The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models

Ricardo Soto
LINA, CNRS, Université de Nantes, France
Escuela de Ingeniería Informática, PUCV, Chile
ricardo.soto@univ-nantes.fr

Laurent Granvilliers
LINA, CNRS, Université de Nantes, France
laurent.granvilliers@univ-nantes.fr

Abstract

This paper presents the first implementation of COMMA, a new solver independent language for modeling constraint-based problems. The combination of a constraint language with an object-oriented framework represents the base of the core of COMMA. Extension capabilities have also been included with the aim of tackling a wide range of applications from combinatorial to continuous problems. A COMMA compiler has been implemented through a three layered architecture including a dynamic parsing system for handling efficiently the mapping process. In particular, COMMA models can be translated to different solvers, currently to ECLiPSe and Gecode/J.

1. Introduction

Constraint programming (CP) allows one to describe systems in terms of unknowns, namely the variables, relations between the unknowns called constraints, and to calculate values of the variables satisfying the constraints. This is a general approach to tackle constraint-based problems that combines rich modeling languages with powerful solving techniques and cooperation mechanisms. CP is said to be declarative since the user has to state properties of the system that must be verified rather than sequences of operations leading to the solutions.

CP languages are in essence very expressive, combining constraint languages with host languages and strategy languages. The constraint language is used for modeling with variables, elementary constraints such as equations or inequalities, and global constraints. The host language is used as a programming language with control operations such as loops and conditionals, types, and sub-programs. Constraint solving may also be tuned using some strategy language, for instance to define the search ordering on the variables. Encoding operational concerns may be the price to pay for efficient solving.

A main drawback in this research area is that the current CP technology is too complex to use for the average engineer. Both, available constraint libraries and CP languages require a considerable background in programming, mathematics, and in the application domain. One of the acknowledged ways [13] to simplify the use of this technology, is to increase the effort on the development of simple, but at the same time expressive enough modeling languages.

Following this research direction, we have implemented the COMMA [2] constraint modeling language and its execution platform. Its design is based on the combination of proven ideas present in existing languages in order to make it expressive but simple enough for use by non CP-specialists.

The system is implemented in Java (20 000 lines) and the ANTLR tool [1] is used for generating lexers and parsers. The core of the architecture can be summarized in the following three features.

- The whole language is built from a combination of a constraint language and an object-oriented language. The constraint language includes usual data structures, control operations, and first-order logic to define constraint-based formulas. The object-oriented language is an abstraction of the Java programming style. This framework clearly provides model structures using composition and inheritance.

- The constraint language is extensible. New functions and relations can be added to existent domains, extending the syntax of the constraint language. This capability makes the architecture adaptable to further upgrades of the solvers.

- The COMMA architecture is solver independent. Currently, COMMA models can be translated to two native solver models, ECLiPSe and Gecode/J. The architecture follows the “model and run” paradigm [23]. The user states one standard model, the system runs the solvers. There is no need for dealing with operational concerns to perform the searching process.
To the best of our knowledge, our approach is unique in including these three important characteristics. We believe that COMMA is simple since it is closer to UML than Java. The expressiveness of COMMA can be increased by extending the syntax of its constraint language. We expect that this work is a step towards defining a unified constraint modeling language \(^1\) such as AMPL or GAMS in the mathematical programming area.

The outline of this paper is as follows. The related work is presented in Section 2. In Section 3 we introduce the main features of the language by means of a well-known CP problem. Section 4 describes the means for extending the constraint language. The object-oriented framework is presented in Section 5. The compilation process is described in Section 6. Experimental results from the system are discussed in Section 7, followed by a conclusion.

2. Related Work

2.1. Constraint Programming

In the past years, significant work has been done on the development of constraint programming languages. For instance the constraint logic programming (CLP) paradigm is considered as the ancestor of current CP languages. In this approach constraint languages and associated solving techniques are embedded in a logic-based computer programming language called the host language. Examples of these languages are CHIP [9], CLP(\(\mathbb{R}\)) [19], and ECLiPSe [11].

