Constrained Objects for Modeling Complex Systems[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

The goal of this research is to develop a programming language and execution environment for modeling complex engineering entities. We propose to model complex systems using a compositional approach. The building blocks in this approach are objects whose internal states must obey a set of invariants, or constraints. When such objects are aggregated to form a complex object, their internal states might further have to satisfy interface constraints. This paradigm of objects is referred to as *constrained objects*, and may be regarded as a declarative form of object-oriented programming. We propose to extend earlier work by providing a richer language, called **Cob**, for specifying constrained objects: We support both the notion of conditional constraints, preferences, and logic variables at the modeling level, and the computational model provides constraint satisfaction, optimization, and incremental recomputation. Cob will be an expressive tool for applications arising in engineering, information and organizational modeling.

1 Introduction

The goal of this research is to develop a programming language and execution environment that will facilitate a principled approach to the modeling of complex engineering entities, such as circuits, trusses, gears, mixers, separaters, 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 is inappropriate, because it is more natural to think of each component as being governed by certain laws, or invariants.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

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

constraints as *constrained objects*, and it may be regarded as a declarative approach to object-oriented programming.

Consider an electrical circuit made of resistors and batteries. We can think of these entities as objects. For example, the state of a resistor may be represented by three variables `V`, `I`, and `R`, which represent 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 express the law of mass balance.

A modeler will 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 constraint satisfaction. A modeler will then go through one or more iterations of *modifying* the component objects followed by re-execution(*re-solve*). 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 possible assignment of values to attributes that will satisfy some given constraints in addition to the ones already present in the constrained objects. Thus, we envisage that a constrained object will not only have a declarative part (constraints), but also a procedural part (to support changes to its state/attribute).

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. 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. 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. We propose to extend earlier work by providing a richer language, called **Cob**, for specifying constrained objects: We support both the notion of conditional constraints, preferences, and logic variables at the modeling level, and the computational model provides constraint satisfaction, optimization, and incremental recomputation. Cob will be an expressive tool for applications arising in engineering, information and organizational modeling.

The remainder of this paper is structured as follows: In section 2 we provide a brief account of related work. Section 3 presents the syntax of our proposed language and examples illustrating the use of constructs of the language for different kinds of problems. Section 4 sketches the overall computational model for constrained objects–the execution engine basically traverses the network of objects, accumulates constraints, and solves them. Finally, section 5 presents our conclusions and areas of further work.

2 Related Work

We now survey closely related language design efforts that also capture some notion of constrained objects. We note that none of these languages support the paradigm to the extent that we propose. In particular, they do not support conditional constraints, logic variables and search, nor do they give the programmer provision for declaratively specifying preferences among multiple solutions or the ability to model optimization problems.

An early forerunner in the field is the pioneering work of Alan Borning on ThingLab, a constraint-based simulation laboratory [Bor81]. ThingLab builds on the object-oriented language Smalltalk, enhancing it to provide constructs that specify constraints within class definitions. Inheritance of constraints and multiple inheritance is permitted. ThingLab is primarily intended for interactive graphical simulations, and provides every constrained object with a visual appearance. It also supports the idea of planning how to solve the constraints, as well as constraint propagation and relaxation.

Another early work, also aimed at graphics applications, is the paradigm of constrained abstract types described by Leler in his programming language Bertrand (which was developed under the supervision of Jayaraman at UNC) [Lel87]. Bertrand employs a technique called *augmented term-rewriting*, which is essentially term-rewriting with simple equation solving. Compared to our proposed language, both these approaches provide only a limited capability for expressing constraints, and also provide little support for handling multiple solutions to constraints. In Thinglab, the only way the programmer can indicate his/her preference is by ordering the methods of constraint satisfaction. In our proposed language, we will make explicit use of the concept of *preference* for this purpose [GJM96].

More recently, the constrained object language Siri [Hor92, Hor93] was proposed by Bruce Horn, and is essentially a superset of the Bertrand language. The notion of *event pattern* is used to declaratively specify state changes: this is done by declaring what relations must hold as a result of the method, and also specifying which attributes may change and which may not in order to satisfy these relations. Siri uses Bertrand's augmented term-rewriting for equation-solving. Consistency and integrity of the object is maintained, since there is no mechanism for directly setting the state of an object that disregards the object's constraint specification.

