22

On the Definition of a Standard Language for Modelling Constraint Satisfaction Problems

Ricardo Soto[1,2], Laurent Granvilliers[1]
[1]*CNRS, LINA, Université de Nantes*
[2]*Escuela de Ingeniería Informática,*
Pontificia Universidad Católica de Valparaíso,
Chile

1. Introduction

A Constraint Satisfaction Problem (CSP) is a declarative representation of a system under constraints. Such a representation is mainly composed by two parts: a sequence of variables lying in a domain and a finite set of constraints over these variables. The goal is to find values for the variables in order to satisfy the whole set of constraints.

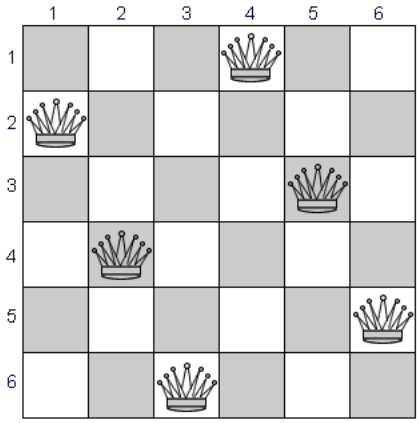


Fig. 1. A solution of the 6-queens problem.

As an example, let us consider the 6-queens problem, which consists in placing 6 chess queens on a 6x6 chessboard such that none of them is able to capture any other using the standard chess queen's moves. A solution requires that no two queens share the same row, column, or diagonal. A CSP for this problem can be stated by means of six variables and three constraints. One variable for each of the six queens to represent their row positions on the chessboard, each variable lying in the domain [1,6] (6 rows). One constraint to avoid that two queens are placed in the same row. One constraint to avoid that two queens are placed in the first diagonal; and finally one constraint to avoid that two queens are placed in the second diagonal. A solution of the 6-queens problem is shown in Figure 1.

Constraint Programming (CP) is called the software technology to solve CSPs. Currently, several CP tools exist which are mainly systems or libraries built on top of a host language such as Prolog (ECLiPSe (Wallace et al., 1997)), C++ (ILOG Solver (Puget, 1994)) or Java (Gecode/J (Schulte & Stuckey, 2004)). These tools, generically named solvers, are able to take as input the CSP and to solve it by exploring and pruning efficiently the search space containing the potential solutions. A main advantage of this approach is that users just need to state a declarative representation of the problem, instead of building complex procedures or algorithms to search the solutions. This is a great asset; however, the language to express these problems is not standard, each tool provides its own semantics with a level of abstraction tied to its host language, which is commonly not easy to use. This is a big concern in the CP field, since users need to deal with the encoding concerns of the solver host language; and moreover to learn a new language each time they want to experiment with a new solver.

In response to this, the CP community has defined the development of a standard language as an important research direction. To this end, a solver-independent three layered architecture has been proposed (Rafeh et al., 2007; Fritsh et al., 2007), including a modelling language -which is expected to be simple enough for use by non CP experts-, a set of solvers and a middle tool mapping models to executable solver code. Thus, the user is able to design one model in a "human-comprehensible" language and to target many solvers.

In this work, we follow this research direction by proposing a solver-independent modelling language but using an object-oriented approach. Our system is called s-COMMA (Soto & Granvilliers, 2007) and it can be regarded as a hybrid built from a combination of an object-oriented language and a constraint language. The s-COMMA constraint language provides typical data structures, control operations, and first-order logic to define constraint-based formulas. The object-oriented part is a simplification of the Java programming style. This framework clearly provides model structures using composition relationships.

The s-COMMA system is written in Java (22000 lines) and it is supported by a solver-independent execution platform where models can be solved by four well-known CP solvers: Gecode/J, ECLiPSe, RealPaver (Granvilliers & Benhamou, 2006) and GNU Prolog (Diaz & Codognet, 2000). We believe s-COMMA is in compliance with the requirements of a standard language. Their simplicity is similar to the state-of-the-art modelling languages (Nethercote et al., 2007; Fritsh et al., 2007). The expressiveness provided is considerable and even it can be increased with extension mechanisms. The solver-independence is the base of the platform which allows experimentations with many solvers.

