# A CLP engine for a general purpose configuration tool

Andrea Calligaris[1], Dario Campagna[2], Christian De Rosa[3], Agostino Dovier[1], Angelo Montanari[1], and Carla Piazza[1]

[1] Dept. of Mathematics and Computer Science University of Udine
(dovier|montana|piazza)@dimi.uniud.it
[2] Dept. of Mathematics and Computer Science University of Perugia
dario.campagna@dipmat.unipg.it
[3] Acritas s.r.l., Martignacco (UD), Italy derosa@acritas.it

**Abstract.** Product configurators are an emerging software technology, developed for helping companies in deploying mass customization strategies. Among the various approaches, constraint-based techniques look particularly encouraging, for both their modeling capabilities and their efficiency. In this paper, we present our work on a product configurator based on Constraint Logic Programming. The system we propose integrates an existing configurator module with a reasoning engine written in SICStus Prolog.

## 1 Introduction

The current competitive pressure leads many companies to offer to customers an increasing product variety. Such a flexibility, however, results into an increase in companies' costs, because of the high complexity introduced by product variety in production processes.

In current markets, customers ask for products closer to their needs, preserving high quality, short delivery times, and affordable costs. The attempt at overcoming the trade-off between costs and delivery times, on the one hand, and product variety, on the other hand, is commonly called *mass customization*. To operate according to mass customization means to sell products satisfying customer's needs, preserving as much as possible the advantages of *mass production* in terms of efficiency and productivity. This operating mode involves a series of difficulties that companies that want to adopt it struggle to resolve using traditional software tools, designed for repetitive productions.

A few years ago, software systems designed for supporting the order cycle of products realized in many variants, both standard and customizable, appeared on the market. These systems are called *software product configurators* and can be used by companies to successfully operate

according to mass customization. A software product configurator is a software tool that allows one to effectively and efficiently deal with problems related to product customization, automatically guaranteeing the satisfaction of all requirements. In addition, it must allow one to determine realization times and costs, generating a detailed plan about all the production process phases, from the order of raw materials to the delivery of the final product. A product configurator can thus be viewed as a sophisticated expert system and the need to study formalisms and to develop algorithms that make it possible to realize efficient and versatile configurators becomes apparent.

The configuration problem has recently attracted a significant amount of attention not only from the application point of view, but also from the methodological one [13]. In particular, significant approaches based on computational logic [6,14] and constraint programming [9,4,12] have emerged. In this paper, we describe the main achievements of our research on product configuration based on Constraint Logic Programming (CLP). We focused our attention on the development of a new CLP-based (product) configuration engine, called MCE, for the existing commercial configuration platform Morphos. In the proposed framework, the description of a product consists of a tree, whose nodes represent the components of a product, and a set of rules. Each node is paired with a set of properties, which express configurable characteristics of the component it represents. Each property is endowed with a finite domain (typically, a finite set of integers or strings), which represents the set of its possible values. The set of rules defines the compatibility relations between properties of product components.

The choice of the CLP paradigm has various motivations. First, many CLP systems providing constraint solvers with different efficiency degrees and constraint expressiveness characteristics are available, e.g., SICStus Prolog [15] and ECL$^i$PS$^e$ [2]. Moreover, the translation of a partial configuration (a partially configured product consisting of a set of nodes, with the associated properties, and a set of constraints about them)) into a CLP program does not present any relevant difficulty and it produces a "natural" encoding of the original description. Finally, given a CLP program that represents a partial configuration, the CLP constraint solver makes it possible to verify the consistency of the partial configuration as well as to infer information about consequences of user's choices.

In its current version, the system offers tools that support product configuration only (not product modeling). It takes advantage of a model of the product and of user choices about property values to create a pro-

gram in SICStus Prolog [15] encoding a Constraint Satisfaction Problem (CSP). Each variable of the CSP represents a property of the product being configured, while the constraints of the CSP encode the rules that specify the relationships among the properties of the product. Once created, the program is executed using the `clpfd` solver of SICStus Prolog. If all constraints are satisfiable, the execution of `clpfd` on the given program returns for each variable the associated, possibly restricted domain; otherwise, it certifies the inconsistency of the encoded CSP. Once the solver computation ends, MCE communicates to Morphos the domain (computed by the solver) of each variable whose domain has been restricted. Using these pieces of information, Morphos is able to prevent user's choices that violate the given constraints.