Constraint solving libraries were also developed for CP. For instance, ILOG Solver [22] and Gecode [3] use C++ as its host language. Koalog [4] and Gecode/J [3] were implemented for Java. In these libraries a constraint language is given by means of built-in methods. Search procedures and search strategies can also be customized.

A generalization of previous constraint systems can be found in Mozart [25]. Support for several programming paradigms such as logic programming constraint programming and concurrency is provided.

CoJava [7] is a recent approach in constraint programming. CoJava extends the Java syntax adding special constructs to make nondeterministic choices, assert constraints and optimize objective variables.

Constraint programming languages and constraint libraries are mostly oriented to a CP-specialist audience. Their constraint vocabulary is very extensive. A significant background in CP techniques is needed. Logic, declarative or object oriented advanced programming skills are in general required to deal with such systems.

\(^1\)This challenge has been quoted as very important by many researchers from the constraint programming community, in particular Jean-François Puget at the 2005 CP conference.

2.2. Constraint Modeling

Modeling languages are based on the following principle: The user states the model and the system solve it. They are considered easier and simpler than constraint programming languages. A little background in mathematics and programming is needed to state models, no knowledge in CP solving techniques is required.

The first systems under this idea were constructed for the mathematical programming field, some examples are AMPL [14] and GAMS [5].

In the CP area, OPL [27] is one of the systems running under this principle. The OPL constraint language is very expressive, complex types and control abstractions are available to write models from diverse application areas.

Zinc [24] is a recent modeling language, its architecture is solver independent and models are translated to the ECLiPSe solver. The Zinc syntax is closer to the constraint language provided by OPL. Predicates and functions can be stated to modularize the models.

Essence [10] is another state-of-the-art language. Its syntax makes it a specification language rather than a modeling language. An Essence execution platform exists where specifications can be mapped to ECLiPSe.

CP problems can be modeled by means of different styles using Zinc, OPL, Essence or COMMA. However, COMMA differs from these languages in two main aspects: (1) COMMA is based on an object-oriented framework, which is a useful feature for modeling problems whose structure is inherently hierarchic. It is common to find this class of problems in applications areas such as design, biology, computer graphics and engineering. (2) COMMA has an extensible constraint language which allows one to increase its expressiveness using a simple description language (see Section 4).

2.3. The Constrained Object Modeling Paradigm

The research of Alan Borning in ThingLab [6] was one of the first attempts in combining an object oriented language with a constraint language. In ThingLab the constraint domain is fixed and the expressiveness of constraints is limited to interactive graphical simulation. This approach was further developed in the Kaleidoscope language [15]. Then, similar ideas were developed in Gianna for modeling constraint-based problems with objects in a visual environment [21]. Automatic configuration applications have also been modeled using these techniques [12]. In general, the constraint language of these approaches was designed to specific application domains.

COB [20] is a more recent language for constrained objects. Its framework is not purely based in this paradigm. In fact, the language is a combination of objects, first order formulas and CLP predicates. Modelica [16] is another
object-oriented approach for modeling problems, but it is mostly oriented towards simulation.

The work done in these approaches was the starting point of the constrained object modeling paradigm. This paradigm raises that hierarchic problems can be modeled in a more natural style using objects subject to constraints rather than a pure logic or declarative constraint language.

COMMA is built on this paradigm, constrained objects can be stated using a high-level modeling language where the state of objects is controlled by constraint solving. In COMMA, we extend earlier work done on this paradigm adding important features not present in aforementioned languages such as: a richer constraint language (conditional constraints, compatibility constraints, reified constraints, global constraints, optimization), an extensible constraint language, a simpler definition of classes (see Section 3) and a solver independent platform. The solver independence is considered as very important in modern modeling languages [24, 10]. This architecture gives the possibility to plug-in new solvers and to process a same model with different solvers, which is useful to learn which solver is the best for the model.