The definition of a standard language is an evident hard task which may require many years and several experimental steps. We believe that the work done on s-COMMA is one of the steps towards the achievement of this important goal.

The outline of this chapter is as follows. Section 2 introduces an overview of the s-COMMA language. The architecture and implementation of the system is explained in Section 3. The related work is presented in Section 4 followed by the conclusions.

2. s-COMMA overview

In this section we give an overview of s-COMMA. We will first present the most important elements of an s-COMMA model and then, we will illustrate these elements by means of three examples, the n-queens problem, the packing squares problem and a production-optimization problem.

2.1 s-COMMA models

An s-COMMA model is composed by two main parts, a model file and a data file. The model file describes the structure of the problem, and the data file contains the constant values used by the model. The model file is composed by import statements and classes; and the data file is composed by constants and variable assignments.

2.1.1 Constants & variable assignments

Constants, namely data variables, are stated in a separate data file and imported from the model file. Constants can be real, integer or enumeration types. Arrays of one dimension and arrays of two dimensions of data variables are allowed. A variable-assignment is an assignment of a value to a variable of an object defined in the model file (as example, see line 3 of the data file in Fig. 4).

2.1.2 Classes

A class is composed by attributes and constraints zones. Single inheritance is permitted and a subclass inherits all attributes and constraints of its superclass.

2.1.3 Attributes

Attributes may represent decision variables or objects. Decision variables must be declared with an integer, real or boolean type. Objects are instances of classes which must be typed with their class name. Arrays of one and two dimensions can be used; they can contain either decision variables or objects. Decision variables and arrays of decision variables can be constrained to a determined domain.

2.1.3 Constraint zones

Constraint zones are used to group constraints encapsulating them inside a class. A constraint zone is stated with a name and it can contain constraints, forall loops, if-else statements, optimization statements, and global constraints.

2.2 The n-queens problem

Let us begin the illustration of these elements by means of the n-queens problem presented in Section 1. An s-COMMA model for this problem is shown in Figure 2. The model is represented by a class called `Queens` which contains an array with n integer decision variables lying in the domain `[1,n]`. The constant value called `n` is imported from the `Queens.dat` file.

At line 6 a constraint zone called `noAttack` contains the three constraints required. One constraint to avoid that two queens are placed in the same row (line 9). One constraint to avoid that two queens are placed in the first diagonal (line 10); and one constraint to avoid that two queens are placed in the second diagonal (line 11). Two for loops ensure that the constraints are applied for the complete set of decision variables.

2.3 The packing squares problem

Let us continue with a more complex problem called packing squares. 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 3 shows a solution for the problem, 8 squares have been placed in a square area of side size 5.

```
1.  //Data file
2.  n:=6;

1.  //Model file
2.  import Queens.dat;
3.
4.  class Queens {
5.    int q[n] in [1,n];
6.    constraint noAttack {
7.      forall(i in 1..n) {
8.        forall(j in i+1..n) {
9.          q[i] <> q[j];
10.         q[i]+i <> q[j]+j;
11.         q[i]-i <> q[j]-j;
12.       }
13.     }
14.   }
15. }
```

Fig. 2. s-COMMA model for the n-queens problem.

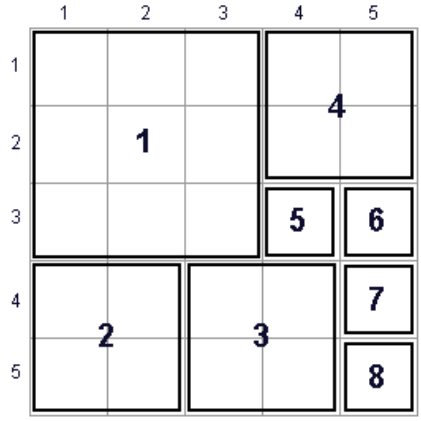


Fig. 3. A solution of the packing squares problem.

