Chapter 1

Introduction

1.1 Motivation and Significance

One of the major tasks for computer scientists is to abstract the physical world into models that a computer can process. Thus computer languages should be designed to narrow the gap between the concepts in the real world and those of computer languages. There are two major categories of programming languages: imperative and declarative. Imperative languages require a step-by-step procedure for solving a problem, while declarative languages focus on the main constraints of the problem. Imperative languages have evolved from procedure languages to object-oriented languages, such as Java and C#, which model real world applications by means of features such as encapsulation, inheritance, aggregation, polymorphism, etc. Meanwhile, the declarative languages make the modeling process simpler by requiring only a specification of the constraints in the applications, and doing the necessary computation automatically. Obviously, it is highly desirable to combine these two kinds of languages together to gain the advantages of both while overcoming their disadvantages.

Significant research work has been done to fulfill this goal and numerous approaches and results have been obtained. Recently a new programming paradigm called *constrained objects* has been proposed in which constraints are embedded in objects [3,11,12]. The idea is that while the object-oriented point of view is natural,

the behavior of an object in many domains, especially engineering, is not naturally modeled by traditional imperative procedures. For example, consider the truss in Figure 1. The two major kinds of objects in the truss are “bar” and “joint”. While bars have constraints expressing the standard law of forces and material, the joints have equilibrium constraints over the bars that are incident at the joint. Thus, bars and joints are most appropriately modeled as constrained objects rather than as traditional objects.

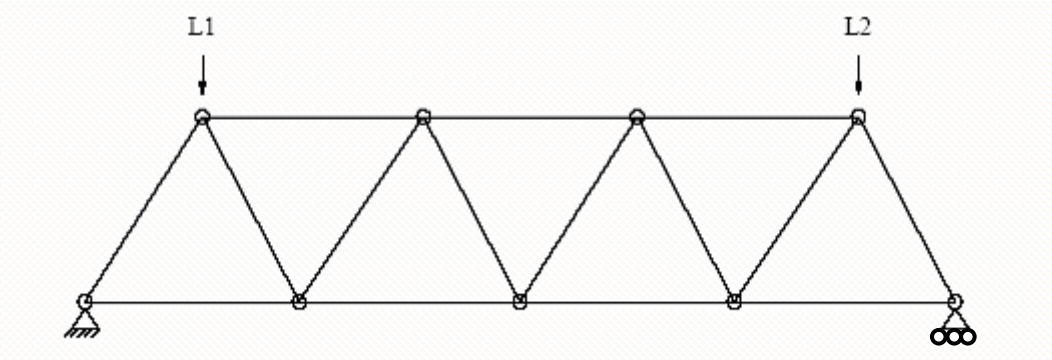


Figure 1. A simple truss

In this thesis we will explore two major modeling domains for constrained objects:

(i) modeling static systems; and

(ii) modeling dynamic systems.

Our primary example of a static system is a building architecture, a domain in which the computer has played an increasingly important role in recent years. However, most computer-aided design programs have evolved from traditional representations such as scale drawings, etc. Like their predecessors, these programs still need much human involvement and correction because they inherit the same informational deficiencies. As explained by Khemlani et al [4], what we need is a

more “intelligent” representation of a building, one which serves as a detailed computation model and also includes semantically meaningful information on all aspects of the building. They also classify building elements into three main categories: architecture, structure, and geometry. To show the applicability of constrained objects in this domain, we provide a detailed example of applying the paradigm in modeling a simple two-level structure according to their categorization of building elements. This example not only gives an in-depth look into the concept and semantics of the constrained objects language, but also the common methodology of applying the paradigm to a given domain.

The second application domain is motivated by the fact that there are many problems where constraints and objects are natural but the behavior of the system is not static, i.e., the objects exhibit time-varying behavior. Examples arise in different domains, for example, voltage and current in an AC circuit, fluid flow between cells in a hydrology system, sensor networks, moving objects, etc. However, there are constraints between the objects involved in the model and the concept of constrained objects is still applicable in these dynamic systems.

As a detailed case study of a dynamic system, consider the nerve system which is the information highway of a living body. In the 1950’s, Hodgkin and Huxley formulated a mathematical model the nerve cell’s behavior. They showed that electrical signals are sent out by the body periodically to control bodily processes such as muscular movement. Furthermore, electrical signals are controlled by ions and their concentrations around the nerve cell. However, the relation between the ion flow and the electrical current follow a set of equations which can be viewed as constraints among them. This model is now referred to as the Hodgkin-Huxley model and these scientists received the Nobel Prize for their work.

The concept of dynamic constrained objects was introduced by Jayaraman and Raux in a recent paper[10]. Thus the second major component of thesis is showing that examples such as the AC circuit and the nerve cell behavior can be modeled as dynamic constrained objects. The methodology behind the computational model for dynamic constrained objects can serve as a basis for the next generation of constrained objects.

1.2 Outline of the Thesis Chapters

Chapter 2 gives a brief overview of constrained objects. We first introduce a language, called Cob, which implements the concept of constrained object. We then describe the complete syntax of Cob and illustrate the paradigm of constrained objects with two applications: DC circuits and trusses. Finally, we sketch the constrained object execution environment including compilation and execution.

Chapter 3 describes the application of the constrained objects for modeling intelligent buildings. First we review the notion of an ‘intelligent building’ described by Khemlani et al, and show how objects and constraints are naturally come into play there. Then we describe the framework of intelligent building design and the associated overall architecture. We subsequently introduce the concept of parametric design and show a parametric-based design for intelligent building. Finally, we assemble all the pieces in order to obtain our results. A higher-level approach to the building problem based on parametric design is also described.

Chapter 4 describes the paradigm of dynamic constrained objects. We first address the weakness of the static constrained objects in modeling dynamic systems. Second, we discuss the concept of *series* variables to represent the attributes that

change with time. A series variable has different states associated with time, and constraints involving previous states and future states can be specified. After showing its syntax and usage, we use dynamic constrained objects to model AC circuits and nerve cell behavior, and we illustrate the computational model.

Chapter 5 presents our conclusions and also addresses future research for constrained objects.

Chapter 2

Overview of Constrained Objects

2.1 Introduction

As we stated in the introduction, constraints and objects are both powerful and have wide application. Cob is a constrained object programming language that supports traditional object-oriented features such as encapsulation, inheritance, aggregation, and polymorphism, as well as declarative constraint language features such as arithmetic equations and inequalities, quantified and conditional constraints, and disequations [3]. Cob also has a modeling/execution environment that includes a compiler, executer, and debugger. In addition, a constrained object system usually has an affiliated visual representation component that is domain specific. This is natural since, often every component of the visual part can be traced back to an object in the underlying model. All of the above features help the user create, modify, and execute a Cob program without a deep understanding of the theory of logic programming or constraint solving techniques. This allows users with domain specific knowledge to write Cob programs easily.

2.1.1 Cob Syntax

A Cob program is basically a series of class definitions. A class definition is made up of attributes, constraints, predicates, and constructors [11].

class_definition ::= [**abstract**] **class** *class_id* [**extends** *class_id*] {*body*}

```
body ::=    [ attributes attributes ]
            [ constraints constraints ]
            [ predicates pred_claues ]
            [ constructors constructor_clauses ]
```

The syntax of attributes in Cob is similar to that of an object-oriented language. Cob has primitive types, including arrays, and also user defined types which are defined by classes.

```
        attributes    ::= [ decl ; ]+
             decl     ::= type id_list
              type    ::= primitive_type_id | class_id | type []
primitive_type_id     ::= real | int | bool | char | string
             id_list  ::= attribute_id [, attribute_id]*
```

Cob constraints can be simple, quantified, creational, or any combination of them.

```
      constraints    ::= [ constraint ;]+
         constraint  ::= simple_constraint | quantified_constraint |
                         creational_constraint
    simple_constraint ::= conditional constraint | constraint_atom
quantified_constraint ::= forall var in enum : (constraints)
                        | exists var in enum : (constraints)
 creational_constraint ::= complex_id = new class_id( terms)
      constraint_atom ::= term relop term | constraint_predicate_id(terms)
               relop  ::= > | < | >= | <= | != | =
```

The constructor is a means of creating an instance of a class and a class can have

one or more constructors.

constructor_clauses ::= *constructor_clause*$^{+}$

constructor_clause ::= *constructor_id(formal_parts) { constructor_body }*

constructor_body ::= *constraints*

To help understand these ideas better we will present a Cob program that shows how these parts are assembled together to model a problem.

2.2 Examples of Cob Programs

2.2.1 Cob Model for DC Circuit

Our first example is a model of a simple electrical DC circuit [3]. We model the components and connections of such a circuit as objects and their properties as constraints on their attributes. We also define an assembly class which is responsible for creating every component of the circuit.

We define a component class as the parent class of any electrical entity (e.g. resistor, voltage source) with two ends (referred to as 1 and 2 respectively). The attributes of this class represent the currents and voltages at the two ends of the entity.

```
abstract class component {
    attributes
        real I1, I2;
        real V1, V2;
    constraints
        I1+I2=0;
}
```

The constraint in class `resistor`, which is a subclass of component, represents Ohm's law. The constraint in class battery shows that one end of the battery is connected to ground (voltage is 0).

```
class resistor extends component{
    attributes
        real R;
    constraints
        V1 - V2 = I1*R;
    constructors resistor(X) {
        R = X;
  }
}
```

```
class battery extends component{
    constraints
        V2 = 0;
    constructors battery(X) {
        V1 = X;
  }
}
```

The class `componentEnd` represents a particular end of a component. We use the convention that the voltage at end 1 of a component is V1 (similarly for current).

```
class componentEnd {
    attributes
        component C;
        real End, V, I ;
    constraints
        V = C.V1 :- End = 1;
        V = C.V2 :- End = 2;
        I = C.I1[1] :- End = 1;
        I = C.I2[1] :- End = 2;
    Constructors componentEnd(C1, E){
        C = C1; End = E;
    }
}
```

The class `node` aggregates a collection of ends. When the ends of components are placed together at a node, their voltages must be equal and the sum of the currents through them must be zero (Kirchoff's law). Notice the use of the quantified constraints (`forall`) to specify these laws.

```
class node {
    attributes
        componentEnd [] Ce;
        real V;
    constraints
        sum X in Ce: X.I = 0;
        forall X in Ce: X.V = V;
    constructors node(L) {
        Ce = L;
  }
}
```

We can create any DC circuit by adding an additional assembly class which we

usually name it as `sampleCircuit`. The syntax to create a new object is very similar to that in C++/Java; however, it is considered as a creational constraint in our programming paradigm. We now show an example, Figure 2, to model with Cob.