It is important to clarify that constrained objects can also be defined in constraint object-oriented languages such as CoJava; and in libraries such as Gecode or ILOG SOLVER. The main difference is that constrained objects must be programmed in the host language (not modeled as in constrained objects modeling languages). Once again, advanced programming skills may be required to perform this task.

3. Mapping CP problems to native solver models

A COMMA model is represented by a set of classes. Each class is defined by attributes and constraints. Attributes may represent decision variables or constrained objects. Decision variables must be declared with a type (Integer, Real, Boolean and Enumeration). Constants are given in a separate data file. A set of constraint zones can be encapsulated into the class with a given name. A constraint zone can contain constraints, loops, conditional statements, optimization statements, and global constraints. Loops can use loop-variables which do not have to be declared (\(i\) and \(j\) in the example shown in Figure 1). The definition of a class is simpler in contrast with constrained-objects languages mentioned in Section 2.3. We avoid encoding concerns that make models complex and difficult to state and understand. For instance, details related to object visibility (protected, private, public) have been omitted. There is no need for object constructors to state a class, direct variable assignment can be done in the constraint zone. We believe that these details are suitable for programming (such as in Gecode or ILOG SOLVER), but not for modeling.

Let us show some of the features included in COMMA by means of the Perfect Squares Problem [17]. The goal of this problem is to place a given set of squares in a square area. Squares may have different sizes and they must be placed in the square area without overlapping each other.

Figure 1 shows a COMMA model for the Perfect Squares Problem. Three constant values are imported from an external data file called PerfectSquares.dat, sideSize represents the side size of the square area where squares must be placed, squares is the quantity of squares (8) to place and size is an array containing the square sizes. At line 3, the definition of the class begins, PerfectSquares is the name given to this class. Then, two integer arrays of decision variables are defined, which represent respectively the x and y coordinates of the square area. So, \(x[2]=1\) and \(y[2]=1\) means that the second of the eight squares must be placed in row 1 and column 1, indeed in the upper left corner of the square area. Both arrays are constrained, the decision variables must have values into the domain \([1,\text{sideSize}]\).

```java
//data
sideSize := 5;
squares := 8;
size := [3, 2, 2, 2, 1, 1, 1, 1];
1. import PerfectSquares.dat;
2. class PerfectSquares {
3.   int x[squares] in [1, sideSize];
4.   int y[squares] in [1, sideSize];
5.   constraint inside {
6.     forall(i in 1..squares) {
7.       int x[i] <= sideSize - size[i] + 1;
8.       int y[i] <= sideSize - size[i] + 1;
9.     }
10.   }
11.   constraint noOverlap {
12.     forall(i in 1..squares) {
13.       for(j:=i+1;j<=squares;j++) {
14.         int x[i] + size[i] <= x[j] or
15.         x[j] + size[j] <= x[i] or
16.         y[i] + size[i] <= y[j] or
17.         y[j] + size[j] <= y[i];
18.       }
19.     }
20.   }
21.   constraint fitArea {
22.     int (sum(i in 1..squares) size[i]^2) - sideSize^2;
23.   }
24. }
25.}
```

Figure 1. A COMMA model for the Perfect Squares Problem

At line 7, a constraint zone called inside is declared. In this zone a forall loop contains two constraints to ensure that each square is placed inside the area, one constraint about rows and the other about columns. Due to extensibility requirements of the language, constraints must be typed.
In fact, the control engine needs to recognize the kind of constraints before sending it to the correct parser (see Section 6.1).

The constraint noOverlap declared at line 14 ensures that two squares do not overlap. The last constraint called fitArea ensures that the set of squares fit perfectly in the square area.

Let us now explain the mapping process of the Perfect Squares model to two equivalent native solver models; one written in Gecode/J and another in ECLiPSe. The translation process is carried out by mapping each COMMA construct to its equivalent solver construct.