Figure 4 shows an s-COMMA model for the packing squares problem. Data values are imported from an external data file called `PackingSquares.dat`, `sideSize` (line 1) is an integer constant that represents the side size of the square area where squares must be placed, `squares` (line 2) represents the quantity of squares to place. `PackingSquares.s` (line 3) is a variable-assignment for the array of `Square` objects declared at line 5 of the model file, here a set of values is assigned to the third attribute (`size`) of each `Square` object of the array `s`. For instance, the value `3` is assigned to the attribute `size` of the first object of the array. The value `2` is assigned to the attribute `size` of the second, third and fourth object of the array. The value `1` is assigned to the attribute `size` of remaining objects of the array. We use standard modelling notation ('_') to omit assignments.

At line 3 in the model file, the definition of the class begins, `PackingSquares` is the name given to this class. Then, an array containing objects from the class `Square` is defined. This class, declared at line 30, is used to model the set of squares. Attributes `x` and `y` represent

respectively the x and y coordinates where the squares must be placed. So, `s[8].x=5` and `s[8].y=5` means that the eighth square must be placed in row 5 and column 5, indeed in the bottom right corner of the square area. Both variables (x,y) are constrained, they must have values into the domain `[1,sideSize]`. The last attribute called `size` represents the size of the square.

```
//Data file
1.  int sideSize :=5;
2.  int squares :=8;
3.  Square PerfectSquares.s := [{_,_,3},{_,_,2},{_,_,2},{_,_,2},
                                {_,_,1},{_,_,1},{_,_,1},{_,_,1}];


//Model file
1.  import PackingSquares.dat;
2.
3.  class PackingSquares {
4.
5.    Square s[squares];
6.
7.    constraint inside {
8.      forall(i in 1..squares) {
9.        s[i].x <= sideSize - s[i].size + 1;
10.       s[i].y <= sideSize - s[i].size + 1;
11.     }
12.   }
13.
14.   constraint noOverlap {
15.     forall(i in 1..squares) {
16.       forall(j in i+1..squares) {
17.         s[i].x + s[i].size <= s[j].x or
18.         s[j].x + s[j].size <= s[i].x or
19.         s[i].y + s[i].size <= s[j].y or
20.         s[j].y + s[j].size <= s[i].y;
21.       }
22.     }
23.   }
24.
25.   constraint fitArea {
26.     sum(i in 1..squares)(s[i].size*s[i].size)) = sideSize*sideSize;
27.   }
28. }
29.
30. class Square {
31.   int x in [1,sideSize];
32.   int y in [1,sideSize];
33.   int size;
34. }
```

Fig. 4. s-COMMA model for the packing 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. Let us note that loops use loop-variables which do not need to be declared (`i` and `j` in the example).

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

2.4 The production problem

Let us finish the s-COMMA overview with a production-optimization problem. Consider a factory that must satisfy a determined demand of products. These products can be either manufactured inside the factory -considering a limited resource availability- or purchased from an external market. The goal is to determine the quantity of products that must be produced inside the factory and the quantity to be purchased in order to minimize the total cost.

Figure 5 shows an s-COMMA model for this problem. At Line 28 of the model, the class to represent products is stated. Each `Product` is composed by its demand, its inside and outside cost, its consumption, and the quantity that must be produced inside and outside the factory. At line 3 the main class of the problem begins, it is first composed by two arrays, one containing the set of products and the other contains the available quantity of resources for manufacturing the products.

At Line 9 a constraint zone called `noExceedCapacity` is stated to ensure that the resource consumption of products manufactured inside do not exceed the total quantity of available resources. At line 15, `satisfyDemand` is posted to satisfy the demand of all the products. Finally, at line 22, an optimization statement is posted to determine the quantity of products that must be produced inside the factory and the quantity to be purchased in order to minimize the total cost.

