Syntax Extensions for a Constrained-Object Language via Dynamic Parser Cooperation*

Ricardo Soto[1, 2], Broderick Crawford[1], Eric Monfroy[3], Fernando Paredes[4]

[1] Pontificia Universidad Católica de Valparaíso,
Av. Brasil 2950, Valparaíso, Chile
ricardo.soto@ucv.cl;

[2] Universidad Autónoma de Chile,
Pedro de Valdivia 641, Santiago, Chile
broderick.crawford@ucv.cl

[3] Universidad Técnica Federico Santa María,
Avenida España 1680, Valparaíso, Chile
eric.monfroy@inf.utfsm.cl

[4] Escuela de Ingeniería Industrial, Universidad Diego Portales,
Manuel Rodríguez Sur 415, Santiago, Chile,
fernando.paredes@udp.cl

Abstract: A modern feature of constraint languages is the ability of compiling a model into a set of solver languages. This allows one to model a problem in a single language and to execute it in a set of solver engines. The idea is to facilitate experimentation as well as model sharing. The common architecture to support this task is composed of three layers: an upper layer for the modeling language, a bottom layer for the solver language, and a middle one for performing the mapping process. However, this architecture has an important inconvenience: there is no mechanism for updating the modeling language. This paper addresses this concern by introducing a simple description language for extending the syntax of the modeling language. The goal is to make the architecture adaptable to further upgrades of the solver layer.

Keywords: Constraint Programming, Programming Languages, Modeling Languages.

1. Introduction

Constraint programming is a modern programming paradigm devoted to the efficient resolution of Constraint Satisfaction Problems (CSP). A CSP is a formal problem representation that mainly consists of a sequence of variables holding a domain and a set of constraint over such variables. The goal is to find a variable-value assignment that satisfies the whole set of constraints.

In the past years, several programming languages and libraries have been designed for CP, for instance, ECLiPSe [24], ILOG SOLVER [14], and OZ [16]. In these approaches, a host language is used to state control operations, a constraint language is used for modeling the variables and constraints, and a strategy language may be used to tune the solving process.

The expertise required by CP languages led to the development of modeling languages such as OPL [23]. Here, a higher-level of abstraction is provided. There is no need for dealing with operational concerns of the host language. The user states the model and the system solves it by means of a fixed underlying solver.

A recent concern is to separate the modeling language from the underlying solver [12, 7]. To this end, a three-layered architecture is proposed, including a modeling language, a[1] solver, and a middle tool mapping models (with a high level of abstraction) to executable solver code (with a low-level of abstraction). Among others, this architecture gives the possibility to plug-in new solvers and to process a same model with different solvers.

An important inconvenience of this architecture is the lack of a mechanism for updating the modeling language. For instance, if a new functionality such as a new method, predicate or global constraint is added in the solver, the unique way to use it from the modeling layer is to update the grammar of the modeling language and to recompile it by hand. Likewise, the mapping tool needs to be modified. The translation of the new functionality from the modeling language to the solver language must be included.

In this paper, we present a simple description language to extend the syntax of a modeling language in order to make the architecture

* A shorter version of this paper was also published in the proceedings of the 21st Workshop on (Constraint) Logic Programming with the title "Dynamic Parser Cooperation for Extending a Constrained Object-Based Modeling Language" [18].

adaptable to further upgrades of the solvers. In addition we present an interesting parsing system for an efficient handling of this extension process. The extension language has been designed as part of the s-COMMA system [4], a three-layered architecture for modeling constrained objects (objects subject to constraints on their attributes [17]). In this architecture constraint satisfaction and optimization models [21, 22] can be translated to three native solver models: ECLiPSe [24], Gecode/J [20] and RealPaver [11].

The s-COMMA modeling language is built from a combination of a constraint language and an object-oriented framework. In this work, we will focus on extending just the constraint language of s-COMMA. We see no apparent necessity for extending the object-oriented framework.

The outline of this paper is as follows. In Section 2 we give an overview of the s-COMMA language. Section 3 describes the extension language. The parsing system is presented in Section 4. The process of updating the architecture is described in Section 5, followed by the related work and conclusions.

2. A Tour of the s-COMMA Language

In order to explain the means of extensions and the way to use it, let us first show the components of an s-COMMA model by using the well-known Social Golfers Problem. This problem considers a group of *n* social golfers which play golf once a week, and always in groups of size *g*. The goal is to arrange a schedule for these players for *w* weeks, such that no two golfers play together more than once.

