Semantics and Applications of Constrained Objects [1]

Bharat Jayaraman
Pallavi Tambay

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

1 Significance and Objectives

We describe the semantics and applications of a novel object-oriented programming language for compositional modeling of complex systems. A domain of particular interest is that of engineering systems, such as circuits, trusses, gears, mixers, separaters, etc. Modeling such entities involves the specification of both the structure and behavior of their constituent components. While structure can be modeled naturally in terms of *objects* and aggregation/inheritance hierarchies, modeling behavior using traditional imperative procedures (or methods) is inappropriate, because it is more natural to think of each component as being governed by certain laws, or invariants. Such behavioral laws are more naturally expressed as *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. 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. We illustrate the paradigm by showing how to model the classes of components that arise in truss structure of figure 1. We represent the bars 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.

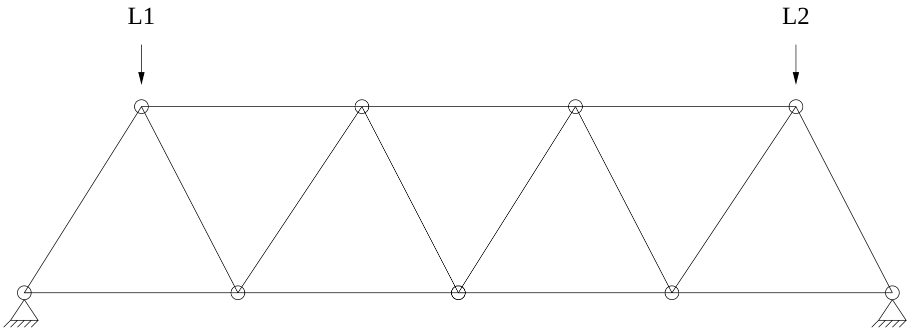

Figure 1: A Simple Truss

```
class Joint {
  attributes
      Bar[] B     % A joint is an aggregation of bars incident at the joint.
  constraints
      sum b in B: b.f_bn * sin(b.angle) = 0.      % Law of equilibrium of forces along the
      sum b in B: b.f_bn * cos(b.angle) = 0.      % x- and y-dimensions.
  constructor Joint(B1) { B = B1 }
}
class Bar {
  attributes
```

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

```
    Real angle, E, Sy, l, w, h, f_bn, f_bk, f_t, σ, I
  constraints
    0 ≤ angle.  angle ≤ 2*π.
    I = f_bk * l² /(π²*E).
    I ≤ w * h³ / 12.
    f_t = Sy * w * h.
    σ = h * l * f_bn / (8 * I)
  constructor Bar(C,S1,l1,w1,h1,a,f1,f2) {
    E = S1*C, l=l1, h=h1, w=w1, angle=a, f_bk=f1, f_bn=f2 }
}
```

The above code illustrates the Cob programming language. The `Joint` class aggregates an array of bars (that are incident at the joint) and incorporates the constraint that the sum of the forces in the horizontal and vertical directions respectively must be 0. The constraints in the `Bar` class express the standard conditions that relate the modulus of elasticity of the material (`E`), yield strength (`Sy`), the dimensions of a bar (`l`, `w`, `h`), the bending, buckling, and tension forces (`f_bn`, `f_bk`, `f_t`), and the stress (σ). The engineering text by Margolis and Mayne provides detailed explanations of these conditions [MM82]. The above example has a direct counterpart in the electrical engineering domain: a truss is similar to an electric circuit; a bar is similar to a resistor, capacitor or other electrical component, and the law of equilibrium of forces at a joint is analogous to Kirchoff's law for currents. In the chemical engineering domain, constrained objects can be used to model mixers and separaters, and constraints can be written to express the laws of heat or mass balance. This paradigm is also applicable in other modeling domains, such as document structures, biological (anatomical) structures, organizational structures, etc.

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 of specifying *preferences* that guide the determination of optimal solutions. For example, if there are many possible designs of an artifact satisfying the specifications, we may wish to adopt the design with either least cost or least weight or least time-to-manufacture, etc. In our earlier work, we developed a paradigm of optimization and relaxation, called preference logic programming (PLP), by extending a constraint logic programming (CLP) [JL87] language with preferences [GJM96]. The use of explicit preference statements helps to declaratively specify the criteria for choosing the optimal solutions to a set of constraints. The computational engine will enumerate alternative solutions to the constraints and prune suboptimal solutions in accordance with the stated preference clauses.

Semantics of Constrained Objects. We provide a set-theoretic semantics for Cob classes and an operational semantics along the lines of CLP and PLP. Declaratively speaking, we adopt a "types as sets" approach [Wan84] to the semantics of Cob classes. Essentially we map a Cob class on to a CLP predicate and derive the set-theoretic semantics of the class from the set of logical consequences of the corresponding predicate. When preferences are also specified in the class definition, we determine the set-theoretic semantics of the class from the set of *preferential consequences* of the corresponding predicate. The concept of preferential consequence differs from logical consequence in that we consider truths not in all possible worlds (logical consequence) but in certain strongly optimal worlds. The operational semantics employs a combination of goal reduction and constraint solving (as in CLP), and pruning suboptimal answers according to the stated preferences (as in PLP).