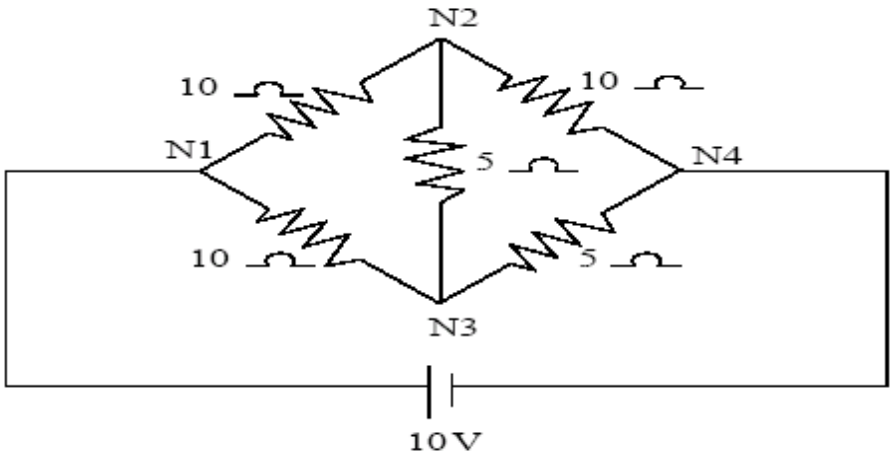


Figure 2. An example of a DC circuit (modified from [3])

```
class samplecircuit {
    attributes
        resistor R12, R13, R23, R24, R34;
        battery B;
        componentEnd Re121, Re122, Re131, Re132, Re231, Re232, Re241,
                     Re242, Re341, Re342, Be1, Be2;
        node N1, N2, N3, N4;
    constructors samplecircuit() {
        R12 = new resistor(10);
        R13 = new resistor(10);
        R23 = new resistor(5);
        R24 = new resistor(10);
        R34 = new resistor(5);
        Re121 = new componentEnd(R12, 1);
        Re122 = new componentEnd(R12, 2);
        Re131 = new componentEnd(R13, 1);
        Re132 = new componentEnd(R13, 2);
        Re231 = new componentEnd(R23, 1);
```

```
        Re232 = new componentEnd(R23, 2);
        Re241 = new componentEnd(R24, 1);
        Re242 = new componentEnd(R24, 2);
        Re341 = new componentEnd(R34, 1);
        Re342 = new componentEnd(R34, 2);
        B = new battery(10);
        Be1 = new componentEnd(B, 1);
        Be2 = new componentEnd(B, 2);
        N1 = new node([Re121, Be1, Re131]);
        N2 = new node([Re122, Re241, Re231]);
        N3 = new node([Re132, Re232, Re341]);
        N4 = new node([Re242, Re342, Be2]);
  }
}
```

All nodes are marked on the figure 2, and given initial values for some of their attributes. The model can then be used to calculate values of the remaining attributes (e.g. the current through a particular component).

2.2.2 Truss Example

To illustrate the use of constrained objects in structural engineering design, we define the Cob classes needed to model a simple truss structure, as shown in Figure 1 [3]. A truss consists of bars placed together at joints. We will use classes to model bar and joint objects, and apply constraints to describe their individual physical properties and connection conditions. There are four different kinds of objects in a truss and we will model them by four classes. We will first introduce the `beam` class which models the basic objects in the truss.

```
class beam {
    attributes
        Real E, Sy, L, W, H, F_bn, F_bk, F_t, Sigma, I, F;
    constraints
```

```
        I = F_bk * L *L /3.141 * 3.141*E;
        I = W * H * H * H / 12;
        F_t = (Sy * (W * H));
        Sigma = (((H * L) * F_bn )/ (8 * I));
        F_t = F :- F > 0;
        F_bk = F :- F < 0;
    constructors bar(E1,Sy1,L1,W1,H1,F_bn1,F_bk1,F_t1,Sigma1,I1,F1) {
        E=E1;  Sy=Sy1;  L=L1;  H=H1;  W=W1;  F_bn=F_bn1;
        F_bk=F_bk1;  F_t=F_t1;  Sigma=Sigma1;  I=I1;  F=F1;
    }
}
```

The constraints in the beam class express the standard relations between its modulus of elasticity (`E`), yield strength (`Sy`), dimensions (`L, W, H`), bending, buckling, and tension forces (`F bn, F bk, F t`), and stress (`Sigma`). Depending upon the direction of the force in a beam (inward or outward), it acts as either a buckling force or a tension force. This relation is expressed as conditional constraints in the beam class. The above details were taken from the text by Mayne and Margolis [14].

```
class bar {
    attributes
        bar B; Real A;
    constraints
        0 <= A;   A <= 2* 180;
    constructors beam(B1, A1) {
        B = B1; A = A1;
    }
}
```

```
class load {
    attributes
        Real F; Real A;
    constraints
        0 <= A;   A <= 2*180;
    constructors load(F1, A1) {
        F = F1; A = A1;
    }
}
```

A bar is a beam placed at an angle (A) and a load is a force (F) applied at an angle (A).

```
class joint {
    attributes
        beam [] Beams; load [] Loads;
```

```
    constraints
       (sum X in Beams: (X.B.F * sin(X.A))) + sum X in Loads: (X.F * sin(X.A))
        = 0;
       (sum X in Beams: (X.B.F * cos(X.A))) + sum X in Loads: (X.F * cos(X.A))
        = 0;
    constructors joint(B1, L1) {
        Beams = B1; Loads = L1 ;
    }
}
```

The `joint` class aggregates an array of bars and an array of loads (that are incident at the joint) and its constraints state that the sum of the forces in the horizontal and vertical directions respectively must be 0.

The classes defined above can be applied to any truss; each individual truss still needs a specific assembly class to model its own unique structure. Here we will show the assembly class corresponds to the truss in Figure 1.

```
class sampletruss {
    attributes
        beam AB, BC, CD, BD, AC;
        bar IAB, IAC, IBA, IBC, IBD, ICA, ICB, ICD, IDB, IDC;
        load IAV, IAH, ICV, IDV;
        bar []  Ba, Bb, Bc, Bd;
        load []  La, Lc, Ld;
        Real Esteel, Sy, Pi;
        Real W1, W2, W3, W4, W5, H1, H2, H3, H4, H5,
            Fab, Fbc, Fcd, Fbd, Fac,
            Fab_bk, Fbc_bk, Fcd_bk, Fbd_bk, Fac_bk,
            Fab_bn, Fbc_bn, Fcd_bn, Fbd_bn, Fac_bn,
            Fab_t, Fbc_t, Fcd_t, Fbd_t, Fac_t,
            Fdv, Fcv, Fav, Fah,
            Sigmaab, Sigmabc, Sigmacd, Sigmaac, Sigmabd,
            Iab, Ibc, Icd, Ibd, Iac;
        joint JA, JB, JC, JD;
```

```
    constraints
        ESteel = 30000000;
        Sy = 30000;
        Pi = 3.141;
        W1 = H1; W2 = H2; W3 = H3; W4 = H4; W5 = H5;
    constructors sampletruss() {
        AB = new beam(ESteel, Sy, 10.4, W1, H1, Fab_bn, Fab_bk, Fab_t,
                      Sigmaab, Iab, Fab);
        BC = new beam(ESteel, Sy, 7.3, W2, H2, Fbc_bn, Fbc_bk, Fbc_t,
                      Sigmabc, Ibc, Fbc);
        CD = new beam(ESteel, Sy, 12.7, W3, H3, Fcd_bn, Fcd_bk, Fcd_t,
                      Sigmacd, Icd, Fcd);
        BD = new beam(ESteel, Sy, 14.7, W4, H4, Fbd_bn, Fbd_bk, Fbd_t,
                      Sigmabd, Ibd, Fbd);
        AC = new beam(ESteel, Sy, 7.3, W5, H5, Fac_bn, Fac_bk, Fac_t,
                      Sigmaac, Iac, Fac);
        IAB = new bar(AB, Pi/4); IAC = new bar(AC, 0); IAV = new load(Fav,
                      Pi/2); IAH = new load(Fah, 0);
        Ba = [IAB, IAC]; La = [IAV, IAH];
        JA = new joint(Ba, La);  IBA = new bar(AB, 5*Pi/4);
        IBC = new bar(BC, 3*Pi/2); IBD = new bar(BD, 11*Pi/6);
        Bb = [IBA, IBC, IBD]; Lb = [];   JB = new joint(Bb, Lb);
        ICA = new bar(AC, Pi); ICB = new bar(BC, Pi/2);
        ICD = new bar(CD, 0);  ICV = new load(15000, 3*Pi/2);
        Bc = [ICA, ICB, ICD];  Lc = [ICV];
        JC = new joint(Bc, Lc);  IDB = new bar(BD, 5*Pi/6);
        IDC = new bar(CD, Pi); IDV = new bar(Fdv, Pi/2);
        Bd = [IDB, IDC]; Ld = [IDV]; JD = new joint(Bd, Ld);
    }
}
```

The Cob classes defined here can model a truss with given applied loads. These classes may be used to determine the forces acting in the beams given the loads and the dimensions of the beams. It can also be used to determine the dimensions (e.g. width) in order support a given maximum load; this capability requires the

computational model to support the solution of nonlinear equations. After we create the constrained object model for the application, the next step is to compile, run and get the result.

2.3 Cob Execution Environment

The Cob compiler compiles the Cob programs into CLP(R) programs, which is basically a Prolog-like language with a built-in constraint solver. Each Cob class is mapped to a CLP(R) predicate such that the constraints of the class form the body of the predicate [2]. Since CLP(R) supports only linear constraints, the Cob implementation has been extended to solve conditional constraints and has also been interfaced with Maple to solve non-linear equations. After successfully compiling the files, a partial evaluator is invoked to produce the optimized code. The optimization basically eliminates the overheads from unnecessary predicate invocations. It also unravels the iteration specified by quantified constraints, and also minimizes the amount of constraint re-checking that takes place when attributes are assigned values. The user then loads the compiled file into CLP(R) and executes one or more queries in order study the model. Here is the overall diagram Figure 3 of the Cob execution environment.

In order to enhance the usability of constrained objects, a domain-specific user interface is generally also provided. Such interfaces have been constructed for simple domains such as circuits and trusses. The domain-specific interface translates (circuit or truss) diagrams into textual Cob code which is then compiled down to CLP(R) for execution. This interface requires knowledge of the basic classes that make up the domain; it is primarily useful in building a specify assembly for testing.

