Implementation Techniques for Constrained Objects

Pallavi Tambay, Bharat Jayaraman
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260-2000
{tambay,bharat}@cse.buffalo.edu

This paper presents the implementation of constrained objects. A constrained object is an object whose attributes are subject to constraints. When a constrained object aggregates other objects, it may impose further constraints on the attributes of these objects. As shown in a recent paper, this programming paradigm is very useful for modeling various systems, especially in the engineering domain [10]. Here, a constraint may be simple, quantified, conditional, or non-linear. A constrained object (Cob) program consists of Java-like class definitions except that methods are replaced by constraints. These constraints may also refer to user-defined CLP-like predicates. Hence our implementation is based on a translation from Cob classes to CLP predicates. Execution of the translated CLP predicates will not yield adequate performance for large-scale systems. Hence we develop a partial execution technique for optimized code generation. This approach also enables developing a visual debugger for Cob programs and also interfacing with a system such as Maple for solving non-linear constraints. The paper presents the syntax and examples of Cob programs, and the scheme for translation and partial execution. The techniques described in this paper have been implemented.

1. INTRODUCTION

This paper presents implementation techniques for a novel programming paradigm called constrained objects. The importance of this paradigm stems from its application in modeling complex engineering systems. While objects can represent the structure of such systems, the behavior of a component in a complex system and its relation with other components is often governed by physical laws, or invariants. The traditional programming concepts of classes, hierarchies and aggregation and polymorphism are appropriate for modeling the structure and categories of components. However, imperative methods are not suitable for specifying the behavioral invariants; declarative constraints are more appropriate for this purpose. In previous work, we have designed a constrained object language, **Cob**, and shown its use in modeling several structures, such as circuits, trusses, mixers, gears, etc. [10].

A Cob program consists of a set of Java-like class definitions except that declarative constraints take the place of imperative methods. A Cob constraint may be simple, quantified, conditional, or non-linear. We explain these constraints in more detail in section 2, and present a few examples of Cob definitions. The execution of a Cob program is initiated by constructing a complex object and setting certain of the attributes, and then solving for the remaining attributes of the constituent objects. We have developed domain-specific graphical interfaces for a few domains (see figure 1) so that the contruction and initialization can be done graphically, rather than textually. Thus constrained objects are a paradigm wherein objects, constraints, and visual representations are brought together in a principled manner for addressing the needs of engineering design.

The implementation of Cob is based upon a translation to CLP, followed by partial execution of the translated code for more efficient execution. The reason we translate Cob to CLP is that the constraints may, in general, make use of CLP predicates. In the engineering design context, we can obtain a more efficient and general implementation by two important techniques: (i) For large-scale structures, we can perform partial execution on the translated CLP program in order to unwrap constraints from their object containers and also unravel loops arising from quantified constraints. Note that the partial evaluator must also work with conditional constraints in our context. (ii) For applications where non-linear constraints might arise, we interface with a more powerful constraint solver, such as Maple. In this way, we overcome some of the limitations arising from CLP. We illustrate this point through the truss example in section 4.

The remainder of this paper is organized as follows: Section 2 presents the syntax and examples of Cob programs; section 3 outlines the translation of Cob classes to CLP predicates, along with an example of sample code that is generated; section 4 presents our partial evaluator in high-level terms, illustrates the improved code generation, and also describes a visual debugger for Cob programs under development. The techniques described in this paper have been implemented.

2. CONSTRAINED OBJECTS

2.1 Cob Syntax

A Cob program is a sequence of class definitions, and each constrained object is an instance of some class.

Class. A class definition consists of attributes, constraints, predicates and constructors.

$$\begin{array}{rcl} \mathit{class\_definition} & ::= & [\ \texttt{abstract}\ ]\ \texttt{class}\ \mathit{class\_id} \\ & & [\ \texttt{extends}\ \mathit{class\_id}\ ]\ \{\ \mathit{body}\ \} \\ \mathit{body} & ::= & [\ \texttt{attributes}\ \mathit{attributes}\ ] \\ & & [\ \texttt{constraints}\ \mathit{constraints}\ ] \\ & & [\ \texttt{predicates}\ \mathit{pred\_clauses}\ ] \\ & & [\ \texttt{constructors}\ \mathit{constructor\_clause}\ ] \end{array}$$

Each of these constituents is optional, and an empty class definition is permitted as a degenerate case. Single inheritance of classes is permitted. There can be more that one constructor for a class. An abstract class is a class without a constructor, and hence cannot be instantiated.

Attribute. 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 types. The size of an array may be left unspecified.

$$\begin{array}{rcl} \mathit{attributes} & ::= & \mathit{decl}\ ;\ [\ \mathit{decl}\ ;\ ]^+ \\ \mathit{decl} & ::= & \mathit{type}\ \mathit{id\_list} \\ \mathit{type} & ::= & \mathit{primitive\_type\_id}\ |\ \mathit{class\_id} \\ & & |\ \mathit{type}[\,] \\ \mathit{primitive\_type\_id} & ::= & \texttt{real}\ |\ \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{char}\ |\ \texttt{string} \\ \mathit{id\_list} & ::= & \mathit{attribute\_id}\ [\ ,\ \mathit{attribute\_id}\ ]^+ \end{array}$$

Constraint. The constraints state inter-class or intra-class relations between the attributes.

$$\begin{array}{rcl} \mathit{constraints} & ::= & \mathit{constraint}\ ;\ [\ \mathit{constraint}\ ;]^+ \\ \mathit{constraint} & ::= & \mathit{simple\_constraint} \\ & & |\ \mathit{quantified\_constraint} \\ & & |\ \mathit{creational\_constraint} \\ \mathit{creational\_constraint} & ::= & \mathit{complex\_id} = \texttt{new}\ \mathit{class\_id}(\mathit{terms}) \\ \mathit{quantified\_constraint} & ::= & \texttt{forall}\ \mathit{var}\ \texttt{in}\ \mathit{enum}{:}(\mathit{constraints}) \\ & & |\ \texttt{exists}\ \mathit{var}\ \texttt{in}\ \mathit{enum}{:}(\mathit{constraints}) \\ \mathit{simple\_constraint} & ::= & \mathit{conditional\_constraint} \\ & & |\ \mathit{constraint\_atom} \\ \mathit{conditional\_constraint} & ::= & \mathit{constraint\_atom} :-\ \mathit{literals} \\ \mathit{constraint\_atom} & ::= & \mathit{term}\ \ \mathit{relop}\ \ \mathit{term} \\ & & |\ \mathit{constraint\_predicate\_id}(\mathit{terms}) \\ & & |\ \mathit{creational\_constraint} \\ \mathit{relop} & ::= & = \ |\ != \ |\ > \ |\ < \ |\ >= \ |\ <= \end{array}$$

A constraint can be simple, quantified or creational. 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 the data domain (e.g. integers, reals, etc.) as well as constants and attributes. A constraint atom may also be a predefined constraint predicate whose properties are known by the underlying system.

