Modeling Engineering Structures with Constrained Objects[1]

Bharat Jayaraman
Pallavi Tambay
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260-2000

E-Mail: {bharat,tambay}@cse.buffalo.edu
Phone: (716) 645-3180 x 111
Fax: (716) 645-3464

Abstract. We present a novel programming language based on the concept of *constrained objects* for compositional and declarative modeling of engineering structures. A constrained object is an object whose internal state is governed by a set of (declarative) constraints. When several constrained objects are aggregated to form a complex object, their internal states might further have to satisfy interface constraints. The resultant behavior of the complex object is obtained by a process of logical inference and constraint satisfaction. Thus, constrained objects are a declarative counterpart of traditional objects (found in object-oriented languages). Our modeling paradigm supports constraints, including quantified and conditional constraints, as well as preferences. We show via several examples that, for the domain of engineering modeling, the paradigm of constrained objects is superior to both a pure object-oriented language as well as a pure constraint language. The current prototype includes tools for authoring the constrained-object class definitions using a modeling tool called CUML (Constraint-based UML); a compiler that translates a CUML specification into an executable CLP(R) program; and a domain-specific visual interface for building and testing constrained objects for the domain of electric circuits. We believe that such tools can be used for pedagogic purposes as well as for more advanced modeling applications.

Keywords : Constrained Objects, Modeling, Object-oriented, Constraint Logic Programming

1 Introduction

In this paper we present a programming language and execution environment that will facilitate a principled approach to the modeling of complex systems. A domain of particular interest is that of engineering entities such as circuits, trusses, gears, mixers, separators, etc. Modeling such entities involves the specification of both their structure and behavior. In modeling structure, it is natural to adopt a compositional approach since a complex engineering entity is typically an assembly of many components. From a programming language standpoint, we may model each component as an *object*, with internal attributes that capture the relevant features that are of interest to the model. The concepts of classes and hierarchies found in object-oriented (OO) languages, such as C++, Java, and Smalltalk, are appropriate to model the categories of components. However, in modeling the behavior of complex engineering entities, the traditional OO approach of using procedures (or methods) is inappropriate, because it is more natural to think of each component as being governed by certain laws, or invariants. From a programming language standpoint, we may express such behavioral laws by *constraints* over the attributes of an object. When such objects are aggregated to form a complex object, their internal attributes might further have to satisfy interface constraints. In general, the resultant state of a complex object can be deduced only by satisfying both the internal and the interface constraints of the constituent objects. We refer to this paradigm of objects and constraints as *constrained objects*, and it may be regarded as a declarative approach to object-oriented programming.

To illustrate the notion of constrained objects, consider a resistor in an electric circuit. (This is a classic example in CLP(R) but there are advantages offered by a constrained object representation, as discussed

[1] This research was partially supported by grants from the National Science Foundation and the Xerox Foundation.

further below.) Its state may be represented by three variables `V`, `I`, and `R`, which represent respectively its voltage, current, and resistance. However, these state variables may not change independently, but are governed by the constraint `V = I * R`. Hence a resistor is a *constrained object.* When two or more constrained objects are aggregated to form a complex object, their internal states may be subject to one or more interface constraints. For example, if two resistor objects are connected in series, their respective currents should be made equal. Similarly, in the civil engineering domain, we can model the members and joints in a truss as objects and we can express the laws of equilibrium as constraints over the various forces acting on the truss. In the chemical engineering domain, constrained objects can be used to model mixers and separaters, and constraints can be written to express the law of mass balance.

Constraints and Preferences. Sometimes there could be multiple solutions to constraints, and we are interested in the optimal solutions. This in turn necessitates some means specifying *preferences* that guide the determination of optimal solutions. To take an example from the engineering context, suppose we model a gear train and its constituent gears as objects whose attributes represent the input and output torques, angular speeds, and transmission powers. The efficiency of the gear train can be given in terms of these attributes by a constraint. In addition to these constraints, one may want to specify a preference that the transmission power be as close to 12hp (horse power) as possible and the efficiency be maximized. In our paradigm, such preferences can be specified declaratively using a preference clause.

Domain Specific Visual Interfaces. In the paradigm of constrained objects that we present, a complex object may also have a visual representation, such as a diagram. This visual representation is compositional in nature; that is, each component of the visual form can be traced back to some specific object in the underlying constrained object model. An end-user, i.e., modeler, will be able to access and modify the underlying model using the visual representation. This capability may be contrasted with currently available tools for engineering design such as AutoCAD, Abaqus, etc., where the visual representation contains only geometric information but does not provide access to the underlying logic or constraints. We expect to have different visual interfaces for different domains but a common textual language (described in section 3) for defining the classes of constrained objects.

Modeling Scenario. We expect a modeler to first define the classes of constrained objects, by specifying the attributes of the various objects, as well as their internal and interface constraints. These classes may be organized into an inheritance hierarchy. Once these definitions have been completed, the modeler can *build* a specific complex object, and execute (*solve*) it to observe its behavior. This execution will involve a process of logical inference and constraint satisfaction [4, 5, 8, 11, 20]. We expect that a modeler will then need to go through one or more iterations of *modifying* the component objects followed by re-execution. Such modifications could involve updating the internal states of the constituent objects, as well as adding new objects, replacing existing objects, etc. The complex object can be queried to find the values of attributes that will satisfy some given constraints in addition to the ones already present in the constrained objects.

Constrained Objects $\geq$ Constraints + Objects. Earlier we explained why the paradigm of constrained objects is preferable to the traditional paradigm of imperative objects for engineering modeling. It is also the case that, for the domain of engineering modeling, the paradigm of constrained objects is preferable to a traditional constraint language or constraint logic programming language (such as CLP(R)): (i) It is more natural to model an engineering artifact as a complex object than as a complex constraint. An object has a direct counterpart in the physical world. (ii) The diagrammatic representation can be more easily traced back to an object representation rather than a constraint representation. (iii) Object structure can help in reporting the cause of model inconsistency, and we believe it will also obtain efficiency in constraint solving, especially incremental solving (when a model is revised).