The Kaleidoscope '91 [BNFB92] language integrates constraints and object-oriented programming, and is also aimed at supporting the development of interactive graphical user interfaces. The language also supports constraint hierarchies, constraint constructors, and multi-methods. The main issue that it addresses is solving constraints over the types of variables as well as constraints over the values of variables. There are three primitive domain constraint solvers, and provision is made so that they can communicate to solve inter-domain constraints. In Kaleidoscope '93, constraints are solved according to their type: class/type constraints, identity constraints, or value/structure constraints.

3 Cob: Syntax and Examples

The grammar below outlines the overall structure of a Cob program. The syntax below is suggestive of the kind of language we envision; not all of the syntactic details are presented here.

A Cob program is essentially a sequence of class definitions, and each constrained object is an instance of some class. Currently, we limit our attention to single inheritance of classes.

$$
\begin{aligned}
\textit{program} \quad &::= \quad \textit{class\_definition}^{+} \\
\textit{class\_definition} \quad &::= \quad \texttt{class}\ \textit{class\_id}\ [\ \texttt{extends}\ \textit{class\_id}\ ]\ \{\ \textit{body}\ \}
\end{aligned}
$$

The body of a class definition consists of the attributes, constraints, functions, predicates, preferences, constructors, and other methods. Each of these constituents is optional, and we permit an empty class definition as a degenerate case.

$$
\begin{aligned}
\textit{body} \quad ::= \quad & [\ \texttt{attributes}\ \textit{attributes}\ ] \\
& [\ \texttt{constraints}\ \textit{constraints}\ ] \\
& [\ \texttt{functions}\ \textit{func\_clauses}\ ] \\
& [\ \texttt{predicates}\ \textit{pred\_clauses}\ ] \\
& [\ \texttt{preference}\ \textit{pref\_clauses}\ ] \\
& [\ \texttt{constructors}\ \textit{constructor\_clauses}\ ] \\
& [\ \texttt{methods}\ \textit{method\_definitions}\ ]
\end{aligned}
$$

A constraint can be either simple or quantified, where the quantification ranges over an enumeration. 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.

$$
\begin{aligned}
\textit{constraints} \quad &::= \quad \textit{constraint}\ [\ .\ \textit{constraint}\ ]^{+} \\
\textit{constraint} \quad &::= \quad \textit{quantified\_constraint} \mid \textit{simple\_constraint} \\
\textit{quantified\_constraint} \quad &::= \quad \forall\ \textit{var} \in \textit{enum}\ .\ \textit{constraint} \\
\textit{simple\_constraint} \quad &::= \quad \textit{constraint\_atom} \mid \textit{conditional\_constraint} \\
\textit{constraint\_atom} \quad &::= \quad \textit{term}\ \ \textit{relop}\ \ \textit{term} \mid \textit{constraint\_predicate\_id}(\textit{terms}) \\
\textit{relop} \quad &::= \quad = \mid\ != \mid\ > \mid\ < \mid\ >= \mid\ <= \\
\textit{term} \quad &::= \quad \textit{constant} \mid \textit{var} \mid \textit{attribute} \mid \textit{function\_id}(\textit{terms}) \mid \sum \textit{var} \in \textit{enum}\ .\ \textit{term} \\
\textit{attribute} \quad &::= \quad \textit{selector}[.\textit{selector}]^{+} \\
\textit{selector} \quad &::= \quad \textit{attribute\_id} \mid \textit{selector\_id}(\textit{terms}) \\
\textit{selector\_id} \quad &::= \quad \texttt{first} \mid \texttt{next} \mid \texttt{last} \\
\textit{terms} \quad &::= \quad \textit{term}\ [\ ,\ \textit{term}\ ]^{+} \\
\textit{conditional\_constraint} \quad &::= \quad \textit{constraint\_atom} :- \textit{literals} \\
\textit{literals} \quad &::= \quad \textit{literal}\ [\ ,\ \textit{literal}\ ]^{+} \\
\textit{literal} \quad &::= \quad [\ \texttt{not}\ ]\ \textit{atom} \\
\textit{atom} \quad &::= \quad \textit{predicate\_id}(\textit{terms}) \mid \textit{constraint\_atom}
\end{aligned}
$$

In the above grammar, we distinguish between ordinary *predicate_id* and *constraint_predicate_id*. The former are defined by the user using Prolog-like rules whereas the latter are a set of predefined predicates (as in CLP-like languages) whose properties are known to the underlying constraint satisfaction system.