Conditional Constraint. 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. We employ Prolog-like syntax for defining conditional constraints. There may be more than one (constraint) atom in the antecedant but the consequent has exactly one (not negated) atom. Examples:

```
F = Sy * W * H :- F > 0;
Day =< 29 :- Month = 2, leap(Year);
```

Quantified Constraint. A quantified constraint is a shorthand for stating a relation where the participants of the relation range over an enumeration (*enum*), i.e., indices of an array or the elements of an explicitly specified set or a set of integers (e.g. `1..10`). e.g.

```
forall C in Cmp:  C.I = I;
exists N in Nodes:  N.Value = 0;
```

Creational Constraint. A creational constraint creates an object of a user defined class and binds it to an attribute. e.g. `R1 = new component(V1, I1, 10);`
A creational constraint may appear as the head of a conditional constraint, and thus object creation can be predicated upon.

Term. Terms can appear in constraints or as arguments to functions, predicates or constructors.

$$\begin{array}{rcl} \mathit{term} & ::= & \mathit{constant}\ |\ \mathit{var}\ |\ \mathit{complex\_id}\ |\ (\mathit{term}) \\ & & |\ \mathit{arithmetic\_expr}\ |\ \mathit{func\_id}(\mathit{terms}) \\ & & |\ \mathit{aggreg\_op}\ \mathit{var}\ \texttt{in}\ \mathit{enum} : \mathit{term} \\ \mathit{aggreg\_op} & ::= & \texttt{sum}\ |\ \texttt{prod}\ |\ \texttt{min}\ |\ \texttt{max} \end{array}$$

A term may be an arithmetic/boolean expression involving attributes and/or constants from the data domain (e.g. integers, reals, etc.). We provide a shorthand for iterative terms, where the iteration is over indices of array or elements of a specified set. This notation is identical to its mathematical equivalent. e.g. the term
`sum X in PC: (X.V)` stands for $\sum_{i=1}^{n}$ `PC[`i`].V` where n = length of PC

Complex_id. A complex identifier is an identifier, an element of an array, or the result of accessing the attributes of a class using the selection operator (`.`) on an attribute one or more times.

$$\begin{array}{rcl} \mathit{complex\_id} & ::= & \mathit{attribute\_id}[.\mathit{attribute\_id}]^+ \\ & & |\ \mathit{complex\_id}[\mathit{term}] \end{array}$$

Literal. A literal is an atom or the negation of an atom.

$$\begin{array}{rcl} \mathit{literals} & ::= & \mathit{literal}\ [\ ,\ \mathit{literal}\ ]^+ \\ \mathit{literal} & ::= & [\ \texttt{not}\ ]\ \mathit{atom} \\ \mathit{atom} & ::= & \mathit{predicate\_id}(\mathit{terms})\ |\ \mathit{constraint\_atom} \end{array}$$

If `p` is an n-ary predicate symbol and $t_1, ..., t_n$ are terms then `p`$(t_1, ..., t_n)$ is an atom [13]. A constraint_atom is an atom with a predefined predicate symbol whose properties are known to the underlying constraint satisfaction system.

Predicate. A predicate is an n-ary relation. 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.

Constructor. A class that is not abstract can have one or more constructors.

$$
\begin{array}{rcl}
\textit{constructor\_clauses} & ::= & \textit{constructor\_clause}^{+} \\
\textit{constructor\_clause} & ::= & \textit{constructor\_id}(\textit{formal\_pars}) \\
 & & \{ \ \textit{constraints} \ \}
\end{array}
$$

The constructor_id must be the same as the name of the class. The body of a constructor contains a sequence of `;` separated constraints. These constraints hold throughout the life of an instance of the class and should not be interpreted as one-time/initialization-only constraints.

2.2 Examples

We model the well-known example of a series/parallel electrical circuit as a constrained object. Several examples of constrained object models of structures from different domains of engineering are given in [10]. We intentionally show the well known electrical circuit example here because it is simple yet illustrative of the language and constraint features of Cob. We model the components and connections of such circuits as objects and their properties and relations as constraints on and amongst these objects.

The `component` class below models any electrical entity that is characterized by its resistance and obey's Ohm's law.

```
class component {
  attributes
      real V, I, R;
  constraints
      V = I * R;
  constructors component(V1, I1, R1) {
      V = V1; I = I1; R = R1;
  }
}
```

The `series` class is recursively defined. It subclasses from the `component` class and aggregates an array of components. The `series` class inherits the voltage, current and resistance attributes and the Ohm's law constraint from the `component` class and further defines some constraints: its effective resistance and voltage is the sum of the resistances and voltages of the components it aggregates respectively; its effective current is equal to the currents through each of its constituent components. These relations are modeled as constraints in the `series` class. The `parallel` class is similarly defined.

```
class series extends component {
  attributes
      component [] Cmp;
  constraints
      forall C in Cmp:  (C.I = I);
      sum C in Cmp:C.V = V;
      sum C in Cmp:C.R = R;
  constructors series(A) {
      Cmp = A;
  }
}
class parallel extends component {
  attributes
      component [] PC;
  constraints
      forall X in PC: X.V = V;
      sum X in PC: X.I = I;
      sum X in PC: 1/X.R = 1/R;
  constructors parallel(B) {
      PC = B;
  }
}
```

The `battery` class represents a source of constant voltage drop. The `connect` class models the connection of a battery across a component and equates their effective voltages.

```
class battery {
  attributes
      real V;
  constructors battery(V1) {
      V = V1;
  }
}
class connect {
  attributes
      battery B; component CC;
  constraints
      B.V = CC.V ;
  constructors connect(B1, C1) {
      B = B1; CC = C1 ;
  }
}
```

We can create a model of an electrical circuit using the classes defined above. This is done by creating instances of constrained objects and assembling them together. Creational constraints are used to create instances of cosntrained objects. Consider the following electrical circuit: a parallel combination of two 20 ohm resistors connected in series with a parallel combination of 10 ohm and 20 ohm resistors. A 30 Volt battery is connected across this assembly of resistors. The class `samplecircuit` models this circuit.

```
class samplecircuit {
  attributes
      battery B;
      connect C;
      component R1, R2, R3, R4, P1, P2, S;
      real V1, V2, V3, V4, I1, I2, I3, I4;
      component[] R12, R34, P12;
  constructors samplecircuit() {
      R1 = new component(V1, I1, 10);
      R2 = new component(V2, I2, 20);
      R3 = new component(V3, I3, 20);
      R4 = new component(V4, I4, 20);
      P1 = new parallel([R1,R2]);
      P2 = new parallel([R3,R4]);
      S = new series([P1,P2]);
      B = new battery(30);
      C = new connect(B, S);
  }
}
```