Prototype Implementation. We currently have developed three tools: (i) A Constraint-based UML tool for authoring Cob class diagrams. This tool allows one to define the classes of constrained objects using a graphical notation a la UML [18] and generates a set of textual Cob class definitions. (ii) A translator that takes Cob class definitions as input and generates equivalent CLP(R) predicates as output. We use the underlying CLP(R) engine for constraint handling. (iii) A domain-specific visual interface (a palette of buttons and drawing primitives) for building electric circuits and generating equivalent executable Cob code.

In this paper we describe a programming language and execution environment called **Cob** (*C*onstrained *Ob*ject) and its use in engineering modeling. While the idea of constrained objects has been discussed to some extent in the literature [3, 2, 10, 13], we extend earlier work by providing a richer set of features for modeling, especially conditional constraints, preferences, and logic variables at the modeling level. We have developed a formal declarative and operational semantics of Cob, but space precludes their description in this paper. The remainder of this paper is organised as follows: Section 2 briefly surveys related work and provides comparisons. Section 3 outlines the syntax of Cob and presents several motivating examples. Section 4 briefly describes the current Cob programming environment. Finally, section 5 describes our current and future research plans.

2 Related Research

Constraints Research. Our approach to constraints and preferences builds upon our earlier work [8, 7], and subsumes the paradigms of CLP as well as HCLP (Hierarchic CLP) [8]. Firstly, CLP does not support conditional constraints or preferences. Our paradigm of preferences is based upon the notion of *stratified preference logic programs* [8], which effectively allow an easy simulation of constraint hierarchies, as shown in [8, 7]. Also, our provision of conditional constraints allows object creation to take place dynamically and be controlled by constraint satisfaction. Research at the UNH Constraints Computation Center under Freuder is closely related to our efforts (`http:www.cs.unh.edu/ccc`). From a language standpoint, there are two important differences: (i) we integrate the concepts of object and constraint, and (ii) we adopt the more general CLP paradigm as opposed to a pure constraint language.

Constrained Objects. An early forerunner in the area of constrained objects is the work of Alan Borning on ThingLab, a constraint-based simulation laboratory [3] intended for interactive graphical simulations. Another work aimed at graphics applications is the language Bertrand. This work was extended by Bruce Horn in his language Siri, which uses the notion of *event pattern* to declaratively specify state changes: by declaring what constraints must hold after the execution of a method, and also specifying which attributes may and may not change during the method execution. Compared to our language, these approaches provide only a limited capability for expressing constraints, and also provide no support for handling multiple solutions to constraints. (Although ThingLab has a preference construct it orders constraints and not their solutions.)

Constraint Imperative Programming. The Kaleidoscope '91 [2] language integrates constraints and object-oriented programming for interactive graphical user interfaces. This 'constraint imperative language' uses constraints to simulate imperative constructs such as updating, assignment, and object identity. For the class of modeling applications that we target, it is not essential for us to consider such imperative concepts. These are important issues in an constraint imperative language [6, 14], but not for a declarative object-oriented language such as Cob. In our modeling scenarios, *model execution* and the *model revision* are carried out in mutual exclusion of one another. Changes are made at the level of the modeling environment. Thus we have a clear separation of the declarative and procedural parts of a constrained object.

Logic and Objects. The Prolog++ language [17] adds to logic programming by allowing storage and modification of the state of a system by providing constructs for classes, methods and assignment. Another approach to integrating logic with object-oriented programming is the Object Logic Integration (OLI) design methodology [12] which allows programming in either or mixed paradigm. From the object point of view, the logic part of OLI is an object with logic programs as states and methods performing logical deduction. The mixed paradigm allows the usual class definitions and methods can be written as queries. However, no notion of constraints is supported by either of these two languages. Two other related logic-based languages are LIFE [1] and Oz [19]. LIFE combines logic with inheritance, however, it does not deal with full-fledged constraints as we do. Oz is a language combinining constraints and concurrency as well as objects. Both these languages do not support the notion of preference nor do they consider engineering applications.

Constraint-based Specifications. For the sake of completeness, we also mention software specification languages that make use of constraints. The need for a formal representation of constraints in object-oriented programming is illustrated by the development of the Object Constraint Language [21]. For practitioners of object-oriented design and analysis, constraints provide an unambiguous and concise tool for expressing the relations between objects. OCL is a specification language that helps in making explicit these relations that would otherwise be implicit in the code but not apparent to the programmer who reads or modifies it. Eiffel is another language which employs constraints for specifying pre- and post-conditions that must hold on the operations of an object [16]. These languages use constraints as specifications; no constraint solving is done at run-time in order to deduce the values of variables. Contracts [9] provide a formal language for specifying behavioral compositions in object oriented systems. They promote an interaction-oriented design instead of class-based design. A contract defines a set of communicating participant classes and their contractual obligations as constraints. A class conforming to a contract must implement the methods exactly as specified in the contract.

3 Cob: An Informal Introduction

Syntax The grammar below outlines the overall structure of a Cob program. A Cob program is essentially a sequence of class definitions, and each constrained object is an instance of some class. The body of a class definition consists of the attributes, constraints, predicates, preferences, and constructors. Each of these constituents is optional, and we permit an empty class definition as a degenerate case.

$$\begin{aligned}
\mathit{program} &::= \mathit{class\_definition}^{+} \\
\mathit{class\_definition} &::= [\ \texttt{abstract}\ ]\ \texttt{class}\ \mathit{class\_id}\ [\ \texttt{extends}\ \mathit{class\_id}\ ]\ \{\ \mathit{body}\ \} \\
\mathit{body} &::= [\ \texttt{attributes}\ \mathit{attributes}\ ]\quad [\ \texttt{constraints}\ \mathit{constraints}\ ] \\
&\ [\ \texttt{predicates}\ \mathit{pred\_clauses}\ ]\quad [\ \texttt{preferences}\ \mathit{pref\_clauses}\ ] \\
&\ [\ \texttt{constructors}\ \mathit{constructor\_clause}\ ]
\end{aligned}$$

For the purpose of this paper, we limit attention to single inheritance of classes and at most one constructor for each class. An abstract class is a class without any constructor, and hence cannot be instantiated. Not all of the syntactic details are presented here; a more complete description of the syntax of constraints is given in Appendix A. Below we briefly discuss the more novel aspects of the language.