The paper is organized as follows. In Section 2, we briefly describe the product configuration problem and existing software product configurators; moreover, we summarize the state of the art of research about the application of constraint programming to product configuration. In Section 3, we present the distinctive features of the configuration platform *Morphos*. The main contributions of the paper are illustrated in Section 4 and Section 5. Some comparisons between MCE and the original script-based Morphos configuration engine are reported in Section 6. Conclusions are drawn in Section 7.

## 2 Product configuration problem

The product configuration problem is fundamental for a large class of companies which offer products in different variants and options. In general, it requires a complex interaction with the customer to find, among the available characteristics and functions, those that best meet his/her needs. Such an interaction can result in combinations that have never been realized before. Products subject to configuration are products whose basic structure is predefined and that can be customized by combining together a series of available components and options (modules, accessories, etc.) or by specifying suitable parameters (lengths, tensions, etc.). The class of "products subject to configuration" includes products of different types, like, for instance, computers, cars, industrial machinery, shoes, etc.. With the term *configurable product* we refer to a type of product offered by a company. Hence, it does not correspond to a specific physical object, but it identifies a set of physical objects that the company can realize. A *configured product* is a single variant of the configurable product, which corresponds to a fully-specified physical object. A configured product is

obtained by customizing a configurable product, that is, by specifying the value of each customizable attribute of the configurable product. The series of activities and operations ranging from the acquisition of information regarding the particular variant of the product requested by the customer to the generation of data for the realization of the requested variant is called *configuration process*.

Companies offering customizable products face a series of management difficulties involving different functional areas. Many problems are related to an inadequate flow of information between the different areas. Traditional solutions adopted for the acquisition of customizable product orders are not, in general, satisfactory remedy to these difficulties. Recently, the major management software vendors have begun to develop and distribute software systems for product configuration. These systems aim at significantly improving the general management of configurable products. Software product configurators make available to the user a set of tools for the management of the customization of products. The "heart" of a software product configurator is the *product model*. It is a logical structure that formally represents the characteristics of the types of product offered by a company; moreover, it specifies a number of constraints on the relationships among these characteristics. The main modules of a product configurators are the *modeler* and the *configuration engine*. The modeler supports the modeling processes. It allows one to create and modify product models by defining characteristics and constraints of configurable products. The configuration engine supports the configuration processes. It facilitates the work of the seller, helping him/her in organizing and managing the acquisition of information about the product variant to be realized. In particular, it makes it possible to immediately check the validity and compatibility of inserted data.

A significant number of contributions witnesses the adequacy of constraint programming as a tool for product configuration, e.g., [3,10], in particular for knowledge representation and problem solving. Generally speaking, a configuration problem can be expressed as a Constraint Satisfaction Problem (CSP) whose solutions represent the possible configured products. A clear example of such an encoding can be found in [5], where the authors takes advantage of the classical $N$-queens problem to present the distinctive features of a product configurator based on constraint programming and preferences. Two applications of constraint programming to concrete instances of the configuration problem are reported in [7] (ILOG (J)Configurator system) and [4] (Lava system).

A common issue discussed in the literature is that of computational efficiency: suitable elements must be introduced to improve the efficiency of constraint solving. Other desirable system features are the ability to organize components in a hierarchy, the availability of tools that provide explanations on system behaviors, and the presence of interactive methods to express preferences and modify choices already made. To support at least in part these features, different variants of the standard CSP model have been proposed in the literature. An extension of standard CSP, called *Conditional CSP*, that makes it possible to manage optional components during the modeling phase in an effective way has been proposed in [9]. Another variant of CSP, close to Conditional CSP, called *Generative CSP*, has been developed in [4]. The handling of CSP at various levels of abstraction (*Composite CSP*) has been illustrated in [12]. A configurator that takes advantage of CSP and soft constraints to implement an iterative approach to the configuration problem is described in [5]. Finally, problems and advantages of configurators based on CSP have been discussed in detail in [1], where the CSP technology is integrated with that of *Truth Maintenance Systems* [8], which originates the so-called *assumption-based CSP*.