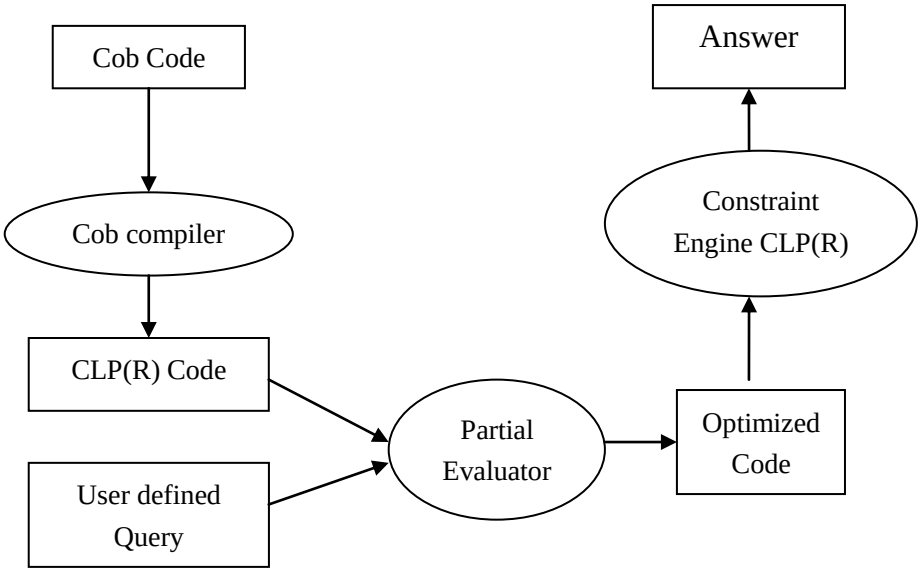


Figure 3. Flow diagram of the Cob computational model

Chapter 3

Constrained Objects for Intelligent Building Design

3.1 Objects and Constraints in Building Architecture

The paradigm of intelligent building representations consists of two main parts: the components and the assembly [4]. The components, which are basically representations of various building elements, have attributes (properties) and a hierarchical structure. Most of the building elements have some variations, e.g., there are various kinds of walls which differ from each other by their function or material. The walls can be grouped by similar properties and organized into a hierarchy. Furthermore, building elements involve many aggregation relations with one another. For example, as shown in Figure 4, the surface element will aggregate wall, edge, and opening elements, while a wall element will aggregate beam and vertex elements. Thus, an object-oriented approach to building element representation is very natural.

On the other hand, the structure of building elements and their relations involve many constraints between them. According to [6] there are *mate constraints* (which join points, axes, plane, etc.), *insert constraints* (which align two circles), *flush constraints* (which make two planes coplanar with their faces aligned in the same direction), and *angle constraints* (which control an angle between two planes). For an example of a mate constraint, the positions of a beam and a column should be the same when they are joined together; for an example of a flush constraint, the height of

all four walls of a room should be the same; etc. Thus constraints play an important role in the assembly and hence it would be ideal to integrate constraints with an object-oriented design.

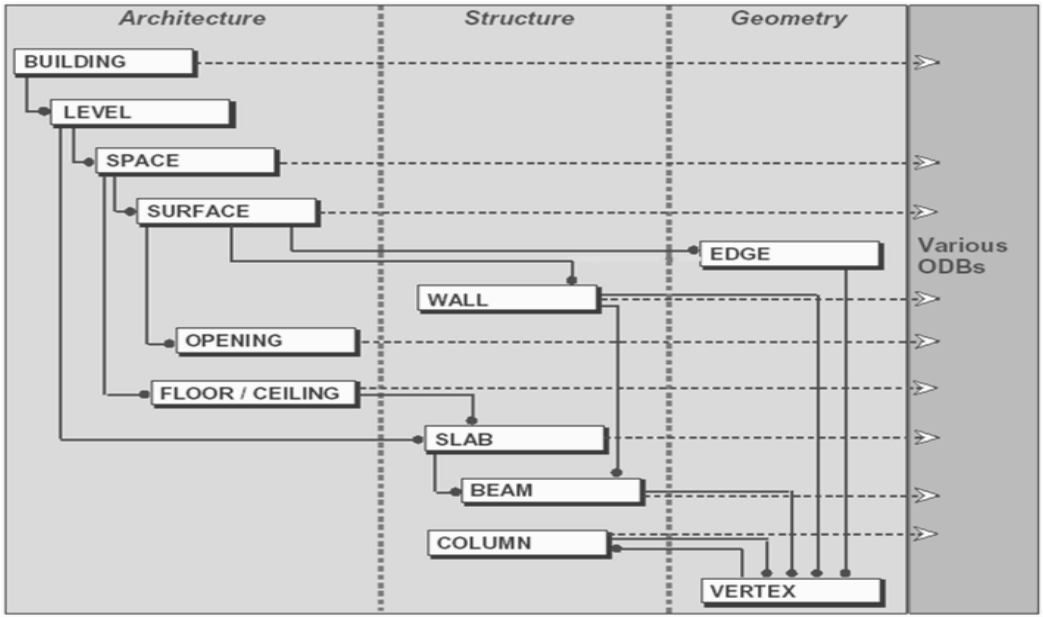


Figure 4. Overall class diagram of all building elements, adapted from[4]

One of our goals in this thesis is to create a constrained-object based model of a building. It is natural to use Cob to implement the idea since Cob has all the features needed for this purpose. Our model can be viewed as an object-oriented database coupled with a powerful constraint-based computation. Consequently, with this model we can analyze the most useful properties that architects need to know about a building.

3.2 Related Work

Nassar's Approach. The use of the constraint-based model in building has been explored by Nassar et al. [6,7]. In their work, a set of constraint-based

assembly operations for generating 3D details of building assemblies are presented. The operations constrain the locations and orientations of the components in a building assembly through a series of constructive steps. There are two operations: declarative constraints, which relate the location of two objects together, and assembly operation. This approach is quite similar to the ODB and PDB in [4]; however they put more emphasis on geometry or structure instead of the architectural features of the building.

Nassar's approach also differs from our approach in several ways. First, Nassar's approach uses more architecture-related knowledge, while our approach pays more attention to fully applying objects and constraints. They use Visual Basic, which is supported by AUTOCAD, to enforce constraints; but this does not fully support the constraint concept. Should some constraints fail, they will not be able to detect it. Secondly, in Nassar's approach they don't have the concept of object while in our approach every building element is an object.

Khemlani's Approach. As stated earlier, the intelligent building representation is divided into two parts: components and assembly. The components are stored in an Object Database (ODB), which stores detailed information about various building elements. Since the elements have the inherited relation we mentioned before, the objects will be stored in a hierarchical structure like Figure 5. The assembly is stored in a Project Database (PDB), which holds information about how these elements are assembled to make up a particular building. The PDB stores all the relations between various components, like which beam should be supported. The main obstacle is a so-called space-structure dilemma [4]. In order to solve this problem, the PDB imports the notion of a purely geometric 'vertex' and also tries to represent not only

structure but also spaces. Thus the assembly rule becomes a little bit complicated and all the building elements are to some extent related to another. On the other hand, it also helped to computationally improve the model.

The intelligent building representation of Khemlani et al supports assessments such as access, egress, and privacy, which will be performed by expert systems or other types of programs, to be developed separately from the building representation. In this sense, it may be regarded as a semantically "rich" model.

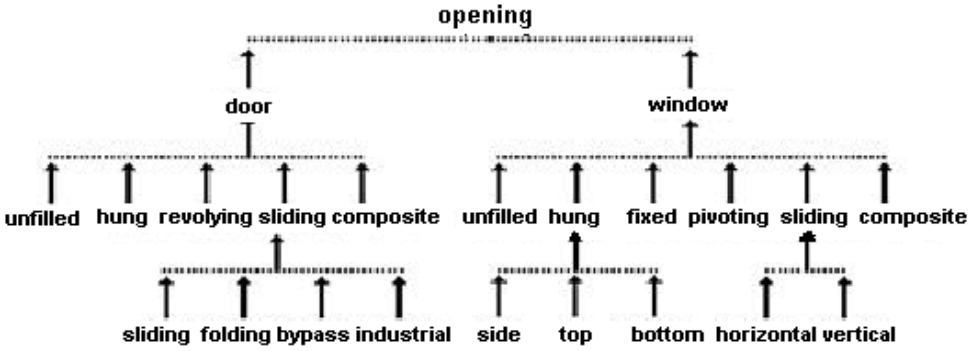


Figure 5. The hierarchy structure of opening

3.3 Constrained Objects Model for Intelligent Building

Our approach is that we map every kind of building element in the intelligent building representation to a Cob object and combine them by applying all the applicable constraints between the objects. Since Cob also supports inheritance, it is straightforward to represent inheritance hierarchies such as that shown in Figure 5. In order to simplify and emphasize the overall relations among all the building elements, we will only show the "top-level" object of every building element. For example, we will use "wall" to represent all kinds of walls, regardless of whether it is a support wall or non-support wall; neither do we distinguish between an internal wall and an

external wall. This is because all the sub-classes of the top-level object will inherit the constraints and thus have the same relation with all the other elements as the top-level object.

Constraints are the key idea when we try to define an assembly in Cob. As we mentioned before, we will have one class for each kind of building element along with the corresponding constraints of the class. Since constraints will come into play when elements are integrated together, they determine whether two elements can be combined or how two elements are combined.

Thus, objects and constraints are two key ideas in this intelligent building representation. However, we still need an overall methodology for creating a building. Typically, there are three types of design—*exact design, parametric design and component design*. Here we will try the last two since they are the more efficient and modern methods of design.

3.3.1 Parametric and Higher Level Organization

The traditional approach of defining a building model requires the user to input all building elements and their locations. The data entry work is very tedious and error-prone for a large-scale building. To solve this problem, we first try the parametric model, in which the designers can vary the design parameters. We can then reuse existing designs and standard elements. In addition, we further simplify the process of input from the user by grouping the building elements into a higher-level virtual component called room and the user will only have to “create” the building room by room. This feature is mainly fulfilled in the domain-specific visualization tool (which is usually written in Java or another object oriented language). With this

software, the user will input the dimension of a room and specify where it is attached to the original building. After the input, the user can compile and run the program.

3.4 Parametric Design

In the traditional computer-aided program, users draw lines, arcs, and circles, which are combined with dimensions and notes to produce the drawings for civil, architectural, or mechanical designs. These programs are based on geometric objects and making a design change requires changing all appropriate components in order to make the drawing correct.

Most modern CAD software utilizes a feature called *parametric design*, a method of linking dimensions and variables to geometry in such a way that when the values change, the part changes as well. Basically, the drawings consist of different elements and each one can be described by some parameters. The user needs to assign these parameters, and the element will be created automatically by the values of these parameters. This is essentially what our approach does. Assigning values to all the parameters of the building elements can create the whole building.

Re-examining Figure 4 we see that there are elements of Building, Level, Space, Surface, Opening, Floor/Ceiling (horizontalSurface), Wall, Slab, Beam, Column, Edge, and Vertex. Furthermore, these elements are grouped into three categories—architecture, structure and geometry. Although they share some common features, the elements in the three categories also have their own characteristics. We will show how to represent building elements in these three categories, and in the process we will also illustrate the Cob language features further.

3.4.1 Architecture, Structure and Geometry

The ‘architecture’ elements are the higher-level elements to which we will add constraints to analyze important or interesting features of the building. All the data needed to depict the building can be obtained from objects in this category. Therefore, new analyses can be imported seamlessly into the existing system without affecting the existing model by means of new constraints.

The ‘structure’ elements are more about the material and their combination. When we assemble the structure elements in the model, we need to ensure that they can be feasibly combined together. These constraints guarantee that the whole building model is feasible in the sense that every building element can be assembled in practice.

The ‘geometry’ elements are important because the three-dimensional location for the building elements. Every object in the geometry category will carry its `xyz` coordinates and thus provide information to other elements when they are assembled together.