Visual Representation. Often a constrained object also has a visual representation, such as a diagram, 3d model, etc. This visual representation is also compositional in nature, i.e., each component of the visual form can be traced back to some specific object in the underlying constrained object model. This is a more authentic form of *visual programming* than is possible in traditional visual languages (such as Visual Basic or Visual C++). We expect that 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 CAD tools such as AutoCAD, Abaqus, etc., where the visual representation contains only geometric information but does not provide access to the underlying logic/constraints. In our approach, a diagram is effectively an executable program. We expect to have different domain-specific visual interfaces for different application domains but a common textual language for class definitions (as illustrated above). Presently we have implemented a visual programming interface for simple electrical components.

Typical Modeling Scenario. In the modeling environment provided by our prototype implementation, a modeler will first define the class hierarchy of constrained objects, including the attributes and internal and interface constraints of each class, as well as how the classes relate with one another (aggregation/inheritance relationships). Once these definitions have been completed, the modeler can *build* a specific complex object, and execute (*solve*) it to observe its behavior. This can be done using the domain-specific visual interface, if one is available for the application, or using the textual interface. The execution of a complex object will involve a process of constraint satisfaction [Fre89, FW95, GJM96, JL87, vH89]. The 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. These modifications are carried out using the domain-specific interface, if available. The complex object can be queried to find possible assignment of values to attributes that will satisfy some given constraints in addition to the ones already present in the constrained objects.

Constrained Objects $\geq$ Constraints + Objects. We believe that the paradigm of constrained objects is a conceptual advance over traditional programming paradigms for a number of application domains. We explained earlier how constrained objects are preferable to the traditional paradigm of imperative objects for engineering modeling. It is also the case that, for many application domains, the paradigm of constrained objects is preferable to a traditional constraint language or constraint logic programming language: (i) It is more natural to model an engineering artifact or organizational structure 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 obtain efficiency in constraint solving, especially in incremental constraint solving, i.e., deciding which constraints to re-solve when a model is modified.

The remainder of this paper is organised as follows: Section 2 briefly surveys closely related work and provides comparisons. Section 3 outlines the syntax of Cob illustrating it with examples. Section 4 describes the declarative and operational semantics of Cob, along with appropriate correctness results.

2 Related Research

While the idea of constrained objects has been discussed to some extent in the literature [Bor81, BNFB92, Hor92, Lel87], we extend earlier work by providing a richer set of features for modeling, especially conditional constraints and preferences, and we also provide declarative and operational semantics for these constructs.

Constraints Research. Our approach to constraints and preferences builds upon our earlier work [GJM96, Gov97], and subsumes the paradigms of CLP as well as HCLP (Hierarchic CLP) [GJM96]. Firstly, CLP does not support conditional constraints or preferences. Our paradigm of preferences is based upon the notion of *stratified preference logic programs* [GJM96], which effectively allow an easy simulation of constraint hierarchies, as shown in [GJM96, Gov97]. 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 [Bor81] 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 [BNFB92] 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 [GL94, Lop97], 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 [Mos94] 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 [LP97] 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 [AKP93] and Oz [Smo97]. 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 address the domain of engineering applications that we do.

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 [War99]. 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 [Mey92]. These languages use constraints as specifications; no constraint solving is done at run-time in order to deduce the values of variables. Contracts [HHG90] 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. 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 a constructor. (We take an abstract class to one without a constructor.) Each of these constituents is optional, and an empty class is a degenerate case.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{class\_definition}^{+} \\
\textit{class\_definition} & ::= & \texttt{class}\ \textit{class\_id}\ [\ \texttt{extends}\ \textit{class\_id}\ ]\ \{\ \textit{body}\ \} \\
\textit{body} & ::= & [\ \texttt{attributes}\ \textit{attributes}\ ]\ \ [\ \texttt{constraints}\ \textit{constraints}\ ] \\
 & & [\ \texttt{predicates}\ \textit{pred\_clauses}\ ]\ \ [\ \texttt{preferences}\ \textit{pref\_clauses}\ ] \\
 & & [\ \texttt{constructor}\ \textit{constructor\_clause}\ ]
\end{array}
$$

For the purpose of this paper, we limit attention to single inheritance of classes and at most one constructor for each class. The body of a constructor is just a collection of constraints. Not all of the syntactic details are presented here, a more complete description of the syntax of constraints is given in Appendix A. Here we briefly discuss the more novel aspects of the language.