$$\begin{aligned}
\mathit{constraint} &::= \mathit{creational\_constraint} \mid \mathit{quantified\_constraint} \mid \mathit{simple\_constraint} \\
\mathit{creational\_constraint} &::= \mathit{attribute} = \texttt{new}\ \mathit{class\_id}(\mathit{terms}) \\
\mathit{quantified\_constraint} &::= \texttt{forall}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{constraint} \mid \texttt{exists}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{constraint} \\
\mathit{simple\_constraint} &::= \mathit{constraint\_atom} \mid \mathit{conditional\_constraint} \\
\mathit{constraint\_atom} &::= \mathit{term}\ \ \mathit{relop}\ \ \mathit{term} \mid \mathit{constraint\_predicate\_id}(\mathit{terms}) \\
\mathit{relop} &::= = \mid\ != \mid > \mid < \mid >= \mid <= \\
\mathit{term} &::= \mathit{constant} \mid \mathit{var} \mid \mathit{attribute} \mid (\mathit{term}) \mid \mathit{function\_id}(\mathit{terms}) \\
&\ \mid \texttt{sum}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{term} \mid \texttt{prod}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{term} \\
&\ \mid \texttt{min}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{term} \mid \texttt{max}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{term} \\
\mathit{conditional\_constraint} &::= \mathit{constraint\_atom} :- \mathit{literals}
\end{aligned}$$

A constraint can be either creational, simple or quantified, where the quantification ranges over an enumeration (referred to as *enum*) which may be the indices of an array or the elements of an explicitly specified set. A simple constraint can either be a constraint atom or a conditional constraint. A constraint atom is essentially a relational expression of the form *term relop term*, where *term* is composed of functions/operators from any data domain (e.g. integers, reals, etc.) as well as constants and attributes. A conditional constraint is a constraint atom that is predicated upon a conjunction of literals each of which is a (possibly negated) ordinary atom or a constraint atom.

Often we would like to augment a predicate definition with preference criteria for determining the best solution(s) to goals that make use of this predicate. In Cob, a preference clause is of the form

p(s1) $\leq$ *p(s2)* :- *clause_body*,

and it states that the solution *s1* is less preferred than solution *s2* for predicate *p* if the condition specified by *clause_body* is true. We also provide `min` and `max` constructs to model optimization problems. The paper [8] provides a more detailed account of this construct.

Date as a Constrained Object Our first example illustrates the basic features of the language, including the use of conditional constraints.

```
class Date {
  attributes
      int day, month, year
  constraints
      1 ≤ year.
      1 ≤ month.    month ≤ 12.
      1 ≤ day.      day ≤ 31.
      day ≤ 30 :- member(month, [4,6,9,11]).
      day ≤ 29 :- month = 2, leap(year).
      day ≤ 28 :- month = 2, not leap(year)
  predicates
      member(X, [X|_]).
      member(X, [_|T]) :- member(X,T).
      leap(Y) :- Y mod 4 = 0, Y mod 100 <> 0.
      leap(Y) :-  Y mod 400 = 0
  constructors Date(d, m, y) { day = d, month = m, year = y }
}
```

We employ Prolog-like syntax for defining predicates and conditional constraints. For example, the conditional constraint
`day ≤ 29 :- month = 2, leap(year)`
requires `day ≤ 29` if the month is February and the year is a leap year. Computationally, an important difference between a conditional constraint and a Prolog rule is the following: If the head of a conditional constraint evaluates to true, then the body need not be evaluated; and, if the head evaluates to false, the body must fail in order for the conditional constraint to be satisfied. In contrast, in Prolog, if the head of a rule unifies with a goal, then the body of the rule must be evaluated; and, if the head does not unify, then the body need not be evaluated.

The above definition can be used to validate a given combination of day, month, and year values, and also be used to generate, for example, a range of month values for a given a combination of day and year. For example, if the day is set to 31 and the year to 1999, the set of possible values for month can be deduced to be any integer between 1 and 12 but not 4, 6, 9, 11, or 2. While 1 $\leq$ month $\leq$ 12 is directly obtained from the unconditional constraints for month, our computational model can deduce, by a process of *constructive negation* of the goal `member(month, [4,6,9,11])`, that month is not 4, 6, 9, or 11. And, it can deduce that month is not equal to 2 from the conditional constraint `day ≤ 28 :- month = 2, not leap(year)`.

Conditional constraints can be used to control object creation dynamically. For example, consider the following conditional constraints over attributes `x`, `y`, and `shape` of a certain class:

```
shape = Rectangle(x, y) :- input = 'rectangle'
shape = Circle(x, y) :- input = 'circle'
```

Together, they can be used to set a `shape` attribute of, for example, a node of a binary tree. In the above example, `x` and `y` stand respectively for the width and height inputs of the `Rectangle` constructor; and they stand respectively for the center and radius attributes of the `Circle` constructor.

Non-Series/Parallel Circuits To further illustrate the syntax of `Cob` and the use of equational and quantified constraints, we present the well-known example of a non-series/parallel electrical circuit (see figure 4 appendix C). We model the components and connections of such a circuit as objects and their properties and relations as constraints on and amongst these objects. The `component` class models any electrical entity (e.g resistor, battery) that has two ends (referred to as 1 and 2). The attributes of this class represent the currents and voltages at the two ends of the entity. The constraint in class `resistor` represent Ohm's law. 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). A `node` is a collection of componentEnds. 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. Using these classes we can model any non-series/parallel circuit. Given initial values for some attributes, this model can be used to find out values of the remaining attributes (e.g. the current through a particular component). Appendix C shows a sample circuit built from these classes.

