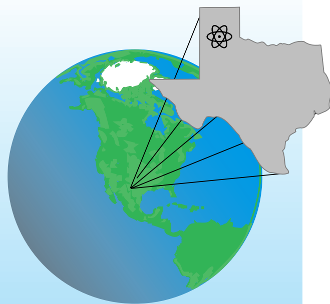

ANRCP-1999-4
February 1999

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

Basic Architecture and Design of Storage Simulation Construction Set

Anil Sehgal and Richard Volz
Department of Computer Science
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.

ANRCP-1999-4

AMARILLO NATIONAL RESOURCE CENTER FOR PLUTONIUM/
A HIGHER EDUCATION CONSORTIUM

A Report on the

Basic Architecture and Design of 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

February 1999

Basic Architecture and Design of Storage Simulation Construction Set

Anil Sehgal and Dr. Richard Volz
Department of Computer Science

Abstract

Regular practice these days has been to build a simulation prototype of an actual entity needing to be built prior to building an actual physical model. Computer simulation and modeling techniques aid greatly in this practice.

In a critical system such as a fissile material storage system, various issues such as safety, security, and automation of handling processes can be assessed through virtual environments, thereby creating a safe atmosphere for research versus hands-on experimentation and possible worker exposures.

By utilizing the approaches presented in this study, the concepts herein can be applied for rapid virtual prototyping of other systems, as this type of system is not limited to the storage of nuclear materials.

TABLE OF CONTENTS

1. INTRODUCTION....1
1.1 SSCS AND ITS CAPABILITIES....1
1.2 EXTENDING SSCS CONCEPT....2
1.3 OVERVIEW OF THE REPORT....2

2. BASIC STRUCTURE OF SSCS....3
2.1 SIMULATION....3
2.1.1 Relationship Between Objects....3

2.2 SSCS – AN OVERVIEW....3

2.3 INTRODUCTION TO TELEGRIP....5
2.3.1 Terminology....5
2.3.2 Geometric Database....5

2.4 OVERALL STRUCTURE OF SSCS....5

3. TECHNICAL APPROACH – SSCS DESIGN 2....7
3.1 OBJECT-ORIENTED DESIGN....7
3.2 APPLICATION OF OO CONCEPTS TO STORAGE SIMULATION....7
3.2.1 Adding Sub-Classes to SSCS....9

3.3 RELATIONSHIPS AND CONTRAINTS....10
3.3.1 Bound Constraints....11
3.3.2 Functional Constraints....13
3.3.3 Adding Sub-Classes....13

3.4 CLASS VAULT....13
3.4.1 Instance of Object Class....15
3.4.2 Instance of Class Constraint....15
3.4.3 Instance of Class Vault....16

4. DESIGN AND ARCHITECTURE....19
4.1 INTERNAL STRUCTURE OF SSCS....19
4.2 DATA FLOW....20
4.2.1 Adding Geometric Models to *Telegrip* Library....20
4.2.2 Adding Sub-Classes to SSCS....20
4.2.3 Creating Simulations through SSCS....21
4.2.4 Checking Constraints....23

4.3 INTERFACE WITH TELEGRIP....24

LIST OF TABLES

Table 1: Super Class 8

Table 2: Class Constraint 11

Table 3: Simplified Constraint 11

Table 4: Represented Constraint 12

Table 5: Negative Tolerance 12

Table 6: Positive Tolerance 12

Table 7: General Form of Bound Constraint 12

Table 8: Class Vault 14

Table 9: Instance of Sub-Class Room 16

Table 10: Instance of Bound Constraint 16

Table 11: Class Newdrum 21

LIST OF FIGURES

Figure 1: Overall Structure of SSCS 5

Figure 2: Class Hierarchy Diagram for SSCS 8

Figure 3: Class Hierarchy for Class "Constraint" 10

Figure 4: Instance of Class Vault 17

Figure 5: SSCS Internal Structure 19

Figure 6: Data Flow in SSCS 22

Figure 7: Data Flow for Constraint Checking 23

1. INTRODUCTION

It has become a regular practice these days to build a simulation prototype of the actual entity to be built, before building a physical model. Building a physical model takes substantial time and cannot be easily modified once built. Computer simulation and modeling techniques provide both flexibility and speed of development in such cases. Virtual prototyping rather than physical prototyping are carried out using 3-D graphical representation of the prototype. The virtual prototype becomes a common platform on which multi-engineering discipline design evaluation and integration can be carried out. The ability to rapidly generate multiple iterations of the design alternatives helps in achieving the optimum design configuration and benefits the user by minimizing engineering costs and human effort.

Virtual environments allow comprehensive analysis and reconfiguration "on the fly" to rapidly evaluate multiple "what if" scenarios. The virtual environment replicates the impact of visual and tactile feedback without the cost of developing a physical prototype. A large storage system for fissile material is a critical system which require virtual prototyping of related automated processes and simulation of mechanical devices including robots. This requires the ability to develop virtual programmable prototypes of robots, which can be programmed like an actual robot and provide feedback in an interactive manner providing a realistic visualization of the automated process.

For developing simulation of systems, such as one for storing fissile material, there are a wide variety of software simulations available. *Telegrip* by Deneb Robotics Inc., is one of such commercial software, which is widely used in industry as a tool for virtual prototyping and simulation of robotic operations. Developing simulations with such software takes substantial time and requires trained personnel. Due to the criticality of fissile material storage system the design iterations are inevitable. For such systems it is desirable to develop and modify simulations with the ongoing design changes. Storage Simulation Construction Set (SSCS) is a tool, which is intended to speed up the process of developing storage system simulations, minimizing the time and effort required.

1.1 SSCS AND ITS CAPABILITIES

In a critical system such as fissile material storage system, there are various issues involved such as safety, security, minimizing human exposure and automating the handling processes. It thus requires multi-engineering design inputs. Speed of development of virtual prototypes of such a storage system and simulation of associated robotic operations thus become important to facilitate rapid, cost effective design iterations. It becomes necessary to develop capabilities for rapid development of virtual prototypes of storage system and associated robotic operations. SSCS is such a capability, which provides tools for rapid virtual prototyping of storage system with *Telegrip*. The models can be modified during different design iterations and the effect of the change can be studied.

The SSCS provides the user with capabilities for developing storage system simulations with the following objectives:

- Ease of development of different storage concepts.
- Reduction in the development time for modeling and simulation of fissile material storage system.
- Minimizing the training and knowledge required for building simulations.

These objectives are achieved by adopting the following approaches:

(a) Creating generic geometric models of storage system components.

(b) Creating geometric models of actual objects from generic geometric models.

(c) Easy assembly of these geometric models.

1.2 EXTENDING THE SSCS CONCEPT

The concept of SSCS is not limited to the storage of nuclear material or in general to a storage system as such, but it can be applied for rapid virtual prototyping of other similar systems also. By following approaches described in the rest of the report and building an application program similar to SSCS, rapid virtual prototyping of systems similar to a storage system can be carried out, achieving speed of development and reduction in the overall development time.

1.3 OVERVIEW OF THE REPORT

The report discusses the architecture for implementation of SSCS. Chapter 2 discusses the basic structure of the SSCS and its interaction with *Telegrip*. Chapter 3 discusses the design of SSCS, and finally Chapter 4 presents the architecture.

2. BASIC STRUCTURE OF SSCS

Before discussing the actual structure of SSCS some concepts of simulation are discussed in Section 2.1. Section 2.2 gives an overview of SSCS, Section 2.3 presents an overview of *Telegrip* software, and Section 2.4 gives the overall structure of SSCS.