$$\begin{array}{rcl}
\textit{constraint} & ::= & \textit{creational_constraint} \mid \textit{quantified_constraint} \mid \textit{simple_constraint} \\
\textit{creational_constraint} & ::= & \textit{attribute} = \texttt{new}\ \textit{class_id}(\textit{terms}) \\
\textit{quantified_constraint} & ::= & \texttt{forall}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{constraint} \mid \texttt{exists}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{constraint} \\
\textit{simple_constraint} & ::= & \textit{constraint_atom} \mid \textit{conditional_constraint} \\
\textit{constraint_atom} & ::= & \textit{term}\ \ \textit{relop}\ \ \textit{term} \mid \textit{constraint_predicate_id}(\textit{terms}) \\
\textit{relop} & ::= & = \mid\ != \mid\ > \mid\ < \mid\ >= \mid\ <= \\
\textit{term} & ::= & \textit{constant} \mid \textit{var} \mid \textit{attribute} \mid (\textit{term}) \mid \textit{function_id}(\textit{terms}) \\
 & & \mid \texttt{sum}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \mid \texttt{prod}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\
 & & \mid \texttt{min}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \mid \texttt{max}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\
\textit{conditional_constraint} & ::= & \textit{constraint_atom} :- \textit{literals}
\end{array}$$

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.

Date Example. Below we provide a declarative specification for a *date* object consisting of three integer attributes day, month, and year.

```
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
  constructor Date(d, m, y) { day = d, month = m, year = y }
}
```

The above example illustrates the use of unconditional and conditional constraints, as well as the need for user-defined predicates in the antecedents of conditional constraints. (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 ≤ 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)`.

Gear Train Example. 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 gears 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 designer can specify a preference that the transmission power be as close to 12hp (horse power). The problem is to maximize the efficiency which is also stated as a preference. In appendix B we show how constrained objects can be used in designing a gear train that meets the designer's constraints and preferences.

4 Semantics

4.1 Declarative Semantics

In traditional object-oriented languages, a class defines an abstract datatype whose behavior is characterized by its operations. Cob classes also define abstract datatypes but their "behavior" is characterized declaratively, i.e., by the constraints on the data. We give a set-theoretic semantics for Cob datatypes using an approach that can satisfactorily account for recursively defined classes. We translate a Cob class c into a CLP-like predicate definition p_c, and define the semantics of the class c in terms of the least model of the predicate p_c. As noted in section 1, when preferences are also specified in the class definition, we determine the set-theoretic semantics of the class from the set of preferential consequences of the corresponding PLP predicate. In the discussion below, we first consider programs without preferences. A Cob program C is thus translated to a CLP-like program P by translating each class in C to a corresponding predicate in P. The top-level call on a constructor class is equivalent to a top-level goal or query. The semantics of C are given in terms of the least model of the program P.

Translation of Constrained Objects to CLP. We first show the translation of Cob programs without preferences. We use $\overline{V_i}$, $\overline{C_i}$, $\overline{p}$ to denote collections of variables, constraints and predicates respectively. For every class c in a Cob program C, we define a predicate p_c with two arguments: the first is a list of the arguments of the constructor of c; and the second is the list of attributes of class c. (For predicates corresponding to abstract classes, the first argument is a null list.) If class c' is a subclass of c, then the second argument of predicate $\mathrm{p}_{c'}$ is the list of attributes of c' prepended with $\overline{V}_c$, the list of attributes of the parent class c. The body of predicate $\mathrm{p}_{c'}$ invokes predicate p_c in order that the constraints of the parent class are met by the corresponding attributes of the subclass. This is illustrated in the translation class *parallel* below. The predicates in a Cob class remain unchanged under the translation. To maintain encapsulation and prevent name-clashes, predicate names within a class are suitably tagged during translation.

`class` c {		
`attributes` $\overline{V_1}$ `constraints` $\overline{\mathcal{C}}_1$	$\Longrightarrow_T$	$\mathrm{p}_c(\overline{V}_2, \overline{V}_1) :- \overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2.$
`predicates` $\overline{p}$		$\overline{p}$
`constructor` $c(\overline{V}_2)\{\ \ \overline{\mathcal{C}}_2\ \ \}$		
}		
`class` c' `extends` c {		
`attributes` $\overline{V}'_1$ `constraints` $\overline{\mathcal{C}}'_1$	$\Longrightarrow_T$	$\mathrm{p}_{c'}(\overline{V}'_2, \overline{V}_c \circ \overline{V}'_1)$ `:-` $\mathrm{p}_c(\_, \overline{V}_c), \overline{\mathcal{C}}'_1, \overline{\mathcal{C}}'_2.$
`constructor` $c'(\overline{V}'_2)\{\ \ \overline{\mathcal{C}}'_2\ \ \}$		
}		

The translation of a Cob class to a predicate also involves translation of the constraints and terms in that class. Below we show some aspects of this translation; the full paper will address all aspects. For example, the i^{th} occurrence of a `forall` quantified constraint is translated to a predicate $\texttt{forall}_i$ with suitable parameters that capture the variables in the constraint in the body of the `forall`