3.4.1.1 Architecture Example

The class definition of “level” in the architecture category is shown in Figure 6. It has four attributes: `building, spaces, slabs` and `volume`. The `level` class, as its name indicates, represents a level in a building. The `Pb` attribute stands for the parent building. The `Spaces` attribute stands for the spaces in this level. Here we use space instead of room to follow the notation from Ref. [4]. In the same way, the `Slabs` attribute stands for the slabs that form this level. Finally, the `volume`

attribute records the total volume inside this level. These attributes, except for `volume`, naturally exist according to the intelligent building representation. Other attributes and constraints can be added depending on what the model is interested in.

```
class level {
 attributes
   building Pb;
   space[] Spaces;
   slab[] Slabs;
   real Volume;
 constraints
   Volume = sum X in Spaces:
                    X.Volume;
```

```
constructors
   level(B,Sp,Sl) {
         Pb = B;
         Spaces = Sp;
         Slabs = Sl;
    }
}
```

Figure 6. The definition level class

We will illustrate how the constraint helps compute the volume here.

```
Volume = sum X in Spaces: X.Volume;
```

This constraint states that the volume of the level equals the sum of all the volumes of the spaces (rooms) in this level. Although it might not be always true since there may be aisles, such a constraint provides the methods for computing the volume of the level. This approach is based on the fact that every space has its volume. It is important to note that the space volume is a constraint and not an assignment; thus, whenever the spaces are assembled, the level's volume will be implicitly defined.

3.4.1.2 Structure Example

We use the "slab" class in the structure category, see Figure 7, to show the common features of the structure element.

```
class slab {
 attributes
   level Pl;
   beam[] Peripheralbeams;
   real Z;

 constraints
   forall  B  in  Peripheralbeams:
                  B.Z = Z;
```

```
constructors
   slab(L,B){
        Pl = L;
        Peripherialbeams = B;
    }
}
```

Figure 7. The definition of slab class

A slab is mainly defined by the beams surrounding it. Therefore we have a variable `Peripheralbeams` of type beam array. The surrounding beams will determine all properties of the slab, such as height and size. In our example we calculate the height of the slab using the constraint:

```
forall B in Peripherialbeams: B.Z = Z
```

However this constraint is more useful than just for computation; it checks if the peripheral beams really form a slab, since they must have the same height. In addition, we can further check if the beams can connect to each other by adding more constraints to match every two pairs. This is done by the ‘mate’ constraint which we mentioned in the previous sections. This kind of constraint is common in building elements of the structure category.

3.4.1.3 Geometry Example

In the intelligent building representation, an edge is tightly associated with a surface and further defined by a vertex. Therefore it is natural to have surface and vertex variables in the definition of edge. Since an edge may also have an opposite edge in order to help separate the two surfaces of the same wall, we also declare a

variable (oe) foe the opposite edge. The remaining three variables are just for the

```
class edge{
 attributes
   surface Ps;
   vertex Vertex;
   edge Oe;
   real Z;
   real HeadX,HeadY;

 constraints
   Z = Vertex.Z;
   HeadX = Vertex.X;
   HeadY = Vertex.Y;
   Oe.Ps.Wall = Ps.Wall;

 constructors
   edge(S,V,E){
        Ps = S;
        Vertex = V;
        Oe = E;
    }
}
```

Figure 8. The definition of edge class

three-dimensional position of the edge's starting point, which is required for every geometry element. The first three constraints of this class serve more as assignments. However the last constraint checks if the opposite edge is really associated with the same wall as this edge.

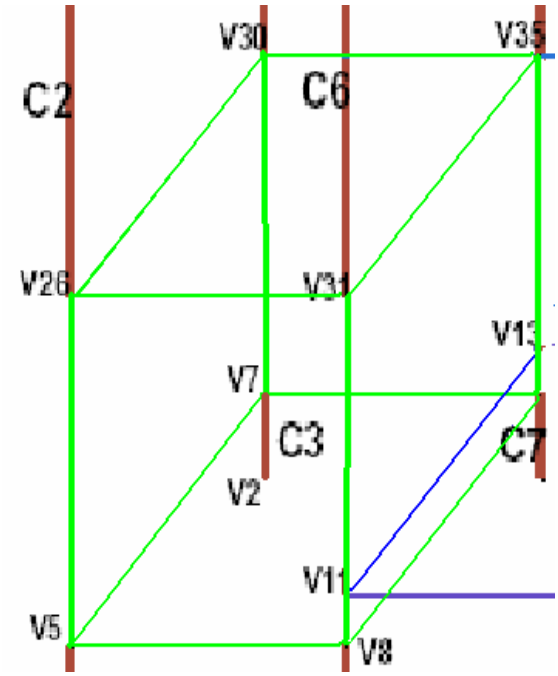
The edge and vertex classes are of great importance for our computation model of the building—because all the other elements rely on geometry directly or indirectly to locate themselves in the three dimensional space.

3.4.2 Assembly

In the Appendix A we provide similar class definitions created for the remaining building elements. The final step consists of creating and assembling all the building elements together. What we have is another instance of an assembly class. We can see from Figure 9 that in order to model a very simple building, we need to declare not only the structure elements but also some architecture elements. Those elements such

as space, surface and edge are mostly virtual elements so they cannot be seen in the real building.

The previous example of level shows that when we create an element we need to specify what the related elements are. Figure 10 shows that in order to define column C6 we need to provide the beginning and end point of that column. In a similar way we can specify the whole structure of the building by merely creating all the elements; the order is not important.




```
space Spaces00;
  slab[] Slabs0;
  slab[] Slabs1;
  surface[] Surfaces00;
  beam[] Beams01;
  slab Slabs00;
  surface Surfaces003, Surfaces002;
  surface Surfaces001, Surfaces000;
  horizontalSurface Ceiling00;
  horizontalSurface Floor00;
  beam B014, B013,B012, B011 ;
  edge Edge003, Edge002;
  edge Edge001, Edge000;
  wall Wall007, Wall002;
  wall Wall001, Wall000;
  column    C8,C4, C6, C2;
  vertex    V8,    V7,    V6, V5;
  vertex V26, V30, V36, V31;
  level Level0;
```

Figure 9. All the elements declared corresponding to the simple building

3.4.3 Result

After writing the definitions of all the elements and creating the building object,

the user is just a few steps away from getting the result, namely compiling and running the model, and possibly debugging. The Cob environment provides tools for all these functions. Thus, the whole process consists of three steps: First, create all the building elements following the overall relations diagram (Figure 4). Second, create the building structure by instantiating all the building elements. Third, compile, run and get the results.

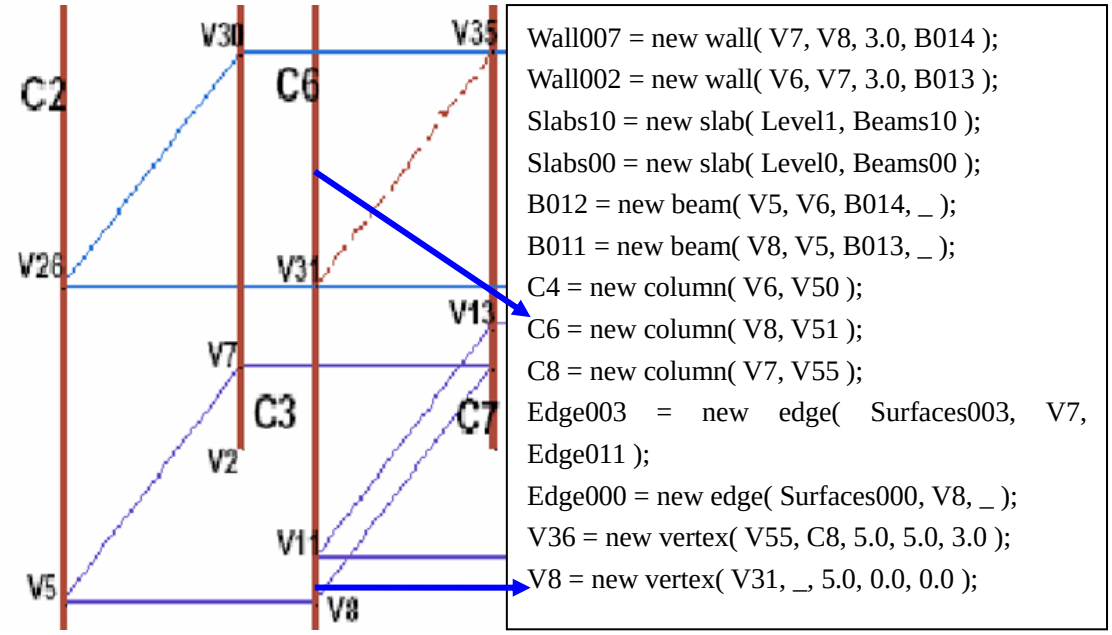


Figure 10. The creation of some elements in the sample building

The output now is the total volume and square footage of the building. Indeed, we can also do other meaningful analysis on the building. We believe the new constrained object model of a building will have a more comprehensive semantic meaning and stronger computational power.

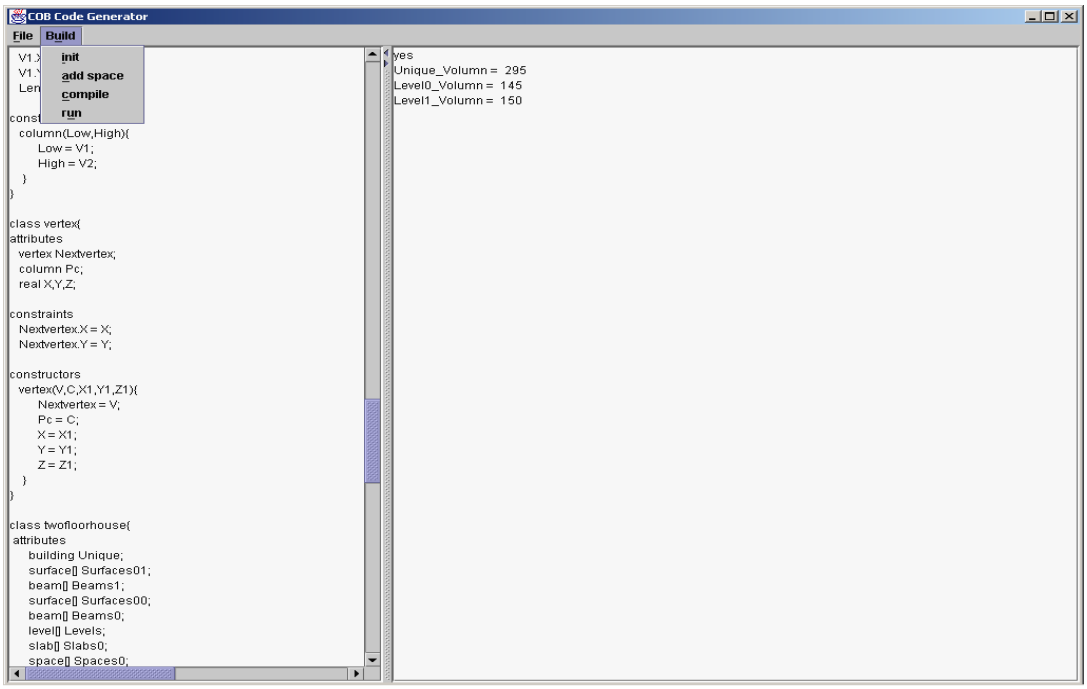


Figure 11. The code and the output of a sample building

3.5 Higher-Level Approach

Although usually the class definitions for building elements are pre-defined or reusable, the building structure is very complicated in the sense of requiring every vertex and every wall to be specified. Needless to say, it is highly error-prone and time consuming. For example, even for a simple two-level house with five spaces, there are 127 building elements for the user to input, and for every element the user needs to specify the exact location and the relation with other elements. It is obvious that for a larger building, the input task would be beyond the capability of a normal human being. We realize the importance of this problem and our solution is to create the building structure with a component-based methodology.