2.1 SIMULATION

A system, such as a storage system, consists of a number of components such as storage racks, material handling equipment, etc. These components have a definite relationship with each other. For building simulation of such a system, the graphical models of the components are built and assembled together. To ensure that the simulation is a correct representation of the physical system, the geometric models of components should be assembled in a way, such that they satisfy the relationship(s) the actual physical components have. Since a graphical model lacks the physical properties such as weight, strength etc., we only consider certain kinds of relationships, such as the positional relationships.

2.1.1 Relationship between Objects

The physical objects composing a system have relationship(s) with each other and thus they exist under certain constraints. To enforce the relationship(s) on the geometric models, we need a mechanism to define the relationship and ensure that they are satisfied. This will help the user to ensure that the simulation is a correct representation of the physical system. To achieve this, properties or attributes are attached to the set of geometric models (the set of attributes describe the characteristics of the model) and constraints are created which represent the relationship between the attributes of the objects. These constraints are expressed in a separate set of constraint objects. To further understand this let us take an example of building a simulation of a table and a block. If it is required that the block lies on top of table, then we construct the geometric models of table and block. These geometric models have attributes such as size and location. To model the situation in which "the block always lies on the table," we bind the value of the location of the block (which is an attribute of the geometric model of block) within some upper/lower limits (which may depend upon the size and location of the table), and enforce the relationship.

An object can have multiple relationships with multiple other objects and therefore, it can have more than one constraint either in absolute value of its attributes or relative to the attributes of various other objects.

2.2 SSCS: AN OVERVIEW

In SSCS, *Telegrip* is used to create a library of graphical models of generic storage system components. The SSCS then incorporates each of these generic components in an object class in the sense of object-oriented programming. These generic components are referred to as 'object types.' Each SSCS object class has the following characteristics:

(a) A graphical model (stored in *Telegrip* library) of the physical object it represents.
(b) A set of attributes, as introduced earlier, describes the characteristics of an object e.g. size and location (with respect to a reference coordinate frame) of an object.
(c) A set of methods that can be invoked. Methods are the operations that can be performed on the objects. By invoking methods we can do the following:
    - Create instance of an object.
    - Modify one or more attributes of an object.
    - Delete instance of an object.

Each method requires some input parameters and performs the required action, e.g., method 'create' will take attributes of the object as its input, and will create an instance of the object with the required attributes.

To enforce the relationships among the actual physical objects, the attributes of the geometric models are constrained. These constraints are represented by a separate class known as 'constraint.' While building a simulation, class constraint can be instantiated to create various types of constraints, which define the relationship(s) between geometric models present in the simulation. These constraints form a list, known as 'constraint list,' which is associated with a particular simulation. At any point of time, new constraints can be added to the list, or some of the constraints present in the list can be deleted. These constraints can be checked to ensure that the relationship between objects is maintained. For example, if we take a simulation of a room and the door of the room, the user can define a constraint that 'height of door should be less than height of room,' and check it when appropriate to ensure that the condition is true.

In SSCS, there will be a set of predefined object classes and the corresponding models in *Telegrip's* library. The user can populate the library by creating geometric models and adding corresponding class to SSCS. To create a specific storage system, called a Vault, the SSCS allows the user to create specific instances of objects present in the library, establish their attributes such as location and size and form a constraint list associated with each storage system model. At any point while composing a simulation the user can perform "constraint checking," to ensure that the set of constraints associated with the storage system are being satisfied.

SSCS also allows the user to create devices (e.g., mobile robot), define the motions that are associated with their use in the system, simulate their operation and allow the user to view the system from user-determined perspective.

The geometric model of a device in *Telegrip* is composed of a number of parts joined together. These parts can move in a particular manner with respect to each other, under certain constraints, e.g., one link joined to another link via a rotary joint. Each part of the device thus has specific characteristics such as type (translation/rotation), limits of movement, etc. Such devices can be controlled in a number of ways, e.g., by defining a series of joint values, by defining the end effector's position/orientation, etc. When a device is built, the *Telegrip* stores all the information relating to the various parts, their types and characteristics, etc., and uses it for controlling the movements of the device.

The SSCS deals with devices at a higher level and provides various ways of controlling the movements of a device, e.g., moving individual joint angles, moving end-effector to certain position/orientation, etc. With such capabilities for effectively controlling the movements of a device, various operations can easily be simulated. By simulating different types of devices to carry out an operation the user can study the performance of each type of device. The simulations of different devices performing an operation can be saved and rerun at a later date, allowing the user to compare and evaluate various modes of operations.

The simulations can be viewed from different perspectives by changing the user viewpoint in a number of ways such as zoom-in and zoom-out, changing angles of view, etc. These changes in views can be combined with the series of device motions and recorded to form a running simulation. At any time while composing a storage system simulation, the partial composition can be saved in a user named file and retrieved afterwards for modification.

2.3 INTRODUCTION TO TELEGRIP

We now introduce *Telegrip* software and discuss its terminology, which will be used while discussing SSCS. *Telegrip* is a commercial robotic simulation and control software by Deneb Robotics, Inc., which provides an interactive, 3D graphic simulation tool for design, evaluation and analysis. Automated processes can be constructed, programmed, and analyzed for collision checks, motion constraints, etc. SSCS uses *Telegrip* for creating and storing geometric models and composing storage system simulations.

2.3.1 Terminology

Subobject: A Subobject is an arbitrary collection of Polygons, Lines, Surfaces, Curves and Text.

Object: Any basic single geometric identity is called as an Object. It consists of one or more Subobjects. A simple example of Object is a block.

Device: A Device is formed by attaching a number of Objects together and defining the relationship between them. The different objects forming a Device are called as its Parts. A robot is a typical example of a Device.

Workcell: A Workcell represents a complete model. It consists of one or more Objects, Devices etc.

Coorsys: A Coorsys represent a three - dimensional point in space along with orientation. Each Object and Device have their Coorsys called Base Coorsys. The World Coorsys refer to the reference Coorsys in a Workcell with respect to which all the Devices and Objects are referenced.

2.3.2 Geometric Database

The information pertaining to a **WORKCELL** along with associated Devices, Objects, etc., are stored in *Telegrip* in the form of an internal geometric data structure. This is a hierarchical data structure in the following order.

Device -> Part -> Object -> SubObject -> Polygons -> Edges -> Lines -> Points etc.

Each device is stored as an n-way tree where each part in the tree can have zero or more child Parts. Every Device has at least one part. Every part in a tree contains a transformation matrix that defines its position and orientation with respect to its parent part. Each part refers to a single geometric Object that is comprised of one or more Subobjects, curves, surfaces, Coorsys, etc. Each Subobject contains zero or more other Subobjects, Points, Lines, Edges, and Polygons. There are one or more coorsys associated with each SubObject, Object, Part, and Device.

2.4 OVERALL STRUCTURE OF SSCS

The overall structure of SSCS is shown in Figure 1.

