.h>
#include "../../objectSuper.h"

// This class inherits from class ObjectBase which is the super class

class SomeClass : public ObjectBase
{
	// declare the output operator as a friend
	friend ostream& operator<<(ostream&, SomeClass *);
	private:
	// Write all the attributes here. This class will inherit the attributes
	// location and orientation from the super class. All other
	// attributes should be declared here. In this template we will declare
	// the attributes as attribute1, attribute2, attribute3

		int attribute1;
		int attribute2;
		int attribute3;
	public:
		SomeClass()         // default constructor
		{
			printf("SomeClass created\n");
		}

		SomeClass(ObjectBase& obj) // takes reference to the base class as
			: ObjectBase(obj) // argument and instantiates the object
		{
		// We assign the values to the attributes here.
		// In the template we have assigned values of 10 to all the
		// attributes
			attribute1 = 10;
			attribute2 = 10;
			attribute3 = 10;

		}
		virtual ~SomeClass()       // Destructor
		{
		}
```

```
            //This is for identifying the class type to make sure that the
            //correct class is getting loaded
            virtual void Show()
            {
                  printf("Show: I am SomeClass\n");
            }

            void Store(ostream& os) {  // This is for saving the class
                  os << this;  // It should write its attributes to output
            }

// Other Methods
            ObjectBase *Rebuild(int , string[]);
            string GetTypeObject(void);
            string GetGeoModel(void);
            string GetNameObject(void);
            string *GetAttributeName(void);
            int GetAttribVal(string);
            int GetAttributeNum(void);

};

// Factory class which is required for the dynamic loading of this class
// This is a sub-class of the ObjectFactory class

class SomeClassFactory : public ObjectFactory
{
      public:
            SomeClassFactory()        // Default Constructor
            {
                  printf("SomeClass Factory Created\n");
            }

            ~SomeClassFactory()       // Destructor
            {
            }

            virtual ObjectBase *CreateObject() // THis method is called while
            {                                 // dynamic loading
                  return new SomeClass;      // It returns a SomeClass object
            }
};


// The "C" linkage factory0() function creates the SomeClass class
// for this library


extern "C" void * factory0(void)
{
            return new SomeClassFactory;
}

// This method return the type of object
// THe type of object is same as the Telegrip file name
// Let us say the Telegrip filename for this object is "some"

string SomeClass::GetTypeObject(void)
{
            return "some";
}

// This Mehtod returns the full path of the Telegrip geometric
```

```
// model of the object that this class represents
// If the full path is /usr/sscs/library/PARTS/

string SomeClass::GetGeoModel(void)
{
              return "/usr/sscs/library/PARTS/";
}

// This method returns the name of the object

string SomeClass::GetNameObject(void)
{
              return nameObject;
}

// This method is for reconstructing the object
// When a saved vault is opened, this method is called which
// rebuilds the object and adds it to the list of objects in Vault

ObjectBase *SomeClass::Rebuild(int number, string list[])
{
              // Create a base class object
              ObjectBase *obj = new ObjectBase(list[0],list[1],list[2],atoi(list[3]),
                     atoi(list[4]) ,atoi(list[5]), atoi(list[6]),atoi(list[7]),
                     atoi(list[8]));
              // Instantiate this class
              SomeClass *class = new SomeClass(*obj);
              ObjectBase *retObj;
              // assign object of this class to a base class object and return
              retObj = class;
              return retObj;
}

// This method returns a list of names of all the attributes of this
// class the number of attributes in this case are 3(location)+3(orientation)
// +3 (of it own) so total is 9

string *SomeClass::GetAttributeName(void)
{
       // Allocate memory for the names
       static string attribs[9];
       for (int i=0; i<9; i++) {
              attribs[i] = new char[1024];
       }
// Copy the common attributes
       strcpy(attribs[0],"locX");
       strcpy(attribs[1],"locY");
       strcpy(attribs[2],"locZ");
       strcpy(attribs[3],"orientX");
       strcpy(attribs[4],"orientY");
       strcpy(attribs[5],"orientZ");
// Now copy the attributes of its own
       strcpy(attribs[6],"attribute1");
       strcpy(attribs[7],"attribute2");
       strcpy(attribs[8],"attribute3");

       return attribs;
}

// Given a string this method return the value of the attribute

int SomeClass::GetAttribVal(string attrib)
{
```

```
// Check for common attributes
       if(strcmp(attrib,"locX")==0)
              return locX;
       else if(strcmp(attrib,"locY")==0)
              return locY;
       else if (strcmp(attrib,"locZ")==0)
              return locZ;
       else if (strcmp(attrib,"orientX")==0)
              return orientX;
       else if(strcmp(attrib,"orientY")==0)
              return orientY;
       else if(strcmp(attrib,"orientZ")==0)
              return orientZ;
// Check for its own attributes and return the values

       else if(strcmp(attrib,"attribute1")==0)
              return attribute1;
       else if(strcmp(attrib,"attribute2")==0)
              return attribute2;
       else if(strcmp(attrib,"attribute3")==0)
              return attribute3;