Our approach is to group certain kinds of elements together and create the building structure based on that higher-level virtual component. That is to say, we no longer have to start from the very beginning and create the building "brick by brick." Instead we will start from a higher level and create the building "room by room." Here the room actually includes the following building elements in the previous definition: space, surface, wall, edge, beam, vertex, slab, etc. We group them together and encapsulate them so that users only have to be concerned about how to place the room one after another. However, notice that the Cob model for an intelligent building remains the same, which means we still need the Cob code similar to the previous one. Here we need a 'meta level' middleware to seamlessly map and generate the lower level Cob code from the component-based design.

3.5.1 Meta-level Layer

This meta-level middleware needs to have a user-friendly interface to gather the user input, create a new room, and finally connect it to the existing building. This process is very complicated, since most building elements are interleaved with each other in the original relations. For example, the new room and the old one will share the same wall and the other two walls are also connected with the "base" at one end. Since beam, edge and vertex are also related to the wall, all these elements are somewhat connected to the old one. When we merge the new room into the base several checks and reassignments need to be done. However, it is still feasible, with some assumptions, to simplify the situation and develop an environment for component-based intelligent building representation (Figure 12).

The merge process is that every time the user creates a new room, we will declare a new space variable. After that, we will initialize all elements associated with it step by step: horizontalSurface, surfaces, walls, edges, beams, etc. The key point is that after the new room is added, some part of the old building is also changed. For example, the beams of the adjacent space will be longer, since it will also support new walls. In order to support this kind of operation efficiently and conveniently, we use Java to write the meta-level software. The advantage is that we can map one-to-one from the classes in Java to the classes in Cob, and hence we also have building, space, level, etc., in Java. Thus when we generate Cob code, we just map it back. However,

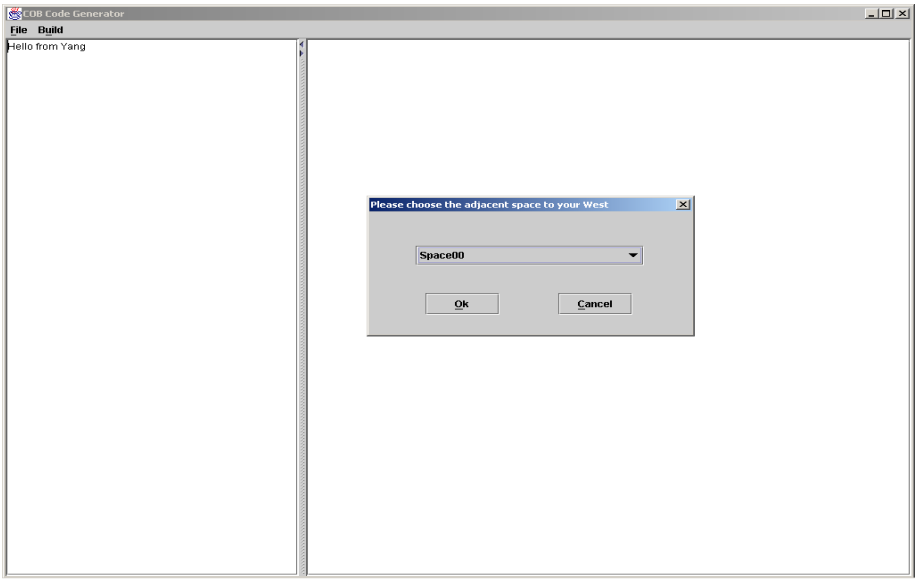


Figure 12. The interface prompts the user to specify the "base room"

the Java classes support some features different from the corresponding Cob class, such as change of attribute values. We also have a registry-like class called ComponentManager, which will provide methods to locate the building elements we need to reference, such as by name. Thus, every time we need to find the element

belonging to the old room, we resort to ComponentManager and then either change its attribute or assign its value to the new room. In this way, we successfully implement the meta-level software, use it to generate an intelligent building room-by-room, and get the results just by clicking *Run* in the build menu.

Chapter 4

Dynamic Constrained Object

4.1 Introduction and Concept

Our constrained object paradigm involves modeling a component as an object possessing a certain set of attributes as well as a set of behavioral laws, which are thought of as constraints that the component must satisfy at all times [10]. Using this approach, we have successfully modeled an application in the architecture domain. However, most of the problems that have been modeled in constrained objects paradigm have been "static," i.e., DC circuits, trusses, etc. The dynamic systems we consider in this chapter are those whose state changes with time. In some instances, this state-change can be characterized in a mathematical way, such as the behavior of AC circuit. In other examples, such as in hydrologic modeling of rain fall and runoff, the rain-fall over time can only be provided explicitly as time-series data because it is largely random. In modeling dynamic systems in general, we also need to maintain information regarding previous states and also enforce constraints that relate a state to those of its previous or succeeding states.

Recently the concept of a 'series' variable was introduced to deal with such dynamic behavior [10]. The series variable can be used to refer to a sequence of values. A reference to a series variable x gives its current value, but a reference such as x' refers to its previous value, whereas a reference such x' refers to its next value. Additionally, we can declare a class as `dynamic` to highlight that the fact that the class

that has series variables in it. In the next section we describe D-Cob, and extension of Cob for modeling dynamic constrained objects.

4.2 D-Cob Syntax and Usage

The syntax of the declaration of a “series” variable in D-Cob is as follows:

```
program ::= class_definition +
class_definition ::= [ abstract ][ dynamic ]class class_id
                     [ extends class_id ] { body }
body ::= [ attributes attributes ]
         [ constraints constraints ]
         [ predicates pred_clauses ]
         [ constructors constructor_clause ]
attributes ::= decl ; [ decl ; ] +
decl ::= type_id_list | series_decl
series_decl::= series attribute_id = series_type
series_type ::= term | [ terms ]
type ::= [ series ] primitive_type_id | class_id | type [ ]
primitive_type_id ::= real | int | bool | char | string
id_list ::= attribute_id [ , attribute_id ] +
```

For example, a series variable `Height` as:

```
series int Height;
```

There are two ways to initialize a series variable: the most common method is to assign an array of values to the variable during its declaration. This assignment specified the value of a series variable at any time slot by the corresponding value of this array. Here is an example:

```
series int Height = [12,14,16,20,26,34,49,59,66,69,72];
```

There is a simplified special version to initialize the series variable when the

value of that series variable does not change. We can just write:

```
series int Height = 68;
```

This form of initialization means the series variable `Height` is always 68, no matter what time it is. However, there are situations where only the first one or two values of the series variable are decided. Therefore, we also support a special form of assignment that we call a boundary initialization. The boundary initialization allows a form of indexing <i> in conjunction with a series variable. E.g., we can say:

```
Height<1> = 0;
Height<2> = 1;
```

After the initialization or assignment we can also use the backquote (‘) or forward quote (’) keyword in order to assign the previous or next value of a series variable:

```
Height’ = 70;
Height‘ = 78;
```

Next we provide a simple example to illustrate how to use dynamic constrained objects as well as constraints over series variables.

4.2.1 Newton’s Method

Our first example shows how to compute a positive real number’s square root by applying Newton’s method of approximation. The formula for the square root of n is:

$$x_n = \frac{x_{n-1} + n / x_{n-1}}{2}.$$

This approximation formula is actually in a series form, which perfectly matches the series variable concept.

```
dynamic class sqr {
   attributes
      series real SQ;
      int Num;
      real EPS, Ans;

   constraints
      SQ =  ( SQ' + Num/SQ' ) / 2;
      Ans =  SQ` :- abs(SQ` - SQ) <= EPS;

   constructors sqr(N, E) {
      Num = N;
      EPS = E;
      SQ<1> = N + 1;
   }
}
```

From this definition we defined a series variable `SQ`, which is of the `real` type. In the constructor part we initialize `Num` and `EPS` variables, which represent the number whose square root we are seeking and the accuracy level, respectively. More importantly, we assign the boundary condition of the series variable `SQ` as `N+1`, using the boundary initialization, which essentially means that the first element in the series is `N+1`. In the constraint part we now have a very simple mapping between Newton's method and the equation involving `SQ` and its previous value. The modeling also makes it easy to determine when we have got the correct answer; we simply check that the difference between two consecutive values in the series is less than accuracy level. This program is concise and runs predictably after we translate it into CLP(R). We will show the translating methodology in the next chapter.

4.3 AC Circuit Model

We have already seen how to model DC circuits with the current version of Cob [3]. One law behind the model is Ohm's law, which describes a linear relationship between voltage and current on a resistor. Another is Kirchoff's law, which asserts that the sum of all currents running through a node is zero. We model the DC circuit by creating node objects to connect all the resistors or voltage sources together and applying Kirchoff's law as the node constraint. This model works very well and there is also a visualization tool for the user to draw a DC circuit and obtain the answer visually. Now we will model a different kind of circuit, an AC (alternating current) circuit. The difference is that the voltage of the source will change over time and we will have capacitors and inductors.

4.3.1 Capacitor and Inductor

Just as Ohm's law specifies the behavior of a resistor, we also have laws for capacitor and inductor [5]. The electrical law for a capacitor is $I = C \times \frac{dV}{dt}$ where C is the capacitance of the capacitor. This law is actually a differential equation involving current and voltage, and Cob is not immediately suitable for this computation. Our new approach in D-Cob is that we approximate it by a difference equation, which is more suitable to depict by series variables. Therefore the differential equation becomes $I = C \times \frac{\Delta V}{\Delta t}$.

Now we can declare both I and V as series variables and we assume that the time interval is just the time difference unit of the two adjacent values in the series. Therefore we now have the Cob code for this law I = C $\times$(V – V').

The law for an inductor is V = L$\times \frac{dI}{dt}$, and this can be transformed similarly into the Cob code V = L $\times$(I – I').

The above approach is a very concise way to describe the electrical laws for capacitors and inductors, and thus the whole model for an AC circuit is formulated in a simple way.

4.3.2 D-Cob Model for AC Circuit

Now we will show an example of a simple AC circuit and its Cob model. Figure 13 shows the circuit whose voltage source provides AC voltage.