An s-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 or Boolean). 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. There is no need for object constructors to state a class; direct variable assignment can be done in the constraint zone. Figure 1 depicts the data file of this problem. It consists of one enumeration and three constants. The enumeration contains the name of the golfers and the constants hold the size of groups, the number of weeks, and the quantity of groups playing per week.

```
1. enum name := {a,b,c,d,e,f,g,h,i};
2. int s := 3; //size of groups
3. int w := 4; //number of weeks
4. int g := 3; //groups per week
```

Figure 1. Data file of the Social Golfers Problem

The model file is divided into three classes (see Figure 2): one to model the groups, one to model the weeks, and a main class to arrange the schedule of the social golfers. The `Group` class owns the players attribute corresponding to a set of golfers playing together, each golfer being identified by a name given in the enumeration from the data file. In this class, the constraint zone `groupSize` restricts the size of the golfers group. The `Week` class has an array of `Group` objects and the constraint zone `playOncePerWeek` ensures that each golfer takes part of a unique group per week. Finally, the `SocialGolfers` main class has an array of `Week` objects and the constraint zone `differentGroups` states that each golfer never plays two times with the same golfer throughout the considered weeks. For a detailed presentation and additional features of s-COMMA please refer to [4] [19].

```
1. import SocialGolfers.dat;
2.
3. class Group {
4.   name set players;
5.   constraint groupSize {
6.     card(players) = s;
7.   }
8. }
9.
10. class Week {
11.   Group groupSched[g];
12.   constraint playOncePerWeek {
13.     forall(g1 in 1..g, g2 in g1+1..g)
14.       card(groupSched[g1].players
15.       intersect
16.       groupSched[g2].players)= 0;
17.   }
18. }
19.
20. main class SocialGolfers {
21.
22.   Week weekSched[w];
23.
24.   constraint differentGroups {
25.     forall(w1 in 1..w, w2 in w1+1..w)
26.       forall(g1 in 1..g, g2 in 1..g)
27.         card(weekSched[w1].
28.         groupSched[g1].players
29.         intersect
30.         weekSched[w2].
31.         groupSched[g2].players) <= 1;
32.     }
33.   }
34. }
```

Figure 2. Model file of the social golfers problem.

3. Extending s-COMMA

In order to present the extensibility features of s-COMMA, we continue with the social golfers problem. Let us consider that a programmer adds to the solver layer (specifically to Gecode/J) a new global constraint to enforce the $a <_{lex} b$ lexicographic ordering. This constraint operates over a set $a = \{x_0, x_1, \ldots, x_n\}$ and a set $b = \{y_0, y_1, \ldots, y_n\}$ of n integer values, ensuring that: $x_0 < y_0$; $x_1 < y_1$ when $x_0 = y_0$; $x_2 < y_2$ when $x_0 = y_0$ and $x_1 = y_1; \ldots; x_{n-1} < y_{n-1}$ when $x_0 = y_0, x_1 = y_1, \ldots,$ and $x_{n-2} = y_{n-2}$, [8]. The $a <_{lex} b$ constraint will be used to remove the symmetries [10] (eliminate redundant solutions) of the already presented social golfers model. To use this new constraint we can extend the semantics of the s-COMMA constraint language. This can be achieved by defining an extension file where the rules of the translation are stated. Such a file may be composed of one or more main blocks (see Figure 3). Main blocks hold the translation rules and denote the solver to which the mapping must be performed. For instance, the first main block defines the mapping rules for the Gecode/J solver.

```
1. GecodeJ {
2.   Constraint {
3.     lexOrder(a,b)->
4.     "gecodeJLexicalOrdering($a$,$b$);";
5.   }
6. }
7.
8. ECLiPSe {
9.   Constraint {
10. ...
11. }
12. ...
```

Figure 3. Adding constraints to s-COMMA.

