Modeling Dynamic Systems with Constrained Objects

Robert Raux
Bharat Jayaraman

Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260-2000
rjraux@cse.buffalo.edu
bharat@cse.buffalo.edu

Abstract. This paper examines the application of constrained objects for modeling dynamic systems. A dynamic system is one whose state changes with time. A constrained object is an object whose internal state is governed by a set of (declarative) constraints. When a complex system is modeled as a set of constrained objects, the resultant behavior is deduced through a process of constraint satisfaction. In previous research, we explored the paradigm of constrained objects for modeling the steady-state behavior of systems. In this paper, we present extensions that allow time-depedent behavior to be modeled as well. A key feature of our proposed approach is that of a time-series variable whose values are in accordance with specified constraints. We provide examples from diverse domains (AC circuits, hydrological modeling, and sensor networks) to illustrate our approach. The current prototype includes a compiler that translates Cob programs into Sicstus prolog Objects with CLP(R) constraints, and a domain specific visual interface for building and testing wireless sensor networks.

1 Introduction

We present a programming language and modeling environment that will facilitate a principled approach to modeling dynamic systems. Our approach is based upon the language paradigm of *constrained objects* introduced in earlier research [5]. This paradigm involves modeling a component as a collection of objects, each with a certain set of attributes, as well as a set of behavioral laws that the component must satisfy at all times. These behavioral laws are thought of as constraints over the attributes of the constituent objects. Using this approach, objects may be aggregated to form complex systems. Such an approach retains many of the advantages of object-oriented languages, such as classes and hierarchies, but allows invariants to be described in a more natural manner.

The motivation for the present paper stems from the observation that most of the problems that have been modeled using Cob have focused on the steady-state behavior of systems such as circuits, trusses, buildings, etc. The dynamic systems that we consider in this paper are those whose state changes with time. In some instances, this the state-change can be characterized in a mathematical way, such as the behavior of an AC voltage source. In other examples, such as in hydrologic modeling of rainfall and runoff, certain aspects of the time-varying behavior, such as water balance, can be characterized by behavioral constraints, while other aspects such as rainfall over time must be provided as time-series data. In modeling such dynamic systems, we also need to maintain information regarding previous states and also enforce constraints that relate a state to those of its previous or succeeding states. We therefore introduce a novel extension, namely, that of a *series variable*, whose values are generally constrained according to a mathematical series. It is also possible to initialize such a variable to a pre-specified series in those cases when a mathematical characterization is not possible. A series variable is one whose values are 'updated' according to a constraint. In this way, we provide for a disciplined form of a mutable object in the language.

We begin with a brief description of the Cob programming language and illustrate some of its uses for engineering modeling. We then explain through examples the problems in modeling complex dynamic systems using Cob, and thereby motivate the proposed language extensions for overcoming these difficulties. The remainder of this paper is organized as follows: Section 2 outlines the syntax of Cob, presents several useful examples, then explores through example the shortcomings of modeling dynamic systems with Cob. Section 3 outlines the syntax additions to Cob and illustrates their effectiveness through examples. Finally, section 4 presents our conclusions, status, and future research plans. All the examples presented in this paper have been tested using our current prototype.

2 Cob: Basic Paradigm

Basic Syntax. The grammar below defines the current structure of a Cob program. A Cob program is a sequence of class definitions and each constrained object is an instance of some class. The body of a class definition consists of attributes, constraints, predicates, and constructors. Each of these constituents is optional.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{class\_definition}^{+} \\
\textit{class\_definition} & ::= & [\ \texttt{abstract}\ ]\ \texttt{class}\ \textit{class\_id}\ [\ \texttt{extends}\ \textit{class\_id}\ ]\ \{\ \textit{body}\ \} \\
\textit{body} & ::= & [\ \texttt{attributes}\ \textit{attributes}\ ] \\
 & & [\ \texttt{constraints}\ \textit{constraints}\ ] \\
 & & [\ \texttt{predicates}\ \textit{pred\_clauses}\ ] \\
 & & [\ \texttt{constructors}\ \textit{constructor\_clause}\ ] \\
\textit{attributes} & ::= & \textit{decl}\ ;\ [\ \textit{decl}\ ;\ ]^{+} \\
\textit{decl} & ::= & \textit{type}\ \textit{id\_list} \\
\textit{type} & ::= & \textit{primitive\_type\_id}\ |\ \textit{class\_id}\ |\ \textit{type}[\,] \\
\textit{primitive\_type\_id} & ::= & \texttt{real}\ |\ \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{char}\ |\ \texttt{string} \\
\textit{id\_list} & ::= & \textit{attribute\_id}\ [\ ,\ \textit{attribute\_id}\ ]^{+}
\end{array}
$$