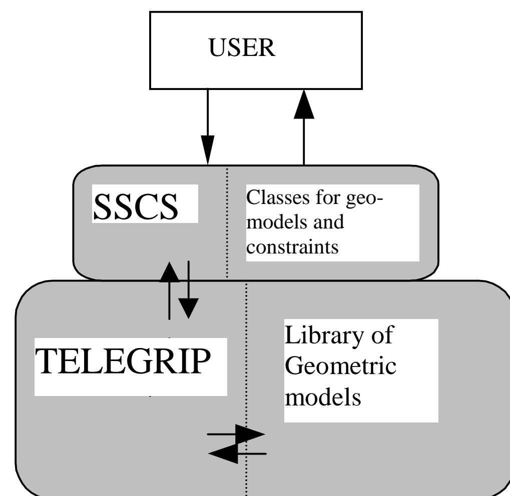

Figure 1: Overall Structure of SSCS

The SSCS is an application program built on top of *Telegrip*. It mainly consists of:

(a) Reusable library of geometric models of storage system components.

(b) User Interface and control program for:

- Getting user input, creating storage system simulation in *Telegrip*, and checking constraints associated with various geometric models present in simulation.
- Creating object classes for geometric models and constraints.
- Creating models of mechanical devices and controlling their motion for simulating an operation.

SSCS aids the user in building storage system simulation, utilizing the library of geometric models, defining the constraints related to these components for a storage system, checking these constraints on user demand, and notifying the user in case of non-conformity.

3. TECHNICAL APPROACH - SSCS DESIGN

The design of SSCS is carried out as per the principles of Object Oriented software design. The relevant basic concepts of Object Oriented design are introduced in Section 3.1. The application of these concepts to the SSCS is discussed in Section 3.2. Section 3.3 discusses Relationships and Constraints, Section 3.4 explains the structure of Class Vault, and finally Section 3.5 gives some advantages of using this design for SSCS.

3.1 OBJECT-ORIENTED DESIGN

In the past decade, the Object Oriented approach to Software Development has become increasingly popular, due to the inherent benefits underlying the approach, especially the fact that object technologies can lead to faster Software Development, higher quality programs and greater interoperability.

The Object Oriented Software Design paradigm exhibits the data and procedural abstractions that lead to effective modularity. A "class" is an Object oriented concept that encapsulates the data and the procedural abstractions that are required to describe the content and behavior of some real world entity (Object). The data abstractions or the properties, which describe the object of a class, are referred to as the attributes of the class. The procedural abstractions or the operations that can be performed on the objects of the class are referred to as the methods of the class. A set of attribute values and a set of methods form an instance of the object belonging to that class.

We have incorporated these object-oriented principals into the SSCS for representation of geometric models and relationship(s) between these models. Each object type is represented by a separate class, with a set of attributes and methods. The relationship(s) between geometric models are represented as instances of a constraint class.

Inheritance is another object-oriented concept that is used in SSCS for defining classes. We define a base class, (also called as the super-class) which has attributes and methods that are common to all the classes. Below this super-class are the derived classes (or sub-classes). Derived classes inherit all the attributes and methods of the super-class. Attributes specific to the derived class can be added to the inherited attributes, and the inherited methods can be redefined. With such class structure the common attributes and methods are defined in the super class and they do not have to be redefined for each derived class. This makes the sub-classes simpler to write and can be easily added by the user.

We apply object-oriented principals to a simulation and represent each of the geometric models by a class. All geometric models have some common attributes such as location and orientation and some common methods such as create and delete. We define a super-class, which represents all the geometric models, and possesses common attributes and methods. We can extend this super-class to represent a particular object such as geometric model of a room. This sub-class (which represents a geometric model of room), will derive attributes and methods from the super-class and it can have its additional attributes such as height, width, length, etc.

3.2 APPLICATION OF OO CONCEPTS TO STORAGE SIMULATION

We have developed a class structure for representing geometric models in a simulation. This structure is shown in Figure 2 and is then explained. In SSCS, a super class represents all the objects. This class has a fixed set of attributes and methods that are shared by all objects, e.g., position and

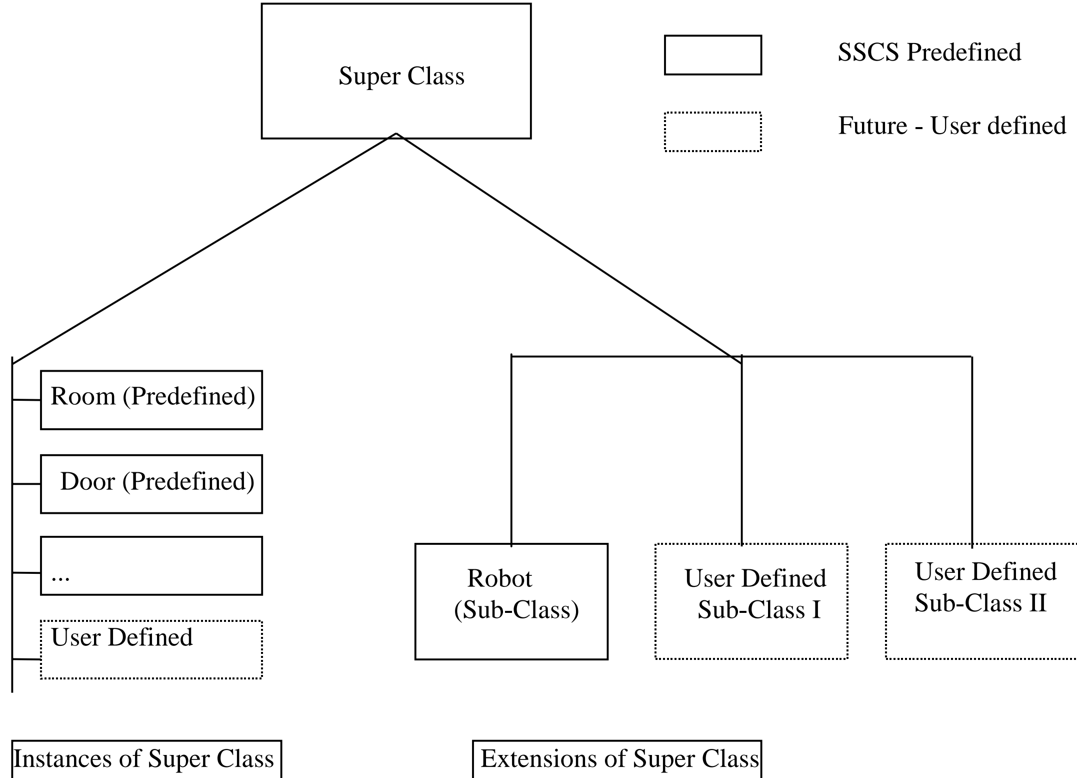

Figure 2: Class Hierarchy Diagram for SSCS

Table 1: Super Class

```
Class Super
Attributes:
Type, Name, Location, Orientation, GeoModel
Methods:
Create, Delete, Modify
```

orientation. A part of this class is shown in Table 1.

The 'Type' attribute represents the object type. The 'Name' is for specifying the name of the object. 'Location' represents the relative location of the object with respect to the world coordinate frame. 'Orientation' is the relative orientation of local frame of the object with respect to the world coordinate frame and 'GeoModel' is the name of the geometric model in *Telegrip's* library.

As shown in Figure 2, an object type can either be an instance of the super-class, or it can be a sub-class that is extension of the super class. The sub-classes (like class 'robot') are an extension of the super-class and have an additional set of attributes.

There will be a set of object types predefined in SSCS, and their corresponding geometric models will be stored in *Telegrip's* library. The object types present in the library can be created in a simulation, by invoking method 'create' for the class representing the object type. The arguments to the method such as the name of the object, its position and orientation, etc., have to given by the user through the SSCS user interface. The execution of method 'create' will result in the instantiation of the class in SSCS and the model of the object will be created in the *Telegrip* **WORKCELL**.