The data file is composed by two enumerations that define the resources and the name's products respectively. At line 3, a variable-assignment for the `capacity` attribute of the class `Production` is stated. At the end, `Production.products` is a variable-assignment for the array `products` defined at line 5 of the model file. This variable-assignment states that the demand of the product klusky is 1000, their inside and outside cost are 6 and 8, respectively; and finally its production requires 5 flour items and 2 eggs. The assignment of the following products is analogous.

```
//Data file
1.  enum resourceList := {flour, eggs};
2.  enum productList  := {klusky, capellini, fettucine};
3.  int Production.capacity := [200,400];
4.  Product Production.products :=
         [klusky:{1000,6,8,[flour:5,eggs:2],_,_},

       capellini:{2000,2,9,[flour:4,eggs:4],_,_},

       fettucine:{3000,3,4,[flour:3,eggs:6],_,_}];


//Model File
1.  import Production.dat;
2.
3.  class Production {
4.
5.    Product products[productList];
6.    int capacity[resources];
```

```
7.
8.    constraint noExceedCapacity {
9.      forall(r in resourceList) {
10.       capacity[r] >= sum(p in productList)
11.         (products[p].consumption[r] * products[p].inside);
12.     }
13.   }
14.
15.   constraint satisfyDemand {
16.     forall(p in productList) {
17.       products[p].inside + products[p].outside >= products[p].demand;
18.     }
19.   }
20.
21.   constraint minimizeCost {
22.     [minimize] sum(p in productList)
23.          (products[p].insideCost  * products[p].inside +
24.           products[p].outsideCost * products[p].outside);
25.   }
26. }
27.
28. class Product {
29.   int demand;
30.   int insideCost;
31.   int outsideCost;
32.   int consumption[resourceList];
33.   int inside in [0,5000];
34.   int outside in [0,5000];
35. }
```

Fig. 5. s-COMMA model for the production problem.

2.5 Extension mechanism

Extensibility is an important feature of s-COMMA. Let us now show this feature using the packing squares problem. Consider that a programmer adds to the Gecode/J solver two new built-in functionalities: a constraint called `inside` and a function called `pow`. The constraint `inside` ensures that a square is placed inside a given area, and `pow(x,y)` calculates the value of `x` to the power of `y`. In order to use these functionalities we can use these new built-ins from s-COMMA by defining an extension file where the rules of the translation are described. This file is composed by one or more main blocks (see Figure 6). A main block defines the solver where the new functionalities will be defined. Inside a main block two new blocks are defined: a `Function` block and a `Relation` block. In the `Function` block we define the new functions to add. The grammar of the rule is as follows:

$$\langle name\rangle(\langle input-parameters\rangle) \rightarrow "\langle solver-code\rangle";$$

In the example, the left part of the rule is `pow(x,y)`, `pow` is the name of the function and `x` and `y` the input parameters. The left part of the rule corresponds to the statement that will be used to call the new function from s-COMMA. The right part corresponds to the code that calls the new built-in method from the solver file. Thus, the code `pow(x,y)` will be translated to `power(x,y)` from s-COMMA to Gecode/J. The translator must recognize the correspondence between input parameters in s-COMMA and input parameters in the solver code. Therefore, variables are tagged with '$' symbols. In the example, the first parameter