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

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

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

```
class componentEnd {
 attributes
  component C.
  real End, V, I
 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 }
}

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

Simple Truss To illustrate the use of constrained objects in engineering design, we model a simple truss structure shown in figure 1. A truss consists of bars placed together at joints. The constraints in the `bar` 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 bar (inward or outward), it acts as either a buckling force or a tension force. This relation is expressed as conditional constraints in the `bar` class. A beam is a bar placed at an angle (`A`) and a load is a force applied at an angle (`A`). The `joint` class aggregates an array of beams and 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 Cob classes defined here can model a truss with loads and may be used to determine the dimensions of the bars such that they can support the loads. These conditions are taken from the text by Mayne and Margolis [15].

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

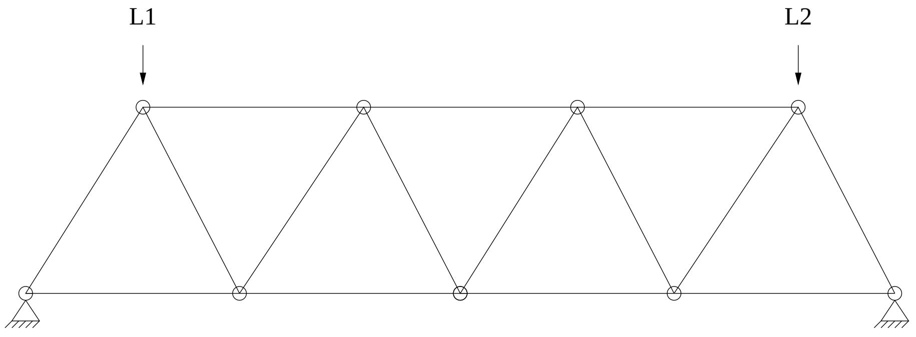


Fig. 1. A Simple Truss

```
 constraints
  Pi = 3.141 .
  I = F_bk * L *L /Pi * Pi*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   }
}
```

```
class beam {
 attributes
  bar B. Real A    % bar B placed at angle A
 constraints
  0 <= A. A <= 360
 constructors beam(B1, A1)
  { B = B1, A = A1 }
}
```

```
class load {
 attributes
  Real F. Real A
 constraints
  0 <= A. A <= 360
 constructors load(F1, A1)
  { F = F1, A = A1 }
}
```

```
class joint {
 attributes
  beam [] Beams. load [] Loads
 constraints
  sum X in Beams: (X.B.F * sin(X.A)) + sum L in Loads: (L.F * sin(L.A)) = 0.
  sum Y in Beams: (Y.B.F * cos(Y.A)) + sum M in Loads: (M.F * cos(M.A)) = 0
 constructors joint(B1, L1) { Beams = B1, Loads = L1 }
}
```

Separators and Mixers A common problem in chemical engineering, is the use of a combination of mixers and separators to produce a chemical that has specific ingredients in a certain proportion. The arrangement of mixers and separators shown in figure 2 has two input raw material streams R1 and R2. Each of these streams has certain ingredients in different concentrations. R1 and R2 are split and a part of each (I1 and I2 respectively) is sent to a separator which separates its ingredients. Each separator supplies certain proportion of each ingredient to the mixer which combines them to produce the desired chemical G. W1 and W2 are the waste streams from the separators. The problem is to produce G while minimizing I1 and I2 thereby minimizing the cost of processing material in the separators.

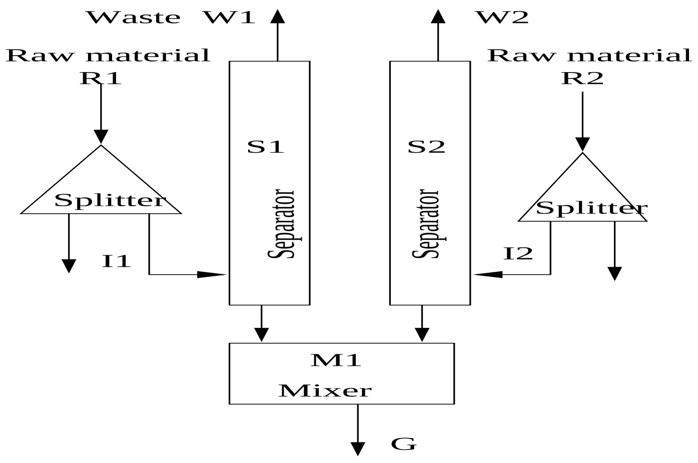


Fig. 2. A Separation Flow Sheet

Figure 2 describes a typical scenario, and we present below some of the key classes needed for this example. A stream is modeled by the class `stream` with attributes for its rate of flow `FlowRate` and the concentrations of its ingredients (`Concentrations` is an array of reals indexed by the ingredients of the stream). The concentrations of all the ingredients of a stream must sum up to 1. The class `equipment` models any piece of equipment having some input streams and output streams. Every equipment that processes streams is constrained by the law of mass balance. Separators, mixers and splitters are instances of the `equipment` class. The class representing figure 2 (not shown here) will have the preference `min I1 +I2`.

