The Cob Programmer's Manual

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

1 Introduction

Cob is a programming language and execution environment based on the concept of *constrained objects* for compositional and declarative modeling of engineering structures. A constrained object is an object whose internal state is governed by a set of (declarative) constraints. When several constrained objects are aggregated to form a complex object, their internal states might further have to satisfy interface constraints. The resultant behavior of the complex object is obtained by logical inference and constraint satisfaction. For the domain of engineering modeling, the paradigm of constrained objects is superior to both a pure object-oriented language as well as a pure constraint language. While the concept of an object and its attributes capture the structural aspects of an engineering entity, the concept of constraint captures its behavioral properties. Our current prototype includes tools for authoring constrained-object class diagrams; a compiler that translates class diagrams to CLP(R) code; and domain-specific visual interfaces for building and testing constrained objects [1, 2]. This manual describes version 1.0 of the Cob programming language.

2 Syntax of Cob Programs

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

$$program \ ::= class\_definition^{+}\,\$$$

All class definitions including the driver class must be in a single file. A Cob program must end with the $ character. Any text appearing after the $ symbol is ignored by the compiler.

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

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

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 any constructor, and hence cannot be instantiated. A class name must begin with a lowercase letter, e.g. `component, series, parallel`.

Attribute Declaration. 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.

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

The size of an array may be constant or left unspecified, e.g.,

```
component [] [] Varsize2DArray;
component [3] FixedsizeArray;.
```

A variable/attribute name must begin with an uppercase letter, e.g., `V, I, R`. An attribute declaration ends with a semi-colon (`;`) and multiple attributes within a declaration are separated by commas (`,`). There are two internal variables used by the Cob compiler (`Cob` and `cob`), and these names must not appear in a Cob program. For the same reason, the _ symbol must not appear as a part of any variable/attribute name. All the attributes of a class must be declared at the beginning of the body of the class following the keyword **`attributes`**. If the class does not have any attributes, the keyword **`attributes`** should be left out.

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

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

Each constraint must end with a semi-colon. 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 any data domain (e.g. integers, reals, etc.) as well as constants and attributes. e.g.

```
V = I * R;
Theta =< 2* 3.141;
X = sin(Theta);
```

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. e.g.

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

A quantified constraint is a shorthand for stating a relation where the participants of the relation may range over enumerations, i.e., indices of an array or the elements of an explicitly specified set. e.g.

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

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.

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

$$\begin{aligned} \textit{term} \ ::= \ & \textit{constant} \mid \textit{var} \mid \textit{attribute} \mid (\textit{term}) \mid \textit{arithmetic\_expr} \mid \textit{func\_id}(\textit{terms}) \\ & \mid \texttt{sum}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\ & \mid \texttt{prod}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\ & \mid \texttt{min}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \\ & \mid \texttt{max}\ \textit{var}\ \texttt{in}\ \textit{enum} : \textit{term} \end{aligned}$$

A term may be an arithmetic/boolean expression involving attributes and/or constants from any 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`
`prod Y in IntArray : Y`	stands for	$\pi_{i=1}^{n}$ `IntArray[`i`]`
`min X in RealArray : X^2`	stands for	min $\{X^2 \mid X \in \texttt{RealArray}\}$

where n = length of PC; m = length of IntArray

Attribute. An attribute 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.

$$\begin{aligned} \textit{attribute} \ &::= \textit{selector}[.\textit{selector}]^{+} \mid \textit{attribute}[\textit{term}] \\ \textit{selector} \ &::= \textit{attribute\_id} \mid \textit{selector\_id}(\textit{terms}) \\ \textit{terms} \ &::= \textit{term}\ [\ ,\ \textit{term}\ ]^{+} \end{aligned}$$

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

$$\begin{aligned} \textit{literals} \ &::= \textit{literal}\ [\ ,\ \textit{literal}\ ]^{+} \\ \textit{literal} \ &::= [\ \texttt{not}\ ]\ \textit{atom} \\ \textit{atom} \ &::= \textit{predicate\_id}(\textit{terms}) \mid \textit{constraint\_atom} \end{aligned}$$