An attribute is a typed identifier, where the type is either a primitive type or a user defined type (i.e. class name) or an array of primitive or user-defined typed. Not all of the syntactic details are presented here; a complete description of all syntax of constraints is given in Appendix A. Below we discuss the more novel aspects of the language.

$$
\begin{array}{rcl}
\textit{constraints} & ::= & \textit{constraint}\ ;\ [\ \textit{constraint}\ ;]^{+} \\
\textit{constraint} & ::= & \textit{creational\_constraint} \mid \textit{quantified\_constraint} \\
 & & \mid \textit{simple\_constraint} \\
\textit{creational\_constraint} & ::= & \textit{attribute} = \texttt{new}\ \textit{class\_id}(\textit{terms}) \\
\textit{quantified\_constraint} & ::= & \texttt{forall}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{constraint} \\
 & & \mid \texttt{exists}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{constraint} \\
\textit{simple\_constraint} & ::= & \textit{conditional\_constraint} \mid \textit{constraint\_atom} \\
\textit{conditional\_constraint} & ::= & \textit{constraint\_atom} : -\ \textit{literals} \\
\textit{constraint\_atom} & ::= & \textit{term}\ \ \textit{relop}\ \ \textit{term} \mid \textit{constraint\_predicate\_id}(\textit{terms}) \\
\textit{relop} & ::= & = \mid\ != \mid\ > \mid\ < \mid\ >= \mid\ <= \\
\textit{term} & ::= & \textit{constant} \mid \textit{var} \mid \textit{attribute} \mid (\textit{term}) \mid \textit{func\_id}(\textit{terms}) \\
 & & \mid \texttt{sum}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\
 & & \mid \texttt{prod}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\
 & & \mid \texttt{min}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\
 & & \mid \texttt{max}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term}
\end{array}
$$

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.[5]

<u>*Series circuits.*</u> To begin to demonstrate the syntax and use of Cob we present the well known example of a series circuit. We have left out the battery class for the sake of brevity. The `resistor` class models a physical resistor conforming to Ohm's law. The `series` class is constructed with an array of resistors. When the resistors are placed in a series, the current that runs through each must be the same. The total voltage and resistance for the series of resistors is the sum. This is represented below through the use of quantified constraints (forall,sum). Given initial values for some attributes, this model can be used to calculate any remaining values.

```
class resistor {
  attributes
    real V, I, R;
  constraints
    V = I * R;
  constructors resistor(V1, I1, R1) { V = V1;   I = I1;   R = R1; }
}
class series {
  attributes
    resistor[] RS;
  constraints
    forall C in RS: C.I = I;
    sum D in RS: (D.V) = V;
    sum E in RS: (E.R) = R;
  constructors series(A) { RS = A; }
}
```

Heat Transfer in a Plate. Our next example is a classic problem. The goal is to model a plate in which the temperature of any of the interior points is the average of the temperature of it's neighboring four points. This can be stated mathematically by using 2d Laplace's equations. The Cob representation below uses this method. The constructor initializes the perimeter points of the 11 x 11 plate. The constraints then calculate all of the internal 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;
  }
}
```

Hydrology. We illustrate in the problems in modeling a dynamic system in the basic paradigm of constrained objects with a simple example from hydrology. Below, we are assuming the following meaningful variables are defined:

```
P = precipitation (gradually drops off to 0.1)
Q = flow between nodes, varies over time
S = volume at the node
F = infiltration (steadily increases)
A = area at the node (constant)
C = constant
```

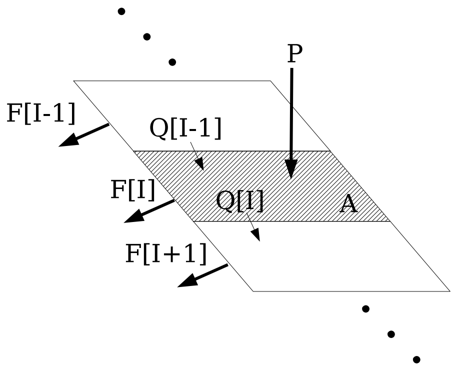

Fig. 1. Node Decomposition