## 3 Morphos System

*Morphos* is a configuration platform developed by Acritas S.r.l. It is simple to use (being as simple as possible was one of the design principles of Morphos) and it can be easily integrated with the information system of a company. The user is guided in the process of creating of a new configured product: using a graphical user interface, he/she can provide a detailed definition of the components of the product; any given answer can be constrained through previously-defined rules that impose or restrict the possible choices. The process of maintaining configurations (storage of product revisions, creation of new configurations from existing ones) also reduces to a little number of immediate and intuitive steps.

The constraints on the characteristics of a product are specified by means of suitable JavaScript scripts. They are exploited by the configuration engine to infer information on product parameters after user's choices during the configuration process. On the positive side, the use of scripts gives a high degree of freedom in constraints definition and it provides an effective tool to manage computational procedures. On the negative side, it makes it difficult to maintain the constraints, it does not allow one to distinguish between constraints and computational procedures, and it

makes constraints consistency verification extremely involved and error-prone. In addition, no specific techniques are available to tackle complex constraints and no user-friendly tools help people having no specific skills in constraint definition.

The awareness of the difficulties inherent to the use of scripts and the intention of extending Morphos with new features and techniques for product modeling and configuration process management lead Acritas S.r.l. to start a collaboration with the Department of Mathematics and Computer Science of the University of Udine. The following sections describe the main achievements of such a collaboration.

## 4 Morphos Product Model

As already pointed out, a *Morphos product model* is defined by a tree, whose nodes represent well-defined components of a product, and a set of rules. Each node has associated a set of properties that describe configurable characteristics of the corresponding component. Each *node property* is characterized by a name and a set of integers or strings representing the set of values it can assume. This set can be given by explicitly enumerating its elements or by specifying a range (interval) of values, defined by a minimum value, a maximum value, and, possibly, a step (such an option is not supported by other systems, such as, for instance, Kumbang [11], where the elements of the domain of a variable must be explicitly given). Figure 1 shows the structure of the tree of a simple product model for a bicycle and the properties of the nodes `Fork`, `Wheel`, and `Crank`.

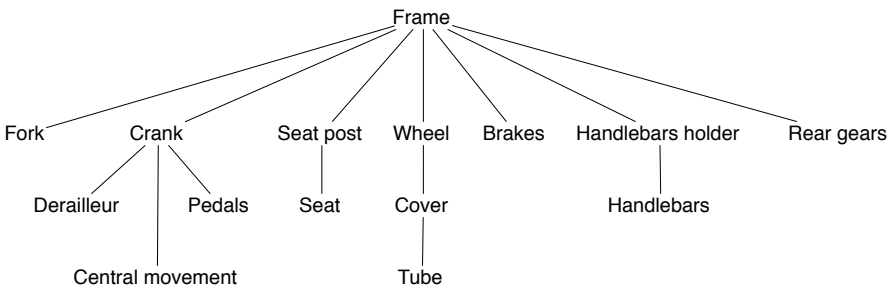
The constraints that define the compatibility relations between the properties of the components of a product can be of three types. To define them, we preliminarily introduce the notions of term, primitive constraint, and node constraint:

**Term.** A property of a node is a term. A constant (integer number or string) is a term. If $t_1$ and $t_2$ are terms, then $t_1 \oplus t_2$ is a term, where $\oplus \in \{+, -, *, /\}$ (notice that all terms are ground).

**Primitive constraint.** A primitive constraint has the form $p\ op\ t$, where $p$ is a node property, $t$ a term, and $op \in \{<, \leq, =, \geq, >, \neq\}$.

**Node constraint.** A primitive constraint is a node constraint. If $F$ and $G$ are node constraints, then $F \wedge G$, $F \vee G$, and $\neg F$ are node constraints.