If p is an n-ary predicate symbol and $t_1, ..., t_n$ are terms then $\texttt{p}(t_1, ..., t_n)$ is an atom [3]. A constraint_atom is an atom with a predifined 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 (whose syntax is shown below), whereas the latter are a set of predefined predicates (as in CLP-like languages) whose properties are known to the underlying constraint satisfaction system.

$$
\begin{aligned}
\mathit{pred\_clauses} &::= \mathit{pred\_clause}\ .\ [\ \mathit{pred\_clause}\ .\ ]^{+}\\
\mathit{pred\_clause} &::= \mathit{clause\_head} :- \mathit{clause\_body}\\
\mathit{pred\_clause} &::= \mathit{clause\_head}.\\
\mathit{clause\_head} &::= \mathit{predicate\_id}(\mathit{terms}')\\
\mathit{clause\_body} &::= \mathit{goal}\ [\ ,\ \mathit{goal}\ ]^{+}\\
\mathit{goal} &::= [\ \texttt{not}\ ]\ \mathit{predicate\_id}(\mathit{terms}')\\
\mathit{terms}' &::= \mathit{term}'\ [\ ,\ \mathit{term}'\ ]^{+}\\
\mathit{term}' &::= \mathit{constant} \mid \mathit{var}' \mid \mathit{function\_id}(\mathit{terms}')
\end{aligned}
$$

Note that the only variables that may appear in a *term* 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.

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

$$
\begin{aligned}
\mathit{constructor\_clauses} &::= \mathit{constructor\_clause}^{+}\\
\mathit{constructor\_clause} &::= \mathit{constructor\_id}(\mathit{formal\_pars})\ \{\ \mathit{constructor\_body}\ \}\\
\mathit{constructor\_body} &::= \mathit{constraints}
\end{aligned}
$$

The type of a formal parameter of the constructor is not specified. Hence selection operation on these parameters is not allowed. However, if they are equated to a local attribute of the class, then the select operation can be performed on the local attribute, since its type is known. 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.

3 Example

Non-Series/Parallel Circuits. To further illustrate the syntax of Cob and the use of equational and quantified constraints, we present the well-known example of a non-series/parallel electrical circuit. We model the components and connections of such a circuit as objects and their properties and relations as constraints on and amongst these objects. The `component` class models any electrical entity (e.g resistor, battery) that has two ends (referred to as 1 and 2). The attributes of this class represent the currents and voltages at the two ends of the entity. The constraint in class `resistor` represent Ohm's law.

The class `end` represents a particular end of a component. We use the convention that the voltage at end 1 of a component is `V1` (similarly for current). A `node` aggregates a collection of ends. When the ends of components are placed together at a node, their voltages must be equal and the sum of the currents through them must be zero (Kirchoff's law). Notice the use of the quantified constraints (`forall`) to specify these laws. Using these classes we can model any non-series/parallel circuit. Given initial values for some attributes, this model can be used to calculate values of the remaining attributes (e.g. the current through a particular component).

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

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

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

```
class end {
 attributes
  component C;
  real E, V, I;
 constraints
  V = C.V1 :- E = 1;
  V = C.V2 :- E = 2;
  I = C.I1 :- E = 1;
  I = C.I2 :- E = 2;
 constructors end(C1, E1)
  { C = C1; E = E1; }
}
class node {
 attributes
  end [] Ce;
  real V;
 constraints
  sum C in Ce: C.I = 0;
  forall C in Ce: C.V = V;
 constructors node(L) {
  Ce = L; }
}
```