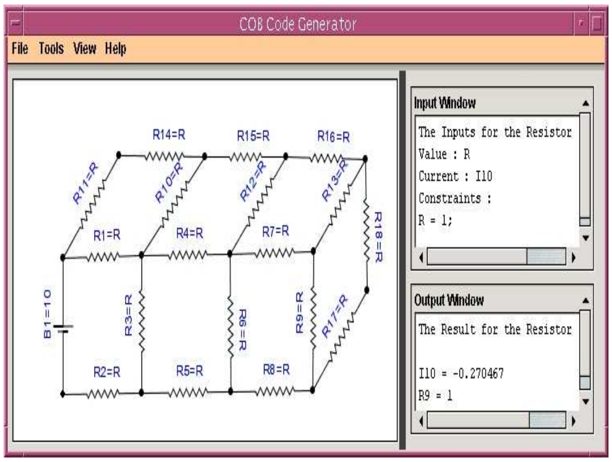

Figure 1: Domain Specific Visual Interface

To create the an instance of the circuit, we use the constraint

```
S1 = new samplecircuit()
```

A call to the constructor of samplecircuit results in the creation of the various instances of `component`, `series`, `parallel`, etc. As a result, the internal and interface constraints of all object instances are solved and the values of voltages and currents across every component are obtained.

2.2.1 Simple Truss

To illustrate the use of constrained objects in engineering design, we define the Cob classes needed to model the simple truss structure in figure 1. A truss consists of bars placed together at joints. The constraints in the `beam` class express the standard relations between its modulus of elasticity (`E`), yield strength (`Sy`), dimensions (`L`, `W`, `H`), moment of inertia (`I`), stress (`Sigma`) and the force (`F`) acting on the beam. Depending on the direction of this force, it may act as the bending, buckling, or tension force and will according be related to the other attributes by different constraints. This relation is expressed as conditional constraints in the `beam` class. A bar is a beam placed at an angle (`A`) and a load is a force applied at an angle. The `joint` class aggregates an array of bars and and an array of loads (that are incident at the joint) and its constraints state that the sum of the forces in the horizontal and vertical directions respectively must be 0. The Cob classes defined here can model a truss with loads and may be used to determine the dimensions of the beams such that they can support the loads. These conditions are taken from the text by Mayne and Margolis [16].

```
class beam {
 attributes
  real E, Sy, L, W, H, F, I, Sigma;
 constraints
  Pi = 3.141 ;
  L > 0; W > 0; H > 0; I > 0;
  I = W * H * H * H / 12;
```