`forall` X	in E: $\mathcal{C}$	$\Longrightarrow_T$	`forall`$(E, \mathcal{V})$
where	`forall`$([\ ], \mathcal{V})$.		
	`forall`$([X\,|\,$`Tail`$], \mathcal{V})$ `:-` $\mathcal{C}_T$, `forall(Tail,`$\mathcal{V}$`)`.		
	$\mathcal{V}$ are the variables in $\mathcal{C}$		
and	$\mathcal{C}_T$ is obtained by translating $\mathcal{C}$		

A constructor invocation in Cob is translated to a CLP goal as shown below.

$X =$ `new` $c(\overline{t})$	$\Longrightarrow_T$	$\mathrm{p}_c(\overline{t}_T, X_T)$	
where	X_T is the translation of X		
and	$\overline{t}_T$ is the translation of $\overline{t}$		

To illustrate the above translations with a concrete example, consider the Cob classes shown below. Class `component` models an electrical component with attributes voltage(V), current(I) and resistance (R) that obey Ohm's law (V=I*R). Class `parallel` represents a collection of components(PC) connected in parallel. Below we show these Cob classes and their translation to CLP.

```
class component {
  attributes   Real V, I, R
  constraints V = I * R
  constructors component(V1,I1,R1){
    V = V1.  I = I1.  R = R1   }
}
class parallel extends component {
  attributes component [] PC
  constraints
  forall X in PC : X.V = V.
  sum Y in PC : Y.I = I.
  sum Z in PC : (1/Z.R) = 1/R.
  constructors parallel(B) {
    PC = B }
}
```

$\Longrightarrow_T$

```
p_component([V1, I1, R1], [V, I, R]) :-
  V = I * R, V = V1, I = I1, R = R1.
p_parallel([B],[V,I,R,PC]):- PC = B,
  p_component(_,[V,I,R]), forall1(PC,X,V),
  N1 = I, sum1(PC,N1,Y,I),
  N2 = 1/R, sum2(PC, N2,Z,R).
forall1([],X,V).
forall1([X|Tail],X,V):-
  X_V=V, X=[X_V,X_I,X_R], forall1(Tail,X,V).
sum2([],0,Z,R).
sum2([Z|Tail],(1/Z_R)+ Sumrest,Z,R):-
  sum2(Tail,Sumrest,Z,R), Z = [Z_V,Z_I,Z_R].
sum1([],0,Y,I).
sum1([Y|Tail],(Y_I)+Sumrest,Y,I):-
  sum1(Tail,Sumrest,Y,I), Y=[Y_V,Y_I,Y_R].
```

Translation of Conditional Constraints. Conditional constraints (see Date example) can be given a semantics along the following lines:

$$
\begin{array}{lll}
A \text{ :- } B & \Longrightarrow_T & \texttt{cc}(A,B) \\
 & & \text{where : } \texttt{cc} \text{ is the predefined predicate with semantics :} \\
 & & \qquad \texttt{cc}(A,B) \text{ :- } A. \qquad \text{(i)} \\
 & & \qquad \texttt{cc}(A,B) \text{ :- } \neg B. \qquad \text{(ii)}
\end{array}
$$

Note: While this translation can be given a declarative semantics, with respect to their operational semantics, it may be noted this translation cannot be directly run in CLP because clause (i) of predicate `cc` should be true if `A` is implied by the program executed so far, while asserting A in CLP would mean adding `A` as another constraint to the pool of already present constraints. Similarly, the second clause of `cc` cannot be expressed as is in CLP. Also, CLP does not support constructive negation. In our operational semantics of `cc`, evaluation of conditional constraints is done only after sufficient information about the arguments is obtained (e.g., they become ground).

We now define some terms required in describing the declarative semantics of Cob programs.

Definition 4.1: *A valuation θ is a mapping from the variables of the program to values in the constraint domain $\mathcal{D}$.*

Definition 4.2: *Given a Cob program C, the CLP program P obtained by translating each class of C is called the CLP-translation of C. If c is a class in C, the predicate p_c obtained by translating c is called the CLP-translation of c.*

Definition 4.3: *Given a Cob program C defining a class c, if P is the CLP-translation C, p_c is the CLP-translation of c, and M_p is the least model of P, then the declarative semantics of c, M_c is defined as:* $M_c = \{\overline{t_2} \mid p_c(\overline{t_1}, \overline{t_2}) \in M_p, \textit{for some } \overline{t_1}, \textit{ where } \overline{t_1}, \overline{t_2} \in \mathcal{D}\}$

Definition 4.4: *Given a Cob program C and a goal $G \equiv$ `new` $c(\overline{t})$, if P is the CLP-translation of C, p_c is the CLP-translation of c, and M_p is the least model of P, then a correct answer to G is a valuation θ for the attributes $\overline{t'}$ of c, such that under the valuation θ, $p_c(\overline{t}, \overline{t'})$ is in the least model ,i.e., $M_p \models_\theta p_c(\overline{t}, \overline{t'})$.*

Cob Programs with Preferences. In Cob programs with more than one correct answer, preferences are a way of specifying the criteria for the optimal answer(s). There are two ways of specifying preferences:

(i) using the `min/max` construct (see Gear Train example), and

(ii) using an explicit preference clause: $p(\overline{t}) \preceq p(\overline{u}) : -\overline{L}$.