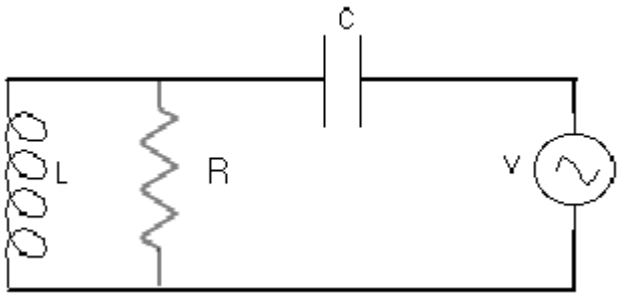


Figure 13. Sample AC circuit

In this circuit a 0.1-henry inductor is in parallel with a 10-ohm resistor and in series with a 0.1-farad capacitor. The frequency of the AC voltage source is ω=0.1 within a range of ± 10 volts, and hence is specified as $10\times\sin(10\times t)$. We now show the Cob code for this circuit:

```
abstract dynamic class component {
  attributes
    series real I1,I2,V1,V2;
  constraints
        I1+I2=0;
}
```

This is the parent class for all the electrical components, as before with DC circuits. Notice that they are all series variables, which means they will change along with time. There is a constraint I1+I2=0 over them and it will hold at all time by the semantics of constraints over series variables.

```
class resistor extends component{
  attributes
    real R;
  constraints
    V1 - V2 = I1*R;
  constructors resistor(D) {
    R = D;
  }
}
```

```
class capacitor extends component {
 attributes
    Real C;
 constraints
        I1 = C*((V1-V2)-(V1'-V2'));
 constructors capacitor(C1) {
    C = C1;
    V1<1> = 0;
    V2<1> = 0;
 }
}
```

Figure 14. D-Cob code for resistor and capacitor

```
class inductor extends component {
   attributes
      Real L;
  constraints
      V1-V2 = L*(I1-I1');
  constructors inductor(L1) {
      L = L1;
      I1<1> = 0;
  }
}
```

```
class voltageSource extends component{
  constraints
       V2 = 0;
  constructors voltageSource(X) {
     V1 = X;
  }
}
```

Figure 15. D-Cob code for inductor and voltageSource

The above four classes are the four basic electrical components. Their constraint parts are the corresponding translations of the electrical laws we described in the previous section. In addition, we need to specify the boundary initialization to prepare for computing the difference equation. The only difference here is that voltage over a component is now represented by the difference between V1 and V2 instead of one variable.

```
dynamic class componentEnd {
  attributes
    component C;
    series real V, I;
    int End;
  constraints
    V = C.V1 :- End = 1;
    V = C.V2 :- End = 2;
    I = C.I1 :- End = 1;
    I = C.I2 :- End = 2;
  constructors componentEnd(C1, E){
    C = C1;
    End = E;
  }
}
```

```
dynamic class node {
  attributes
    componentEnd [] Ce;
    series real[] V;
  constraints
    sum X in Ce: X.I = 0;
    forall X in Ce: X.V = V;
  constructors node(L) {
    Ce = L;
  }
}
```

Figure 16. D-Cob code for componentEnd and node

The `componentEnd` class is used to represent the two sides of an electrical component. The node class is the class to connect all the componentEnds together and apply Kirchoff's law as a constraint. The law is represented by two universally quantified equations, which describe two simple facts:

1. All the voltages of the component end's that are attached to a node are equal.
2. The sum of all currents running through a node is zero.

```
dynamic class samplecircuit {
  attributes
    resistor R;
    real[] Voltages;
    voltageSource B;
    capacitor C;
    inductor I;
    componentEnd
R1,R2,B1,B2,C1,C2,I1,I2;
    node N1, N2, N3;
constructors samplecircuit() {
    R = new resistor(10);
    C = new capacitor(0.2);
    I = new inductor(0.1);
    Time[1] = 0;
    Voltages = 10*sin(0.1*Time);
    B = new voltageSource(Votages);
    B1 = new componentEnd(B, 1);
    B2 = new componentEnd(B, 2);
    R1 = new componentEnd(R, 1);
    R2 = new componentEnd(R, 2);
    C1 = new componentEnd(C, 1);
    C2 = new componentEnd(C, 2);
    I1 = new componentEnd(I, 1);
    I2 = new componentEnd(I, 2);
    N1 = new node([C1, B1]);
    N2 = new node([B2, R1, I1]);
    N3 = new node([C2, R2, I2]);
  }
}
```

Figure 17. D-Cob code for AC circuit

This is the assembly class, which actually declares and initializes all the electrical components and nodes that connect them. To model a user-defined circuit, only this class has to be modified corresponding to the structure and values of the circuit elements. As we see, this class is large even with only four components in the circuit. However, a drawing tool can help the user draw the circuit and generate the above code mechanically.

We have shown a model for an AC circuit using dynamic constrained objects. Next we provide a different illustration.

4.4 Nerve Cell Behavior Model

In the early 1950s, Hodgkin and Huxley were awarded the Nobel Prize for their work on nerve cells. They did experiments on the giant axon of the squid and found the details of the electrophysiology characteristic of the cell membrane [2]. They found that a semi-permeable cell membrane separates the interior of the cell from the extra-cellular liquid and acts as a capacitor. If an input current I(t) is injected into the cell, it may add further charge on the capacitor, or leak through the channels in the cell membrane that consists mainly of Cl^- ions. Because of active ion transport through the cell membrane, the ion concentration inside the cell is different from that in the extra-cellular liquid. They further discovered that there are three different types of ion flow, viz., sodium, potassium, and a leak current. The ion gates are protein channels that regulate ion flow into and out of the cell. There are three gates that are associated with the action potential: m, h, and n; the m and h gates control sodium flow, while the n gate controls potassium flow.

4.4.1 Mathematical Model

Under the Hodgkin and Huxley model, the total current flow through a cell membrane is the sum of capacitive and resistive current flows. The capacitive current is described by the equation: $I = C \times \frac{dv}{dt}$, where C and V denote the membrane capacitance and trans-membrane potential. A resistive current depends on the transmembrane potential, V, the equilibrium potentials of the individual ions E_{ion} and the conductance of the ion channels g_{ion}. All resistive currents can be generally described by equations of the form:

$I_{ion} = g_{ion} \cdot (V - E_{ion})$.

Hodgkin and Huxley's experiments further demonstrated that among the three currents only sodium and potassium are time variant. Therefore the total resistive current can be described by:

$$I_{res} = g_{Na} \times m^3 \times h \times (V - E_{Na}) + g_k \times n^4 \times (V - E_K) + g_L \times (V - E_L)$$

where m, h and n are three coefficients that also depend on V, which satisfies the following general equation:

$$\frac{dx}{dt} = \alpha_x(v) \times (1 - x) - \beta_x(v) \times x$$

in which x stands for m, h or n and α_x and β_x are coefficients depend that on V and associate with m, h or n respectively.

4.4.2 D-Cob Model for Nerve Cell Behavior

We now show how to use dynamic constrained objects to solve this recursive specification by using the parameters that Hodgkin and Huxley found in their experiment [9]. Just as in the AC circuit, we use difference equations in D-Cob to model differential equations. The main difference is that we have second order differential equations. We stratified the equations into two levels since m, h and n depend on V. Therefore m, h and n are at a second level and we make them depend on V' instead. Figure 18 shows the code for the Hodgkin and Huxley model. We use the series variable V to represent the voltage between the inner and outer side of the cell and I for the current. We use M, H and N to represent each coefficient for the different resistive current. We can see that the mapping from the mathematical form into a Cob expression is quite straightforward. We show how to execute a D-Cob program and

obtain the result in the next section.

```
dynamic class HodgkinHuxley {
   attributes
      series real V,M,H,N;
      real I;
   constraints
      V-V' = I- (120*pow(M,3)*H(V+155) + 36*pow(N,4)*(V-12) +
              0.3*(V+10.6));
      M-M' = (1-M)*((V'+25)/10)/(exp((V'+25)/10)-1) - M*4*exp(V'/18);
      H-H' = (1-H)*0.07*exp(V'/20)- H/(1+exp((V'+30)/10));
      N-N' = (1-N)*0.1*((V'+10)/10)/(exp((V'+10)/10)-1) -
               N*0.125*exp(V'/80);

   constructors HodgkinHuxley(A) {
       I = A;
      V<1> = 0;    M<1> = 0;    H<1> = 0;    N<1> = 0;
   }
}
```

Figure 18. The D-Cob code for the Hodgkin and Huxley model

4.5 Dynamic Cob Computational Model

4.5.1 Synchronize and Translate

The advantage of dynamic Cob is its ability to compute a series of results over time. Now we will show how to execute a dynamic Cob program to get the results. There usually is more than one series variable declared in a dynamic Cob program, hence they must be synchronized to avoid confusion in order. Essentially, we should make time to progress at the same pace for all series variables at the same interval. To achieve this we have an internally defined a class called *dobject* (see Figure 19).

All classes that are declared dynamic and their subclasses will inherit from this dobject class automatically. The semantics of the dobject is that it defines a time slot with a length specified at the time of creation. The constraint:

```
forall X in 1..Len: TimeSlots[X] = X;
```

will initialize the `TimeSlots` array the same as its index, e.g. TimeSlots[8] = 8. Another key array variable is `Time`. This is the variable that all the series variables use to synchronize with each other when necessary. We used a special form, which we will discuss in detail in the next section, to initialize the whole array.

```
class dobject {
  attributes
    int[] Time,TimeSlots;
    int Len,Interval;

  constraints
    forall X in 1..Len:
        TimeSlots[X] = X;
    forall T > 1 in TimeSlots:
        Time[T]=
        Time[T-1]+Interval;

   constructors dobject() {
      Time[1] = 0;
   }
}
```



```
dobject([],[TimeSlots,Time,Len,Interval])
:-
   Len = 10,
   Interval = 0.1,
   makearray(1,TimeSlots),
   makearray(1,Time),
   Cob80  =  0,
   index(Time,1,Cob80),
   makelistfromto(1,Len,Cob82),
   coball17(Cob82,[Cob81,TimeSlots]),
   TimeSlots = [_|SSlot],
   coball18(SSlot,
            [Interval,Cob85,Time]).
```

Figure 19. Cob and CLP(R) code for dobject

In another step to synchronization we translate all series variables into a corresponding array with one more dimension, i.e.:

```
series real SQ;          ➔   real []  SQ;
series real [10]  P;     ➔   real [10] [] P;
```

This translation captures the semantics of the series variable. In order to translate a constraint involving a series variable, we will first wrap it with a universal quantifier: `forall T in TimeSlots`. Secondly, we transform any series variable `X` in the constraint into `X[T]`, captures the semantics that a series variable must obey its constraints for all time. We illustrate the translation with Newton's method for square roots (see Figure 20). In this example, the constraint

```
SQ = ( SQ' + Num/SQ' ) / 2
```

is transformed into

```
SQ[T] = (SQ[T-1]+Num/SQ[T-1])/2
```

because `SQ'` means the previous value of `SQ`. We can see that the mapping for a series variable `X`

`X` → `X[T]`

`X'` → `X[T-1]`

`X‘` → `X[T+1]`

can be applied to any constraints involving series variables, and thus can serve as a standard in translating D-Cob to Cob. However, it is not the final step, since the current Cob system is not designed and optimized for this kind of computation.