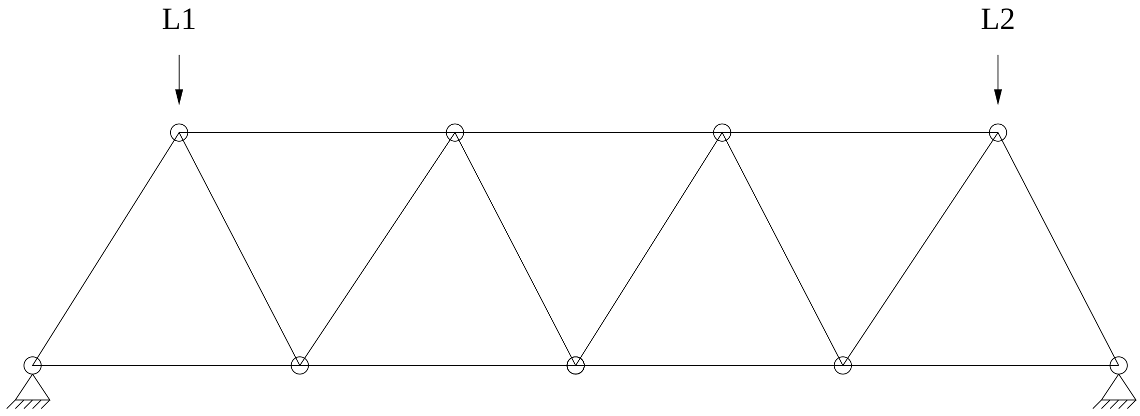

Figure 1: A Simple Truss

Simple Circuit as a constrained object We present an example to illustrate the syntax of Cob and the use of equational and quantified constraints. This is the well-known example of an electrical circuit consisting of a series-parallel combination of resistors. 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 main property of interest in this example is Ohm's law, namely, `v = i*r`, where `v` stands for voltage, `i` for current and `r` for resistance. Notice the use of the quantified constraints ($\forall$ and Σ) for specifying the laws of series and parallel circuits.

```
class Component {
  attributes
      Real v, i, r
  constraints
      v = i * r
  constructor Component(v1, i1, r1) {
      v = v1, i = i1, r = r1
}}
class Series extends Component {
  attributes
      Component [] C
  constraints
      ∀c∈C. i = c.i.
      v = ∑c∈C. c.v.
      r = ∑c∈C. c.r
  constructor series(c) { C = c }
}
```

```
class Parallel extends Component {
  attributes
      Component [] C
  constraints
      ∀c∈C. v = c.v.
      i = ∑c∈C. c.i.
      1/r = ∑c∈C. 1/c.r
  constructor parallel(c) { C = c }
}
```

A Simple Truss as a Constrained Object To illustrate the use of constrained objects in representing an engineering design, we provide two classes below to model bars and joints in a simple truss structure shown in figure 1. The `Joint` class aggregates an array of bars (that are incident at the joint) and effectively states 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). These conditions are taken from the text by Mayne and Margolis [MM82]. The following formulation is just one possible way to model the problem, and undoubtedly there are other approaches.

```
class Joint {
 attributes
   Bar[] B
 constraints
   ∑ b∈B. b.f_bn * sin(b.angle) = 0.
   ∑ b∈B. b.f_bn * cos(b.angle) = 0
 constructor Joint(B1) { B = B1 }
}
```

```
class Bar {
 attributes
   Real angle, E, Sy, l,
   w, h, f_bn, f_bk, f_t, σ, I
 constraints
   0 ≤ angle.  angle ≤ 2*π.
   I = f_bk * l^2 /(π^2*E).
   I ≤ w * h^3 / 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
 } }
```

Document Layout as a Constrained Object Problem Another novel feature of our work is that we support both the notion of constraints as well as preferences at the modeling level. Preferences are useful in choosing the optimal solution(s) to a set of constraints/predicates. Suppose we are interested in formatting the contents of a 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 the setting of this example, there are certain important constraints that we must deal with, e.g.: "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 class `Para`, where we state that the format with lesser badness is the preferred one as : `format(L1, B1) < format(L2, B2) :- B1 > B2.`

```
class Book {
 attributes
   Chapter[] chapters
 constraints
   first(chapters).begin_pg = 1.
   ∀ c ∈ chapters.
   ((c.end_pg+1=next(c).begin_pg)
   :-not is_last(c))
 predicates
   format(Bpgs)
   :-format(chapters,Bpgs).
   format([], []).
   format([C1|C],[CP1|CPgs])
   :-C1.format(CP1),format(C,CPgs).
}
```

```
class Chapter {
 attributes
   Section[] sections.
   Int begin_pg, end_pg
 constraints
   first(sections).begin_ln = 1.
   end_pg = begin_pg + Numpgs - 1.
   ∀ s ∈ sections.
   ((s.end_ln+1=next(s).begin_ln)
   :-not is_last(s)).
 predicates
   format(Cpgs)
   :-format(sections,SLines),
   form_pgs(SLines,Cpgs,Numpgs).
   format([], []).
   format([S1|S],[SL1|SLines])
   :-S1.format(SL1),format(S,SLines).
}
```

```
class Section {
 attributes
   Para[] paras.
   Int begin_ln, end_ln
 constraints
   begin_ln mod 50 != 0.
   end_ln mod 50 != 1.
   begin_ln = first(paras).begin_ln.
   end_ln = last(paras).end_ln.
   ∀ p ∈ 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.
   Char[] words
 constraints
   end_ln = begin_ln + Numlines - 1.
   width = 80
 predicates
   format(Lines, Badness) :-
      ... details omitted ...
 preference
   format(L1, B1) < format(L2, B2) :-
      B1 > B2.
}
```