Within the `GecodeJ` block, a `Constraint` block has been defined. This block owns the mapping rule of the new constraint to be added. This rule consists of two parts. The left part of the rule defines the statement used to call the new function from the s-COMMA language, and the right part defines the statement used to call the new built-in method from the solver file. In this way, the rule states that `lexorder(a,b)` will be translated to `gecode JLexicalOrdering(a,b)` in the mapping process from s-COMMA to Gecode/J. To facilitate the translation of the input parameters, variables (`a` and `b`) must be tagged with '$' symbols. In the example, the first parameter and the second parameter of the new s-COMMA constraint will be translated as the first parameter and the second parameter of the Gecode/J method call, respectively. The use of the new constraint in the social golfers problem is shown in Figure 4.

```
1. import lexOrderings.ext;
2. ...
3.
4. main class SocialGolfers {
5.
6.  Week weekSched[w];
7.
8.  constraint differentGroups {
9.    forall(w1 in 1..w, w2 in w1+1..w)
10.    forall(g1 in 1..g, g2 in 1..g)
11.     card(weekSched[w1].groupSched[g1].
12.      players intersect weekSched[w2].
13.      groupSched[g2].players) <= 1;
14.  }
15.
16.  constraint removeSymmetries {
17.    forall(w1 in 1..weeks, g1 in 1..
18.    groups-1)
19.     lexOrder(weekSched[w1].
20.     groupSched[g1].players,
21.     weekSched[w1].
22.     groupSched[g1+1].players);
23.
24.    forall(w1 in 1..weeks-1)
25.     lexOrder(weekSched[w1].
26.     groupSched[1].players,
27.     weekSched[w1+1].
28.      groupSched[1].players);
29.  }
30.}
```

Figure 4. Removing symmetries from the social.

4. Adding Functions

To present the usefulness of this feature, let us introduce the Sudoku problem. This problem consists in filling a 9x9 matrix so that each column, each row, and each of the nine 3x3 sub-matrices contains different digits from 1 to 9.

A model for this problem is depicted in Figures 5 and 6. The data file is composed of two constants and a variable assignment. The constant n defines the size of the matrix and `s` the size of the sub-matrices. The variable assignment is used to fill some of the cases of a two-dimensional array called `puzzle`. This array is stated at line 5 of the model file and represents the matrix of the problem. The constraint zones of the model are defined next.

The `differentInRowsAndColumns` constraint zone ensures that every row and column of the matrix contains different values, and `different InSubMatrices`

guarantees that all the 3x3 sub-matrices get different values.

```
1. int s := 3;
2. int n := 9;
3. int Sudoku.puzzle :=
     [[_, _, _, _, _, _, _, _, _],
     [_, 6, 8, 4, _, 1, _, 7, _],
     [_, _, _, _, 8, 5, _, 3, _],
     [_, 2, 6, 8, _, 9, _, 4, _],
     [_, _, 7, _, _, _, 9, _, _],
     [_, 5, _, 1, _, 6, 3, 2, _],
     [_, 4, _, 6, 1, _, _, _, _],
     [_, 3, _, 2, _, 7, 6, 9, _],
     [_, _, _, _, _, _, _, _, _]];
```

Figure 5. Data file for the Sudoku problem.

```
1. import Sudoku.dat;
2.
3. main class Sudoku {
4.
5.   int puzzle[n,n] in [1,n];
6.
7.   constraint differentInRowsAndColumns
{
8.     forall(k in 1..n, i in 1..n, j in
9.     i+1..n) {
10.      puzzle[k,i] != puzzle[k,j];
11.      puzzle[i,k] != puzzle[j,k];
12.    }
13.  }
14.
15.  constraint differentInSubMatrices {
16.    forall(x1 in 1..s, y1 in 1..s, x2
in
17.    1..s) {
18.      forall(y2 in 1..s, x3 in 1..s,
y3
19.      in 1..s) {
20.        if(x2 != x3 and y2 != y3)
21.          puzzle[(x1 - 1) * s + x2,
22.          (y1 - 1) * s + y2] !=
23.          puzzle[(x1 - 1) * s + x3,
24          (y1 - 1) * s + y3];
25.      }
26.    }
27.  }
28.}
```

Figure 6. Model file for the Sudoku problem.