Using the above class definitions we give a constrained object definition of the circuit diagram in Figure 1.

```
class samplecircuit {
  attributes
  resistor R12, R13, R23, R24, R34;
  battery B;
  end Re121, Re122, Re131, Re132, Re231,Re232 ,
      Re241, Re242, Re341, Re342, Be1, Be2;
  node N1, N2, N3, N4;
  constructors samplecircuit(X) {
    R12 = new resistor(10);
    R13 = new resistor(10);
    R23 = new resistor(5);
    R24 = new resistor(10);
    R34 = new resistor(5);
    Re121 = new end(R12, 1); Re122 = new end(R12, 2);
    Re131 = new end(R13, 1); Re132 = new end(R13, 2);
    Re231 = new end(R23, 1); Re232 = new end(R23, 2);
    Re241 = new end(R24, 1); Re242 = new end(R24, 2);
    Re341 = new end(R34, 1); Re342 = new end(R34, 2);
    B = new battery(10);
    Be1 = new end(B, 1); Be2 = new end(B, 2);
    N1 = new node([Re121, Be1, Re131]);
    N2 = new node([Re122, Re241, Re231]);
    N3 = new node([Re132, Re232, Re341]);
    N4 = new node([Re242, Re342, Be2]);
    dump([R12, R23, R34, R24, R13]);
  }
}
```

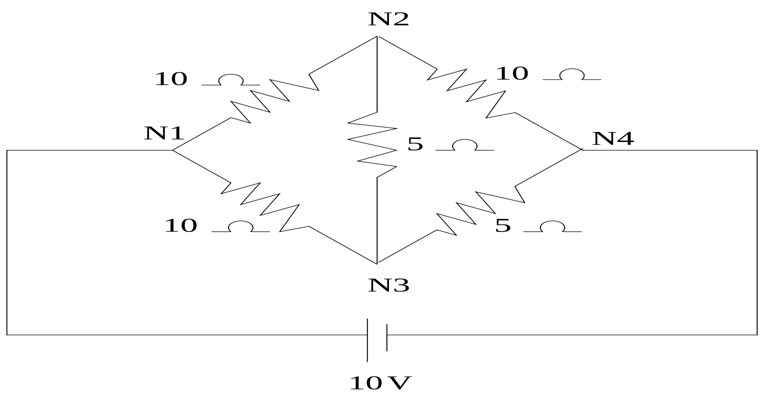

Fig. 1. Simple Non-Series Parallel Circuit

For more examples of Cob programs for modeling engineering structures, see files with `.cob` extension in the directory
`/projects/tambay/Cob/`

4 Compiling and Running Cob Programs

4.1 Compilation Modes.

The compiler translates a Cob program to a CLP(R) program. Essentially each class definition translates to one predicate clause. We use the underlying CLP(R) engine for constraint handling. Depending upon whether the programmer wants to work with the compiled clpr file or directly run the executable, there are different modes of compilation described below.

1. To compile a Cob file named `foo.cob`, into a file named `foo.clpr` use the command `$ /projects/tambay/temp/compiler/cob foo.cob foo.clpr`
   The clp program in `foo.clpr` should be run in the sicstus clpr module. To do this, start a prolog process and load in the file `foo.clpr`. Cob queries can now be evaluated at the prolog prompt. Note however, that the queries must be calls to the constructor of some constrained object appended with an extra argument.
   ```
   $ prolog
   |?- ['foo.clpr'].
   |?- samplecircuit(_,_).
   ```

2. An alternate method of compilation is to compile a Cob file named `foo.cob`, into a file named `foo.clpr` and place the compiled (in prolog) clpr file in foo.run using the command
   `$ /projects/tambay/temp/compiler/cob   foo.cob   foo.clpr   -e foo.run`
   To run the executable, type in its filename and press return. A prolog prompt will be displayed. Queries should be given at this prompt. Note that the queries must be calls to the constructor of some constrained object appended with an extra argument.
   ```
   $ foo.run
   |?- samplecircuit(_,_).
   ```
3. A third way to compile is with the `-c` option. This will create the executable as in the previous command but with a cobinterface above the prolog interpreter. To compile in this way, run the command
   `$ /projects/tambay/temp/compiler/cob   foo.cob   foo.clpr   -e foo.run -c`
   To run the executable, type in its filename and press return. A cob prompt will be displayed. Queries can be entered at this prompt in the form of calls to constructors of constrained objects.
   ```
   $ foo.run
   cob_query ?- samplecircuit(_).
   ```
   During the evaluation of the query, if there is an exception, then control will return to the prolog prompt. To return to the `cob_query` mode, type the `cob_prompt.` command.