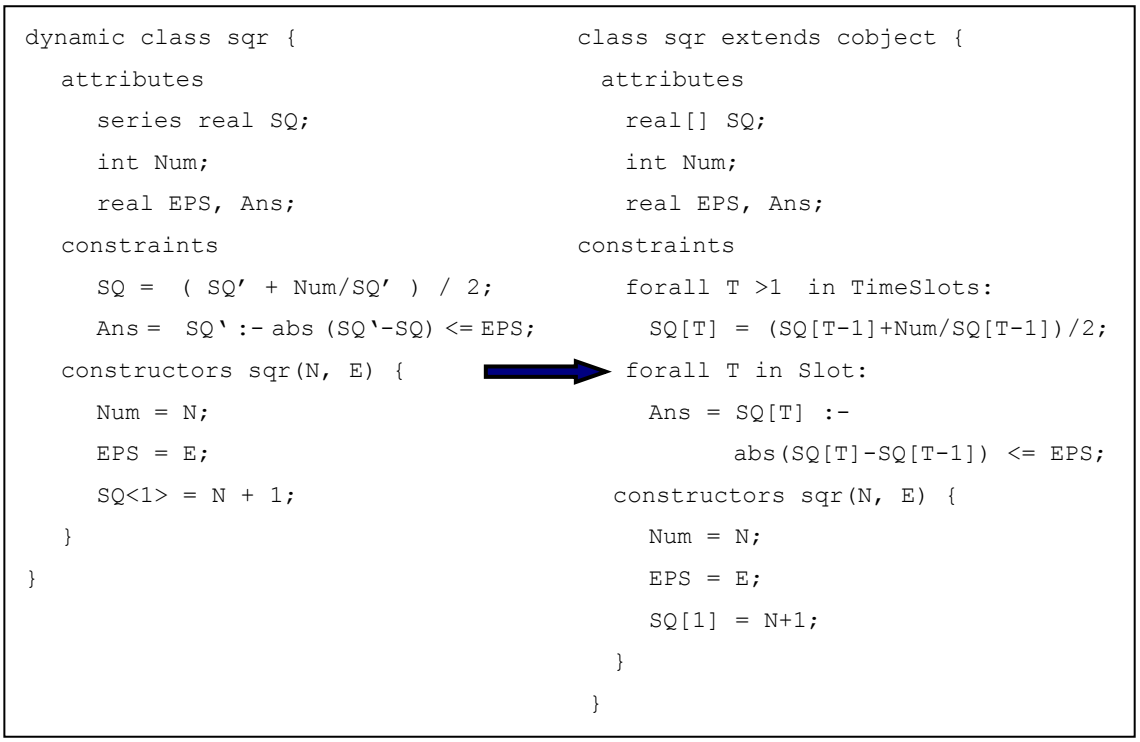


Figure 20. Newton's method D-Cob and its translation into Cob

4.5.2 Compute in CLP(R)

One of the problems is the boundary initialization. If a series variable is involved in difference equations, it most likely will need to have boundary initialization. Therefore we employ a special form; instead of `forall T in TimeSlots`, we use

```
forall T > 1 in TimeSlots
```

which will not apply the constraints on the first element of the array. Here is the translation of these two forms in the CLP(R).

```
forall T in TimeSlots  ➔  coball2(TimeSlots,[EPS,Num,Cob3,SQ,Len,Ans])

forall T>1 in TimeSlots ➔

              TimeSlots=[_|SSlot], coball2(SSlot,[EPS,Num,Cob3,SQ,Len,Ans])
```

In the similar way, we use `forall T > 2 in TimeSlots` to adapt to the "double" boundary initialization.

Although the mapping process is almost finished, in order to run the D-Cob program we still need two more system variables. The first is `Len` in *dobject*, which determines how much iteration of the series variables will be computed. This value can be determined several ways:

1. In the Newton's method case, the number of iterations is determined by an end condition (`error<EPS`). Thus we can set a very high value, which serves as a boundary, although in most cases it will not reach this value.
2. In the AC circuit and other examples that use a constant period, the stages depend on the frequency 'ω' of the voltage source and the time interval. That is because we only need to compute one cycle of system performance and the result will repeat itself.
3. In other cases users can specify how many "steps" they want to observe.

The second variable is the time `Interval` between each iteration. The time is a variable that is initialized in the *dobject* class and every class that extends *dobject* will use it to synchronize with others. As we see, time is actually an array that starts from 0*t* and increases a certain amount for every index. This amount, called `Interval`, needs to be determined by the user before starting to run the program. The choice of the value of `Interval` is a trade off between performance and accuracy. That is because the semantics of `Interval` is the sampling rate for the model, i.e., how often we sample the real system. Therefore the trade off is as follows:

1. The smaller the interval, the more accurately we simulate the real system.
2. The larger the interval, the better the modeling performance.

4.5.3 Results for AC Model

To solve the circuit of Figure 13, we choose pick `Len` as 200 and `Interval` as 0.01. We translated the D-Cob program shown in Figure 17 into Cob using the mapping shown in 4.5.2 section, and further translated it into CLP(R).

To show the results we focus on the voltage on the capacitor. We first choose the input voltage of voltage source as `V = 10×sin(10×t)`. We also used the standard method in electrical engineering to solve the differential equations and obtained the actual formula of the voltage on the capacitor. First we show the result of the mathematical formula:

$$V_C = -\frac{100}{399} \times e^{-\sqrt{t}} \times \sqrt{399} \times \sin(\tfrac{1}{2} \times \sqrt{399} \times t) +$$

$$100 \times e^{-\sqrt{t}} \times \cos(\tfrac{1}{2} \times \sqrt{399} \times t) - 100 \times \cos(10 \times t) + 10 \times \sin(10 \times t)$$

The solution is depicted in Figure 21.

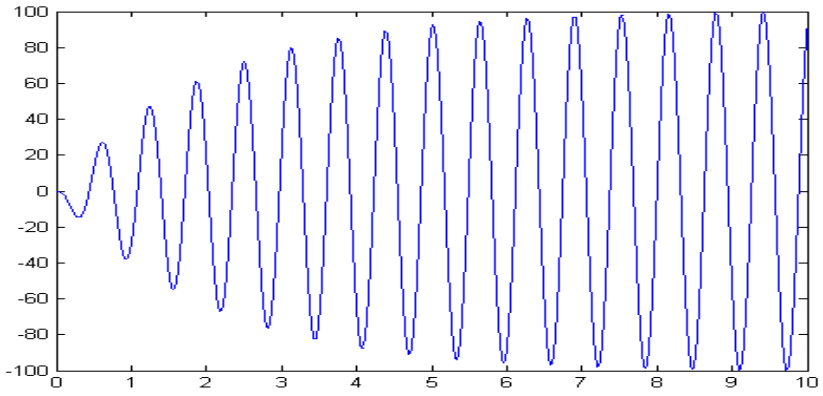


Figure 21. Diagram from the numerical solution of AC circuit V_B = 10*sin(10*t)

In order to show the results more thoroughly, we changed the input voltage into $V = 10 \times \sin(\pi \times t)$. Now the mathematical formula is changed to:

$$V_C = \pi \times 0.012134723 \times e^{-\sqrt{t}} \times \cos(\tfrac{1}{2} \times \sqrt{399} \times t)$$

$$- 0.0055577143 \times e^{-\sqrt{t}} \times \sqrt{399} \times \sin(\tfrac{1}{2} \times \sqrt{399} \times t)$$

$$- 12.134723 \times \cos(\pi \times t) + 0.35312367 \times \sin(\pi \times t)$$

The solution is depicted in Figure 22.

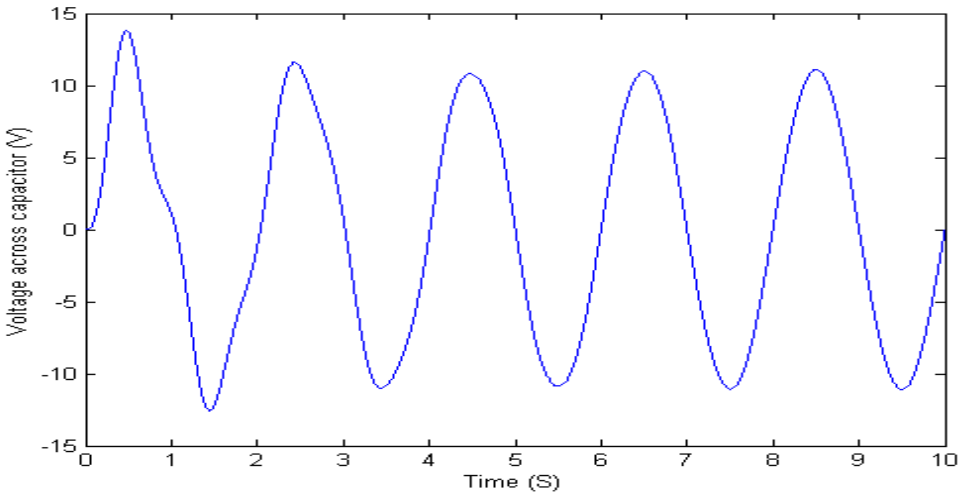


Figure 22. Diagram from the numerical solution of AC circuit V_B = 10*sin(π*t)

The AC circuit example shows that D-Cob is capable of solving problems whose mathematical solution is complicated. The modeling is simple and straightforward, and the result is accurate.

4.5.4 Results for Nerve Cell Behavior Model

We can also follow the process we mentioned in section 4.5.2 to run the nerve cell behavior example. Figure 23 is the intermediate Cob code that we translated by the rule for mapping from D-Cob to Cob. (We omit the *dobject* class as it is the same everywhere.) After translating the Cob into CLP(R) code, we can run this model and check the result. We choose a current of 0.5 amp and check the V, which is a key

variable.

We next run the example with a current of 3 amps applied on the cell and show the results in Figure 24. It is also possible to adjust the coefficient of the Hodgkin-Huxley model and get useful cell nerve function data by running the dynamic constrained objects model.