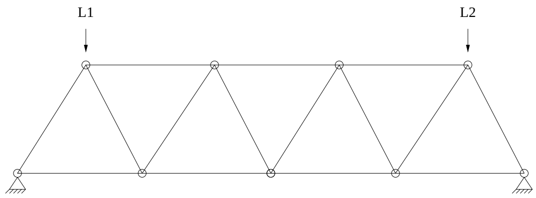

Figure 2: A Simple Truss

```
  Sigma = (H * L * F_bn )/ (8 * I);
  (I = - F * L *L /Pi * Pi*E) :- F < 0;
  (F = Sy * W * H) :- F > 0;
 constructors beam(E1,Sy1,L1,W1,H1,Sigma1,I1,F1) {
    E=E1; Sy=Sy1; L=L1; H=H1; W=W1;
    Sigma=Sigma1; I=I1; F=F1;
 }
}

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

class load {
 attributes
  real F; real A;
 constraints
  0 <= A; A <= 360;
 constructors load(F1, A1) {
  F = F1; A = A1;
 }
}

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

2.3 Comparison with Related Work

The earliest work in the area of constrained objects is the constraint-based simulation laboratory ThingLab [2]. It was intended for interactive graphical simulations of experiments in physics and geometry. Within a ThingLab class, the programmer can specify a constraint by stating a rule and a set of methods for satisfying the constraint. The rule is used to test how well the constraint is satisfied. The system uses this information, to decide on a constraint satisfaction plan when state change occurs. The constraint solver uses local propagation for satisfying constraints. In contrast to this, the Cob programmer does not specify methods for solving constraints. It is done automatically in Cob by the underlying engine.

The constrained object language Siri [7], integrates constraints into an object-oriented framework. The central idea of a constraint pattern is used to model object encapsulation, inheritance, constraints as well as message passing. Constraints define relationship between the attributes which may be equality, arithmetic or user-defined. Constraint patterns are an extension of the pattern in BETA [11]. Siri extends another language Bertrand [12] intended for graphics applications. It uses Bertrand's augmented term-rewriting for equation-solving. Due to this, Siri can solve only a limited type of simple constraints as opposed to Cob which can handle linear, non-linear, conditional and quantified constraints.

The Kaleidoscope '93 [5, 15] language integrates constraints and object-oriented programming for developing interactive graphical user interfaces. It simulates imperative constructs like updating, assignment and object identity through constraints. It uses multi-method lookup to find an appropriate constraint constructor to solve a constraint. The constraint constructor simplifies user-defined constraints till they reduce to constraints on primitive types at which point a primitive solver based on local propagation and Gaussian elimination is used for computation. However, it is not possible to express conditional and quantified constraints in Kaleidoscope. Partial evaluation of Cob programs allows the use of powerful external solvers which enable the computation of more complex (non-linear) constraints than possible in Kaleidoscope.

Modelica [4] is an object-oriented language for physical system modeling and simulation. Within a class, statements under the algorithm clause perform state change and those within the equations clause specify the relations between attributes of a class. Constraints can be a conjunction or disjunction of logical term involving arithmetic expressions. Language constructs are provided for expressing conditional and quantified constraints can be expressed. Although the use and some constraint constructs of this language are similar to Cob, an important advantage of Cob is the ability to predicates within class definitions. Our paradigm supports underspecified models, logic variables, answer constraints, user defined constraint predicates, symbolic constraint solving and visual interactive execution via the underlying object structure.

3. COB TO CLP TRANSLATION

We describe the translation of a Cob program to a CLP program. A Cob program C is 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 class constructor is equivalent to a top-level goal or query. We describe this translation for the key components of a Cob program.

Class. We use V_i, C_i, P_i 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. For classes with more than one constructor, one predicate is defined per constructor. 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 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. 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 renamed during translation.

`class` c `{`	$\Longrightarrow_T$	$P_1 \ \cup \ \{defn(p_c)\}$
`attributes` V_1		`where` $defn(p_c)$ `is:`
`constraints` C_1		$\mathrm{p}_c(V_2, V_1) : -C_1, C_2.$
`predicates` P_1		
`constructor` $c(V_2)$`{`		
C_2		
`}`		
`}`		
`class` c' `extends` c `{`	$\Longrightarrow_T$	$P_2 \ \cup \ \{defn(p_{c'})\}$
`attributes` V_1'		`where` $defn(p_{c'})$ `is:`
`constraints` C_1'		$\mathrm{p}_{c'}(V_2', V_c \circ V_1')$ `:-`
`predicates` P_2		$\mathrm{p}_c(\_, V_c), C_1', C_2'.$
`constructor` $c'(V_2')$`{`		
C_2'		
`}`		
`}`		

In an inheritance hierarchy, a subclass may redeclare a variable of its superclass. Our approach to handling variable redeclaration involves a systematic renaming of variables in subclasses. Suppose a class C_1 is a superclass of C_2 and the class C_2 is a superclass of C_3 and so on. Then the variables in the n^{th} subclass C_n in this hierarchy are renamed as follows. If C_n declares a variable X which is also declared in any of the classes $C_1, ..., C_{n-1}$, then X and all of its occurrences in C_n are renamed to a new variable name. If C_n uses a variable Y without declaring it, then we inspect the nearest class C_i (with the highest value for i less than n) that declares Y. If Y in C_i was renamed to $NewY$, then all occurrences of Y in C_n are replaced with $NewY$. If Y in C_i was not renamed, then Y in C_n is also not renamed.

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.

Constraint Atom. A constraint atom is translated by translating its constituent terms. Translation of terms is required since they might be performing array indexing, or using the selection operation. A list of constraints is translated by translating every element of the list.

$\mathcal{T}$ *relop* $\mathcal{U}$	$\Longrightarrow_T$	$\mathcal{T}_T$ *relop* $\mathcal{U}_T$
where		$\mathcal{T}_T$ is obtained by translating term $\mathcal{T}$
		$\mathcal{U}_T$ is obtained by translating term $\mathcal{U}$
and		*relop* is a relational operator.

Quantified Constraint. 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: $\mathcal{C}$	$\Longrightarrow_T$	$\texttt{forall}_\texttt{i}(E, \mathcal{V})$
where	$\texttt{forall}_\texttt{i}([\ ], \mathcal{V})$.	
	$\texttt{forall}_\texttt{i}([X\texttt{|Tail}],\mathcal{V})$:- $\mathcal{C}_T$, $\texttt{forall}_\texttt{i}(\texttt{Tail},\mathcal{V})$.	
	$\mathcal{V}$ is a list of the variables in $\mathcal{C}$	
and	$\mathcal{C}_T$ is obtained by translating constraint $\mathcal{C}$	

Quantified variables like X are renamed before carrying out the above translation to ensure the correct translation of multiple quantified constraints using the same quantified variable name X. Similarly, if the enumeration is an interval of integers, it is translated as shown below. Note that the predicate `makelist` is a predefined predicate whereas the predicate $\texttt{forall}_\texttt{i}$ will have to defined during translation.

`forall` X in $I..J$: $\mathcal{C}$	$\Longrightarrow_T$	$\texttt{makelist}(I, J, ItoJ)$, $\texttt{forall}_\texttt{i}(ItoJ, \mathcal{V})$
where	$\texttt{makelist}(N, M, NtoM)$: $NtoM$ is the list of integers from N to M.	

Existentially quantified constraints are translated along the same lines, with an appropriate definition for the $\texttt{exists}_i$ predicate.

Conditional Constraint. Conditional constraints are translated along the following lines (see Truss example in section 3.2.1).

A :- B	$\Longrightarrow_T$	$\texttt{cc}(A_T, B_T)$
where	A_T is obtained by translating constraint A	
	B_T is obtained by translating constraint B	
and	`cc` is the predefined predicate with semantics:	
	$\texttt{cc}(X,Y)$:- X. (i)	
	$\texttt{cc}(X,Y)$:- $\neg Y$. (ii)	

It should 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 evaluating `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. The entailment of a primitive or user defined constraints cannot be evaluated in CLP. 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).

Term. Terms appear in constraints and as arguments to functions (sin, cos, etc) and constructors. Translation of constraints involves the translation of terms. We show some aspects of this translation. The i^{th} occurrence of a summation term is replaced by a new variable Sum_i whose value is computed by an invocation of the predicate $\texttt{sum}_i$ with suitable parameters that capture the variables in the term in the body of the `sum` term.

`sum` X in E: $\mathcal{T}$	$\Longrightarrow_T$	Sum_i
where	$\texttt{sum}_\texttt{i}(E, \mathcal{V}, Sum_i)$	
where	$\texttt{sum}_\texttt{i}([\ ], \mathcal{V}, 0)$.	
	$\texttt{sum}_\texttt{i}([X\texttt{|Tail}],\mathcal{V}, \mathcal{T}_T + SumRest)$:- $\texttt{sum}_\texttt{i}(\texttt{Tail},\mathcal{V}, SumRest)$.	
	$\mathcal{V}$ is a list of the variables in term $\mathcal{T}$	
and	$\mathcal{T}_T$ is obtained by translating term $\mathcal{T}$	

Product terms are translated similarly with a `*` instead of the `+`. Terms computing the minimum of a collection of terms (`min`) are translated similarly by iteratively computing the minimum of two terms instead of the sum as shown above.

Array. Array are represented as lists. Multidimensional arrays are represented as list of lists of lists and so on depending on the dimension. Array declarations are translated to calls on the predefined predicate `makearray`. An array of undefined size is represented as the list `[_]`.

$typeName$ `[`I`][`J`]` X	$\Longrightarrow_T$	`makearray([`I`,`J`],` X`)`
where	`makearray(`$ListDim$`,` Y`)`: Y a list of size N where N is the length of the list $ListDim$. If the n^{th} element of $ListofDim$ is M, then the n^{th} element of Y is a list of size M.	

As array declarations of unspecified size are bound to [_], and elements can be inserted into such an array one at a time, the tail of the array may be a nonvar ([_]). Hence the actual definition of recursive predicates like $\texttt{forall}_\texttt{i}$, $\texttt{sum}_\texttt{i}$, etc. contain a conditional statement that checks for the presence of a non-var tail.

Indexing. Indexed terms are translated to a new variable whose value is obtained by a call to the predefined predicate `index`. Translation of multilevel indexing involves multiple calls to `index`, the result of one call is indexed by the next. A list of terms is translated by translating each element of the list.

X `[`$\mathcal{I}$`]`	$\Longrightarrow_T$	X_I
where	$\texttt{index}(X, \mathcal{I}_T, X\_I)$	
and	$\mathcal{I}_T$ is obtained by translating $\mathcal{I}$	
X `[`$\mathcal{I}$`][`$\mathcal{J}$`]`	$\Longrightarrow_T$	X_I_J
where	$\texttt{index}(X, \mathcal{I}_T, X\_I)$	
and	$\texttt{index}(X\_I, \mathcal{J}_T, X\_I\_J)$	
and	$\mathcal{J}_T$ is obtained by translating $\mathcal{J}$	

Array indexing involves list traversal and hence can lead to a slowdown. Our scheme of partial evaluation processes away calls to the `index` predicate.

Selection. Selection terms of the type $A.B$ are translated by retrieving the list of attributes of the class t_A (type of A) and extracting attribute B from it as shown below.

$A.B$	$\Longrightarrow_T$	A_B
where	A = $A\_\mathrm{V}_{t_A}$	
and	t_A is the type of A	
	V_{t_A} is the list of attributes of the class t_A.	
	$A\_\mathrm{V}_A \equiv$ prefix every element of V_A with $A\_$	

Cob permits polymorphism in that a variable declared to be of type C can at runtime be equated with an instance of a subclass of C.

Constructor. A constructor invocation in Cob is translated to a CLP goal as shown below. A call to the constructor of class c is translated to an invocation of the predicate p_c which is obtained by translation of the class c. The list of constructor arguments form the first argument of the call to p_c and the variable being equated to the newly formed object of class c becomes the second argument. Note that a Cob class is represented by a predicate definition, and a Cob object is represented by a list of variables of the class. For this translation of constructor calls to be meaningful, it is required that in classes with more than one constructor, each of the constructors should be of a different arity.

$$X = \mathtt{new}\ c(\bar{t}) \quad \Longrightarrow_T \quad \mathrm{p}_c(\bar{t}_T,\ X_T)$$

where p_c is the translation of class c
X_T is the translation of X
and $\bar{t}_T$ is the translation of $\bar{t}$

An important point to note is that any or all of the arguments to the constructors may be non-ground (unbound) variables.

3.1 Example

We have a prototype compiler that implements the above translation. 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;
        }
}
```