Constants written in the data file are stated directly as constants in both native models. Decision variables are mapped to their equivalent ones in the solver language. For instance the array int x[squares] in \([1,\text{sideSize}]\) is translated as follows in Gecode/J. A new array of decision variables is declared. It is filled with 8 decision variables of integer type which represent the positions in the x-axis of the area. The constructor of the IntVar is composed by the JavaSpace of Gecode/J (this), the name of the variable, and the domain of the variable \(([1,\text{sideSize}])\). In the end, the new array is added to the vars object which will be used for the labeling process.

```java
VarArray<IntVar> x = new VarArray<IntVar>();
for (int i=0; i<squares; i++) {
    x.add(new IntVar(this,"x" + i,1,sideSize));
} vars.addAll(x);
```

The ECLiPSe code (see below) declares the array in the first line. SQUARES defines the size of the array. In the second line the domain for each variables is defined. As in Gecode/J, the latest lines are used for adding the new array to the array used for labeling.

```eclipse
dim(X, [SQUARES]),
X:=[1..SIDESIZE],
X=[\_ |VARS X],
append([\_],VARS_X,L_X),
```

Iterations are translated to their equivalent ones in the solver language. For instance the constraint zone inside is translated as follows in Gecode/J. Two constraints are posted using the method post, expressions are built using the Expr constructor. The parameter IRT_LQ represents the less or equal relation \((-\leq\)) and \(\cdot\) represents the addition and the subtraction functions respectively.

```eclipse
for (int i=0; i<squares; i++) {
    post(this, new Expr().p(x.get(i)), IRT_LQ, 
          new Expr().p(x.get(i).m(size[i]).p(i)));
    post(this, new Expr().p(y.get(i)), IRT_LQ, 
          new Expr().p(sideSize).m(size[i]).p(i));
}
```

The ECLiPSe code is as follows. Variables \(X\), \(Y\), SIDESIZE and SIZE must remain constant across the loop. This is stated explicitly using the param statement.

```eclipse
{for(1,1, SQUARES),param(X,Y,SIDESIZE,SIZE) do
    X[I] += SIDESIZE - SIZE[I] + 1, 
    Y[I] += SIDESIZE - SIZE[I] + 1),
}
```

The last constraint zone called noOverlap uses a disjunction constraint using the logical operator or. This constraint can be mapped directly to ECLiPSe by means of the same operator.

```eclipse
X[I] + SIZE[I] \#< X[J] or 
X[I] + SIZE[J] \#< X[I] or ... 
```

In the Gecode/J translation some adjustments have to be done. For each constraint in the disjunction a boolean var (BoolVar) is declared and a reified constraint is posted. Reified constraints reflects the truth value of a constraint in a boolean value. For instance, the truth value of \(x[i] + size[i] \leq x[j]\) is reflected in the BoolVar b1. In the end, all the boolean variables from the reified constraints are stated in a boolean constraint representing the entire disjunction. Let us note that reified constraints can also be stated in COMMA using the built-in function reify.

```eclipse
BoolVar b1 = new BoolVar(this,"b1");
post (this, new Expr().p(x.get(i)).p(size[i]),
    IRT_LQ, new Expr().p(x.get(j)).b1);
```

```eclipse
BoolVar b2 = new BoolVar(this,"b2");
...
post (this, new BExpr(b1).or(b2).or(b3).or(b4));
```

Finally, the code generated is merged with the code lines required by the solver host language such as imported libraries, headers, declarations, and specific constructs. Let us notice that important features such as optimization statements; and global constraints such as alldifferent, are also provided by COMMA.