For populating *Telegrip's* library, to add new object types, the following steps are required:

(a) Create the geometric model of the object in *Telegrip*.

(b) Create the associated class in SSCS. This class can be created by either:

- Instantiating super-class (if attributes of the object are the same as that of the super-class), through SSCS menu.

- Extending super-class to form a sub-class, having the additional set of attributes (the methodology and details for adding sub-classes to SSCS is discussed in the following section).

3.2.1 Adding Sub-Classes to SSCS

The user can create additional sub-classes in a predefined directory location. Each sub-class (which is C++ code) shall be defined in a separate file and the name of the file shall be the same as the name of the sub-class it contains. A sub-class template shall be provided to the user, which will provide a basic structure for writing C++ code for sub-classes. Sub-classes will extend super-classes by defining additional attributes or redefining the methods of the super-class.

The new sub-classes defined by the user have to be integrated with SSCS, so that the sub-class can be accessed when the SSCS is running. The user-defined sub-classes shall be compiled as a shared object file and will be dynamically loaded at run-time. The code integration of new sub-classes with SSCS will be carried out as follows:

1. Read the predefined directory for files that contain sub-classes.
2. Get the name of the sub-classes and display it to the user for selection.
3. Get the user selection and load the shared object file, so that the attributes and methods of the sub class can be accessed by the SSCS.

The template for defining sub-class will have the following key methods, which will have to be defined by the user in any sub-class :

(a) `getAttributeNum` which returns the number of attributes in the sub-class.

(b) `getAttributeNames` which returns the names of all the attributes of the sub-class.

(c) `getAttributeRef` which takes the name of an attribute as argument and returns a reference or a pointer to the attribute.

This mechanism provides the seamless integration of the user defined sub-class code with the SSCS code.

We take an example in which the user is adding a new type of drum to the library. The following tasks should be performed.

- Create the geometric model of drum in *Telegrip's* library.

- Add a sub-class for this drum at a predefined directory location. The user can define some attributes specific to the drum (in the sub-class) such as diameter, height etc.

- Compile the sub-class code with the provided 'Makefile,' to form a shared object file.

- Start the SSCS, and this sub-class will be available to the user for using in the simulations.

3.3 RELATIONSHIPS AND CONSTRAINTS

There exists a relationship(s) between various storage system components. These relationships can be enforced on the geometric models by expressing the relationship as set of constraints. These constraints are nothing but a functional relationship between a set of objects.

Because of the nature of the storage system, many of the constraints can be expressed in terms of simple limit tests on the attributes. In SSCS the constraints are implemented as a set of constraint objects. Due to the common nature of most of the constraints, we define a super-class, which can be instantiated to expresses the common type of constraints called as Bound constraints. The super-class is extended to express the other types of constraints known as the Functional constraints. The class hierarchy for implementing constraints is shown in Figure 3, and the types of constraints are explained in the following text. Basically we have divided the constraints into two types; the first ones are simple bound constraints (which are the instances of the super-class) and the second ones are functional constraints (which are the extensions of the super-class). Bound constraints are the ones, which simply constrain the value of an attribute within some upper or lower limits. These limits can be an absolute value or they can be the value of an attribute (of some other object), e.g., 'the

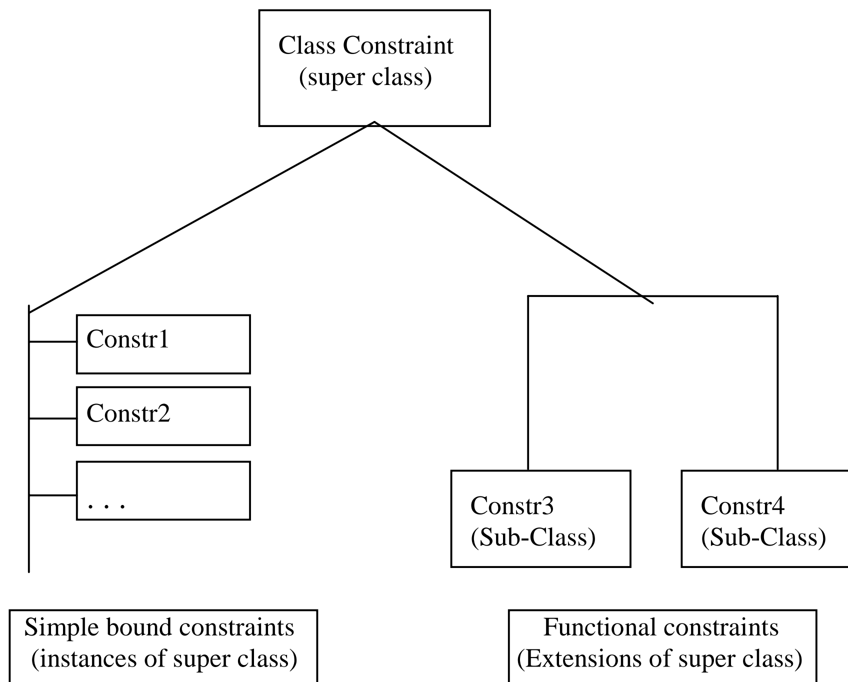

Figure 3: Class Hierarchy for Class 'Constraint'

Table 2: Class Constraint

```
Class Constraint
Attributes: Name, ListAttribRef, ListAttribTol, NumAttribs,
            RelOp
Methods:   create(Name, ListAttribRef, ListAttribTol,
RelOp,
            NumAttribs), delete(Name), check(Name).
```

height of door is less than height of room' is an example of a bound constraint. The functional constraints are the ones in which the bound in the value of an attribute is a function of one or more other values (which may be the attributes of other objects). In other words, "if the height of room is 'x' then the height of door should be 'y' otherwise it should be 'z'," is an example of a functional constraint.

All the constraints are expressed by a class 'constraint' (super-class), which are shown in Table 2. The class constraint is a super-class, which has the following attributes:

- 'Name' which is the name of the constraint.
- 'ListAttributeRef' refers to the list of references (or pointers) to the attributes that participate in the constraint.
- 'ListAttributeTol' refers to the list of tolerance in the value of the attributes
- 'NumAttribs' is the number of attribute references in the list.
- 'RelOp' is the relational operator for a pair of attributes.
-

(The concept of tolerance and RelOp is discussed in the next section).

Method 'create' takes the list of attribute references, their tolerances, and Relational operator as arguments, and creates an instance of the constraint. For creating a constraint it is required that the objects, whose attributes are to be constrained, are present in the model. Method 'delete' is for deleting an instance of an existing constraint, 'check' is a boolean method which gets the values of attribute references in the list, and returns true if the values satisfy the constraint, or otherwise false. These two types of constraints, namely Bound constraints and Functional constraints, are further explained in the following sections.

3.3.1 Bound Constraints

We first consider the case of simple constraints, which are the instances of the super class 'constraint.' The structure of the constraints is explained by first taking a simple form of the constraint and then expanding it to the full form. The simplified form of the constraint is shown in Table 3 in which tolerance is not considered.