A Cob program containing preference clauses can be translated to a Preference Logic Program (PLP) in a natural way. The preference clauses of the latter kind are translated as is (with the necessary translation of subterms) while the `min/max` clauses are translated as shown below. We use $\overline{V_i}$, $\overline{C_i}$, $\overline{p}$ to denote collections of variables, constraints and predicates respectively.

$$
\begin{array}{lll}
\texttt{class } c\ \{ & & \\
\quad \texttt{attributes } \overline{V_1} \quad \texttt{constraints } \overline{\mathcal{C}}_1 & \Longrightarrow_T & \mathrm{p}_c(\overline{V}_2, \overline{V}_1) : -\ \overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2. \\
\quad \texttt{predicates } \overline{p} \quad \texttt{preferences min } v & & \overline{p} \\
\quad \texttt{constructor } c(\overline{V}_2)\{\ \ \overline{\mathcal{C}}_2\ \ \} & & \mathrm{p}_c(X, V_1) \preceq \mathrm{p}_c(X, V_1') \text{ :- } v \geq v' \\
\} & &
\end{array}
$$

The paradigm of PLP is capable of expressing both hierarchic as well as recursive optimization problems [GJM96]. Space precludes a full discussion of these issues in this extended abstract. Following [GJM96], we give a possible-worlds semantics for preference logic programs. Consider a predicate p with one or more preference clauses associated with it. The different models of p form different worlds, and an ordering over the worlds is determined by the preference clauses. The preferential consequences of the program are the set of atoms that are true in the *strongly optimal worlds*, i.e., worlds such that there are no other worlds that are better according to the preference clauses. This is in contrast with *logical consequence* which refers to truth in *all* worlds. A *strongly optimal answer* to the query G is a valuation θ such that $G\theta$ is a *preferential consequence* of the preference logic program.

Definition 4.5: *Given a Cob program C with preferences, the PLP program P obtained by translating each class of C is called the PLP-translation of C. If c is a class in C containing preferences, the predicate* p_c *obtained by translating c is called the PLP-translation of c.*

Definition 4.6: *Given a Cob program C containing preferences, if its PLP-translation is P, then the declarative semantics of C is defined to be the set of preferential consequences of the program.*

Definition 4.7: *Given C a Cob program with preferences and an invocation* $c(\overline{t})$*, if the equivalent PLP program is P and the equivalent goal is G, then the correct optimal answer to G is a valuation* θ *such that (G* θ*) is a preferential consequence of P.*

4.2 Operational Semantics

We first consider Cob programs without preferences. 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 $< A, E, I, D, C, N >$, where A is a multiset of atoms and constraints and E, I, D, C, N are respectively multisets of equations, inequations, disequations(`!=`), conditional constraints and negated conjunction of atoms. $\mathcal{T}$ is the constraint theory. Given a goal $G \equiv$ `new` $c(\overline{t})$, the *start state* is $< G, \phi, \phi, \phi, \phi, \phi >$. 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. $< A \cup a, E, I, D, C, N > \rightarrow_r < A \cup B, E \cup \{\overline{t_1} = \overline{t_2}\}, I, D, C, N >$
   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. $< A \cup a, E, I, D, C, N > \rightarrow_r fail$
   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. $$< A \cup c, E, I, D, C, N > \rightarrow_c \begin{array}{ll} < A, E \cup c, I, D, C, N > & \text{if c is an equation} \\ < A, E, I \cup c, D, C, N > & \text{if c is an inequation} \\ < A, E, I, D \cup c, C, N > & \text{if c is a disequation} \\ < A, E, I, D, C \cup c, N > & \text{if c is a conditional constraint} \end{array}$$

4. $$< A, E, I, D, C \cup \{p : -q\}, N > \rightarrow_{cc}$$
   $$\begin{array}{ll} < A, E, I, D, C, N > & \text{if } P, \mathcal{T} \models \overline{\forall}(p \leftarrow (E \cup I \cup D \cup C \cup N)) \\ < A, E, I, D, C, , N \cup \{\neg q\} > & \text{if } P, \mathcal{T} \models \overline{\forall}(\neg p \leftarrow (E \cup I \cup D \cup C \cup N)) \\ < A, E \cup p, I, D, C, N > & \text{if } P, \mathcal{T} \models \overline{\forall}(q \leftarrow (E \cup I \cup D \cup C \cup N)) \text{ and p is an equation} \\ < A, E, I \cup p, D, C, N > & \text{if } P, \mathcal{T} \models \overline{\forall}(q \leftarrow (E \cup I \cup D \cup C \cup N)) \text{ and p is an inequation} \\ < A, E, I, D \cup p, C, N > & \text{if } P, \mathcal{T} \models \overline{\forall}(q \leftarrow (E \cup I \cup D \cup C \cup N)) \text{ and p is a disequation} \end{array}$$