4. Extending COMMA

One of the interesting aspects of COMMA is the capability of extending the constraint language. We are conscious that this feature may require skills in programming, but it is an important approach to extend a constraint language without recompiling it by hand. Let us go back to the Perfect Squares Problem. Consider that a Gecode/J programmer adds in the solver two new built-in functionalities: a constraint called noOverlap and a function called sumArea. The constraint noOverlap will ensure that two squares do not overlap and sumArea will sum the square areas of a square set. In order to use these functionalities we need to extend the constraint language. To this end, we define a new file (called extension file) where the rules of the translation are described. The language used for this description is strongly based on term rewriting [8].

This file is composed by three blocks (see Figure 2): an Option block, a Function block, and a Relation block. In the Option block we state the domain and the solver

2The grammar of these functionalities can be found in [2].
where the new functionalities will be defined, in this case the integer domain (`int`) and the Gecode/J solver are selected. Consequently the integer parser will be updated automatically to include these new functionalities (see Section 6.1). Functionalties involving more than one domain must be included in the mixed domain.

In the `Function` block we define the new functions to add. The grammar of the rule is as follows:

```
(COMMA-code) (input-parameters) -> (solver-code)
```

In the following example, the `COMMA` code is `sumArea`. This function name will be used to call the new function from `COMMA`. The input parameter of the new `COMMA` function is an array (`a[]`). Finally, the corresponding Gecode/J code is given to define the translation. The new function will be translated to `sumArea(s);`. This code allows us to call the new built-in method from the solver file. The translator must implement the correspondence between input parameters in `COMMA` and input parameters in the solver code. Therefore, variables must be tagged with `$` symbols. Here, the input parameter of the `COMMA` function will just be translated as the input parameter in the Gecode/J function.

```
Option {
  domain: int;
  solver: Gecode/J;
}
Function {
  sumArea(a[]) -> "sumArea($a$);";
}
Relation {
  noOverlap(x[], y[], size[], squares)
  -> "noOverlap($x$, $y$, $size$, $squares$);";
}
```

**Figure 2. Extension for Gecode/J Translation**

In the `Relation` block we define the new constraints to add. We use the same grammar as for functions. In the example, a new constraint called `noOverlap` is defined, it receives four parameters. The translation to Gecode/J is given. Once the extension file is completed, it can be called by means of an import statement. The resultant `COMMA` model using extensions is shown below.

```
import PerfectSquares.dat;
import PerfectSquaresDecode.ext;

class PerfectSquares {
  int x[squares] in [1,sideSize];
  int y[squares] in [1,sideSize];
  
  constraint placeSquares {
    forall(i in squares) {
      int x[i] <= sideSize - size[i] + 1;
      int y[i] <= sideSize - size[i] + 1;
    }
    int noOverlap(x,y[],size[],squares);
    int sumArea(size) = sideSize^2;
  }
  
  constraint dim {
    int volume > cCase.volume;
  }
}
```

**Figure 3. Car Engine**

Let us note that extension mechanisms are very important in modeling languages. If a new global constraint, method, predicate or functionality is added in the solver, the unique way to use it from the modeling layer is to extend the modeling language.

5. Mapping Hierarchic CP problems to native solver models

Complex structures as circuits, mixers, engines, computer graphics, molecules are in general entities composed by many pieces. These pieces have often their own composition rules and constraints between other pieces and/or between its attributes.

Modeling these structures is not quite natural using either a pure logic or a pure declarative CP language. It seems more appropriate to state the model as a Hierarchic CP problem where pieces of the system are represented by objects under constraints.