Table 3: Simplified Constraint

Field 1	RelOp	Field 2
AttribRef1	LT,GT,EQ,LE,GE	AttribRef2

Field 1 represents a reference to the attribute to be constrained and the Field 2 represents the bound, which can be a reference to an attribute or an absolute number. The 'RelOp' is represented by a string and can be any of the above 5 symbols. Depending upon the value of the RelOp, the mathematical expressions that this constraint will represent are:

LT: `Attribute 1 < Attribute 2`
{Attribute1 should have value less than Attribute2 }

GT: `Attribute 1 > Attribute 2`
{Attribute1 should have value greater than Attribute2 }

LE: `Attribute 1` $\leq$ `Attribute 2` {Attribute1 should have value less than or equal to Attribute2 }

GE: `Attribute 1` $\geq$ `Attribute 2` {Attribute1 should have value greater than or equal to Attribute2 }

EQ: `Attribute 1 = Attribute 2` {Attribute1 should have value equal to the value of Attribute2 }

We take an example of representing a constraint which is expressed as 'width of the cart should be less than the width of the room', and assume that objects 'cart' and 'door' are present in the model and have an attribute 'width'. This constraint can be expressed in Table 4 by making Field 1 as a reference to attribute 'width' of object 'cart,' and Field 2 a reference to attribute 'width' of object 'door'. The value of the RelOp in this case will be 'LT'.

Table 4: Constraint

Field 1	RelOp	Field 2
Cart_widthRef	'LT'	Door-width Ref

This equivalent mathematical expression is shown below.

```
{   cart_width    <
door_width    }
```

Now we extend this for expressing constraints of slightly complex form and show how the concept of Tolerance can be applied. We take an example of constraint 'the minimum clearance between the width of door and width of cart should be 5 units.' In other words we can say that 'the width of cart has to be less than (door-width – 5).' This can be mathematically represented as follows.

```
{  cart_width    <
(door_width  -  5)  }
```

This expression can be expressed by our constraint object in Table 5:

Table 5: Negative Tolerance

Field 1		RelOp	Field 2	
Attribute1	Tol1		Attribute 2	Tol2
cart-width Ref	0	'LT'	door-width Ref	- 5

Since we can convert all negative tolerances into equivalent positive tolerances, we express all the tolerances as positive for simplicity. This is illustrated by expressing the negative tolerance in Table 5 into an equivalent positive tolerance as shown in Table 6:

Table 6: Positive Tolerance

Field 1		RelOp	Field2	
Attribute1	Tol1		Attribute2	Tol2
cart_widthRef	+ 5	'LT'	door_widthRef	0

The equivalent mathematical expression will be `{(cart_width + 5) < door_width }`, which represent exactly similar relationships as the original one.

The general form of the Bound constraint is shown in the following table and expressed in the form of a mathematical equation afterwards.

Table 7: General Form of Bound Constraint

Field 1		RelOp	Field 2	
AttribRef1	Tol1	LT,GT,EQ, LE,GE	AttribRef2	Tol2

Mathematical equation representing the bound constraint.

$$\{Attribute1 + Tol1 \ (<,>,=,\leq,\geq) \ Attribute2 + Tol2 \}$$

Due to the nature of the storage system, most of the relationship between its components can be expressed in the form of one or more constraints of the above types, e.g., constraint 'width of cart should be greater than 10 units

and less than width of door' can be expresses as the following 2 constraints:

- cart_width > 10
- cart_width < door_width

Bound constraints are instances of the super-class. For a particular simulation, such types of constraints can be created by the user through SSCS menu. The user has to provide the names of the attributes, their tolerances and the appropriate relational operator for creating such a constraint. The method 'check' for such constraints, will get the values of attributes with the tolerances, and evaluate the mathematical expression representing the constraint.

3.3.2 Functional Constraints

The functional constraints are those that cannot be expressed in terms of one or more attribute bound constraints. These constraints are represented by sub-classes that are formed by extending the super-class. Each sub-class represents a functional relationship between a set of objects in the simulation. When method 'create' is invoked, it will ask the user to supply the list of objects and their attributes that are required to be constrained. For creating an instance of a constraint, it is necessary that the object whose attributes are to be constrained, are present in the simulation. Each of the sub-classes will have a different method check, which will get the list of attribute values as input, and will evaluate the function.

An example of this type of constraint may be: 'If the height of room is more that 15 units, then height of door should be 7 units, otherwise it should be less than 7 units.' This constraint is represented by a sub-class. The method 'create,' for creating an instance of the constraint in this case, will take arguments as:

Name: A string, which represents the name of the constraint.

ListAttribRef: List of attribute references which in this case will be a reference to attribute 'height' of object 'room' and a reference to attribute 'height' of object 'door'.

NumAttribs: The number of attributes, which is 2 in this example.

Let us take a situation in which the room height is 20 units and the door height is 10 units. If the method 'check' is invoked then it will get the values of participating attributes, which in this case will be 20 for the room height and 10 for the door height. The height of room being greater than 15, the height of the door should be 7 (as per the constraint). Since the height of door is 10, the constraint is not satisfied and the method check will return a false result.

3.3.3 Adding Sub-Classes

The basic procedure for adding a sub-class (to represent a functional constraint) is the same as that discussed before. The main considerations for such classes are:

(a) All sub-classes will be defined in a predefined directory, in a separate file.
(b) The sub-class name and the file name shall be same.
(c) A sub-class template shall be provided to the user, which will provide a basic structure for writing C++ code for sub-classes.
(d) User has to define the method 'check' for each sub-class.

3.4 CLASS VAULT

A class in SSCS known as class vault represents a *Telegrip* **WORKCELL**. A part of this class is shown in Table 8:

Table 8: Class Vault

```
Class Vault
Attributes:
Name, TrgipWorkcell, ListObjRef, ListConstrRef

Methods:
create, delete, addObjRef, delObjRef, addConstrRef,
delConstrRef, checkCons
```

The attributes of the class are:

Name: A string that represents the name of the vault.

TgripWorkcell: A string that represents the path and name of the *Telegrip* workcell (represented by the vault).

ListObjRef: This is the list that contains the reference to the instances of the objects. The graphical models corresponding to these instances will be present in the *Telegrip* workcell.

ListConstrRef: This is the list of references to the instances of constraint objets in which the vault participates.

The methods of the class are described below:

- create (Name, TgripWorkcell): This method takes the name of the vault & the name of *Telegrip* Workcell as arguments and instantiate the class vault. The corresponding workcell (initially empty) is also created in *Telegrip*.

- delete (Name): Method delete takes the name of vault as argument and deletes the instance of the vault. The corresponding workcell is also deleted in *Telegrip*.

- addObjRef (RefObj): This method takes the reference to the instance of object as argument and adds the reference to 'ListObjRef'. The corresponding graphical model (representing the instance of the object) is added to the *Telegrip's* **WORKCELL**.

- delObjRef (RefObj): This method takes the reference to the instance of object as argument and deletes the reference to the instance from the 'ListObjRef.' The corresponding graphical model is deleted from the *Telegrip* workcell.

- addConstrRef (RefCons): This method takes the reference to the instance of constraint object as argument and adds the reference to the list 'ListConstrRef.'

- delConstrRef (RefCons): This method takes the reference to the instance of constraint object as argument and deletes the reference from the list 'ListConstrRef.'

- CheckCons: This method successively invokes method 'check' of the instances of constraint objects which are referred in the list 'ListConstrRef.'

When the user invokes method create of the class vault, he or she has to supply the name of the vault and the name of corresponding *Telegrip* workcell. On invoking method 'create,' an instance of class vault is created (with an empty list for object

and constraint references) and the corresponding **WORKCELL** (which is empty) is started in *Telegrip*. When the user creates an instance of object, its reference is added to the 'ListObjRef' and the corresponding graphical model is created in the *Telegrip* **WORKCELL**. Similarly as the instances of the constraint objects are created, their references are added to 'ListConstrRef.'