Let us now consider that three new functions operating over two-dimensional arrays are added to Gecode/J: a function to get the rows, another to get the columns and a third one to get sub-matrices. Figure 7 depicts the corresponding extension file. The parameter `mat` corresponds to the matrix on which the function acts, `i` and `j` are the indexes of the row and of the column to be obtained, respectively. The third function has four parameters, the pair (`i1,j1`) represents the coordinates of the upper-left corner of the sub-matrix and the pair (`i2,j2`) represents the lower-right corner of the sub-matrix.

```
1. GecodeJ {
2.   Constraint {
3.     lexOrder(a,b) ->
4.     "gecodeJLexicalOrdering($a$,$b$);";
5.   }
6.   Function {
7.     getRow(mat,i) ->
8.     "gecodeJGetRow($mat$,$i$);";
9.     getColumn(mat,j) ->
10.    "gecodeJGetColumn($mat$,$j$);";
11.    getSubMatrix(mat,i1,i2,j1,j2) ->
12.    "gecodeJGetSubMatrix($mat$,$i1$,
13.    $i2$,$j1$,$j2$);";
14.  }
15.}
16.....
```

Figure 7. Adding new functions.

The resulting model using these new functions is depicted in Figure 8. Here, we can see that the model has been defined in a more concise and elegant way. In addition, the use of the alldifferent [1] constraint will improve the resolution process of the problem.

```
1. main class Sudoku {
2.
3.   int puzzle[n,n] in [1,n];
4.
5.   constraint differentInRowsAndColumns {
6.     forall(i in 1..n) {
7.       alldifferent(getColumn(puzzle, i));
8.       alldifferent(getRow(puzzle, i));
9.     }
10.  }
11.
12.  constraint differentInSubMatrices {
13.    forall(i in 1..s, j in 1..s)
14.      alldifferent(getSubMatrix(puzzle,
15.      (i-1)*s + 1,i*s,(j-1)*s + 1,j*s));
16.  }
17. }
```

Figure 8. Using the new functions in the Sudoku problem.

5. Dynamic Parser Cooperation

The s-COMMA system is written in Java and the ANTLR [13] tool has been used for generate lexers, parsers and tree walkers. The system is supported by a three layered architecture: a modeling layer, a mapping layer and a solving layer (see Figure 10). The compiling system is composed by three compilers. One for the s-COMMA language, one for the data and another for the extension files. This system is the basis of the mechanism to extend the constraint language.

The s-COMMA compiler (see Figure 11) 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).

In order to get the abstract syntax tree (AST) from the parsing process, several cooperations between those parsers are performed at running time. A control engine manages this cooperation by sending each line of the s-COMMA model to the correct parser. Lines are syntactically checked by the parser and then transformed into an AST which is returned to the control engine. The control engine takes this AST and attaches it to the AST of previously parsed lines. Let us clarify this process by means of a simple example.

```
class Coop {
  int a;
  real b;
}
```

Figure 9. Attributes from different domains

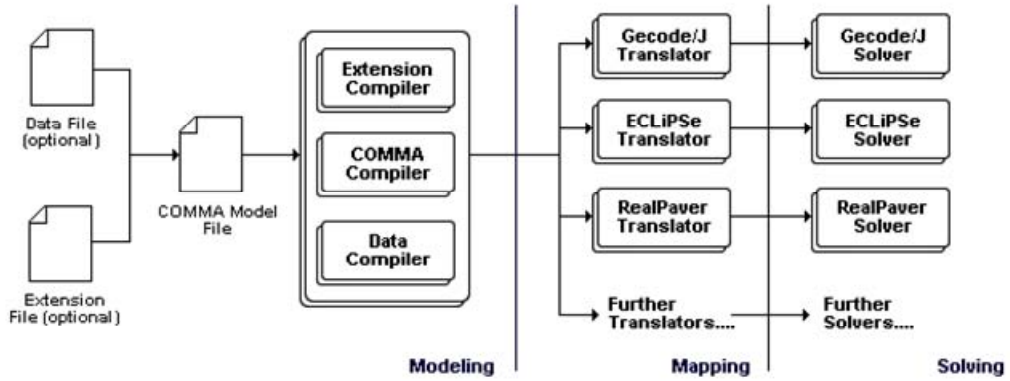

Figure 10. The s-COMMA Architecture

Figure 9 depicts an s-COMMA class called `Coop` which has attributes from different domains. The parsing process of this file is as follows: At line `1` the class is declared. This is part of the base language, so the base parser builds the corresponding AST for this line. Then, the control engine detects at line `2` an `int` type, this line is sent to the integer parser which builds the corresponding AST. The control engine takes this AST and appends it at the end of the previous AST. Once the AST of the model is complete, a semantic checking is performed by means of two tree walkers which check types, class names, variable names, inheritances and compositions. Then, the AST is transformed into a Java object storing the model information, data information and extensions information using efficient representations. Finally, this Java object is translated to the executable solver file.