```
class stream {
 attributes
  real FlowRate.  real [] Concentrations
 constraints
  sum C in Concentrations : C = 1
 constructors stream(Q, C) {  FlowRate = Q, Concentrations = C  }
}
class equipment {
 attributes
  stream [] inStream, outStream.  int nIngedients
 constraints
  forall I in 1..nIngredients :
   (sum J in inStream : (J.FlowRate * J.Concentrations[I])) =     % law of mass balance
   (sum K in outStream: (K.FlowRate * K.Concentrations[I]))
 constructors equipment(In,Out,NumIng) {
  inStream = In, outStream = Out, nIngredients = NumIng }
}
```

Heat Transfer in a Plate This is another classic program in CLP(R). The problem is to model a plate in which the temperature of any point in the interior of the plate is the average of the temperature of its neighbouring four points. This can be stated mathematically by using 2d Laplace's equations. The Cob representation is shown below in a class called `heatplate`. The constructor initializes the temperature of the border of the 11 x 11 plate. Compared to its CLP(R) representation, the Cob representation of this problem is very concise, owing to the use of quantified constraints. The Cob class below calculates the heat at all the interior points.

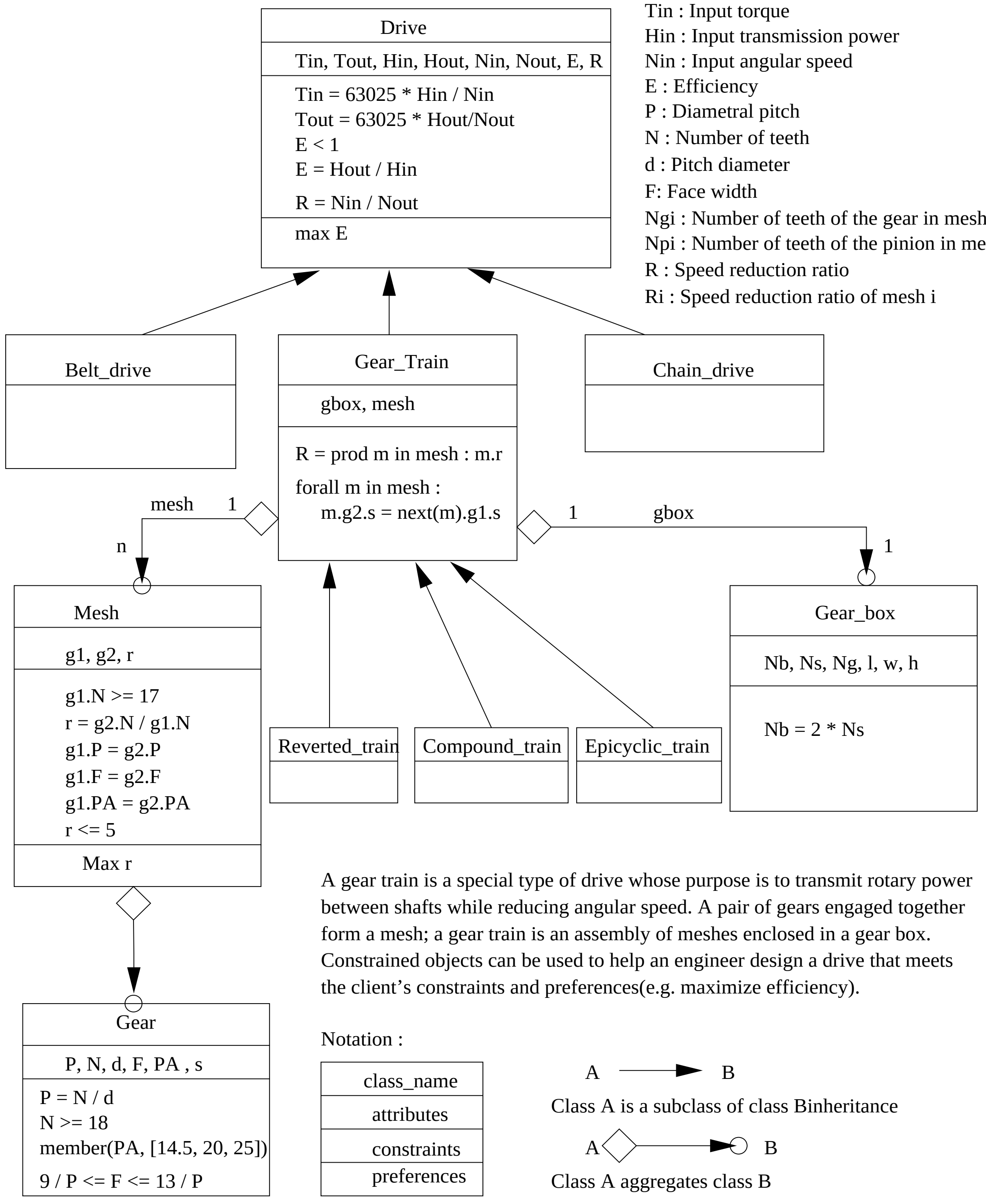
```
class heatplate {
  attributes
      Real [][] Plate
  constraints
      forall I in 2..10:
         forall J in 2..10:
            4 * Plate[I,J] =
               (Plate[I-1,J] + Plate[I+1,J] + Plate[I,J-1] + Plate[I,J+1])
  constructors heatplate(A,B,C,D) {
      forall M in 1..11: Plate[1,M] = A,
      forall N in 1..11: Plate[11,N] = B,
      forall K in 2..10: Plate[K,1] = C,
      forall L in 2..10: Plate[L,11] = D
  }
}
```

Gear Train We now present a more detailed illustration of constrained objects. The design of a gear train is a problem from the industrial engineering context. A gear train is a special kind of drive whose purpose is to transmit rotary power between shafts while reducing angular speed. It consists of an assembly of pairwise engaged gears (meshes) enclosed in a gear box. The efficiency (`E`) of a drive is given as a relation between its input and output torques, angular speeds and transmission powers. We model this as a constraint in the `Drive` class. The attributes and constraints of the remaining Cob classes are described in the class diagram in figure 3. The problem is to design a gear train that meets the designer's constraints and preferences while maximizing its efficiency (which is stated as a preference in the `Drive` class). The Cob model of figure 3 can be used to solve this problem. Given values for `Tin`, `Hin`, and `Nin`, the underlying Cob computational model calculates the numer of meshes and teeth(of a gear) required to maximize the efficiency of the drive. Then using the number of meshes and teeth obtained earlier, the values of `P` (diameteral pitch), `d` (pitch diameter) and `F` (face width) are computed.

Document Representation The use of constrained objects to model documents brings up the need for constraint optimization and relaxation. In a simplified version of this problem, a book may be thought of as a complex object that aggregates many chapters, each of which aggregates many sections. Each section in turn aggregates many paragraphs, and each paragraph aggregates many words. Thus, the elements of a book, namely, chapter, section, and paragraph become the basic classes of objects in the Cob model. Suppose we are interested in formatting the contents of the book. We may think of a `format` predicate within each class of book-element (chapter, section, etc.), whose purpose is to obtain the optimal (best) layout for the element (The reason for having a *predicate*, instead of a method, will become clear below.) Each formatted element would have a measure of how good or bad its formatting is, viz., its so-called *badness*. We assume that the `format` predicate incorporates some such method for calculating its badness. Also, the `format` predicate of one element would invoke the `format` predicate of its constituent elements in order to construct its optimal layout.

In this example, there are certain important constraints, namely: no section shall begin on the last line of a page, and no section shall end on the first line of a page. Such restrictions are better specified separately from the formatting algorithm, since the algorithm may be modified independently while these still hold. The `constraints` clause is used for specifying these restrictions. Now suppose the formatting algorithm comes up with a format that violates one of these constraints, it would then need to produce another format which will satisfy all constraints. (This is why having a format predicate is more convenient than a method.) Of course, this assumes that such a format exists; otherwise, we need some means of relaxing the constraints. The possibility of multiple solutions necessitates a preference criterion for choosing the best (most preferred) one. This is shown in the class Para, where we state that the format with lesser badness is the preferred one:

```
format(L1, B1) < format(L2, B2) :- B1 > B2.
```



Tin : Input torque
Hin : Input transmission power
Nin : Input angular speed
E : Efficiency
P : Diametral pitch
N : Number of teeth
d : Pitch diameter
F: Face width
Ngi : Number of teeth of the gear in mesh i
Npi : Number of teeth of the pinion in mesh i
R : Speed reduction ratio
Ri : Speed reduction ratio of mesh i

A gear train is a special type of drive whose purpose is to transmit rotary power between shafts while reducing angular speed. A pair of gears engaged together form a mesh; a gear train is an assembly of meshes enclosed in a gear box. Constrained objects can be used to help an engineer design a drive that meets the client’s constraints and preferences(e.g. maximize efficiency).

Notation :

Class A is a subclass of class Binheritance

Class A aggregates class B

Fig. 3. Gear Train as Constrained Objects

```
class Section {
  attributes
      Para[] paras.
      int begin_ln, end_ln
  constraints
      begin_ln mod 50 != 49.
      end_ln mod 50 != 1.
      begin_ln = first(paras).begin_ln.
      end_ln = last(paras).end_ln.
      forall p in paras: (p.end_ln + 1 = next(p).begin_ln)  :-  not is_last(p))
  predicates
      format(Sec_lines) :- format(paras, Sec_lines, Badness).
      format([], [], 0).
      format([P|Paras], [L|Lines], Badness) :-
          P.format(L,Badness'),
          format(Paras, Lines, Badness''),
          Badness = Badness' + Badness''.
}
class Para {
  attributes
      int begin_ln, end_ln, Numlines.
      string[] words
  constraints
      end_ln = begin_ln + Numlines.
      width = 30
  predicates
      format(Lines, Badness) :- ... details omitted ...
  preference
      format(L1, B1) < format(L2, B2) :- B1 > B2.
}
```

4 Cob Programming Environment

Compiler. We have developed formal declarative and operational semantics of Cob. For the interested reader, we highlight/summarize the rules for operational semantics of Cob in appendix B. In order to validate the language and its semantics, we have developed a prototype Cob compiler that translates a Cob program into a CLP(R) program. Essentially each class definition translates to one predicate clause. The constraints and constructor clauses of all classes are translated to appropriate CLP(R) constraints. Since conditional constraints do not have a direct equivalent in CLP(R), we have implemented them separately. Disequations (!=) and constructive negation are handled through special predicates. Inheritance and compound attributes are translated by expanding the attributes of a class to include the attributes of its superclass. Quantified and aggregation constraints (`forall`, `sum`, etc.) are translated to predicate clauses that iterate over elements of an enumerated type. We use the underlying CLP(R) engine for constraint handling. We are presently extending the tool to handle preferences.

CUML. We have developed a constraint-based extention of the Unified Modeling Language [18]. This extension, called CUML, allows one to define attributes, constraints, and preferences associated with every constrained object class definition. The tool generates textual equivalent of the CUML specification. Figure 3 illustrates the CUML notation for the gear train example. CUML facilitates quick and error-free code development. The notation being employed is explained at the bottom of Figure 3.

Domain Specific Visual Interface. Currently we have a domain-specific visual interface tool for building electric circuits and generating textual Cob code that instantiates the appropriate classes. This interface is a palette of buttons and drawing primitives for creating instances of electrical components like batteries, resistors and wires. Each component has a predefined Cob class definition and the value of any of its attributes (e.g., current through a wire or voltage of a battery), can be specified through this interface. The components can be placed in any desired configuration and the appropriate classes representing this configuration are automatically instantiated. Once a model/configuration is created, it is translated using the Cob compiler to CLP(R) code. The constraints of the model are then solved, and values of attributes of all the components are displayed. If the model has an inconsistency (some constraint cannot be satisfied), it is detected during this *execution* of the model. Code generation and model execution are important capabilities of this tool not present in drawing tools like AutoCAD.

5 Conclusions, Status, and Further Work

The concept of constrained objects is broadly applicable in the modeling of complex engineering structures. We have illustrated this point in this paper through a variety of examples. We believe the Cob definitions presented are concise and clear. With the aid of domain-specific visual interfaces, the resulting paradigm has considerable potential for both pedagogic purposes as well as for more advanced applications. On a technical level, our language advances previous work by showing the use of a number of new features in modeling, especially conditional and quantified constraints, as well as preferences.

We are working on a number of modeling applications with researchers in the civil, chemical, computer, mechanical, and industrial engineering departments in the University at Buffalo. These applications include: rapid product configuration in agile manufacturing, constraint-based product design, physically-based modeling, hierarchic modeling in chemical process synthesis, and structural design. Experience gained in these areas will in turn help refine and advance the paradigm, making it a more robust platform for engineering modeling and design.