```
class hydro {
  attributes
    Real[10][10] Q, S;   Real[10] P, F;   Real C, A;
  constraints
    S[1,2] = S[1,1] + (P[2] - F[2])*A - Q[1,1];
    forall Node in 1..10:
       forall Time in 1..10:
         Q[Node,Time] = C * S[Node,Time];
    forall N in 1..10:
       S[N,1] = (P[1] - F[1])*A - Q[N,1];
    forall T in 2..10:
       forall N in 2..10:
         S[N,T] = S[N,T-1] + (P[T] - F[T])*A + (Q[N-1,T-1]- Q[N,T-1]);
  constructors hydro(Answer) {
    P = [1.0, 1.0, 1.0, 0.8, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1];
    F = [0.05, 0.15, 0.25, 0.35, 0.45, 0.55, 0.60, 0.65, 0.70, 0.75];
    A = 1;   C = 0.5;
    forall J in 1..10: Answer[J] = Q[10,J];
  }
}
```

This simple model computes the flow at the last node (10) over a time interval of 10 seconds. The quantified constraints specify the constraints for determining the volume of water accumulating at any node N at any time T based upon the values at the previous node (N-1) and also the flow from node N-1 to N during time T-1. Note that the an explicit time-dimension is needed for the flow (Q) and volume (S) arrays. As a result, the storage requirements for modeling grow intractably for larger problem sizes. Also, the repetitive use of the time dimension for the various arrays also leads to cumbersome specification for larger problems.

It should be noted that certain variables P and F (for precipitation and infiltration) have been initialized with time-series data. An alternative approach is to declare a predicate as a function of time which requires as its input *time* and will proceed to generate the expected value at the specified time instance. This is a viable option, but is not immediately clear, nor is it very elegantly declared in the language.

Simple Wireless Sensor Network. To further illustrate the difficulties of modeling a dynamic system with the current Cob language, we consider modeling a very basic wireless sensor network of simple sensors in a two-dimensional environment. These sensors are capable of detecting a single target and communicating the data with other sensors. We do not specify which sensors should receive this data; there are also no frequency, bandwidth, or other constraints as might be needed in a more advanced model. The purpose of this model is to act as a computational driver for a visualization component, which will graphically display the sensors, the target, and sensor data sharing in a two-dimensional environment over a period of time. The visualization component should be able to receive information from the Cob program at every time instance, allowing it to update its on-screen representations.

```
class sensor{
  attributes
    real X,Y,TX,TY,DSQ;
    real DSense, DExposure;
    bool Detect;
  constraints
    Sensing > 0.0;    Exposure > 0.0;
    DSQ = (X - TX)^2 + (Y - TY)^2;
    Detect = true :- DSQ < DSense^2, DSQ > DExposure^2;
    Detect = false :- DSQ >= DSense^2;
    Detect = false :- DSQ =< DExposure^2;
  constructors sensor ([DS,DE,SX,SY],[TPosX,TPosY]){
    DSense = DS;    DExposure = DE;
    X = SX;    Y = SY;
    TX = TPosX;    TY = TPosY;
  }
}
class environment {
  attributes
    int NoS;
```