We distinguish three different types of constraint:




| Node | Property | Type | Value |
|---|---|---|---|
| Fork | Type | String | Racing, MTB |
| | Material | String | Aluminum, Carbon |
| | ShockAbsorber | String | 80 mm, 100mm, 85-130 mm, SingleChassis |
| Crank | Type | String | Racing, MTB |
| | Central movement | String | Integrated, None |
| | Gears | String | 46/36, 48/38, 50/34, 44/32/22 |
| Wheel | Type | String | Racing, MTB |
| | Material | String | Aluminum, Carbon |
| | Diameter | String | 22",24",26",28",29" |
| | Spokes number | Integer | From 18 to 24 with step 2 |
| | Cover | String | Traditional, Tubeless |

**Fig. 1.** Bicycle: tree structure and node properties.

**Rule.** A rule is defined by a *condition*, a node constraint, and a series of *actions* (commands) to be executed by Morphos when the condition holds or a set of node constraints that must be satisfied when the condition holds. An example of rule for the bicycle product model is the following one:

**Condition:** `Fork.Type = Racing`

**Action:** `Fork.ShockAbsorber = SingleChassis`

**Constraint.** A constraint is a rule whose condition always holds (`True` condition), and whose action consists of a node constraint, that is, a constraint defines a condition that must be satisfied by all configurations. An example of constraint for the bicycle product model is the following one:

**Condition:** `True`

**Action.** `Wheel.Cover = Cover.Typology`

**Relation:** A relation is a table that relates two or more properties. It defines a set of admissible tuples of values, that is, an admissible

subset of the Cartesian product of the set of values of the properties it relates. An example of relation for the bicycle product model is given in Table 1.

| Crank.Type | Crank.Gears |
|---|---|
| Racing | 46/36 |
| Racing | 48/38 |
| Racing | 50/34 |
| MTB | 44/32/22 |

**Table 1.** Bicycle: an example of relation.

As we will show in Section 5, in the current version of the system, product model constraints behave as *global rules*: they hold for all the ordered tuples of instance properties they involve. However, taking advantage of the tree structure of the product model one may think of introducing *local rules*, that is, constraints which hold only for tuples of instance properties belonging to the nodes of a specific subtree. Such an ability of dealing with local rules will be incorporated in the next release of the engine.

The above-described solution is quite different from the one supported by the original script-based Morphos configuration engine: a script was associated with each node, which was executed whenever a change in a property of the node took place. No special mechanisms to control script execution were available. As a consequence, the original Morphos engine was unable to prevent unacceptable behaviors, such as infinite computations, from occurring, as shown by the following simple example. Let us assume that we have two nodes *Node1* and *Node2*, which respectively contain properties $A$ and $B$. Both properties have the set {0,1} as their domain. In addition, *Node1* and *Node2* are tied up with the scripts Script1 and Script2, respectively, which are reported in Table 2. The resulting behavior can be summarized as follows. Whenever property $A$ changes, then Script1 is executed setting the value of $B$ accordingly to that of $A$. Similarly, whenever property $B$ changes, Script2 sets a new value for $A$, but this time the value for $A$ is the negation of that for $B$. It can be easily checked that this can lead to an infinite computation. For instance, let us assign value 0 to $A$. Since $A$ has changed, Script1 is executed. Due to the first rule of Script1, value 0 is assigned to $B$. The change in the value of $B$ causes the execution of Script2, whose first rule assigns value 1 to

| | |
|---|---|
| A = 0 ⇒ B := 0 | B = 0 ⇒ A := 1 |
| A = 1 ⇒ B := 1 | B = 1 ⇒ A := 0 |

**Table 2.** Script1 (left) and Script2 (right)

$A$. Such a change forces the engine to execute Script1 again and, since the value of $A$ is 1, it assigns value 1 to $B$. The change in the value of $B$ forces the execution of Script2 that assigns value 0 to $A$, the value that was assigned to $A$ at the very beginning. Hence, whenever we assign the value 0 (resp., 1 to $A$ (resp., $B$) an infinite cycle is generated.