The Cob compiler translates the above code to

```
pcomponent([V1, I1, R1], [V, I, R]) :-
  V = I * R,
  V = V1,
  I = I1,
  R = R1.
pparallel([B],[V,I,R,PC]):-
  PC = [_],
  PC = B,
  pcomponent(_,[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].
```

Compiler built_ins: Some of the above transformations use predicates `makelist/3`, `makearray/2` and `index/3`. These are predefined predicates built into the compiler. We refer to such predicates as the compiler built_ins. Note that the Cob programmer does not invoke these predicates. They appear in the translated CLP code only. The scheme for partial evaluation described in section 4 will process away these predicates.

Cob built_ins: Side effects like reading/writing to stdout or a file can be achieved in a Cob program through the use of the `dump/1` and or the CLP built-in predicates `print/1, tell/1, told/1, read/1`. These are generated by the non-terminal *constraint_predicate_id* in the grammar in section 3.1. `dump/1` is a predefined Cob predicate written using the CLP built-in predicate `print/1`. The Cob programmer can directly use these predicates in a Cob program. We refer to such predicates as the Cob built_ins.

In our earlier version of the compiler mentioned in [10], the translated code was run on CLP(R). The output of the current version of the Cob compiler is run on the clpr module provided by SICStus Prolog [19]. In Sicstus Prolog, the clpr module is implemented as a quote-eval language. Constraints and symbols must be explicitly enclosed in {} in order to be interpreted. The translation described previously in this section generates eval-quote programs where all symbols are interpreted unless quoted. To transform the eval-quote programs of the above translation to quote-eval programs for input to Sicstus CLP, we use the `expand/0` predicate provided for this purpose by Sicstus Prolog. As a result of using `expand/0`, all constraints get explicitly enclosed in {}. A slightly modified translation can achieve a similar output by explicitly enclosing constraints within {} during the translation itself.

3.2 Properties of Translated Program

Basically, a Cob program with n class definitions is translated into a CLP program with one predicate for each of these classes. A class with k constructors results in a predicate with k clauses.

Termination. A class cannot extend itself or any of its subclasses. Hence if a class c is translated to predicate p_c, the only case when p_c will be recursive is if c creates an instance of itself within its class definition. For the class of engineering problems that we model, a recursive class definition would mean that an instance of a structure creates another instance of itself. Since an engineering assembly is finite, there can be only a finite depth of such predicate invocations. We also assume that the user-defined predicates used in constraints terminate on all invocations.

CLP subset. The predicates in the translated CLP program either correspond to Cob classes or Cob built-ins or compiler built-in predicates or those defined as a result of translation (forall, sum, etc). Thus the types of predicates and constructs appearing in the CLP program are a strict subset of the CLP language. A query to the CLP program is always obtained by translating a call to a Cob class constructor. Every predicate invocation in the translated program arises from translating a Cob constructor invocation (unless it is a call on user-defined CLP predicate). The constraints of the original Cob program are straightforward to identify since they are explicitly enclosed in $\{\}$.

Non-logical constructs. Translated CLP programs do not have cuts except in definition of the compiler built-in predicate `index`. Negated goals which can appear only in conditional constraints are evaluated only when ground.

4. PARTIAL EVALUATION

The execution of a Cob program is initiated by a call to the constructor of a Cob class. e.g. X `= new c(Args)`.
The evaluation of this call entails the creation of instances of constrained object classes and solving of their constraints. In the context of engineering modeling, such an evaluation creates an assembly of Cob class instances and represents a configuration of a complex engineering structures. The arguments (`Args`) to the constructor determine the type, arrangement and attributes of the components of the configuration.

In the Cob language the arguments to the constructor of a class need not be ground. Hence a call to a constructor may represent either a specific configuration or a collection of configurations. For example, recall the classes defined in section 2.2 relating to electrical circuits. The call `P = new parallel(X)` represents a parallel combination of any number of electrical components depending on the size of X. Whereas the call `P' = new parallel([R1, R2])` represents parallel combinations of any two components. The exact values of all the attributes (voltage, current, etc.) of `P'`, `R1` and `R2` will be known only after more information about these components is known. While some attributes of both are unknown, the difference between `P` and `P'` is that the assembly of subcomponents of `P'` is fixed whereas that of `P` is not. For the remainder of this section we will be referring to family of structures of the kind that `P` represents.

One case when the query may represent different assemblies is when creational constraints appear as part of conditional or quantified constraints. i.e. when constraint solving can lead to object instantiation.