and the second parameter of the s-COMMA function will be translated as the first parameter and the second parameter in the Gecode/J function, respectively.
Within the `Relation` block we define the new constraints to add. In the example, a new constraint called `inside` is defined, it receives four parameters. The translation to Gecode/J is given in the same way. Once the extension file is completed, it can be called by means of an import statement. The resultant s-COMMA model using extensions is shown in Figure 6.

```
//Extension File
1.  GecodeJ {
2.    Function {
3.      pow(x,y) -> "power($x$,$y$)";
4.    }
5.    Relation {
6.      inside(a,b,c,d) -> "inside($a$,$b$,$c$,$d$);";
7.    }
8.  }
9.
10.  ECLiPSe {
11.    Function {
12.   ...
13.  GNUProlog {
14.    Function {
15.   ...
16.  RealPaver {
17.    Function {

//Model file
1. import PackingSquares.dat;
2. import PackingSquares.ext;
3.
4. class PackingSquares {
5.
6.    Square s[squares];
7.
8.    constraint placeSquares {
9.      forall(i in 1..squares) {
10.       inside(s[i].x,s[i].y,s[i].size,sideSize);
11.       forall(j in i+1..squares) {
12.         s[i].x + s[i].size <= s[j].x or
13.         s[j].x + s[j].size <= s[i].x or
14.         s[i].y + s[i].size <= s[j].y or
15.         s[j].y + s[j].size <= s[i].y;
16.       }
17.     }
18.   }
19.
20.   constraint fitArea {
21.        (sum(i in 1..squares) (pow(s[i].size,2)) = pow(sideSize,2);
22.   }
23.}
```

Fig. 6. s-COMMA model for the packing squares problem using extensions.

3. s-COMMA architecture

The s-COMMA system is supported by a three-layered architecture: Modelling, Mapping and Solving (see Fig 7). On the first layer, models are stated; extension and data files can be

given optionally. Models are syntactically and semantically checked. If the checking process succeeds, an intermediate model called flat s-COMMA is generated, the aim of this model is to simplify the task of translators. Finally, the flat s-COMMA file is taken by the selected translator which generates the executable solver file.

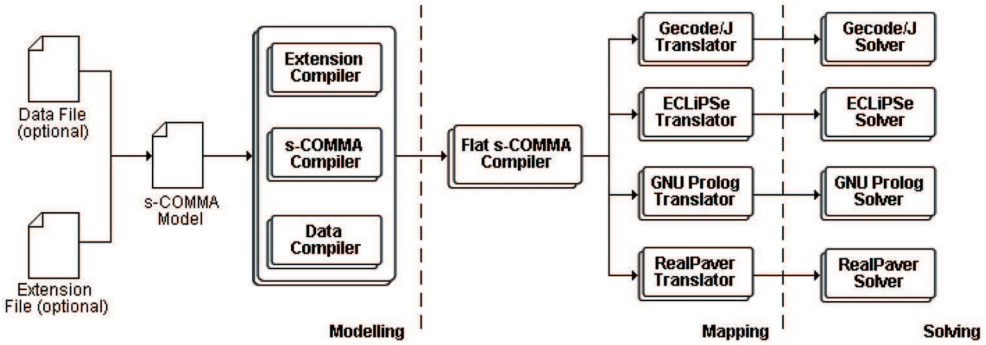


Fig. 7. s-COMMA architecture.

3.1 From s-COMMA to flat s-COMMA

A direct translation from s-COMMA to executable solver code is feasible (in fact, we have studied this in (Soto & Granvilliers, 2007)). However, many statements provided by s-COMMA are not supported by solvers. Thus, for performing this direct mapping, many model-transformations must be carried out at the level of translators. This makes translators bigger (in terms of code lines) and, as consequence, difficult to develop and maintain. A well-known technique to simplify code generation is to include an intermediate phase where the non-supported features are transformed to simpler (or supported) features. We state this transformation on an intermediate model called flat s-COMMA. The set of performed transformations from s-COMMA to flat s-COMMA are described below.

Flattening composition. The hierarchy generated by composition is flattened. This process is done by expanding each object declared in the main class adding its attributes and constraints in the flat s-COMMA 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.

Loop unrolling. Loops are not widely supported by solvers, hence we generate an unrolled version of loops.

Enumeration substitution. In general solvers do not support non-numeric types. So, enumerations are replaced by integer values, original values are stored to give the results.