## 5 Morphos Configuration Engine (MCE)

To improve the capabilities of the Morphos engine we developed a new configuration engine, called *Morphos Configuration Engine (MCE)*, taking advantage of CLP techniques. Figure 2 pictorially shows how the Morphos MCE system manages the configuration process. First, Mor-

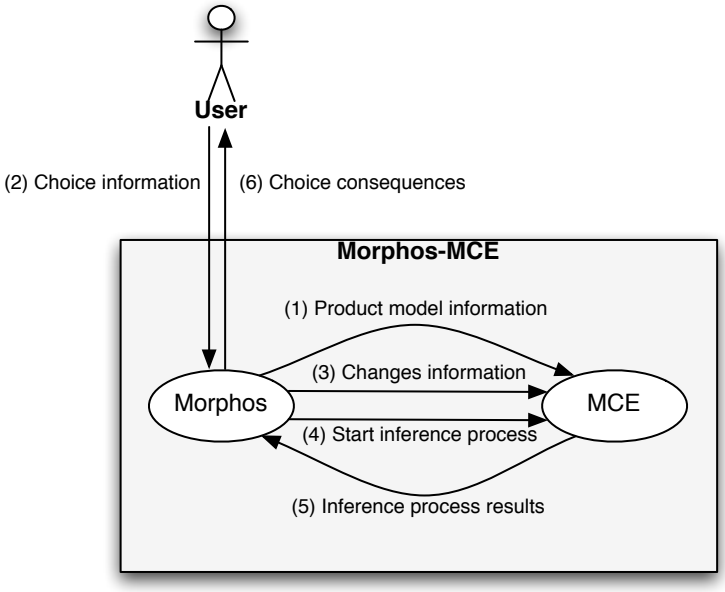



**Fig. 2.** Morphos-MCE: the configuration process.

phos initializes MCE (1) sending to it information about the model of the product to be configured, that is, nodes, properties, and constraints

defined by the product model. After such an initialization phase, the interaction with the user begins. The user makes his/her choices using the Morphos interface (2). He/she can add or remove node instances (product components) and set the values of node properties (product component characteristics). Morphos communicates to MCE each data variation specified by the user (3) and MCE updates the current partial configuration accordingly. MCE can then be exploited to infer information from the resulting partial configuration (4). Once the inference process ends, MCE returns to Morphos the results of its computation (5). Morphos in its turns shows to the user the consequences of his/her choices on the (partial) configuration (6).

The Morphos MCE system also allows the user to modify past choices about property values. Figure 3 shows how the Morphos MCE system manages the assignment revision process. We illustrate such a capabil-

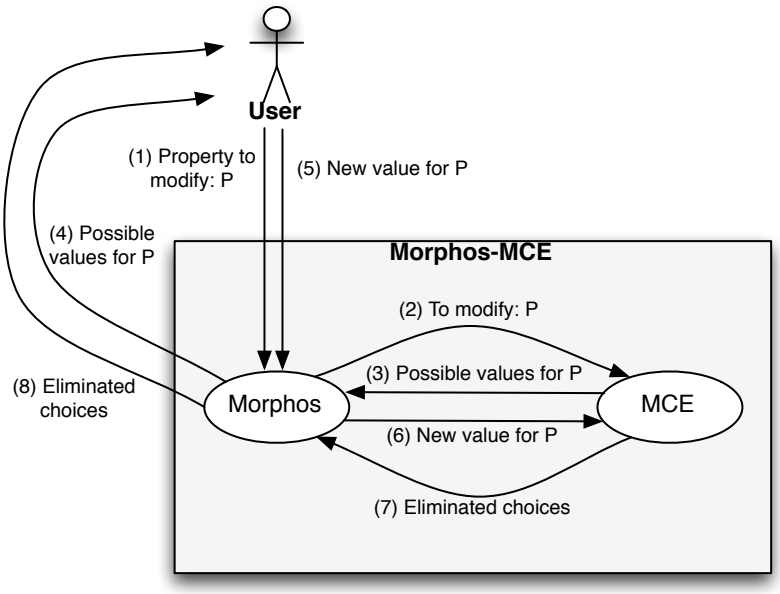



**Fig. 3.** Morphos-MCE: the assignment revision process.

ity of the Morphos MCE system by means of a simple example. Let us assume that the user would like to modify the value he/she previously assigned to property `P`. The user selects property `P` using Morphos (1). Morphos communicates to MCE that the user would like to modify the

value of `P` (2). MCE computes the set of values that can be assigned to `P` without incurring in conflicts with constraints and choices made before the one concerning property `P`, and it communicates the result to Morphos (3). Morphos shows to the user the values he/she can assign to `P` (4). The user chooses the new value for `P` (5). Morphos communicates it to MCE (6). MCE computes a maximal subset of the choices made after the one concerning `P` that can be maintained without generating any conflict with constraints, previous choices, and the new choice for `P`, and it communicates to Morphos the choices that have to be withdrawn (7). Morphos shows to the user which choices must be withdrawn to keep the configuration consistent (8). The user can change the subset of maintained choices according to his/her own preferences. In doing that, he/she can impose to the system to reintroduce some choices it withdrawn, thus forcing it to recompute the subset of choices to withdraw.