4.2 Example

Let the name of the above Cob program be `kirchoff.cob`. Below is a script showing the compilation and running of the program.

```
tambay@kulta:compiler@1:06:40pm>cob kirchoff.cob kirchoff.clpr -e ckt
% restoring /projects/tambay/CobToSicstus/cobcompiler...
...
yes
...
% consulting /projects/tambay/temp/compiler/kirchoff.clpr...
....
% /projects/tambay/temp/compiler/ckt.sav created in 110 msec
yes

tambay@kulta:compiler@1:06:52pm>ckt
% restoring /projects/tambay/temp/compiler/ckt...
...
| ?- samplecircuit(A,_).
R12 =  [1.0E+01,4.375,0.5625,-0.5625,1.0E+01,_1172]
R23 =  [4.375,3.75,0.12500000000000006,-0.12500000000000006,5.0,_2904]
R34 =  [3.75,0.0,0.7500000000000001,-0.7500000000000001,5.0,_4636]
R24 =  [4.375,0.0,0.43749999999999994,-0.43749999999999994,1.0E+01,_3770]
R13 =  [1.0E+01,3.75,0.625,-0.625,1.0E+01,_2038]
A = [_A] ?
yes
```

4.3 Error Messages and Warnings

The cob compiler will point out a syntax error by naming the class in which it occurs, the attribute/constraint/constructor definition in which it occurs and the index of the constraint/attribute declaration. If the error is in the ith constraint, it means, it is in the ith semi-colon separated constraint. Line number of error is not given. Undeclared variables will not be caught unless the selection operation (`.`) is being performed on them. If the compilation hangs, please send e-mail with the uncompilable cob program to **tambay@cse.buffalo.edu**

If the executable is being formed along with compilation, then errors (if present) will be detected by the prolog interpreter. These should be removed by correcting the cob program and re-compiling it.

If the clp translation of a cob program is loaded within prolog manually, then there may be a series of warnings of singleton variables. In most cases, these can safely be ignored. However, errors shown during this compilation must not be ignored. They should be removed by correcting the cob program and re-compiling it.

5 Miscellaneous

Printing Two functions are provided for printing the values of attributes to standard output.

1. dump: To print the value of a variable, use the built-in Cob predicate dump/1. If A,B,C are Cob program variables with values 1,2 and unknown respectively, and X is an undeclared variable, then `dump([A,B,C,X])` will print
```
A = 1
B = 2
C = _some_internal_name
X = _some_internal_name
```
2. print: To print a string or just the value of a variable, use print/1. `print('Sample String')` will print `Sample String` on standard output.

Type checking Currently type checking is performed only on the variables on which the select/access "." operation is performed. This type inference is done at run-time.

Tracing The translated CLP program can be traced by using Prolog's `trace` command. Once in trace mode, the normal debugging commands of Prolog can be used to trace the program.

Using underscore The underscore character (_) can be given as argument to constructors or predicates. The programmer should get familiar with its use Prolog before using it in Cob.

6 Bug Reports and Other Comments

Please submit bug reports and other comments to **tambay@cse.buffalo.edu**

References

1. B. Jayaraman and P. Tambay. Constrained Objects for Modeling Complex Structures. In *Object-Oriented Programming Languages Systems and Applications, Companion*, pages 71–72, 2000.
2. B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In *Proc. Symposium on Practical Aspects of Declarative Languages*, 2002.
3. J. Lloyd. *Foundations of Logic Programming*, pages 6–7. Springer-Verlag, 2nd edition, 1987.