5. $< A, E, I, D, C, N > \rightarrow_s < A, E, I, D, C, N >$ if $consistent$(E, I, D, C, N)
   $< A, E, I, D, C, N > \rightarrow_s fail$ if $\neg consistent$(E, I, D, C, N)

The case-analysis in rule 4 above presents the formal operational semantics of a conditional constraint. (Refer to the Date Example of section 3 for the informal semantics of a conditional constraint.) For example, a case such as $P, \mathcal{T} \models \overline{\forall}(p \leftarrow (E \cup I \cup D \cup C \cup N)$, states that p is implied by the current set of constraints in $E \cup I \cup D \cup C$ possibly together with the negated atoms in N. This checking is carried out by the underlying constraint solver.

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* $< G, \phi, \phi, \phi, \phi, \phi >$, where successive states are obtained by the application of computation rules.

Definition 4.8: *If a derivation for a goal G w.r.t. a program P ends in the state* $< \phi, E, I, D, C, N >$ *such that consistent(E, I, D, C, N), we say that the derivation is successful.* E, I, D, C *and* N *are the answer constraints. A valuation* θ *is a computed answer if the answer constraints evaluate to true under* θ*, i.e.,* $P, \mathcal{T} \models_\theta (E \cup I \cup D \cup C \cup N)$

We assume that the existence of a (sound and complete) constraint solver for determining a computed answer from a set of answer constraints. For example, the creation of a complex object, e.g., `new component(10, I, 5)`, will result in the value for `I`. If the input is not sufficiently instantiated to compute the values of all the remaining variables in top-level query, then the computation results in the computed answer for some of the variables and possibly some answer constraints.

Theorem 4.1 (Soundness): *Given a Cob program* C*, its CLP-translation* P *and a goal G, if a finite derivation for G w.r.t. P ends in* $< \phi, E, I, D, C, N >$ *such that consistent(E, I, D, C, N), then* $M_p \models (E \cup I \cup D \cup C \cup N) \rightarrow G$.

Theorem 4.2 (Completeness): *Given a Cob program* C*, its CLP-translation* P *and a goal G, if a valuation* θ *is such that* $M_p \models_\theta G$*, then there exists a finite derivation with start state* $< G, \phi, \phi, \phi, \phi, \phi >$ *and ending in* $< \phi, E, I, D, C, N >$ *such that consistent(E, I, D, C, N) and* $P, \mathcal{T} \models_\theta (E \cup I \cup D \cup C \cup N)$.

The proof strategy for the soundness and completeness results follows those of the CLP language. That is, our soundness proof is by induction on the length of the derivation and the completeness proof requires the intermediate step of a fixed-point semantics. The presence of user-defined predicates in the antecedant of a conditional constraint can give rise to nontermination. Hence the proof of completeness assumes the termination of the goals involving these predicates.

The operational semantics given above can be compared with the semantics of CLP languages given in [JM94]. The variables E, I, D form the set of active constraints, i.e., the set of constraints given to the constraint solver, and the variables C, N form the set of passive constraints which are periodically examined by the compiler to see if any of them can be passed to the constraint solver.

Cob Programs with Preferences. We provide a derivation scheme called Pruned Tree SLD-derivation, along the lines of [GJM96], for efficiently computing the strongly optimal answers to queries. In essence, PTSLD-derivations incrementally construct the search tree from a goal and use the preference clauses as advice to prune paths that compute non-optimal answers. Essentially a tree T_1 **derives** a tree T_2, written $T_1 \Rightarrow T_2$, if T_2 is obtained from T_1 by reducing a goal in a leaf of T_1 in all possible ways and creating the corresponding offspring nodes. Paths that compute sub-optimal solutions are pruned by consulting the preference clauses. We refer the reader to [GJM96] for the details of the operational semantics. The soundness of PTSLD-derivations requires, among other things, that the predicates be sufficiently non-ground when invoked. The completeness of PTSLD derivations requires that the search tree for every optimization goal is finite, because the optimal solution cannot in general be computed unless the search tree is exhaustively explored.

Current Status. We have developed three tools: (i) A Constraint-based UML tool called CUML for authoring Cob class diagrams. The output of this tool is 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 textual Cob code that instantiates the appropriate classes.

References

[AKP93] H. Ait-Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993.

[BNFB92] 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.