// in case there is an error, return a large value
       return 9999999;
}

// This method return the number of attributes(common and its own)

int SomeClass::GetAttributeNum(void)
{
       return 9;
}

// THis is for writing the object to the output, which is to a file in
// this case. Only the name of the class needs to be modified here

ostream& operator<<(ostream &os, SomeClass *obj) {

       os << "<" << "Object.SomeClass." << " >" << "<"
     <<  obj->GetTypeObject() << " ,"
      <<  obj->GetGeoModel()   << " ,"
      <<  obj->GetNameObject() << " ,"
      <<  obj->GetLocX() << " ," << obj->GetLocY() << " ," <<
                    obj->GetLocZ() << " ," <<  obj->GetOrientX() << " ," <<
                    obj->GetOrientY() << " ," << obj->GetOrientZ()  << " ," << endl;

       return os;
}
```

```
// Filename SomeConstraint.C

// Template for a user defined constraint
// We will refer to the subclass as "SomeConstraint"
```

```
// For writing a class from this template "SomeConstraint" should
// be replaced by the actual class name(which is the name of the constraint)
// The file name should also be same as the class name
// Any class deriving from this class should be added in
// the directory /subclass/constraint


#include <iostream.h>
#include "../../objectSuper.h"
#include "../../SscsVault.h"

extern Vault *thisVault;

// This is a subclass of the class Constraint

class SomeConstraint : public Constraint
{
	// Declare the output operator as a friend of the class
	// so that it can access its attributes
	friend ostream& operator<<(ostream&, SomeConstraint *);
	private:

	public:
		SomeConstraint()		// Default constructor
		{
			printf("SomeConstraint created \n");
		}
		// This constructor instantaites the base class and then
		// forms the sub class
		SomeConstraint(Constraint& con) : Constraint(con)
		{
			printf("SomeConstraint created \n");
		}
		virtual ~SomeConstraint()  // Destructor
		{
		}
		virtual void Show() // Just for debugging to make sure that
		{		    // correct class is getting loaded
			printf("Con Show: SomeConstraint\n");
		}
		void Store(ostream& os)		// for storing the constraint
		{				// in a file(when vault is saves)
			os << this;
		}
		Constraint *ReturnConstraint(void)
		{
			return this;
		}
// Other method declarations
		Constraint *Rebuild(int, string[]);
		string GetNameConstraint(void);
		int GetNumArgConstraint(void);
		int IsSimple(void);
		int Check(Vault *);
};

// This is the factory class which is required for the
//	dynamic loading of this constraint class
// THis is a sub-class of the class ConstraintFactory

class SomeConstraintFactory : public ConstraintFactory
{
	public:
```

```
        SomeConstraintFactory()          // Default constructor
        {
              printf("SomeConstraint Created\n");
        }
        ~SpConstraintFactory()           // Destructor
        {
        }
        // Return an instance of SomeConstraint
        virtual Constraint *CreateConstraint()
        {
              return new SomeConstraint;
        }
};


// the C linkage factory create the class for the library


extern "C" void * factory0(void)
{
      return new SomeConstraintFactory;
}
// THis method is for reconstructing the constraint object
// Let us say we have a constraint which expresses relation
// between three atrubutes. Let these object.attributes be
// object1.attribute1, object2.attribute2, object3.attribute3

Constraint *SomeConstraint::Rebuild(int num,string list[])
{
      assert(num == 1);
      assert(list[0] !=NULL);
      static string *sp_obj_list = (string *) malloc(10);
      static string *sp_att_list = (string *) malloc(10);
      static int *sp_tol_list = (int *) malloc(10);
      sp_obj_list[0] = CopyString("object1");
      sp_obj_list[1] = CopyString("object2");
      sp_obj_list[2] = CopyString("object3");
      sp_att_list[0] = CopyString("attribute1");
      sp_att_list[1] = CopyString("attribute2");
      sp_att_list[2] = CopyString("attribute3");
      sp_tol_list[0] = 0;       // tolerance can also be defined if required
      sp_tol_list[1] = 0;
      sp_tol_list[2] = 0;

// construct a constraint object
      Constraint *con_str = new Constraint("name_of_the_constraint",3,sp_obj_list,
            sp_att_list, sp_tol_list, "S");

// construct a SomeConstraint object
// assign the SomeConsrtaint object to a Constraint object and return it.

      SomeConstraint *new_con = new SomeConstraint(*con_str);
      Constraint *retCon = new_con;
      return retCon;
}

// Return the name of the constraint
string SomeConstraint::GetNameConstraint(void)
{
      return constrName;
}

// return the number of arguments
```

```
int SomeConstraint::GetNumArgConstraint(void)
{
	return numArgs;
}

// This method returns 1 if it is a simple constraint 0 otherwise
int SomeConstraint::IsSimple(void)
{
	return 0;
}

// This is the check method which checks the constraint
// it returns 1 if the constraint is satisfied and 0 otherwise
int SomeConstraint::Check(Vault *temp)
{
	int rhs, lhs, val1, val2, val3;
	ObjectBase *oop1 = temp->returnObjRef(constrObj[0]);
	ObjectBase *oop2 = temp->returnObjRef(constrObj[1]);
	ObjectBase *oop3 = temp->returnObjRef(constrObj[2]);
// if there are more than 3 attributes which have to be related
// then add them here
	val1 = oop1->GetAttribVal(attrib[0]);
	val2 = oop2->GetAttribVal(attrib[1]);
	val3 = oop3->GetAttribVal(attrib[2]);
// This is the check that has to be performed on the different
// values of the attributes, we have just demonstrated a simple example
	if((val1 - val2) > val3)
		return 1;	// return 1 if the constraint is satisfied
	return 0;		// otherwise return 0
}


// This is for writing the constraint to the output operator
ostream & operator<<(ostream &os, SomeConstraint *con)
{
	os << "<" << "Constraint.SomeConstraint." << " >" << endl;
}
```