We are presently investigating a few important issues relating to this paradigm: (i) inconsistency detection, (ii) incrementality, (iii) abstraction. When the state of a model violates its constraints, a response to the effect that an error has occured is often not sufficient. The underlying computational engine should be able to provide a narrow range of possible places where the programmer can look for and correct the error. It is also possible sometimes that there are no solutions to the constraints, and the modeler is interested in understanding the cause of this inconsistency. This is also referred to as an over-constrained system [5] in the literature. A constraint violation could occur due to an incorrectly stated constraint, or an inconsistent value assigned to an attribute and can be corrected with the help of the programer. In conjunction with a visual representation for constrained objects, it is possible to develop techniques showing where the constraint violation occurred.

Another issue of special interest is incrementality. Since a modeler will typically make several changes to an initial model, it is important to support incremental constraint satisfaction, i.e., we should try to compute the new state of the complex object without re-solving all the constraints. When complex object is very large (i.e., consists of many subobjects), it may be very time-consuming to do a detailed simulation. In this case it may be necessary to work with a simplified model, i.e., a "cross-section" of the model so as to get an approximate answer in reasonable time. This problem may be called *model abstraction.*

References

1. H. Ait-Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993.
2. A. Borning B. N. Freeman-Benson. Integrating Constraints with an Object Oriented Language. In *Proc. European Conference On Object-Oriented Programming*, pages 268–286, 1992.
3. A. Borning. The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS*, 3(4):252–287, 1981.
4. E.C. Freuder. Partial Constraint Satisfaction. In *Proc. 11th Intl. Jt. Conf. on Artificial Intelligence*, pages 278–283, 1989.

5. E.C. Freuder and R.J. Wallace. Heuristic Methods for Over-Constrained Constraint Satisfaction Problems. In *Proc. CP'95 Workshop on Overconstrained Systems*, 1995.
6. A. Borning G. Lopez, B. N. Freeman-Benson. Constraints and Object Identity. In *Proc. European Conference On Object-Oriented Programming*, 1994.
7. K. Govindarajan. *Optimization and Relaxation in Logic Languages.* PhD thesis, Department of Computer Science, SUNY - Buffalo, 1997.
8. K. Govindarajan, B. Jayaraman, and S. Mantha. Optimization and Relaxation in Constraint Logic Languages. In *Proc. 23rd ACM Symp. on Principles of Programming Languages*, pages 91–103, 1996.
9. R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioural Compositions in Object-Oriented Systems. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 1990.
10. B. Horn. Constraint Patterns As a Basis For Object Oriented Programming. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 1992.
11. J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.
12. J.H.M. Lee and P.K.C. Pun. Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language. *Computer Languages*, 23(1):25–42, 1997.
13. W.J. Leler. *The Specification and Generation of Constraint Satisfaction Systems.* Addison-Wesley, 1987.
14. G. Lopez. *The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language.* PhD thesis, University of Washington, 1997.
15. R. Mayne and S. Margolis. *Introduction to Engineering.* McGraw-Hill, 1982.
16. B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.
17. C. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming.* Addison-Wesley, 1994.
18. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.
19. G. Smolka. Constraint Programming in Oz (Abstract). In *Proc. Intl. Conference on Logic Programming*, pages 37–38, 1997.
20. P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
21. Warmer, J., Kleppe, A. *The Object Constraint Language.* Addison-Wesley, 1999.

Appendix A: Cob Syntax

$$
\begin{array}{rcl}
\textit{attributes} & ::= & \textit{decl}\ [\ .\ \textit{decl}\ ]^+ \\
\textit{decl} & ::= & \textit{type}\ \textit{id\_list} \\
\textit{type} & ::= & \textit{primitive\_type\_id} \mid \textit{class\_id} \mid \textit{type}[] \\
\textit{primitive\_type\_id} & ::= & \texttt{Real} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Char} \mid \texttt{String} \\
\textit{id\_list} & ::= & \textit{attribute\_id}\ [\ ,\ \textit{attribute\_id}\ ]^+ \\
\textit{constraints} & ::= & \textit{constraint}\ [\ .\ \textit{constraint}\ ]^+ \\
\textit{attribute} & ::= & \textit{selector}[.\textit{selector}]^+ \mid \textit{attribute}[\textit{term}] \\
\textit{selector} & ::= & \textit{attribute\_id} \mid \textit{selector\_id}(\textit{terms}) \\
\textit{selector\_id} & ::= & \texttt{first} \mid \texttt{next} \mid \texttt{last} \\
\textit{terms} & ::= & \textit{term}\ [\ ,\ \textit{term}\ ]^+ \\
\textit{literals} & ::= & \textit{literal}\ [\ ,\ \textit{literal}\ ]^+ \\
\textit{literal} & ::= & [\ \texttt{not}\ ]\ \textit{atom} \\
\textit{atom} & ::= & \textit{predicate\_id}(\textit{terms}) \mid \textit{constraint\_atom} \\
\textit{pred\_clauses} & ::= & \textit{pred\_clause}\ .\ [\ \textit{pred\_clause}\ .\ ]^+ \\
\textit{pred\_clause} & ::= & \textit{clause\_head}\ :-\ \textit{clause\_body} \\
\textit{pred\_clause} & ::= & \textit{clause\_head} \\
\textit{clause\_head} & ::= & \textit{predicate\_id}(\textit{terms}') \\
\textit{clause\_body} & ::= & \textit{goal}\ [\ ,\ \textit{goal}\ ]^+ \\
\textit{goal} & ::= & [\ \texttt{not}\ ]\ \textit{predicate\_id}(\textit{terms}')
\end{array}
$$

$$\begin{aligned}