[Bor81] A. Borning. The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS*, 3(4):252–287, 1981.

[Fre89] E.C. Freuder. Partial Constraint Satisfaction. In *Proc. 11th Intl. Jt. Conf. on Artificial Intelligence*, pages 278–283, 1989.

[FW95] E.C. Freuder and R.J. Wallace. Heuristic Methods for Over-Constrained Constraint Satisfaction Problems. In *Proc. CP'95 Workshop on Overconstrained Systems*, 1995.

[GJM96] 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.

[GL94] A. Borning G. Lopez, B. N. Freeman-Benson. Constraints and Object Identity. In *Proc. European Conference On Object-Oriented Programming*, 1994.

[Gov97] K. Govindarajan. *Optimization and Relaxation in Logic Languages.* PhD thesis, Department of Computer Science, SUNY - Buffalo, 1997.

[HHG90] 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.

[Hor92] B. Horn. Constraint Patterns As a Basis For Object Oriented Programming. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 1992.

[JL87] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Prog.*, 1994.

[Lel87] W.J. Leler. *The Specification and Generation of Constraint Satisfaction Systems.* Addison-Wesley, 1987.

[Lop97] G. Lopez. *The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language.* PhD thesis, University of Washington, 1997.

[LP97] 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.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[MM82] R. Mayne and S. Margolis. *Introduction to Engineering.* McGraw-Hill, 1982.

[Mos94] C. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming.* Addison-Wesley, 1994.

[Smo97] G. Smolka. Constraint Programming in Oz (Abstract). In *Proc. Intl. Conference on Logic Programming*, pages 37–38, 1997.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[Wan84] M. Wand. A types-as-sets semantics for Milner-style polymorphism. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, pages 158–164, 1984.

[War99] Warmer, J., Kleppe, A. *The Object Constraint Language.* Addison-Wesley, 1999.

Appendix A: Cob Syntax

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{class\_definition}^{+} \\
\mathit{class\_definition} & ::= & \texttt{class}\ \mathit{class\_id}\ [\ \texttt{extends}\ \mathit{class\_id}\ ]\ \{\ \mathit{body}\ \} \\
\mathit{body} & ::= & [\ \texttt{attributes}\ \mathit{attributes}\ ][\ \texttt{constraints}\ \mathit{constraints}\ ] \\
 & & [\ \texttt{predicates}\ \mathit{pred\_clauses}\ ][\ \texttt{preference}\ \mathit{pref\_clauses}\ ] \\
 & & [\ \texttt{constructor}\ \mathit{constructor\_clause}\ ] \\
\mathit{attributes} & ::= & \mathit{decl}\ [\ .\ \mathit{decl}\ ]^{+} \\
\mathit{decl} & ::= & \mathit{type}\ \mathit{id\_list} \\
\mathit{type} & ::= & \mathit{primitive\_type\_id} \mid \mathit{class\_id} \mid \mathit{type}[] \\
\mathit{primitive\_type\_id} & ::= & \texttt{Real} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Char} \mid \texttt{String} \\
\mathit{id\_list} & ::= & \mathit{attribute\_id}\ [\ ,\ \mathit{attribute\_id}\ ]^{+} \\
\mathit{constraints} & ::= & \mathit{constraint}\ [\ .\ \mathit{constraint}\ ]^{+} \\
\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{attribute} & ::= & \mathit{selector}[.\mathit{selector}]^{+} \mid \mathit{attribute}[\mathit{term}] \\
\mathit{selector} & ::= & \mathit{attribute\_id} \mid \mathit{selector\_id}(\mathit{terms}) \\
\mathit{selector\_id} & ::= & \texttt{first} \mid \texttt{next} \mid \texttt{last} \\
\mathit{terms} & ::= & \mathit{term}\ [\ ,\ \mathit{term}\ ]^{+} \\
\mathit{conditional\_constraint} & ::= & \mathit{constraint\_atom} :- \mathit{literals} \\
\mathit{literals} & ::= & \mathit{literal}\ [\ ,\ \mathit{literal}\ ]^{+} \\
\mathit{literal} & ::= & [\ \texttt{not}\ ]\ \mathit{atom} \\
\mathit{atom} & ::= & \mathit{predicate\_id}(\mathit{terms}) \mid \mathit{constraint\_atom} \\
\mathit{pred\_clauses} & ::= & \mathit{pred\_clause}\ .\ [\ \mathit{pred\_clause}\ .\ ]^{+} \\
\mathit{pred\_clause} & ::= & \mathit{clause\_head} :- \mathit{clause\_body} \\
\mathit{pred\_clause} & ::= & \mathit{clause\_head} \\
\mathit{clause\_head} & ::= & \mathit{predicate\_id}(\mathit{terms}') \\
\mathit{clause\_body} & ::= & \mathit{goal}\ [\ ,\ \mathit{goal}\ ]^{+} \\
\mathit{goal} & ::= & [\ \texttt{not}\ ]\ \mathit{predicate\_id}(\mathit{terms}') \\
\mathit{terms}' & ::= & \mathit{term}'\ [\ ,\ \mathit{term}'\ ]^{+} \\
\mathit{term}' & ::= & \mathit{constant} \mid \mathit{var}' \mid \mathit{attribute} \mid \mathit{function\_id}(\mathit{terms}') \\
\mathit{pref\_clauses} & ::= & \mathit{pref\_clause}\ .\ [\ \mathit{pref\_clause}\ .\ ]^{+} \\
\mathit{pref\_clause} & ::= & \mathit{predicate\_id}(\mathit{terms}') \leq \mathit{predicate\_id}(\mathit{terms}') :- \mathit{clause\_body}
\end{array}
$$