The call X `= new c(Args)` is translated to the CLP query p_c`(Args,` X`)`. As explained earlier, this query represents a family of structures with a fixed assembly of subcomponents. Given the properties of the translation in section 3.2, it is possible to partially evaluate such queries. The output of partial evaluation is a collection of constraints which can be solved for different instances of the assembly. For example, consider the definition of `sampleciruit` given in section 2.2. A call to `S1 = new samplecircuit()` creates an instance of the sample circuit. The partial evaluation of the corresponding query `samplecircuit([ ],S1)` gives the following constraints:

```
V1 = I1 * 10 = I2 * 20 = V2
V3 = I3 * 20 = I4 * 20 = V4
V1 = S.I * P1.R
S.I = I1 + I2
1/P1.R = 1/10 + 1/20
V3 = S.I * P2.R
S.I = I3 + I4
1/P2.R = 1/20 + 1/20
30 = S.I * S.R
30 = V1 + V3
S.R = P1.R + P2.R
```

The full name of every variable above is actually prefixed by "S1." since each variable is a local variable of S1. For simplicity, we do not show the prefix above. These constraints can now be solved for different values of the variables. In this case, it just so happens that the equations are sufficient to calculate the values of all the variables. In general, this may not be the case as illustrated by the examples in section 4.2. In the next section, we describe our scheme for partial evaluation.

4.1 Strategy

The partial evaluator takes as input a CLP program obtained by translating a Cob program and a query obtained by translating a constructor call, and returns a list of constraints. A solution to this list of constraints is a solution to the query. Such a partial evaluation is possible because the CLP program has the properties mentioned in section 3.2 and it is also known that: (i) evaluation of Cob built-in predicates does not generate constraints; (ii) quantified constraints are generated from calls to $forall_i$, sum_i predicates, hence these predicates must be unraveled to obtain the constraints; (iii) the syntax and type of constraints is known from the syntax of Cob programs. Formally, the partial evaluator is described as:

Input: A CLP($\mathcal{D}$) program P satisfying the properties in section 3.2 and a goal G.
Output: A set of constraints C such that $\mathcal{P}, \mathcal{D} \models C \rightarrow G$.

In the following discussion, it is understood that a Cob program is given as input to the partial evaluator. We do not show the input program explicitly as an argument to the predicate `parEval/2` which defines the partial evaluator.

```
parEval(G, C) :-
   parEval(G,[ ],C).
```

The predicate
`parEval(+Goal,+InputConstraints,-OutputConstraints)` takes the input `Goal` and an initially empty list of constraints `InputConstraints` and returns the output list of constraints in `OutputConstraints`.

```
parEval((A,B), Cin, Cout) :-
   parEval(A, Cin, Cout1),
   parEval(B, Cout1, Cout).
parEval(G, C, C) :-
   cob_built_in(G), call(G).
parEval(G, C, C) :-
   compiler_built_in(G), call(G).
parEval(A→B;C, Cin, Cout):-
   call(A)→ parEval(B, Cin, Cout1);
   parEval(C, Cout1, Cout).
parEval(cc(A,B), Cin, Cout ) :-
   evalCC(A,B,Result),
   (Result -> (Cin = Cout);
    (Result=callA -> Cout = [A | Cin];
     Cout = [cc(A,B) | Cin])).
parEval({X = N}, C, C) :-
   var(X), number(N), call({X = N}).
parEval(G, C, C) :-
   is_unification(G), call(G).
parEval(G, C, [G | C ] ):-
   predicate_property(G, imported_from(clpr)).
parEval(G, C, C) :-
   clause(G,Body), parEval(Body).

evalCC(A, B, true) :- entailed(A).
evalCC(A, B, callA) :- entailed(B).
evalCC(A, B, true) :- ground(B).
evalCC(A, B, cannot_prove).
```

We describe some of the interesting clauses from the above definition.

- The Cob built-in and compiler built-in predicates are for side effects and appropriate translation of selection, enumeration or indexing of Cob program variables. These are predefined predicates whose definitions do not contain any constraints. The partial evaluator evaluates such goals to completion without further analysis. Similarly unification goals are also evaluated without further analysis.
- A conditional constraint (`cc(A,B)`) is evaluated using the `evalCC` predicate. If the consequent entails from the current information, the conditional constraint is true and the partial evaluation proceeds to the next goal. If the antecedant entails in the current state, the consequent is added to the list of output constraints. If the antecedant is ground and does not entail, the conditional constraint is true. If there is insufficient information to prove the conditional constraint, it is added to the list of output constraints. The Sicstus Prolog predicate `entailed/1` tells whether or not a given constraint is entailed by the current constraint store.
- A non-conditional constraint is detected using the Sicstus Prolog predicate `predicate_property` which returns the value `imported_from(clpr)` for constraints. Constraints of the form $X = number$ are evaluated because they represent variable initialization. All other constraints are added to the existing list of constraints.
- If the goal is not handled by any of the above rules, it is resolved into a collection of goals using the rules in program P.

Given a goal CLP program P and a goal G, suppose the output of the partial evaluator is the list of constraints C. An instance g of G is evaluated by solving the corresponding instance c of C. This is illustrated in the next section.

4.2 Examples

We now give two examples to illustrate the result of the above scheme for partial evaluation.

Heat Transfer in a Plate. The problem is to model a plate in which the temperature of any point in the interior of the plate is the average of the temperature of its neighbouring four points. This can be stated mathematically by using 2d Laplace's equations. In Cob, we model every point on the plate as a `cell` with four neighbouring cells. The value of the `Flag` attribute indicates whether it is an interior cell (`Flag=1`) or not (`Flag=0`). If a cell on the interior, its temperature `T` is related to its neighbours. This is modeled by the conditional constraint in class `cell` The Cob representation of a 2D plate is shown below in a class called `heatplate`. Compared to its CLP(R) representation, the Cob representation of this problem is more comprehensible, and the quantified and conditional constraints make it reusable for different sizes of the plate.

```
class cell {
 attributes
  cell Left, Right, Up, Down;
  real T, Flag;
 constraints
  (T = (Left.T+Right.T+Up.T+Down.T)/4) :- Flag=1;
 constructor cell(T1, F1) {
  T = T1; Flag = F1;
 }
}
class heatplate {
 attributes
  int Size;
  cell [Size][Size] Plate;
 constraints
  forall I in 2..Size-1:
     (forall J in 2..Size-1:
       (Plate[I,J].Up = Plate[I-1,J];
       Plate[I,J].Right = Plate[I,J+1];
       Plate[I,J].Left = Plate[I,J-1];
       Plate[I,J].Down = Plate[I+1,J];););
 constructors heatplate(S, A,B,C,D) {
  Size = S;
  forall I in 2..Size-1:
     forall J in 2..Size-1:
       (Plate[I,J] = new cell(_, 1););
```

```
  forall K in 1..Size:
    (Plate[1,K] = new cell(A,0);
      Plate[Size,K] = new cell(B,0););
  forall L in 2..Size-1:
    (Plate[L,1] = new cell(C,0);
     Plate[L,Size] = new cell(D,0););
 }
}
```