```
    sensor [] Sensor;
  constructors environment(InS,Target,OutS) {
    sizeof(Sensor,NoS);
    forall I in 1..NoS:
       Sensor[I] = new sensor(InS[I],Target);
    forall I in 1..NoS:
       OutS[I] = Sensor[I].Detect;
  }
}
```

The basic operating constraints of the sensor is that it can detect a target provided the target is within the sensing range (`DSense`) and the target is not within the exposure range (`DExposure`). The reasoning is that if the target is too close to the sensor it can "jam" the sensor communication. This logic is expressed by the constraints in the class `sensor`. A sensor is created with a specified 2d location `SX, SY` and also a specific sensing range `R` and an exposure range `E`. Note that the location of the target `TX, TY` is also passed as an argument to the constructor of sensor.

The class `environment` represents the two-dimensional environment at a single point in time. The variables `InS` and `OutS` in the program have the following meanings: `InS` is an input array of all sensors, and `OutS` is a corresponding boolean array indicating which sensors of the input array have detected the target.

The motion of the target is not specified within the Cob program, because a change in location of the target cannot be modeled in a purely declarative way, given the nature of the problem formulation. The change in target location is modeled external to the Cob program. Thus, each time the target moves, we need to instantiate a new environment object, which in turn will instantiate new sensor objects and give each of them the new target location. Thus the entire model must be reconstructed in order to query the sensors to determine if they detect the target. For a large number of sensors and targets, the above approach becomes intractable, and hence we demonstrate in the next section a better solution to this problem.

3 Dynamic Cob: An Informal Introduction

Syntax Additions. To address the shortcomings in the basic Cob paradigm, we introduce and explain two new features: the `series` variable, and **dynamic** class.

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

$$
\begin{aligned}
\textit{attributes} &::= \textit{decl} \ ; \ [\ \textit{decl} \ ; \ ]^{+} \\
\textit{decl} &::= \textit{type id\_list} \mid \textit{series\_decl} \\
\textit{series\_decl} &::= \texttt{series}\ \textit{attribute\_id} \ = \ \textit{series\_type} \\
\textit{series\_type} &::= \textit{term} \mid [\ \textit{terms}\ ] \\
\textit{type} &::= [\ \texttt{series}\ ]\ \textit{primitive\_type\_id} \mid \textit{class\_id} \mid \textit{type}[\,] \\
\textit{primitive\_type\_id} &::= \texttt{real} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{string} \\
\textit{id\_list} &::= \textit{attribute\_id}\ [\ ,\ \textit{attribute\_id}\ ]^{+}
\end{aligned}
$$

The keyword `dynamic` specifies that the class has constraints which specify dynamic behavior. This class may then make use of the special variable called *Time*, which is a real number greater than or equal to 0 (i.e. how much time has elapsed). This allows for constraints to be written as functions of time.

The keyword `series` specifies that the variable is used in constraints which require its value(s) at previous and/or future values in a series. The modeler may then reference these specific values with respect to the current state of the series in one of two ways:

1. The '(quote) operator specifies values in the past
2. The ‘(backquote) operator specifies values in the future.

Consider a series variable `Speed` defined as:

```
series real Speed
```

We can provide a base case such as:

```
Speed = 1.
```

Now if we wish to increment the speed by 0.5 for every time instance, we define a constraint such as:

```
Speed = Speed' + 0.5.
```

We can observe the values in `Speed` over a small time series.

Time	Speed'2	Speed'	Speed
0	X	X	1
1	X	1	1.5
2	1	1.5	2.0
3	1.5	2.0	2.5

The values of a series variable may also be initialized in one of two ways: The first is to set the variable equal to an explicitly specified array of values, e.g.,

```
series P = [1.0,0.9,0.8,0.7];
```

Defining the variable in this manner restricts the model to discrete steps. The second method of declaration is to define the values of the variable as a function of time, as shown below. The `Time` variable can be used in the declaration.

```
series N = sin(Time+0.5);
```

Square Root using Newton's Method. The series variable is not only useful for modeling a dynamic time-based system, but also for general other forms of series, as illustrated by the definition of the square root of a number by Newton's method of approximations.

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}$$

The class below calculates the square root of a positive number `N` within a given epsilon `EPS`. The final result is `Ans` when the difference between x_i and x_{i+1} is within the tolerance `EPS`.

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

    SQ‘= (SQ + (N/SQ)) / 2;
    Ans = SQ‘:- abs(SQ‘- SQ) <= EPS;

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

Hydrology Revisited. A more space efficient solution to the hydrology example presented earlier can be formulated using the series variable. In the code below, note that time is not explicitly treated, and thus the nested forall constraints of the earlier formulation become a single forall constraint.

```
dynamic class hydro {
  attributes
    series real P = [1.0, 1.0, 1.0, 0.8, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1];
    series real F = [0.05, 0.15, 0.25, 0.35, 0.45, 0.55, 0.60, 0.65, 0.70, 0.75];
    series Real[10] Q, S;
    Real C, A;
  constraints
    forall N in 1..10: Q[N] = C * S[N];
    S‘[1]= S[1] + (.85)*A - Q[1];
    forall N in 1..10: S[N] = .95*A - Q[N];
    forall N in 2..10:
       S[N] = S’[N] + (P - F)*A + (Q’[N-1] - Q’[N]);
  constructors hydro(Answer) {
    A = 1;
    C = 0.5;
    Answer = Q[10];
} }
```

Although the aesthetic improvements of the Cob program are clear, the runtime improvements do not become immediately apparent until the data field surpasses a given magnitude. Below is the runtime comparison of the original and the improved formulations of a 100-node hydrology model. Note that calculations past 40 seconds were not possible in the original forumulation due to memory overflow.

Time(sec)	Nodes	Original (sec)	Improved (sec) Runtime(sec)
10	100	8.5	26
20	100	54	55
40	100	330	200

Wireless Sensor Network Revisited. In the model below, we now have a moving target object and fixed sensors (although we could just as easily have them move). Note the introduction of the `target` class, with series variables X and Y which provide the time-varying behavior of the target. Although this model looks similar to the original, the use of dynamic classes and series variables allow the sensors to be created just once, and the constraints of the sensors are re-solved for each time instance, as the target location changes.