Let us now show the object-oriented framework included in `COMMA` by means of an academic problem from the configuration and design area. We consider the task of configuring a car engine using a hierarchical and compositional approach (see Figure 3). The engine at the first level is built from a crankcase, a cylinder system, a block and a cylinder head at the second level. The cylinder system is a subsystem made of a valve system, a piston system and an injection. Both valve and piston systems have their own composition rules.

```
class Engine {
  class CrankCase {
    CrankCase cCase; enum type in {a,b,c};
    CylSystem cSyst; int oilVesselVol;
    Block block; int bombePower;
    CylHead cHead; int volume
    int volume; }
  constraint dim {
    int volume > cCase.volume;
  }
}
The class \texttt{CylSystem} has a more complex declaration. The first attribute called \texttt{type} represents the cylinders configuration of the form $ab$ where $a$ is the number of cylinders and $b$ is the structure (in line or V). This attribute is called a constrained attribute because its domain is constrained to a limited number of possible values 4L, 6L, 6V and 8V. The attribute is represented by an enumeration type (\texttt{enum}).

The cylinder system has three subsystems denoted by \texttt{inj}, \texttt{vSys} and \texttt{pSys}. Then, a constraint zone called determinePressure is declared to state a conditional constraint. This conditional constraint represents that 4L-engines have a distance between cylinder bigger that 6. In others kinds of engines the distance must be bigger than 3. In order to represent this constraint, an if-else statement is stated. If the condition is true, the first constraint is activated. Otherwise, the second constraint is activated.

```java
class CylSystem {
    enum type in {4L, 6L, 6V, 8V};
    int distBetCyl in [3, 18];
    Injection inj;
    ValveSystem vSyst;
    PistonSystem pSyst;
    constraint determinePressure {
        if (type = 4L)
            int distBetCyl > 6;
        else
            int distBetCyl > 3;
    }
}
```

The subsystem injection is composed of three attributes called \texttt{type}, \texttt{admValve}, and \texttt{pressure}. The injection class has also a compatibility constraint \cite{18} between its components. A compatibility constraint limit the combination of allowed values for the decision variables to a given set. For example, for values \texttt{type}, \texttt{admValve} and \texttt{pressure} just four combination of values are allowed. The possible values are described inside the compatibility built-in constraint.

```java
class Injection {
    enum type in {direct, indirect};
    enum admValve in {small, medium, large};
    int pressure;
    constraint compValues {
        (type,admValve,pressure) {
            (direct, small, 80);
            (direct, medium, 90);
            (indirect, medium, 100);
            (indirect, large, 130);
        }
    }
}
```

Remaining elements of the engine are not presented because they can be modeled with the elements already shown in previous subsections. It is important to mention, that inheritance is allowed in \texttt{COMMA}. As in Java, we can use the reserved word \texttt{extends} to inherit all attributes and constraints defined in the superclass. Multiple inheritance is not allowed.

Let us now explain the mapping process of the Engine model. As in the previous explanation attributes are directly translated to their corresponding elements. In general solvers do not support non-numeric types. So, enum types are replaced by integer values, for example \texttt{admValve} in \{\texttt{small}, \texttt{medium}, \texttt{large}\} is replaced by \texttt{admValve} in \{1,2,3\}, original values are stored to give the results. If a variable type is not supported by the host language of the solver, a warning is given in the translation process.

Conditional statements are transformed to logical formulas. For instance, if $a$ then $b$ else $c$ is replaced to $(a \Rightarrow b) \cap (a \cup c)$. In the Gecode\textit{J} the translation the formula is written directly. In Gecode\textit{J}, we use the reified constraint as in the disjunction showed in the Perfect Squares problem. Then, the boolean formula is stated.

```java
post (this, new BExpr(a).imp(b));
post (this, new BExpr(a).or(c));
```

Compatibility constraints are also translated to a logical formula. For instance, in Gecode\textit{J} we create a conjunctive boolean expression for each n-tuple of allowed values \texttt{BEExpr} (\texttt{b1}, and (\texttt{b2}), and (\texttt{b3}). Each constraint of the n-tuple is stated as a reified constraint.

```java
post (this, new Expr().p(type), IRT_EQ, new Expr().p(1),b1);
post (this, new Expr().p(admValve), IRT_EQ, new Expr().p(1),b2);
post (this, new Expr().p(pressure), IRT_EQ, new Expr().p(80),b3);
```

In the end, we create a disjunctive boolean expression between the conjunctive boolean expression created before.