From the results of nerve cell behavior model and the AC circuit model we can see that the dynamic constrained object paradigm is capable of modeling dynamic systems whose states are changing, and provides an alternate solution for differential equations. The computational model is also simple and easy to extend from the static domain.

```
class HodgkinHuxley extends dobject{
   attributes
      real[] V,M,H,N;
      real I;
   constraints
       forall T>1 in TimeSlots:
          V[T]-V[T-1] = I[T]-(120*pow(M[T],3)*H[T]*(V[T]+155)+36*
                        pow(N[T],4)*(V[T]-12)+0.3*(V[T]+10.6))/Interval;
       forall T>1 in TimeSlots:
          M[T]-M[T-1] = (1-M[T])*((V[T-1]+25)/10)/(exp((V[T-1]+25)/10)-1)
                        - M[T]*4*exp(V[T-1]/18))/Interval;
       forall T>1 in TimeSlots:
           H[T]-H[T-1] = ((1-H[T])*0.07*exp(V[T-1]/20)-
                          H[T]/(1+exp((V[T-1]+30)/10)))/Interval;
       forall T>1 in TimeSlots:
          N[T]-N[T-1]=(1-N[T])*0.1*((V[T-1]+10)/10)/(exp((V[T-1]+10)/10)
                         -1)-N[T]*0.125*exp(V[T-1]/80))/Interval;

   constructors HodgkinHuxley(A) {
        V<1> = 0;     M<1> = 0;      H<1> = 0;      N<1> = 0; I = A;
   }
}
```

Figure 23. The Cob code translated for the Hodgkin and Huxley model

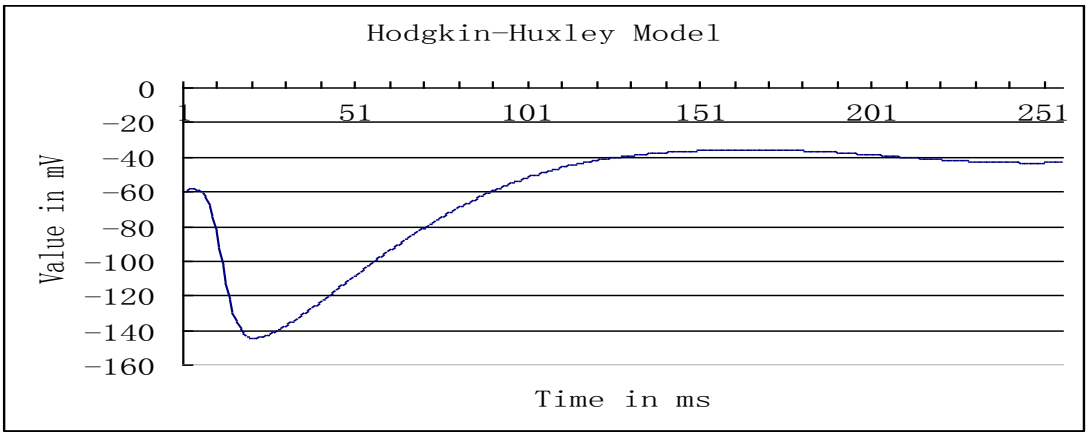


Figure 24. A set of data obtained from the Hodgkin-Huxley Model

Chapter 5

Conclusions and Further work

The idea of combining constraints and objects is applicable for modeling a variety of systems [4, 5]. In this thesis we explored examples of static as well as dynamic systems. In the former case, we showed a novel illustration of how constrained objects can be applied to the building architectural domain, by showing a detailed model for an "intelligent" representation. In our approach, the user can input the model of a building in a relatively easy way and achieve the comparable results as in a component-based approach. This not only shows that the idea of intelligent building representation is correct in general, but also that the Cob environment can adapt to the architectural domain. Our meta-level software also shows how to apply component-based design with Cob.

In the area of modeling dynamic systems, we showed examples in two different domains (AC circuits and nerve-cell modeling) to illustrate the features of the D-Cob language. A key idea in both examples is the use of difference equations to model differential equations. The execution process is simple and the results are encouraging.

We have seen that changing the values of attributes is very common during the process of building and experimenting with a model. Such changes are made using meta-level software, but it may be desirable to make them in Cob itself. This is a fundamental dilemma because there is no assignment in declarative semantics. The current approach to deal with the changing is to have variables that can refer to its previous and future value. However, we still need to explore more examples to see if

this concept is sufficient for more broad domains. The immediate improvement can be a compiler to translate from dynamic cob directly into CLP(R).

The Cob environment is relatively slow for handling a large amount of objects due to its limited computational efficiency. It is obvious that we need to improve the efficiency in order to tackle real-world buildings. One approach may be to support an interface with state-of-the-art solvers in specific domains. This will improve the solving ability so that we can solve more problems with efficiency. Another more fundamental approach is to migrate the underlying system to C/C++, which is to some extent more efficient and friendly to most problems. In this way, we can find and improve the bottleneck of the system and make it more robust and efficient.

References

1. Barták, R. “Constraint Programming: In Pursuit of the Holy Grail,” in *Proceedings of the Week of Doctoral Students*, Part IV, MatFyzPress, Prague, pp. 555-564, June 1999.

2. Guckenheimer, J. and Oliva, R. A. “Chaos in the Hodgkin–Huxley Model,” *SIAM J. Applied Dynamical System*. Vol. 1, No. 1, pp. 105–114.

3. Jayaraman, B. and Tambay, P. “Modeling Engineering Structures using Constrained Objects,” *Fourth International Symposium on Practical Aspects of Declarative Languages*, pp. 28-46, Springer-Verlag, 2002.

4. Khemlani,L., Timerman, A., Benne,B. and Kalay Y.E. “Intelligent Representation for Computer Aided Building Design,” *Automation in Construction* 8(1), pp. 49-72, 1998.

5. Lovell, B., Downs, T. “Capacitors and Inductors,” *Lecture Notes for ENG1030 Electrical Physics and Electronics*, The University of Queensland, Brisbane, Australia. http://www.itee.uq.edu.au/~engg1030/lectures/1perpage/lect5.pdf

6. Nassara, K., Thabetb, W., Beliveauc, Y. “Building Assembly Detailing using Constraint-Based Modeling,” *Automation in Construction* 12, pp. 365-379, 2003.

7. Nassara, K., Thabetb, W., Beliveauc, Y. “A procedure for Multi-Criteria Selection of Building Assemblies,” *Automation in Construction* 12, pp. 543-560, 2003.

8. Otter, M., Elmqvist, H. “Modelica – Language, Libraries, Tools, Workshop and EU-Project RealSim,” *Simulation News Europe*, pp. 3-8, December 2000.

9. Pollinger, T. “Hodgkin-Huxley Cells,” http://www.math.rutgers.edu/courses/338

/hodhuxPrg/html/.

10. Raux R.J., Jayaraman B. “Modeling Dynamic systems with Constrained Objects,” Dept. of Computer Science and Engineering, University at Buffalo, Technical Report 2004-05, 2004.

11. Tambay, P. and Jayaraman, B. “The Cob programmer’s Manual,” Dept. of Computer Science and Engineering, University at Buffalo, Technical Report 2003-01, February 2003.

12. Tambay, P. “Constrained Objects for Modeling Complex System,” *Ph.D Dissertation*, Dept. of Computer Science and Engineering, University at Buffalo, February 2004.

13. Mayne, R., Margolis, S. “Introduction to Engineering,” *McGraw-Hill*, 1982

Appendix A

```
class building{
 attributes
   level[] Levels;
   real Volume;
 constraints
   Volume =
    sum X in Levels: X.Volume;
 constructors
   building(L){
     Levels = L;
   }
}
```

```
class level{
 attributes
    building Pb;
    space[] Spaces;
    slab[] Slabs;
    real Volume;
 constraints
   Volume =
    sum X in Spaces: X.Volume;
constructors
   level(B,Sp,Sl){
         Pb = B;
         Spaces = Sp;
         Slabs = Sl;
    }
}
```

Class definition of *building and level*

```
class space{
 attributes
   level Pl;
   surface[] Surfaces;
   horizontalSurface Floor;
   horizontalSurface Ceiling;
   real Height,Width,Length;
   real Volume;
   real X1,X2,Y1,Y2;
 constraints
   forall S in Surfaces:
          S.Height = Height;
   X2 = Surfaces[1].Edge.HeadX;
   X1 = Surfaces[2].Edge.HeadX;
   Y2 = Surfaces[3].Edge.HeadY;
   Y1 = Surfaces[2].Edge.HeadY;
   Length = X2-X1;
   Width = Y2-Y1;
   Volume = Height*Length*Width;
   Floor.Z-Floor.Distance+Ceiling
   .Z-Ceiling.Distance = Height;
 constructors
   space(L,S,F,C){
        Pl = L;
        Surfaces = S;
        Floor = F;
        Ceiling = C;
    }
}
```

Class definition of *space*

```
class surface{
   attributes
      space Ps;
      wall Wall;
      edge Edge;
      opening[] Openings;
      real Height;
   constraints
      Height = Wall.V1.Z+Wall.Height-Edge.Z;
 constructors
   surface(S,E,O,W){
      Ps = S;   Edge = E;   Openings = O;    Wall = W;
   }
}
```

Class definition of *surface*

```
class edge{
   attributes
      surface Ps;
      vertex Vertex;
      edge Oe;
      real Z;
      real HeadX, HeadY;
  constraints
      Z = Vertex.Z;
      HeadX = Vertex.X;
      HeadY = Vertex.Y;
      Oe.Ps.Wall = Ps.Wall;
 constructors
   edge(S,V,E){
       Ps = S;  Vertex = V;   Oe = E;
   }
}
```

Class definition of *edge*

```
class opening {
    attributes
        string Type;
        surface Ps;
        real Width,Height,Distance;

    constructors
        opening(T,S,W,H,D){
            Type = T;
            Ps = S;
            Width = W;
            Distance = D;
            Height = H;
    }
}
```

Class definition of *opening*

```
class horizontalSurface{
    attributes
        space Ps;
        slab Slab;
        real Distance;
        real Z;
    constraints
        Z = Slab.Z+Distance;
    constructors
        horizontalSurface(S,Sl,Dis){
            Ps = S;
            Slab = Sl;
            Distance = Dis;
   }
}
```

Class definition of *horizontalSurface*

```
class slab{
    attributes
        level Pl;
        beam[] Peripherialbeams;
        real Z;
    constraints
        forall B in Peripherialbeams: B.Z = Z;
    constructors
        slab(L,B){
            Pl = L;
            Peripherialbeams = B;
        }
}
```

Class definition of *slab*

```
class beam{
    attributes
        vertex Vertex1,Vertex2;
        beam Parallelbeam;
        beam Continuousbeam;
        real Z;
    constraints
        Vertex1.Z = Z;
        Vertex2.Z = Z;
    constructors
        beam(V1,V2,Pb,Cb){
            Vertex1=V1;
            Vertex2=V2;
            Parallelbeam = Pb;
            Continuousbeam = Cb;
    }
}
```

Class definition of *beam*

```
class wall{
    attributes
        vertex V1;
        vertex V2;
        real Height;
        beam Beam;
    constraints
        V1.Z = V2.Z;
        V1.Z = Beam.Z;
    constructors
        wall(F,S,H,B){
            V1 = F;
            V2 = S;
            Height = H;
            Beam = B;
    }
}
```

Class definition of *wall*

```
class column{
    attributes
        vertex V1,V2;
        real Length;
    constraints
        V1.X = V2.X;
        V1.Y = V2.Y;
        Length = V2.Z - V1.Z;
    constructors
        column(Low,High){
            V1 = Low;
            V2 = High;
    }
}
```

Class definition of *column*

```
class vertex{
    attributes
        vertex Nextvertex;
        column Pc;
        real X,Y,Z;
    constraints
        Nextvertex.X = X;
        Nextvertex.Y = Y;
    constructors
        vertex(V,C,X1,Y1,Z1){
            Nextvertex = V;
            Pc = C;
            X = X1;
            Y = Y1;
            Z = Z1;
    }
}
```

Class definition of *vertex*