```
dynamic class target {
 attributes
   real X0,Y0;
   series real X = X0 + 0.5 * 9.8 + Time^2;
   series real Y = Y0 + 9.8 * Time;
 constructors target(Xinit, Yinit) {
   X0 = Xinit; Y0 = Yinit;
 }
}
dynamic class sensor {
 attributes
   real X,Y,DSense,DExposure;
   series real DSQ;
   series bool Detect;
   target T;
 constraints
   DSense > 0.0;   DExposure > 0.0;
   DSQ = (X - T.X)^2 + (Y - T.Y)^2;
   Detect = TRUE :- DSQ < DSense^2, DSQ > DExposure^2;
   Detect = FALSE :- DSQ >= DSense^2;
   Detect = FALSE :- DSQ =< DExposure^2;
 constructors sensor([DS,DE,Xinit,Yinit], Target){
   X = Xinit;   Y = Yinit;
   DSense = DS;   DExposure = DE;
   T = Target;
 }
```

```
}
dynamic class environment {
 attributes
   target Target;
   sensor [] Sensor;
 constructors environment(InS,OutS){
   sizeof(InS,NoS);
   Target = new target(0,0);
   forall I in 1..NoS:
    Sensor[I] = new sensor(InS[I], Target);
   forall I in 1..NoS:
    OutS[I] = Sensor[I].Detect;
 }
}
```

4 Conclusions, Status, and Further Work

The concept of constrained objects is broadly applicable in the modeling static as well as dynamic systems. We have illustrated this point through a variety of examples and explanations. We believe that the programs presented are both clear and concise, and are representative of a larger spectrum of possible applications. On a technical level, our new language features improve upon an already advanced Cob paradigm, thereby allowing simple, conditional, and quantified constraints to be interwoven with dynamic behavior.

We are also working on domain specific visualization models that take advantage of the new dynamic Cob features. A wireless sensor network visualization is currently under development. This system allows the modeler to create a specified number of static and/or dynamic sensor and targets in a three dimensional environment. There are various types of sensors and targets as well as communication schemes to choose from. Once this initial information is entered, the modeler may observe the model over an arbitrary time frame, playing, pausing, and rewinding at their discretion.

Prototype Implementation. In order to validate the language and its semantics, we have developed a new *Dynamic Cob* compiler that translates a Cob program into a Sicstus Prolog representation using the objects library to model each class and the CLP(R) library to solve all but linear constraints. Each class is essentially represented by a corresponding Sicstus object with the appropriate predicates. Variables that are defined as type `series` are implemented by detecting the amount of time specific information that must be available. This data is then stored as part of the specific class attributes. Series ariables that are initialized are implemented by registering them with a Cob internal *state* class. This class keeps track of every state variable and the corresponding class. When the internal *time* class is incremented, the state class alerts every object that they are entering a new time instance and to recalculate the current value of their variable. When a class is defined as type `dynamic`, it is automatically registered with the same Cob internal *state* class. When the internal time is changed, the state class ensures that the `Time` variable is passed into the constraint predicate of that class. To advance the model, a simple call to the time class `time::increment(Amount)` is all that is required. This call resolves all of the appropriate time constraints. The fundamental comparisons of the Cob paradigm to other logic programming languages like Bertrand [6], Siri [3], COB [7], LIFE [1], Oz [8], and Kaleidoscope[2] are discussed extensively in previous work [4, 5]. Each language fails to be both declarative and dynamic.