The independence of parsers has been done for three reasons. (1) It gives us the adequate modularity to easily maintain the parsing engine. (2) It avoids us to recompile 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.

6. Updating the Architecture

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 (See Figure 12).

The control engine adds the new tokens to a table of symbols. The rules of translation are stored in a XML file. This file is used to perform the translation of the new functionalities, from the s-COMMA language to the solver file. The control engine manages ambiguities by checking the new tokens of the extension with the existing tokens in the symbol table.

7. Related Work

Extensibility has been studied widely in the area of programming languages. Language extensions can define new language constructs, new semantics, new types and translations to the target language. Many techniques support each of these extensions. Some examples are syntactic exposures [2], hygienic macro expansions [5] and adaptable grammars [3].

exactly as we use in our extension language: The initial terms are transformed into the target language terms (initial-terms -> target-language-terms). As far as we know, this is the first attempt to make syntax-extensible a modeling language for constraint-based problems.

8. Conclusion and Future Work

We have shown by means of a practical example how the constraint language of s-COMMA can be extended using a simple description language. The process of extending the constraint language is handled by a control engine and a set of independent parsers. The parser independence provides us the adequate modularity to avoid recompiling the whole language each time a new extension is added. This leads to a faster extension process since just the updated domain is recompiled.

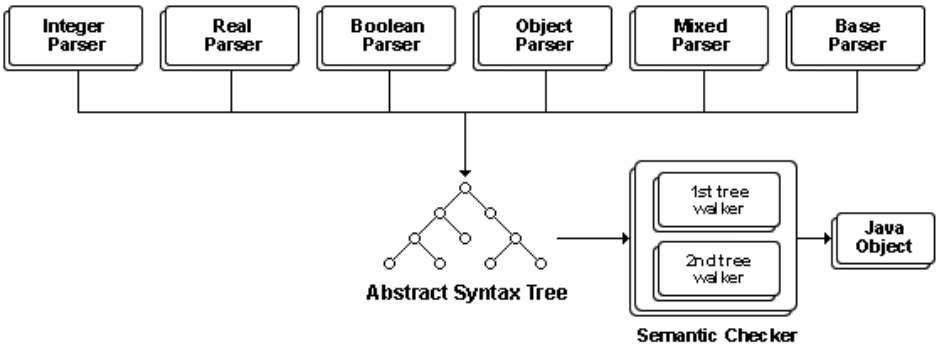

Figure 11. The compiling process

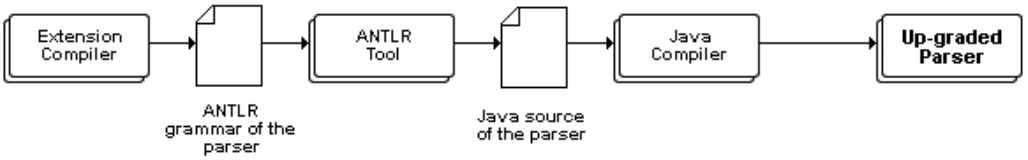

Figure 12. The extension process

These approaches are in general dedicated to extend the whole syntax of a language; consequently they provide a bigger framework than we need. For this reason we have chosen term rewriting [6] as the base of our extension language. This technique allows one to define a clear correspondence between two sets of terms,

The work done in this extension language may be improved adding customizable functions.

For instance, functions for which the priority and the notation (prefix, infix, postfix) can be defined, such as math operators (+,*,-,/). An extension manager may be useful to control which functionalities could be eliminated or maintained.