\textit{terms}' &::= \textit{term}' \ [\ ,\ \textit{term}'\ ]^+ \\
\textit{term}' &::= \textit{constant} \mid \textit{var}' \mid \textit{attribute} \mid \textit{function\_id}(\textit{terms}') \\
\textit{pref\_clauses} &::= \textit{pref\_clause}\ .\ [\ \textit{pref\_clause}\ .\ ]^+ \\
\textit{pref\_clause} &::= \textit{predicate\_id}(\textit{terms}') < \textit{predicate\_id}(\textit{terms}') :- \ \textit{clause\_body} \\
&\quad \mid \texttt{min}\ \textit{term} \mid \texttt{max}\ \textit{term}
\end{aligned}$$

Appendix B : Cob Operational Semantics

The top down execution of a Cob program C translated to a CLP program P, is described in terms of a sequence of transitions on states. A *state* is represented by the tuple $< A, E, I, D, C, \theta, N >$, where A is a multiset of atoms and constraints and E, I, D, C, θ, N are respectively multisets of equations, inequations, disequations(`!=`), conditional constraints, valuations (attribute value pairs), and negated conjunction of atoms. Given a goal $G \equiv \texttt{new}\ c(\bar{t})$, the *start state* is $< G, \phi, \phi, \phi, \phi, \phi, \phi >$. Note that G is an appropriate translation of $\texttt{new}\ c(\bar{t})$ to a CLP goal. There is also a state called $fail$. A *transition* changes one state to another. The types of transitions that may be applied to a given state is described below as a case analysis on the state.

1. $< A \cup a, E, I, D, C, \theta, N > \rightarrow_r < A \cup B, E \cup \{\overline{t_1} = \overline{t_2}\}, I, D, C, \theta, N >$
   if a is an atom selected by the computation rule and $h \leftarrow B$ is a rule of P and a and h have the same outermost predicate and $a = p(\overline{t_1})$ and $h = p(\overline{t_2})$.
2. $< A \cup a, E, I, D, C, \theta, N > \rightarrow_r fail$
   if a is an atom selected by the computation rule and there does not exist a rule $h \leftarrow B$ of program P such that a and h have the same outermost predicate and $a = p(\overline{t_1})$ and $h = p(\overline{t_2})$
3. $$< A \cup c, E, I, D, C, \theta, N > \quad \rightarrow_c \quad \begin{array}{ll} < A, E \cup c, I, D, C, \theta, N > & \text{if c is an equation} \\ < A, E, I \cup c, D, C, \theta, N > & \text{if c is an inequation} \\ < A, E, I, D \cup c, C, \theta, N > & \text{if c is a disequation} \\ < A, E, I, D, C \cup c, \theta, N > & \text{if c is a conditional constraint} \end{array}$$
4. $< A, E, I, D, C, \theta, N >$
   $$\begin{array}{ll} \rightarrow_e < A, E', I, D, C, \theta \cup \theta', N > & \text{where } (\theta', E') = solve_e(E) \\ \rightarrow_{ie} < A, E \cup E', I', D, C, \theta \cup \theta', N > & \text{where } (\theta', E', I') = solve_i(I) \\ \rightarrow_d < A, E, I, D', C, \theta, N > & \text{where } D' = solve_d(D) \\ \rightarrow_{cc} < A, E \cup E', I \cup I', D \cup D', C \cup C', \theta \cup \theta', N' > & \text{see details of } \rightarrow_{cc} \text{ below} \end{array}$$

   $solve_e(E)$ is an equation solver that returns attribute-value pairs and a simplified set of equations E' such that $D \models \theta' \cup E'$ iff $\theta \cup E$
   $solve_i(I)$ is an inequation solver that returns attribute value pairs θ' and simplified sets of equations E' and inequations I' such that $D \models \theta' \cup E' \cup I'$ iff $\theta \cup E \cup I$
   $solve_d$ tries to verify a set of disequations, and returns those disequations that cannot be verified.
5. $< A, E, I, D, C \cup \{p : -q\}, \theta, N > \quad \rightarrow_{cc}$
   $$\begin{array}{ll} < A, E, I, D, C, \theta, N > & \text{if } \mathcal{D} \models \overline{\forall}(p \leftarrow (E \cup I \cup D) \\ < A, E, I, D, C, \theta, N \cup \{\neg q\} > & \text{if } \mathcal{D} \models \overline{\forall}(\neg p \leftarrow (E \cup I \cup D) \\ < A, E \cup p, I, D, C, \theta, N > & \text{if } \mathcal{D} \models \overline{\forall}(q \leftarrow (E \cup I \cup D) \text{ and p is an equation} \\ < A, E, I \cup p, D, C, \theta, N > & \text{if } \mathcal{D} \models \overline{\forall}(q \leftarrow (E \cup I \cup D) \text{ and p is an inequation} \\ < A, E, I, D \cup p, C, \theta, N > & \text{if } \mathcal{D} \models \overline{\forall}(q \leftarrow (E \cup I \cup D) \text{ and p is a disequation} \end{array}$$
6. $< A, E, I, D, C, \theta, N > \rightarrow_s < A, E, I, D, C, \theta, N >$ if $consistent$(E, I, D)
   $< A, E, I, D, C, \theta, N > \rightarrow_s fail$ if $\neg consistent$(E, I, D)

A *computation rule* selects a type of transition and an approriate element of A to compute the next state in the sequence. A *derivation* for a goal G w.r.t. a program P is a finite sequence of transitions starting at the *start state* $< G, \phi, \phi, \phi, \phi, \phi, \phi >$, where successive states are obtained by the application of computation rules.

If a derivation for a goal G w.r.t. a program P ends in the state $< \phi, E, I, D, C, \theta, N >$ such that *consistent*(E, I, D, C), we say that the derivation is *successful*, the valuation θ is the *computed answer* and E, I, D, C and N are the *answer constraints*.

Appendix C: Cob model of a Non-Series/Parallel Circuit

The circuit in Figure 4 can be modeled in cob by the class `samplecircuit` given below. Given the resistance of each resistor and the volate of the battery, the cob model calculates the current flowing through every component and the voltage at every node of the circuit.

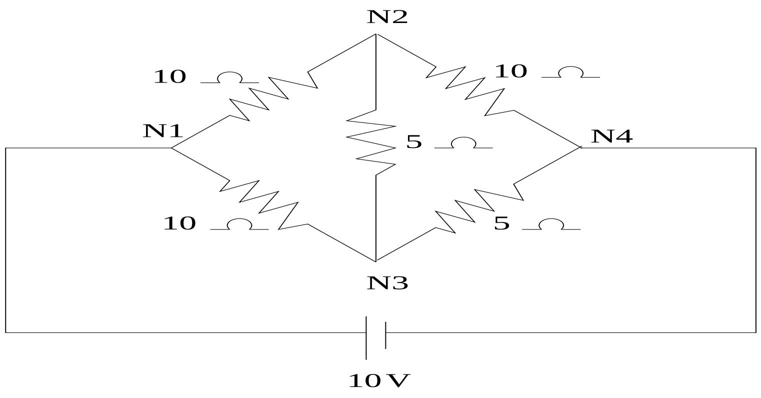


Fig. 4. A Non-series/parallel circuit

```
class samplecircuit {
 attributes
  resistor R12, R13, R23, R24, R34.   battery B.   node N1, N2, N3, N4.
  componentEnd Re121, Re122, Re131, Re132, Re231,Re232 , Re241, Re242, Re341, Re342, Be1, Be2
 constructors samplecircuit(X) {
  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])
 } }
```