Data substitution. Data variables are replaced by its value defined in the data file.

Conditional removal. Conditional statements are transformed to logical formulas. For instance, `if a then b else c` is replaced by $(a \Rightarrow b) \wedge (a \vee c)$.

Logic formulas transformation. Some logic operators are not supported by solvers. For example logical equivalence $a \Leftrightarrow b$ and reverse implication $a \Leftarrow b$. We transform logical equivalence expressing it in terms of logical implication. Reverse implication is simply inverted $b \Rightarrow a$.

Finally, the generated flat s-COMMA code is taken by the selected translator which generates the executable solver file.

4. Related work

s-COMMA is closely related to recent standard modelling language proposals as well as object-oriented languages for modelling constraint problems.

4.1 The definition of a standard modelling language

The definition of a standard modelling language for CP is a recent trend. First encouragements on this issue were done by J-F. Puget in (Puget, 2004). He suggested to develop a ``model and run'' paradigm such as in Math Programming. The paradigm involved a standard file format for expressing models and a CP library to solve them. Then, at *The Next 10 Years of CP* (Benhamou et al., 2007), this challenge was confirmed as an important research direction. Recently, at CP 2007 Conference, MiniZinc (Nethercote et al., 2007) was proposed as a standard modelling language. MiniZinc can be seen as a subset of elements provided by Zinc (Rafeh et al., 2007). The syntax is closely related to OPL (Van Hentenryck, 1999) and its solver-independent platform allows translating MiniZinc models into Gecode and ECLiPSe solver models. Essence (Fritsch et al., 2007) is another good basis to define such a standard. This core is focused on users with a background in discrete mathematics; this style makes Essence a specification language rather than a modelling language. The Essence execution platform allows mapping specifications into the ECLiPSe solver.

We believe s-COMMA may be a good starting point too, the constraint language of s-COMMA is closely related to OPL and Zinc. The solver-independent platform is an adequate support to map models to four different solvers. The object-oriented framework and the extensibility are also important features not present in the aforementioned proposals.

4.2 Objects + Constraints

The first attempt in combining an object oriented language with a constraint language was on the development of ThingLab (Borning, 1981) which was built for interactive graphical simulation. A next version of this approach was developed in the Kaleidoscope language (Freeman-Benson, 1992). Then, similar ideas were developed in Gianna (Paltrinieri, 1994) for modelling constraint-based problems with objects in a visual environment. COB (Bharat & Tambay, 2002) is a more recent approach for the engineering field, the language is a combination of objects first order formulas and CLP (Constraint Logic Programming) predicates. Modelica (Fritzson & Engelson, 1998) is another object-oriented system for modelling engineering problems, but it is mostly oriented towards simulation. In general, the constraint language of these approaches was designed to specific application domains. They also lack of extensibility and solver-independence.

5. Conclusion

In this chapter we have presented s-COMMA, a new language for modelling CSPs. The basis of s-COMMA is a combination of an object-oriented language with an extensible constraint language which allows users to state modular models using the benefits of the object-oriented style. s-COMMA is also supported by a three layered architecture where models can be mapped to four well-known solvers. We believe that these capabilities make s-COMMA a good basis to define a standard modelling language.

To reach this final purpose, several aspects could be developed, for instance: more work on benchmarks, solver cooperation, new global constraints and translation to new solvers. The development of a tool for modelling CSPs through a UML-like language will be useful too.

6. References

Benhamou, F.; Jussien, N. & O'Sullivan, B. (2007). Trends in Constraint Programming. ISTE, ISBN: 9781905209972, England.

Bharat, J. & Tambay, P. (2002). Modelling Engineering Structures with Constrained Objects. Proceedings *of Principles and Practice of Declarative Languages (PADL)*, pp. 28-46, LNCS Springer-Verlag, Portland, USA.