The concept of Vault is explained further by explaining the structure of instance of classes representing objects and constraints. An instance of class Vault is then explained.

3.4.1 Instance of Object Class

Each object type has a series of attributes whose values must be provided when an instance is created. We take an example of creating an instance of sub-class room, which represents the geometric model of a room in a simulation. We assume that the sub-class room is defined in SSCS and the corresponding graphical model of the room is present in the *Telegrip's* library. For creating the instance of class 'room,' the user has to invoke method 'create' for the class through the graphic interface of SSCS. On invoking method 'create,' the user is prompted to supply the values of the attributes through the interface. After supplying the values, the instance is created. An example of the instance is shown in Table 9.

In this instance, the values such as "Room1" for Name, (10,10,10) for the location and (0,0,0) for the orientation are supplied by the user. The Location represents the offset of local frame of model of room with respect to the world coordinate frame of the **WORKCELL.** Orientation represents the relative roll, pitch and yaw of the room with respect to the world coordinate frame.

When the instance of classroom is created, a reference to this instance is added to list of object references in the vault and the corresponding graphical model is created in the **WORKCELL** (with the required parameters such as location and orientation).

3.4.2 Instance of Class Constraint

An instance of Bound constraint is shown in Table 10. For creating an instance, the user has to input the following through the graphical interface.

(a) Name of the constraint.

(b) Name of the participating attributes.

(c) Value of tolerance for the attributes.

(d) The number of participating attributes.

(e) Symbol for the relational operator.

These all are passed to the method create of the class constraint and an instance of constraint is created. (Note that although the user supplies the name of the participating attributes, a reference to those attributes are fetched and passed on as arguments to the method create). For creating this constraint the instance of room named 'Room1' should be present in the simulation. The notation 'Room1.W' and 'Room1.L' are actually the references to the attributes of 'Room1.'

Table 9: Instance of Sub-Class Room

Instance of Sub-Class Room	
Type	Room
Name	"Room1"
Length	100
Width	50
Height	20
Location	10, 10, 10
Orientation	0,0,0

Table 10: Instance of Bound Constraint

Instance of Constraint	
Name	'room1 width less than room1 length'
ListAttribRef	(Room1.W, Room1.L)
ListAttribTol	(0,0)
NumAttribs	2
RelOp	'LT'

3.4.3 Instance of Class Vault

When the user starts a new vault, the class vault is instantiated in SSCS and a new **WORKCELL** is started in *Telegrip*. As the user creates the instances of objects, the following actions take place in SSCS and *Telegrip* respectively:

(a) The class that represents the object is instantiated in SSCS and a reference to that instance is added to 'ListObjRef' of the object vault.
(b) The graphical model of the object is retrieved from the *Telegrip* library and added to the *Telegrip's* **WORKCELL**.

Figure 4 shows an instance of class vault, which contains a reference to the instances of classroom, door and rack (the graphical models of these are present in the *Telegrip* **WORKCELL**). It also has a reference to the constraint list, which is a collection of instances of class 'constraint.' The fact that an instance of constraint has a reference (or a pointer) to the object is also shown in the following figure.

An instance of class vault in SSCS represents a **WORKCELL** in *Telegrip*. The **WORKCELL** is retrieved in Telegrip when an instance of vault is retrieved in SSCS and the **WORKCELL** is saved when the instance of vault is saved. At any stage, while composing a simulation, the user can invoke the method 'CheckCons' of Vault, which will traverse the constraint list and report back to the user if some constraint is being violated.

Name	Vault1
Object List	
Constraint List	

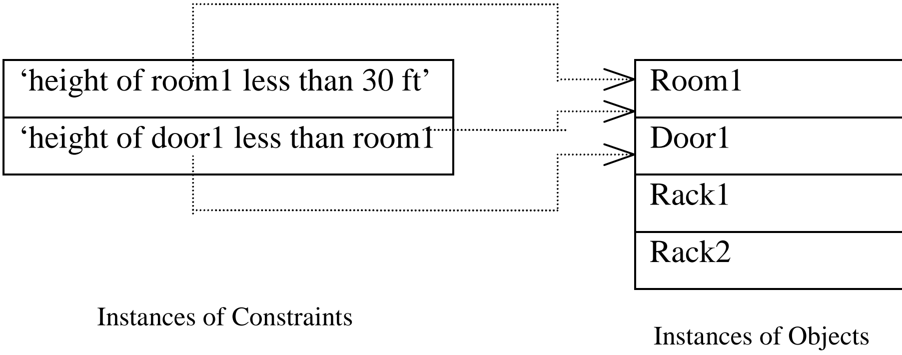

Figure 4: Instance of Class Vault

4. DESIGN AND ARCHITECTURE

This chapter discusses the architecture for implementation of SSCS. Section 4.1 presents the architecture, Section 4.2 explains the internal data flow, Section 4.3 discusses interfacing with *Telegrip.*

4.1 INTERNAL STRUCTURE OF SSCS

This section presents the internal structure of the SSCS. The diagram shows the various operations that a user may perform while using SSCS (discussed in the next section) and the internal structure of the SSCS. The various components of SSCS along with the functional description are given in Figure 5.