The Cob compiler translates the above program to the CLP program P using the translations in section 4. Suppose we want to compute the heat at all the interior points of a 4x4 grid and suppose that the temperatures along the top, bottom, left and right border of the grid are `A`, `B`, `C` and `D` respectively. We can partially evaluate the query
`par_Eval(heatplate(4,A,B,C,D),Constraints).`
to obtain the constraints:

X_1 = (A + X_2 + C + X_3)/4,
X_3 = (A + X_4 + X_1 + D)/4,
X_2 = (X_1 + C + B + X_4)/4,
X_4 = (X_3 + B + X_3 + D)/4

where X_i are the Cob program variables (`Plate[I,J]`) The array indexing is processed by the partial evaluator. The conditional constraints in the cell class are simplified by the partial evaluator since the value of the flag variable is known for all cells. Suppose the above set of constraints is denoted as C(A,B,C,D). We can now obtain solution for any 4x4 grid by solving these constraints. For example, a 4x4 grid, with the top row set to 0 and the rest of the border set to 100, can be evaluated simply by solving the constraints C(0,100,100,100).
These constraints can also be displayed in terms of the original Cob program variables as:

```
Plate[3][3] = (Plate[2][3]+100+Plate[3][2]+100)/4
Plate[3][2] = (Plate[2][2]+100+100+Plate[3][3])/4
Plate[2][3] = (0+Plate[3][3]+Plate[2][2]+100)/4
Plate[2][2] = (0+Plate[3][2]+100+Plate[2][3])/4
```

By solving these constraints we get the values:

```
    Plate[3][3] = 87.5
    Plate[3][2] = 87.5
    Plate[2][3] = 62.5
    Plate[2][2] = 62.5
```

The constraints for this example were linear equations and were solved in CLP. In general the constraints may be non-linear and hence cannot be solved in the clpr module of SICStus prolog. The next example shows how non-linear constraints can be solved using partial evaluation.

Truss Design. For the truss problem in section 3.2.1, we encounter several non-linear constraints. A concrete example of a truss and its Cob model are given in [10]. A sample truss is created using five steel beams of having a square cross-section. The length of the beams, the angle at which they are placed at a joint and the load at each joint is given. The problem is to determine the thickness required of each beam to be able to support the load. The query `T = sampletruss()` is partially evaluated to obtain a set of constraints. We show below the constraints for one of the instances of the beam (AB) and joint (JA) class:

```
Iab=-Fab*(12*10.4)*(12*10.4)/(3.141*3.141*3.0E+07) :-
    Fab<0
Fab=3.0E+04*(W*H) :- Fab > 0
Iab = (W1 * W1 * W1 * W1) / (12)
Fab * cos(3.141/4) + Fac * cos(0) +
    Fav * cos(3.141/2) + Fah * cos(0) = 0
Fab * sin(3.141/4) + Fac * sin(0) +
    Fav * sin(3.141/2) + Fah * sin(0) = 0
```

These non-linear constraints cannot be solved in the CLP system. Hence we make use of Maple [20], which is a computing system for interactive algebra, calculus, discrete mathematics, graphics, numerical computation and many other areas of mathematics. From the above set of non-linear constraints, we give the non-conditional constraints to Maple. The answers obtained include the value of `Fab` which is -13440.80489. Using this value, we can simplify the conditional constraints. The second conditional constraint dissappears because its antecedant is false. The first conditional constraint simplifies to yield the constraint
`Iab =-(-13440.80489)*(12*10.4)*(12*10.4)/(3.141*3.141*3.0E+07)`
Now the new set of constraints is again solved to obtain the values
`S1.AB.W = 1.70684719`
`S1.AB.H = 1.70684719`
Our current system has been interfaced with Maple.

Efficiency. The partial execution is implemented in Sicstus Prolog. efficiency in solving achieved by this partial evaluation. The time taken for the partial evaluation and the time taken to solve is less than the time taken to run the program in Sicstus clpr.

4.3 Interactive Execution

The result of partial evaluation of a Cob program or model is a flat list of constraint. For large programs, if a constraint failed, it would be difficult to trace the error back to the object that that gave rise to the constraint. We introduce the notion of a constrained object graph which serves to give a meaningful representation of constraints and can be used for interactive execution and debugging of partially evaluated Cob programs.

Constrained Object Graph. A complex object may be depicted by an object graph. Each node represents an object (i.e., instance of some class) and each directed edge represents an aggregation relationship. Replacing each node of the object graph with the constraints of the object it represents results in the formation of the constrainted object graph. In the case of Cob models of engineering artifacts, the structure of this graph implicitly represents the assembly of components of the artifact.

We have implemented a tool that generates the constrained object graph for Cob programs. Figure 3 shows the constrained object graph for the samplecircuit defined in section 2.2. In this example, all instances of resistors, battery, etc are created in the class `samplecircuit`. In general this need not be the case. The components `R1` and `R2` might be created inside the parallel class. In either case, the object graph will be identical to the one in figure 3. This is because, the edges of the object graph denote aggregation. They do not indicate where the instance was created.

Augmented Translation. In order to build an object graph, the translation must include information regarding object creation and aggregation a map of Cob variable names and their corresponding CLP variables. The translation shown in section 3 has to be augmented in order to store this information. We show below only the modified parts of the translation. The rest of the translation is identical to that given in section 3.

The translator inserts calls to predefined predicates for building the object graph, collecting constraints at every node and maintaining information about which node is currently being evaluated (to indicate the node to which subsequent constraints will belong).

```
class c {                      ==>T   P1 ∪ {defn(pc)}
  attributes V1                       where defn(pc) is:
  constraints C1                       pc(V2, V1) :-
  predicates P1                            edge(V1),
  constructor c(V2){                       mapvar(V1, NamesV1)),
     C2                                    C1, C2.
  }                                   and
}                                      NamesV1: the names of
                                       the variables in V1
```

Creational constraints are translated as shown below. Calls to the predefined predicates `in` and `out` are placed before and after the creational constraints. This indicates to the partial evaluator that a new node is being created and that all the subsequent constraints between these two calls belong to one node.