4 Computational Model

We now describe the computational engine of Cob developed so far. Given a complex object, i.e., an assembly of constrained objects, the Cob Abstract Machine tries to solve the constraints and arrive at values for the attributes of the constrained objects. Depending upon the constraints/predicates/preferences that are present, a particular computational strategy is chosen. There are several important cases that arise: (i) linear/non-linear, simultaneous equality constraints (e.g. as in the electric circuit); (ii) linear inequality constraints with preference criteria, or objective functions; (iii) linear/non-linear inequality constraints; (iv) conditional constraints and (v) constraints and predicates with preferences. Each of the above-mentioned cases is a full-fledged topic in its own right. We will discuss cases (i), (iii) and (iv) here.

A complex object may be depicted by an object graph. Each node of an object graph represents an object (i.e., instance of some class) and each directed edge represents an aggregation relationship. We introduce the notion of a constraint graph which is based on the object graph. Replacing each node of the object graph with the constraints of the object it represents results in the formation of the constraint graph. The structure of a complex object is used in building a plan to solve its constraints. This structure is reflected both by the object graph and the constraint graph. The initial constraint solving is done by a bottom up traversal of the constraint graph of the complex object. Some of the attributes at the nodes of the constraint graph may have values assigned to them. The values of the attributes and the constraints at a node, are used by the abstract machine to calculate the values of the remaining attributes and rearrange the constraints into a simpler form. This phase of constraint solving leaves the complex object in a state where the current values of its attributes satisfy the given constraints and the values of no more attributes can be inferred. The state of the complex object can now be *modified* or *queried* by invoking state-change methods or predicates. When the state changes, constraints are re-solved to bring the object back to a state where no constraints are violated. We give below, constraint *solving* for each of the three types of constraints : equality, inequality and conditional (separately and together). Notation used:
CG: Constraint graph.
Q: Bottom up, breadth first order on nodes of CG.
θ: A substitution of attribute-value pairs, where value is a ground term.
ρ: A substitution of attribute-expression pairs, where expression may have non-ground terms.
$(X\theta)$: Normalized application of substitution θ to X, to remove all occurances of variables of θ from X. If this deduces values of any attributes, then these values are used to perform further substition on X, continuing till no more attribute-value pairs are obtained.
`normalize`(E): rewrite the set of linear equations/inequations such that a variable on the l.h.s. of any equation (called dependent variable) does not appear on the r.h.s. of any equation.
`solve`(E): Solve the set of equations (or inequations) in set E simultaneously, to obtain attribute-value (and attribute-expression) pairs.

Constraint Solving for Linear Equations:
`INPUT`: CG
`OUTPUT`: θ, ρ
$\theta \leftarrow \phi, \rho \leftarrow \phi$
while (Q not empty) {
    $n = \texttt{head}(Q)$
    $\theta_n = \texttt{solve}(E_n)$
    $\rho_n = \texttt{normalize}(E_n\ \theta_n)$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n$
    $Q \leftarrow \texttt{tail(Q)}$ }
$(\rho\ \theta)$

Constraint Solving for Linear Inequations:
`INPUT`: CG
`OUTPUT`: θ, ρ, σ
$\theta \leftarrow \phi, \rho \leftarrow \phi, \sigma \leftarrow \phi$
while (Q not empty) {
    $n = \texttt{head}(Q)$
    $(\theta_n, \rho_n) = \texttt{solve}(I_n \cup \sigma)$
    $\sigma_n = \texttt{normalize}(I_n\ \theta_n\ \rho_n)$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n,\ \sigma \leftarrow \sigma \cup \sigma_n$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $\sigma \leftarrow \texttt{normalize}\ (\sigma\ \theta_n\ \rho_n)$
    $Q \leftarrow \texttt{tail}(Q)$ }
$(\rho\ \theta), (\sigma\ \theta)$