Borning, A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. ACM Transactions on Programming Languages and Systems., 3(4): pp. 353-387, ISSN:0164-0925.

Schulte, C. & Stuckey, P. (2004). Speeding Up Constraint Propagation. *Proceedings of Principles and Practice of Constraint Programming (CP)*, pp. 619-633, LNCS Springer-Verlag, Toronto, Canada.

Diaz, D. & Codognet, P. (2000). The gnu prolog system and its implementation, *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 728–732, ACM Press, Villa Olmo, Italy.

Frisch, A.; Grum, M.; Jefferson, C.; Martínez, B. & Miguel, I. (2007). The design of Essence: A constraint language for specifying combinatorial problems, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 80–87, Hyderabad, India.

Freeman-Benson, B. (1990). Kaleidoscope: Mixing Objects, Constraints and Imperative. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pp. 77-88, SIGPLAN Notices 25(10), Ottawa, Canada.

Fritzson, P. & Engelson, V. (1998). Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pp. 67-90, LNCS Springer-Verlag, Brussels, Belgium.

Granvilliers, L & Benhamou, F. (2006). Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software., 32(1):pp.138-156, ISSN:0098-3500.

Nethercote, N.; Stuckey, P.; Becket, R.; Brand, S.; Duck, G. & Tack, G. (2007). Minizinc: Towards a standard cp modelling language. *Proceedings of Principles and Practice of Constraint Programming (CP)*, pp. 529–543, LNCS Springer-Verlag, Providence, USA.

Paltrinieri, M. (1995). A Visual Constraint-Programming Environment. *Proceedings of Principles and Practice of Constraint Programming (CP)*, pp. 499–514, LNCS Springer-Verlag, Cassis, France.

Puget, J. (1994). A C++ implementation of CLP. *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore.

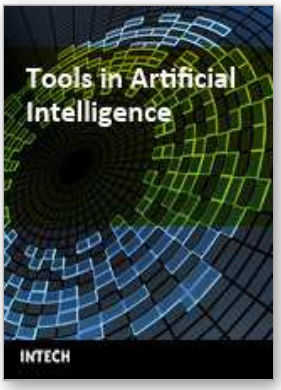
Puget, J. (2004). Constraint programming next challenge: Simplicity of use. *Proceedings of Principles and Practice of Constraint Programming (CP)*, pp. 5–8, LNCS Springer-Verlag, Toronto, Canada.

Rafeh, R.; García de la Banda, M.; Marriott, K. & Wallace, M. (2007). From zinc to design model. *Proceedings of Principles and Practice of Declarative Languages (PADL)*, pp. 215–229, LNCS Springer-Verlag, Nice, France.

Soto, R. & Granvilliers, L. (2007). The Design of COMMA: An extensible framework for mapping constrained objects to native solver models. *Proceedings of the International Conference on Tools with Artificial Intelligence* (ICTAI), pp. 243–250, IEEE Computer Society, Patras, Greece.

Van Hentenryck, P. (1999). Constraint Programming in OPL. *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pp. 98-116, ACM Press, Paris, France.

Wallace, M.; Novello, S. & Schimpf, J. (1997). Eclipse: A platform for constraint logic programming, Technical report, IC-Parc, Imperial College, London, England.



Tools in Artificial Intelligence

Edited by Paula Fritzsche

ISBN 978-953-7619-03-9
Hard cover, 488 pages
Publisher InTech
Published online 01, August, 2008
Published in print edition August, 2008

This book offers in 27 chapters a collection of all the technical aspects of specifying, developing, and evaluating the theoretical underpinnings and applied mechanisms of AI tools. Topics covered include neural networks, fuzzy controls, decision trees, rule-based systems, data mining, genetic algorithm and agent systems, among many others. The goal of this book is to show some potential applications and give a partial picture of the current state-of-the-art of AI. Also, it is useful to inspire some future research ideas by identifying potential research directions. It is dedicated to students, researchers and practitioners in this area or in related fields.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821