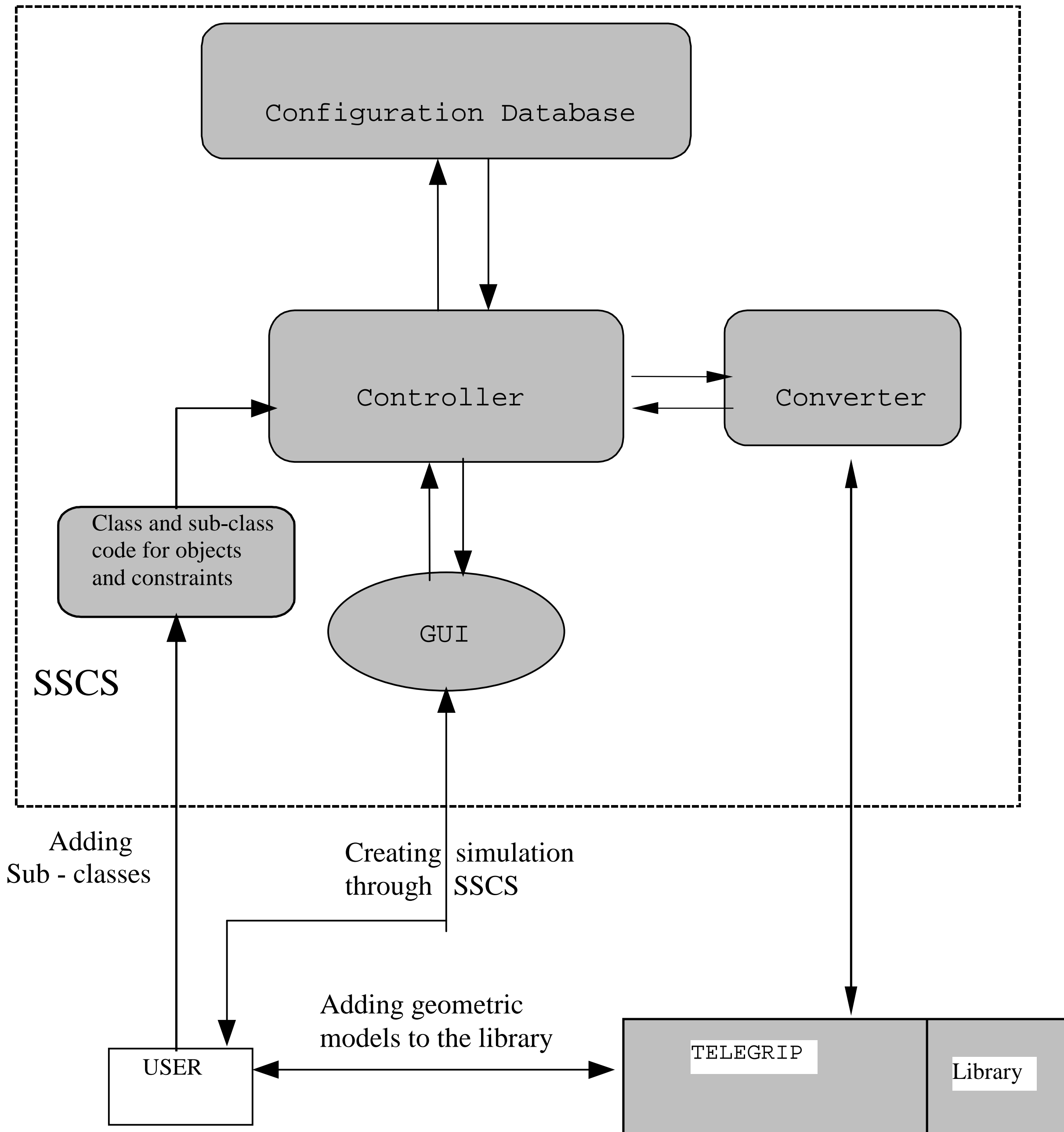

Figure 5: SSCS Internal Structure

(a) **CONTROLLER**:
The **CONTROLLER** contains the control program for SSCS. The main responsibility of the **CONTROLLER** is to coordinate the flow of information between other components of the SSCS. It gets the users input via **GUI**, and coordinates data flow between the other components (the data flow is explained in the next section). It performs various operations on the classes such as creating an instance of a class or invoking one of the class methods, etc. It is also responsible for getting the feedback, like an error condition, and passing it on to the **GUI** for display.

(b) **CONFIGURATION DATABASE:**
The **CONFIGURATION DATABASE** manages the data files which stores the instance of object Vault (for the loaded **WORKCELL**). It is invoked by the **CONTROLLER** for supplying data, such as the value of attributes required for checking constraints. Whenever an object is created, deleted or modified, it is invoked by the **CONTROLLER** for updating the data files.

(c) The **CONVERTER** is the interface with *Telegrip*. It receives the commands from the **CONTROLLER** for execution of methods. Its responsibility is to convert the user invoked methods to an equivalent *Telegrip* understandable command and pass on to *Telegrip*. The *Telegrip* executes the command and provides feedback in the form of a code back to the **CONVERTER**. The **CONVERTER** passes this feedback on to the **CONTROLLER**, which displays an appropriate message to the user through **GUI**.

4.2 DATA FLOW

The main operations that a user can perform while using SSCS can be broadly divided into the following categories. This is further explained in the following sections.

1. Add new object types to SSCS -
   This operation involves the following activities to be performed.
   - Add geometric model of the object to the Telegrip library.
   - Add Sub-class for the object to SSCS.
2. Add user defined functional constraints (this involves adding sub-classes).
3. Build simulations in *Telegrip* with SSCS using the geometric models present in the *Telegrip's* library.

4.2.1 Adding Geometric Models to the Telegrip Library

For populating the *Telegrip's* library with the geometric model of a user defined object type, the user has to create the geometric model using *Telegrip's* menu. The model should be saved as a user named file in a predefined directory path. This directory path and filename will be used for associating this geometric model with the SSCS sub-class.

We take an example in which the user is adding a new type of drum to the SSCS. The first step would be to construct the geometric model of the new drum in *Telegrip* (through *Telegrip's* menu). Let us say that the geometric model is made and is saved in a file called 'newdrum' in the predefined directory. The name 'newdrum' has to be used to associate the corresponding sub-class with this model.

4.2.2 Adding Sub-Classes to SSCS

For adding sub-classes to SSCS, the user has to write the C++ code and save it in a file in the predefined directory path. This

sub-class will be an extension of the super-class with additional attributes and redefined methods. The C++ code for the sub-class would be as per the sub-class template provided with SSCS. After writing the sub-class the code has to be compiled with the 'makefile' provided with the SSCS. On compiling the code there will be a shared object file created (.so).

When the SSCS is started, it reads the predefined directory for getting the names of all the sub-classes and displays it to the user. When the user selects a particular name, the shared object for that sub-class gets dynamically loaded and an instance of the sub-class is returned. With this instance of the sub-class we can access its attributes and methods.

In the example from the previous section, a sub-class has to be written for the "newdrum," by extending the super-class. A part of this class is shown in Table 11.

The class newdrum is an extension of the class super (the super-class). The user may define some attributes specific to the drum such as diameter, height, etc. This sub-class has to be saved in a file named "newdrum.C" in a predefined directory (note that the filename is the same as the class name, and the extension ".C" indicates it is a file with C++ code). The code can be compiled by typing 'make.' Now these sub-classes can be used with SSCS. (The sub-classes for the functional constraints can be added in the same manner.)

4.2.3 Creating Simulations through SSCS

We have taken an example in which, while creating a storage system simulation, the user wanted to use the "newdrum," which was added in the previous examples. The data flow in SSCS for this operation is shown in Figure 11, and is explained in the following text.

The user invokes a menu button for creating an object. This brings up a list of objects (which are present in the library) that can be created. After the user chooses the particular object, he or she has to enter the user data such as location and orientation through **GUI**. The **GUI** checks for the validity of strings and numbers.

Table 11: Class Newdrum

```
Class newdrum : class super    // newdrum is a sub-class of
                                  class super.
Attributes
Diameter, height, . . .
Methods
Create, delete, . . .
```

- A valid string starts with a character.
- A valid number is a valid integer having a maximum value of 2147483647. This is the maximum integer value that C++ can support (a long integer is 32-bits long).

If there is an error, it reports it back to the user; otherwise, it passes it on to the **CONTROLLER** for further processing.