```java
post (this, (new BEExpr(b1).and(b2).and(b3)).
     or(new BEExpr(b4).and(b5).and(b6)).
     or(new BEExpr(b7).and(b8).and(b9)).
     or(new BEExpr(b10).and(b11).and(b12)));
```

In ECLiPSe just one big formula is stated.

\begin{align*}
\textsc{CASE} \#1 \textsc{ADMVALVE}\#1 \textsc{PRESSURE}\#80 \text{ or } \\
\textsc{CASE} \#1 \textsc{ADMVALVE}\#2 \textsc{PRESSURE}\#90 \text{ or }
\end{align*}

\textbf{5.1. Mapping constrained objects to solver source code}

The common way to map constrained objects to the solver source code is to eliminate the hierarchy by building a flat solver file. This process is done by expanding each constrained object declared in the main class adding its attributes and constraints in the flat file. The name of each attribute has a prefix corresponding to the concatenation of the names of objects of origin in order to avoid name redundancy. The process is done recursively. For instance, the Engine model is expanded as follows.

```java
Case_type in (a,b,c)
```

```java
case_oilVesselVol
```

```java
case_bombePower
```

```java
case_volume
```

```java
cSystem_type in {4L, 6L, 6V, 8V}
cSystem_distBetCyl in [3, 18]
cSystem_inj_type in {direct, indirect}
```
Constraints derived from the expansion of objects are added in the same way. This technique has two main problems. First, the growth of code can be huge. This code explosion depends on the levels of compositions; and on the amount of constraints and attributes of constrained objects added. Second, benefits gained by the object-oriented style at the modeling level are lost at the solver level. This is important for programmers who want to tune the solver code generated by the model.

In our platform we introduce a new solver translation style, where constrained objects are not expanded, they are just translated to their equivalent constrained objects in Gecode/J; and to their equivalent predicates in ECLiPSe. In the example shown below, each COMMA object is defined as a Java object, the class Gecode Methods contains the necessary methods to perform the communication between classes, for example the method getVar used in the constraint to get the decision variable volume of the cCase object. In the end, we add getters for each object of the class.

```java
public class Engine extends Comma {
    private CrankCase cCase;
    private CylSystem cSyst;
    ...
    public Engine(Space sp) {
        cCase = new CrankCase(sp);
        cSyst = new CylSystem(sp);
        ...
        post(sp,new Expr().p(volume),
            IRT_GR,
            new Expr().p(cCase.getVar("volume")));
    }
    public CrankCase getCCase() {
        return cCase;
    }
    ...
}
```

The same technique can be applied for CLP predicates in ECLiPSe. Each class is represented by a predicate. Variables of predicates represent the attributes of the object. The name of the object of origin is added as a postfix to avoid ambiguities. Both examples show a clear correspondence between COMMA and native solver models. Programs are modular and reusability is enhanced. The overhead in code lines is just due to the communication between classes.