X = `new` $c(\bar{t})$ $\Longrightarrow_T$ `in(`X_T`),`
$\mathrm{p}_c(\bar{t}_T,\ X_T)$
`out(`X_T`).`

where p_c is the translation of class c
X_T is the translation of X
and $\bar{t}_T$ is the translation of $\bar{t}$

Partial Evaluator built_ins: We refer to the predicates `edge, mapvar, in, out,` as the partial evaluator built_ins. `edge` is used for building the edges of the object graph. `mapcompoundvar` is used for building the CobCLP variable map for variables of the kind X.Y
`in, out` are used for building the Cob to CLP variable map and as the beginning and end markers for distinguishing the constraints of each node.

Augmented Partial Evaluator. The partial evaluator is also modified in order to build the object graph. The predicate `parEval/3` defined in section 4 is changed changed to `parEval/7`. It maintains a map (`Min, Mout`) of variable names and keeps track of the current node (`Nin, Nout`) of the object graph.

parEval(G, Cin, Cout, Min, Mout, Nin, Nout) :-
par_eval_built_in(G),
call_approp(G).

On all goals except the partial evaluator built_ins, the partial evaluator behaves as before. The partial evaluator built_ins are called appropriately using the current state of the object graph, the variable map, and the current node being visited. The output of this partial evaluator is used to build the constrained object graph.

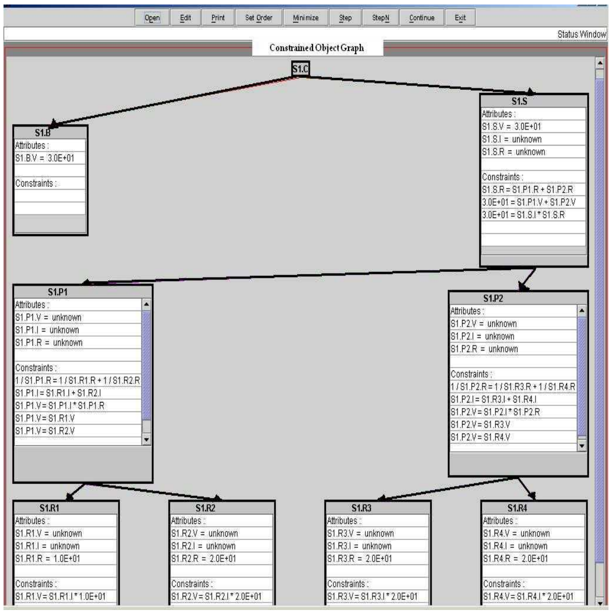

Figure 3: Object Graph of Samplecircuit

4.4 Comparison with Related Work

Partial evaluation of logic programs has been studied and described in [1, 14].The techniques used to perform partial evaluation include program transformations using unfolding, folding and definition [17]. Given a program and a query, they return a program specialized for all instances of the query. The program returned by partial evaluation is expected to be more efficient than the original program. A scheme for partial evaluation of constraint logic programs [8] is discussed in [6] which uses the fundamental transformations: specialization, unfolding and factorization. Specialization replaces a query with a call to a specialized version of the predicate. Unfolding is done by resolving a goal with the body of the specific clause it matches. Redundant code is eliminated and constraint simplification is done by detecting tautologies and contradictions.

Specialization techniques can be applied to constraint logic programs to obtain determinism [3], i.e., only one clause is applicable for goal reduction at a time. Again this too uses specialized fold, unfold and partitioning operations. [18] describes the fold and unfold operations for constraint logic programming with dynamic scheduling.

Our work differs from the earlier approaches to partial evaluation of constraint logic programs in the following ways. We deal with a special class of CLP programs: those obtained by translating Cob programs. Such CLP programs have different types of predicates: Cob built-in predicates, conditional constraints, quantified constraints. The explicit denotation of CLP(R) constraints in Sicstus Prolog and domain knowledge allows us to obtain the constraints given a program and a query.

5. CONCLUSIONS

The implementation of constrained objects involves two main steps: translation of Cob classes to CLP predicates, and partial execution of the translated program. Although the class definitions define essentially abstract data types, the translation to CLP predicates is quite natural, compact, and straightforward. However, the translated programs are not suitable for immediate execution. There are overheads arising from the representation of arrays by lists and indexing by list traversal; and also due to the loop iterations caused by quantified constraints. Partial evaluation allows us to process away these operations resulting in a highly reduced set of operations that need to performed.

We have developed an implementation of all the techniques presented in this paper. Our initial experiments with partial evaluation shows that there is a 10-30 fold improvement in performance. We have also found the interface to Maple to be a very effective means of handling nonlinear constraints.

Presently we are examining applications in structural engineering and also in hydrological modeling, in collaboration with researchers in these specialties. We believe that our implementation will provide a basis for developing efficient and effective applications in these areas.

6. ACKNOWLEDGMENTS

This research was partially funded by grants from the National Science Foundation and the Xerox Foundation.

7. REFERENCES

[1] H. Abramson and M. Rogers. *Meta-Programming in Logic Programming*. MIT Press, 1989.

[2] A. Borning. The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS*, 3(4):353–387, 1981.

[3] F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and strategies for contextual specialization of constraint logic programs. *Electronic Notes in Theoretical Computer Science*, 30(2), 1999.

[4] P. Fritzson and V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *12th European Conference on Object-Oriented Programming*, pages 67–90, 1998.

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

[6] T. J. Hickey and D. A. Smith. Toward the partial evaluation of clp languages. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 43–51. ACM Press, 1991.

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

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

[9] B. Jayaraman and P. Tambay. Semantics and Applications of Constrained Objects. Technical Report 2001-15, Department of Computer Science and Engineering, University at Buffalo, 2001.

[10] B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In *Proc. Symposium on Practical Aspects of Declarative Languages*, pages 28–46, 2002.

[11] B. Kristensen, O. Madsen, B. Moller-Perdersen, and K. Nygaard. Object-Oriented Programming in the BETA Programming Language. Norwegian Computing Center, Oslo and ComputerScience Department, Aarhus University, Aarhus, Denmark, 1989.

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

[13] J. Lloyd. *Foundations of Logic Programming*, pages 6–7. Springer-Verlag, 2nd edition, 1987.

[14] J. Lloyd and J. Shepherdson. Partial evaluation in logic programming. Technical Report 87-09, Univ. of Bristol, 1987.

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

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

[17] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Department of Telecommunication and Computer Systems, Stockholm, Sweden, March 1991.

[18] Sandro Etalle and Maurizio Gabbrielli and Elena Marchiori. A transformation system for CLP with dynamic scheduling and CCP. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 137–150. ACM Press, 1997.

[19] SICS-AB. SICStus Prolog, 2002.

[20] Waterloo Maple Inc. Maple 7, 2001.