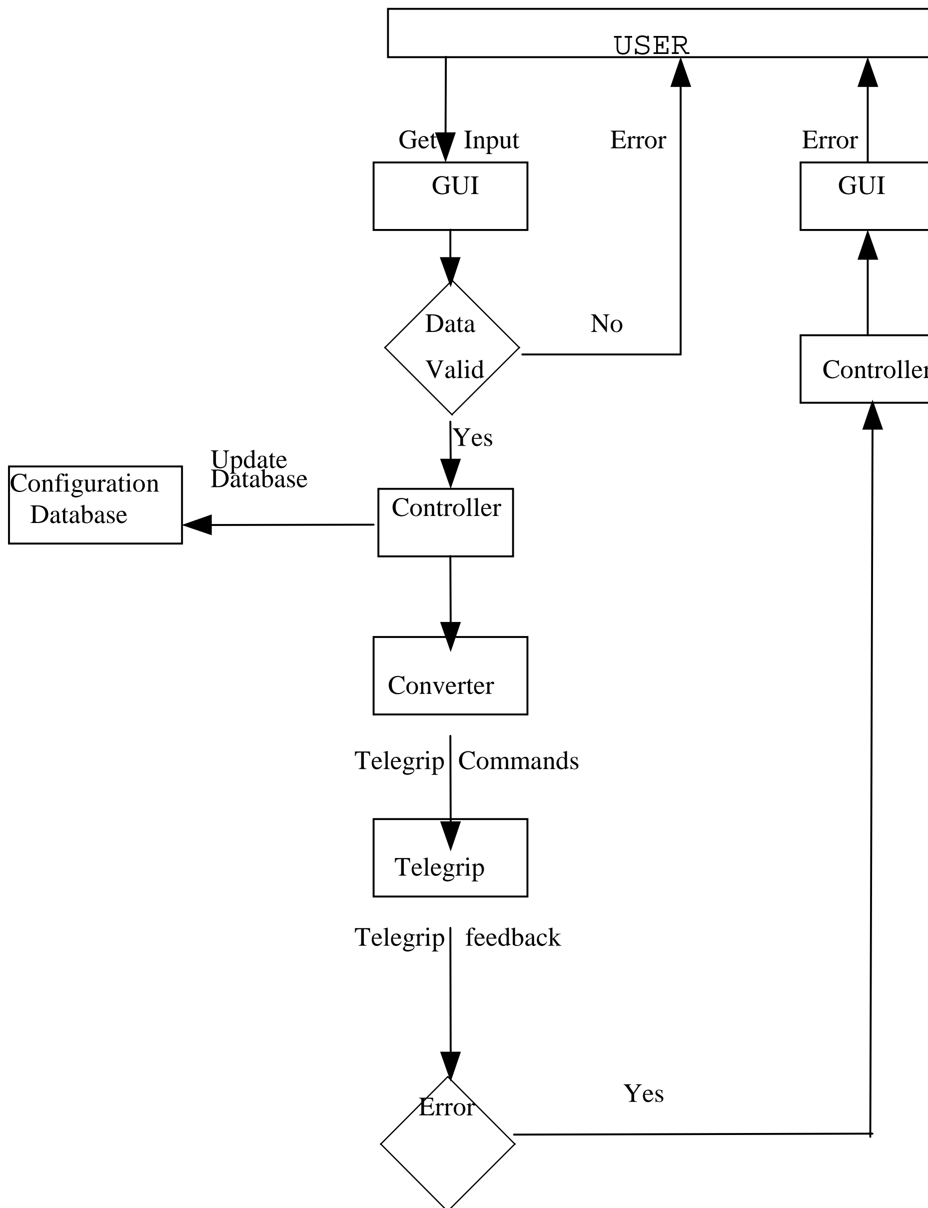

Figure 6: Data Flow in SSCS

The **CONTROLLER** asks the **CONFIGURATION DATABASE** to update object Vault with the changes arising out of creation of the drum. In this case, an instance of the drum will be added to the list of objects in the Vault. The **CONTROLLER** passes on the request for creating the drum, and the parameters required such as the location and orientation, to the **CONVERTER.**

The **CONVERTER** receives the request for creation of drum and issues an equivalent CLI (command line interpreter) command or invokes an Axxess function to carry out execution in *Telegrip* (these methods of interfacing with *Telegrip* are explained in the next section). It waits for a response from *Telegrip*. If no error is reported then the method executed successfully. If *Telegrip* reports an error, it is returned to the **CONTROLLER** and then to **GUI** for displaying to the user. If there is no error, then no feedback is given to user, indicating that the drum has been successfully created in *Telegrip*. After this, the user should be able to see the model of the drum in the *Telegrip* **WORKCELL**.

4.2.4 Checking Constraints

The data flow related with the constraint checking in SSCS is shown in Figure 7.

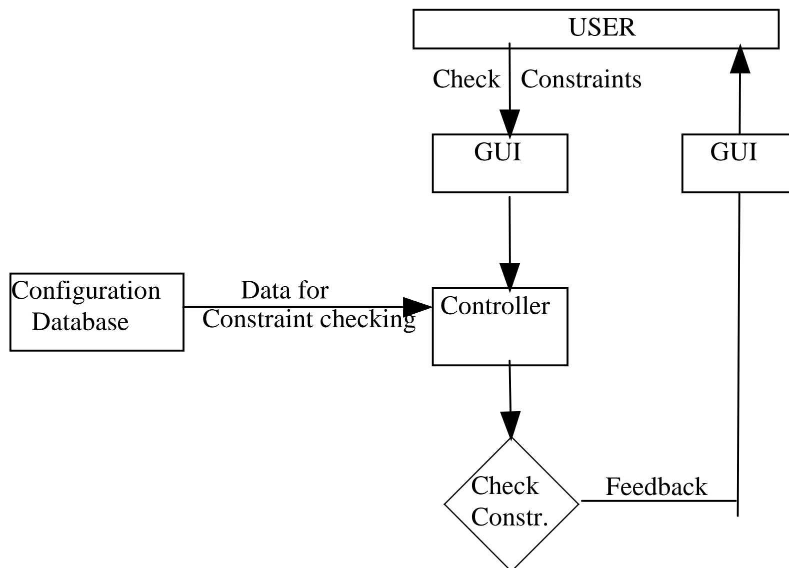

Figure 7: Data Flow for Constraint Checking

A request for checking constraints is made by the user through **GUI**, it is passed on to the **CONTROLLER** which gets the data for checking constraints from the **CONFIGURATION DATABASE**. The **CONTROLLER** evaluates the constraints and provides feedback to the user, through **GUI**, concerning any violation of constraints.

4.3 INTERFACE WITH TELEGRIP

The various modes of the Telegrip interface and their usage by SSCS is discussed in this section. *Telegrip* can be controlled in the following ways:

- *Telegrip* menu system
- Graphic Simulation Language
- Command line interpreter
- Axxess programming interface.

(a) *Menu System*: The menu system provides a mouse driven, graphical interface to the user. It is most flexible and provides a higher level of control to the user. In SSCS, this mode will only be used for adjusting the views of the models. Only the display functions such as CRUISE, ROTATE and VIEW will be used through the *Telegrip* menu. All the changes in geometry will be managed through the SSCS. This is important because SSCS would not be aware of any changes made to geometry directly through *Telegrip*.

(b) *Graphical Simulation Language (GSL)*: It is a higher level programming language for use in graphic simulation. It is mainly used for programming the devices and their control. SSCS will use this for executing a series of CLI commands.

(c) *Command Line Interpreter (CLI)*: The CLI processes a command language that interacts with *Telegrip's* simulation environment. It can load a **WORKCELL**, manipulate Devices, Paths etc., and load GSL programs for running a simulation. Most buttons in the Menu System have a corresponding CLI command or a set of commands. In addition, CLI can perform functions that are not possible through the menu buttons. The CLI commands can be sent through an external program like SSCS for controlling *Telegrip*. *Telegrip* will execute the command and return a code; the value of code indicates whether there was an error while executing the command and if so, what the type of error was.

(d) *Axxess Programming Interface*: Axxess is a flexible API (Application Programmer Interface) framework in which the user can easily integrate their own software with a *Telegrip* kernel. It provides flexibility in coding and portability. While CLI can manage workcells, it cannot modify the geometry of the models. Axxess accesses the internal functions and data structures within *Telegrip* via the Axxess API and can modify the geometry of the models. The Axxess library of functions provides a comprehensive set of routines that the developers may use to build their own applications. SSCS utilizes the Axxess functions for manipulating the internal data structure of the objects and thus controls their geometry.

The **CONVERTER** will communicate will *Telegrip* mainly through the CLI commands (for **WORKCELL** manipulation operations such as loading, saving, etc.) and the Axxess API (for manipulating geometry of models). The menu in *Telegrip* will only be used for changing the views and not for changing the geometry of the objects as explained earlier.

REFERENCES

This report did not cite any references.