Domain Specific Visual Interface. We currently have a domain-specific visual interface tool for modeling a wireless sensor network. The visualization program, which is written in C and OpenGL, is completely decoupled from the corresponding Cob program. The visualization acts only as a three dimensional observation environment. This allows for various types of wireless sensor networks to be created in Dynamic Cob which conform to a common communication interface with the visualization component. The visualization program allows the user to navigate the wireless sensor network with a series a key commands which correspond to moving the camera. It also allows the user to pause the simulation and inspect the network at any time instance.

Acknowledgments: This research was supported by a grant from the National Science Foundation. The basic paradigm of Cob was developed by Pallavi Tambay as part of her recently completed Ph.D. dissertation. The sensor networks example presented in this paper is a simplied version of the formulation due to Shiva Ramanna and Anantharaman Ganesh. Thanks to Yang Zhang for comments and suggestions on the paper.

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. B. Horn. *Constrained Objects.* PhD thesis, CMU, November 1993.
4. B. Jayaraman and P. Tambay. Semantics and Applications of Constrained Objects. Technical Report 15, Department of Computer Science and Engineering, University at Buffalo, 2001.
5. B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In S. Krishnamurthi and C. Ramakrishnan, editors, *Symp. on Practical Aspects of Declarative Languages*, pages 28–46. Springer-Verlag, 2002.
6. W.J. Leler. *The Specification and Generation of Constraint Satisfaction Systems.* Addison-Wesley, 1987.
7. R. Peak. Automating product data-driven analysis using multifidelity multidirectional constrained objects. Invited Presentation, NASA STEP for Aerospace Workshop, Jet Propulsion Lab, Pasadena CA, Jan 2000.
8. G. Smolka. Constraint Programming in Oz (Abstract). In *Proc. Intl. Conference on Logic Programming*, pages 37–38, 1997.

Appendix A: Basic Cob Syntax

The following grammar elaborates on the syntax defined in section 3.

$$
\begin{array}{rl}
\mathit{attribute} & ::= \mathit{selector}[.\mathit{selector}]^{+} \mid \mathit{attribute}[\mathit{term}] \\
\mathit{selector} & ::= \mathit{attribute\_id} \mid \mathit{selector\_id}(\mathit{terms}) \\
\mathit{selector\_id} & ::= \texttt{first} \mid \texttt{next} \mid \texttt{last} \\
\mathit{terms} & ::= \mathit{term}\ [\ ,\ \mathit{term}\ ]^{+} \\
\mathit{literals} & ::= \mathit{literal}\ [\ ,\ \mathit{literal}\ ]^{+} \\
\mathit{literal} & ::= [\ \texttt{not}\ ]\ \mathit{atom} \\
\mathit{atom} & ::= \mathit{pred\_id}(\mathit{terms}) \mid \mathit{constraint\_atom} \\
\mathit{pred\_clauses} & ::= \mathit{pred\_clause}\ .\ [\ \mathit{pred\_clause}\ .\ ]^{+} \\
\mathit{pred\_clause} & ::= \mathit{clause\_head}\ :-\ \mathit{clause\_body} \\
\mathit{pred\_clause} & ::= \mathit{clause\_head} \\
\mathit{clause\_head} & ::= \mathit{pred\_id}(\mathit{terms}') \\
\mathit{clause\_body} & ::= \mathit{goal}\ [\ ,\ \mathit{goal}\ ]^{+} \\
\mathit{goal} & ::= [\ \texttt{not}\ ]\ \mathit{pred\_id}(\mathit{terms}') \\
\mathit{terms}' & ::= \mathit{term}'\ [\ ,\ \mathit{term}'\ ]^{+} \\
\mathit{term}' & ::= \mathit{constant} \mid \mathit{var}' \mid \mathit{attribute} \mid \mathit{function\_id}(\mathit{terms}') \\
\mathit{pref\_clauses} & ::= \mathit{pref\_clause}\ .\ [\ \mathit{pref\_clause}\ .\ ]^{+} \\
\mathit{pref\_clause} & ::= \mathit{pred\_id}(\mathit{terms}')\ \leq\ \mathit{pred\_id}(\mathit{terms}')\ :-\ \mathit{clause\_body} \\
 & \mid \texttt{min}\ \mathit{term} \mid \texttt{max}\ \mathit{term}
\end{array}
$$

Note that the only variables that may appear in a *term* (defined in section 3) are attributes or those that are introduced in a quantification. These variables are generated by the non-terminal *var*. In the above grammar, the variables that appear in a *pred_clause* are the usual logic variables of Prolog. These are referred to as *var'* in the above syntax.