```
engine(L):-
    crankcase( CCASE_TYPE, CCASE_OILVESSELVOL, 
                CCASE_BOMBEPOWER, CCASE_VOLUME),
    ...
    VOLUME #> CCASE_VOLUME,
    ...
```

6. The compiler

The compiling process is carried out by three independent compilers [26]. One for the COMMA language, one for the data and another for the extension files. The COMMA compiler is composed of one parser per constraint domain (Integer, Real, Boolean and Objects), one parser for constraints involving more than one domain (Mixed parser) and one base parser for the rest of the language (classes, import and control statements). Models are syntactically and semantically checked by two ANTLR tree walkers. In order to check types, class names, variable names, inheritances and compositions. Several cooperations between those parsers are performed at running time to generate the resultant ANTLR syntactic tree.

If the compiling process succeeds, the output is a Java object storing the model information, data information and extensions information using efficient representations. Then, another ANTLR tree walker explores the Java object and generates the attributes and constraints code for the corresponding solver. Finally, the code generated is merged with the code lines required by the solver host language. This file is compiled and the solving process can be launched.

6.1. Automatic Parser Update

A control engine is able to automatically update the necessary parsers when a new relation or function is added as an extension. The process is as follows: when a new extension file is detected by the base parser in a model, the extension compiler is called. The extension file is parsed, and then translated to an ANTLR grammar. This grammar is merged with the previous domain grammar to generate a new grammar from which the ANTLR tool generates the parser in Java code. The new parser for the updated domain is compiled and then it replaces the previous domain parser. The control engine adds the new tokens to a table of symbols. Ambiguities are managed by checking the new tokens of the extension with the existing tokens in the table of symbols.

The independence of parsers has been implemented for three reasons. (1) It gives us the adequate modularity to easily maintain the parsing engine. (2) It allows to avoid recompiling the base parser (and parsers not involved in the extension) each time a new extension is added. This leads to a faster extension process since it is not necessary to update and recompile the whole language, we recompile just the updated domain. (3) This is a necessary condition to avoid ambiguities between identifier tokens that may arise from new extensions added. For instance, the same function defined for two different domains.

7. Experimental Results

The tests have been performed on a 3GHz Pentium 4 with 1GB RAM running Ubuntu 6.06. Benchmarks were taken from the Gecode and ECLiPSe set of examples. In the end, we add the Engine example with a flat translation version and a hierarchic translation version. Tests have been done to contrast the size of COMMA models with na-
Table 1. Statistics

<table>
<thead>
<tr>
<th>Problem</th>
<th>COMMA tokens</th>
<th>Gecode/J tokens</th>
<th>ECLiPSe tokens</th>
<th>COMMA ms</th>
<th>Gecode/J ms</th>
<th>ECLiPSe ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send+More</td>
<td>587</td>
<td>112</td>
<td>172</td>
<td>231</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>N-Queens</td>
<td>148</td>
<td>193</td>
<td>148</td>
<td>155</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>Perfect Squares</td>
<td>184</td>
<td>528</td>
<td>119</td>
<td>510</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td>893</td>
<td>218</td>
<td>232</td>
<td>98</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>20 Linear Equations</td>
<td>530</td>
<td>2327</td>
<td>2347</td>
<td>790</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>Engine (Hierarchic)</td>
<td>148</td>
<td>221</td>
<td>210</td>
<td>155</td>
<td>135</td>
<td>218</td>
</tr>
<tr>
<td>Engine (Flat)</td>
<td>148</td>
<td>221</td>
<td>210</td>
<td>155</td>
<td>135</td>
<td>218</td>
</tr>
</tbody>
</table>

Table 1 gives the sizes of models in number of tokens, and translations times in milliseconds (ms) from COMMA to Gecode/J and ECLiPSe. The results show that COMMA models are considerably more compact than native solver models. Since COMMA is a modeling language, encoding details are not needed as in programming languages. For the Engine example a bigger difference is shown between both models. This is explained by the elimination of the hierarchy produced by the expansion of the constrained objects. As explained in Section 5, this process gives a bigger flat version file of the model.

8. Conclusions and Future Work

The first implementation of COMMA has been presented. We have shown through examples how COMMA is able to model typical benchmarks and complex structured problems using simple declarative constructs in an object-oriented approach. A simple definition of classes has been introduced with the spirit of avoiding the encoding details that make models complex and difficult to state and understand. The extensibility of COMMA is an important feature to increase the expressiveness of the language. The compactness of COMMA models in relation to native solver models has been demonstrated in benchmarks. We believe that this language may be a step towards a general constraint modeling language. For this purpose, several aspects could be developed, for instance: more work on benchmarks, new built-in global constraints; cooperation between solvers; definition of a UML profile for graphical components; translation to new solvers. Finally, we plan to make the COMMA system open source for the research community.

References