REFERENCES

1. BESSIÈRE, C., E. HEBRARD, B. HNICH, T. WALSH, **The Complexity of Global Constraints**. In proceedings of AAAI, 2004, pp. 112-117.
2. CHIBA, S., **A Metaobject Protocol for C++**. In proceedings of OOPSLA, 1995, pp. 285–299.
3. CHRISTIANSEN, H., **A Survey of Adaptable Grammars**. SIGPLAN Notices, vol. 25, no.11, 1990, pp. 35–44.
4. CHENOUARD, R., L. GRANVILLIERS, R. SOTO, **Model-driven Constraint Programming**. In proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), ACM Press, 2008, pp. 236-246.
5. CLINGER, W., J. REES, **Macros that Work**. In proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL), 1991, pp. 155–162.
6. DERSHOWITZ, N., J. JOUANNAUD, **Rewrite Systems**. In Handbook of Theoretical Computer Science, vol. B: Formal Models and Sematics (B), 1990, pp. 243–320.
7. FRISCH, A., W. HARVEY, C. JEFFERSON, B. MARTÍNEZ-HERNÁNDEZ, I. MIGUEL, **Essence: A Constraint Language for Specifying Combinatorial Problems**. Constraints, vol. 13, no. 3, 2008, pp. 268-306.
8. FRISCH, A., B. HNICH, Z. KIZILTAN, I. MIGUEL, T. WALSH, **Global Constraints for Lexicographic Orderings**. In proceedings of the 8th International Conference of Principles and Practice of Constraint Programming (CP), vol. 2470 of LNCS, 2002, pp. 93-108.
9. GAMBINI, I., **Quant aux carrés carrelés**. PhD thesis, L'Université de la Méditerranée aix-Marseille II, 1999.
10. GENT, I., B. SMITH, **Symmetry Breaking in Constraint Programming**. In proceedings of 14th European Conference on Artificial Intelligence (ECAI), 2000, pp. 599-603.
11. GRANVILLIERS, L., F. BENHAMOU, **Algorithm 852: Realpaver: An Interval Solver using Constraint Satisfaction Techniques**. ACM Trans. Math. Softw., vol. 32, no. 1, 2006, pp. 138–156.
12. MARRIOTT, K., N. NETHERCOTE, R. RAFEH, P. J. STUCKEY, M. GARCIA DE LA BANDA, M. WALLACE, **The Design of the Zinc Modelling Language**. Constraints, vol. 13, no. 3, 2008, pp. 229-267.
13. PARR, T., K. FISHER, **LL(*): The Foundation of the ANTLR Parser Generator**. In proceedings of the 32th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM Press, 2011, pp. 425-436.
14. PUGET, J. F., **A C++ Implementation of CLP**. In proceedings of SCIS, Singapore, 1994.
15. RAFEH, R., M. GARCÍA DE LA BANDA, K. MARRIOTT, M. WALLACE, **From Zinc to Design Model**. In proceedings of 9th International Symposium on Practical Aspects of Declarative Languages (PADL), vol. 4354 of LNCS, 2007, pp. 215–229.
16. SMOLKA, G., **The Oz Programming Model**. In Computer Science Today, vol. 1000 of LNCS, 1995, pp. 324–343.
17. SOTO, R., L. GRANVILLIERS, **The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models**. In proceedings of the 19th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE Computer Society, 2007, pp. 243-250.
18. SOTO, R., L. GRANVILLIERS, **Dynamic Parser Cooperation for Extending a Constrained Object-Based Modeling Language.** In proceedings of the 21st Workshop on (Constraint) Logic Programming (WLP), Technical Report 434, University of Würzburg, 2007, pp. 70-78.
19. SOTO, R., **Controlling Search in Constrained-Object Models**. In proceedings of the 12th Ibero-American Conference on Artificial Intelligence (IBERAMIA), vol. 6433 of LNAI, 2010, pp. 582-591.

20. SCHULTE, C., G. TACK, **Perfect Derived Propagators.** In proceedings of International Conference of Principles and Practice of Constraint Programming (CP), vol. 5202 of LNCS, 2008, pp. 571-575.

21. TALMACIU, M., E. NECHITA, **Some Combinatorial Optimization Problems for Weak-Bisplit Graphs.** Studies in Informatics and Control, ICI Publishing House, vol. 19, no. 4, 2010, pp. 427-434.

22. TANGOUR, F., P. BORNE, **Presentation of Some Metaheuristics for the Optimization of Complex Systems.** Studies in Informatics and Control, ICI Publishing House, vol. 17, no. 2, 2008, pp. 169-180.

23. VAN HENTENRYCK, P., **The OPL Optimization Programming Language**. The MIT Press, 1999.

24. WALLACE, M., S. NOVELLO, J. SCHIMPF, **Eclipse: A Platform for Constraint Logic Programming**, Technical Report IC-Parc, Imperial College, 1997.