Constraint Solving for Conditional Constraints:
INPUT : CG
OUTPUT: θ, ρ, σ, γ
$\theta \leftarrow \phi, \rho \leftarrow \phi, \sigma \leftarrow \phi, \gamma \leftarrow \phi$
while (Q not empty) {
    $n = \texttt{head}(Q)$
    $b_n \leftarrow \phi$
    foreach $c \in C_n$ {
        if ($\texttt{not head}(c)$)
        $b_n \leftarrow b_n \cup$ (not body(c)) }
    $(\theta_n, \rho_n, \sigma_n) = \texttt{solve}(b_n \cup \gamma)$
    $\gamma_n = \texttt{normalize}\ (b_n\ \theta_n\ \rho_n)$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n,$
    $\sigma \leftarrow \sigma \cup \sigma_n,\ \gamma \leftarrow \gamma \cup \gamma_n$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $\sigma \leftarrow \texttt{normalize}(\sigma\ \theta_n\ \rho_n)$
    $\gamma \leftarrow \texttt{normalize}(\gamma\ \theta_n\ \rho_n)$
    $Q \leftarrow \texttt{tail}(Q)$ }
$(\rho\ \theta), (\sigma\ \theta), (\gamma\ \theta)$

Combined Constraint Solving:
INPUT : CG
OUTPUT: θ, ρ, σ γ
$\theta \leftarrow \phi,\ \rho \leftarrow \phi,\ \sigma \leftarrow \phi,\ \gamma \leftarrow \phi$
while (Q not empty) {
    $n = \texttt{head}(Q)$
    $\theta_n = \texttt{solve}(E_n)$
    $\rho_n = \texttt{normalize}(E_n\ \theta_n)$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n$
    $(\theta_n, \rho_n) = \texttt{solve}(I_n \cup \sigma)$
    $\sigma_n = \texttt{normalize}(I_n\ \theta_n\ \rho_n)$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n,\ \sigma \leftarrow \sigma \cup \sigma_n$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $b_n \leftarrow \phi$
    foreach $c \in C_n$ {
        if ($\texttt{not head}(c)$)
        $b_n \leftarrow b_n \cup$(not body(c)) }
    $(\theta_n, \rho_n, \sigma_n) = \texttt{solve}(b_n \cup \gamma)$
    $\gamma_n = \texttt{normalize}\ (b_n\ \theta_n\ \rho_n)$
    $\theta \leftarrow \theta \cup \theta_n,\ \rho \leftarrow \rho \cup \rho_n,$
    $\sigma \leftarrow \sigma \cup \sigma_n,\ \gamma \leftarrow \gamma \cup \gamma_n$
    $Q \leftarrow Q\ \theta_n\ \rho_n$
    $\sigma \leftarrow \texttt{normalize}(\sigma\ \theta_n\ \rho_n)$
    $\gamma \leftarrow \texttt{normalize}(\gamma\ \theta_n\ \rho_n)$
    $Q \leftarrow \texttt{tail}($ Q $)$ }
$(\rho\ \theta), (\sigma\ \theta), (\gamma\ \theta)$

5 Conclusions and Future Work

We have presented several illustrative examples of the paradigm of constrained objects in this paper, and also briefly sketched its computational model. Our work 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 believe that these features will be especially useful in the modeling of complex engineering structures. Presently we are working on a number of modeling applications with faculty 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 lifecycle 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 also investigating an implementation of the underlying engine for Cob, having earlier completed a prototype implementation of constraints and preferences. We are also providing a visual representation for objects, along the lines of a tool such as AutoCAD, except that we also provide a constrained object representation for each component that makes up a visual representation.

There are a number of areas of future work. 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. Identifying patterns of constraints between objects and providing the programmer with a library of constraint patterns will enable faster modeling and reuse. 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 [FW95] 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.

In some applications it may be necessary to work with not just linear or non-linear equations, but differential equations. This would arise in electrical circuits with inductors and capacitors. Here, the resultant behavior of the complex object is time-varying, and has more of the flavor of a simulation. When a 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*. One issue of special interest in relation to constrained objects 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.

References

[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.

[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.

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

[Hor93] B. Horn. *Constrained Objects*. PhD thesis, CMU, November 1993.

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

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