Appendix B: On the next page we illustate the Constraint-UML notation for Cob classes using the Gear Train design example.

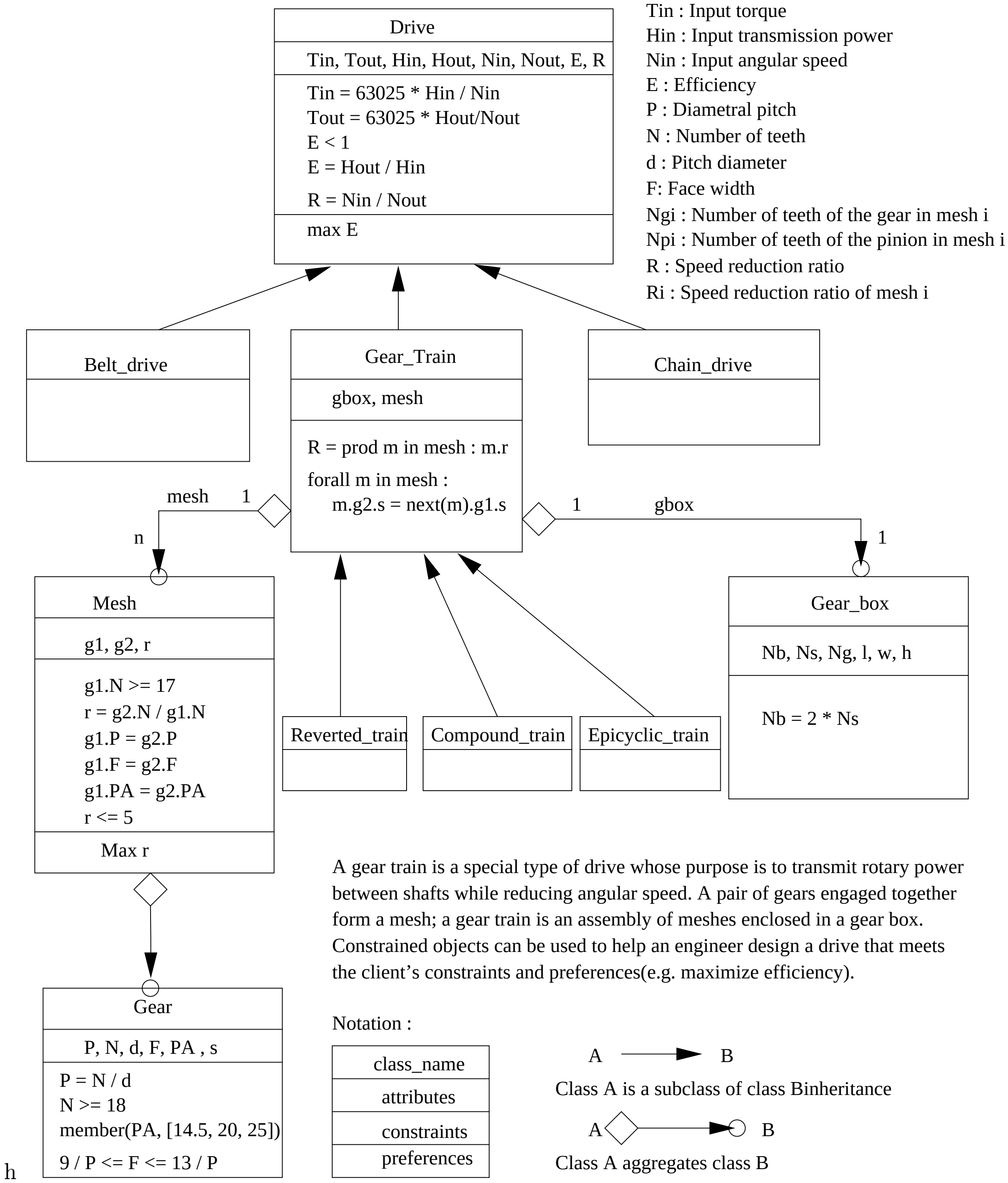

Figure 2: Class Hierarchy for Gears