Let us describe now the CLP encoding of the product configuration problem. Starting from information about the product model and the partial configuration (node instances and choices on property values), MCE defines a SICStus Prolog program encoding a CSP. The variables of the CSP represent the properties of nodes instances in the partial configuration, the domains of the variables are the sets of possible values for the properties defined in the product model, the constraints of the CSP are the rules, the constraints, and the relations about the properties of node instances in the partial configuration (choices of property values are viewed as primitive constraints). Rules, constraints, and relations are translated into SICStus Prolog constraints as follows. Let us consider first the case of node constraints.

- A primitive constraint $p\ op\ t$ is translated into a constraint `P RelOp t`, where `RelOp` is the SICStus Prolog relational operator corresponding to *op*.
- A node constraint $F \wedge G$ (resp., $F \vee G$, $\neg F$) is translated into a constraint `F #/\ G`, (resp., `F #\/ G`, `#\ F`), where `F` and `G` are the translations in SICStus Prolog of $F$ and $G$, respectively.

Constraints defining compatibility relations among the properties of product components are translated in SICStus Prolog as follows.

- A rule with condition defined by a node constraint $C$ and actions defined by node constraints $A_1, \ldots, A_n$ is translated into a SICStus Prolog constraint `C #=> A_1 #/\ ... #/\ A_n`, where `C`, `A_1`, `A_2`, etc. are the translations in SICStus Prolog of the node constraints $C, A_1, \ldots, A_n$.

– A constraint with action defined by node constraint $A$ is translated into a SICStus Prolog constraint `A`.
– A relation on properties $p_1, \ldots, p_n$ consisting of the tuples $\langle t_{1,1}, \ldots, t_{1,n} \rangle$, $\ldots$, $\langle t_{m,1}, \ldots, t_{m,n} \rangle$ is translated into a set of SICStus Prolog constraints of the form `p_i #= t_(j,i) #=> (p_1 in {...} #/\...#/\ p_i-1 in {...} #/\ p_i+1 in {...} #/\...#/\ p_n in {...})`.

It is worth pointing out that a configuration may include multiple instances of each node defined in the product model. As an example, a bicycle configuration contains two instances for node `Wheel` and two instances for node `Cover`. Each constraint in the product model must hold for all the ordered tuples of instance properties it involves, e.g., the constraint with action `Wheel.Cover = Cover.Typology` must hold for all the four couples of instance properties `Wheel.Cover` and `Cover.Typology`. As a consequence, for each constraint in the product model and each ordered tuple of instance properties it involves, a constraint is inserted in the SICStus Prolog program defined by MCE.

Given a program with the above-described characteristics, the `clpfd` solver of SICStus Prolog computes the possibly restricted domain associated with each variable, provided that all constraints are satisfied; otherwise, it detects the inconsistency of the encoded CSP. When the computation of the solver ends, MCE communicates to Morphos the domain (computed by the solver) of each variable whose domain has been restricted. By exploiting information about (restricted) domains, Morphos can prevent user's choices that would violate the constraints, that is, user's choices incompatible with previous ones. Moreover, if there are commands to be executed, MCE communicates them to Morphos as well.

We conclude the section with a short description of how MCE uses `clpfd` during the assignment revision process. Let us consider again the case when the user would like to modify the value he/she assigned to property `P`. To compute the set of alternative values that can be assigned to `P` without generating conflicts with constraints and choices made before the one concerning property `P`, MCE defines a SICStus Prolog program encoding a CSP as in the configuration process, but without considering the primitive constraints representing choices on property values made after the one the user would like to modify. To compute a maximal subset of the choices made after the one concerning `P` that can be maintained without generating conflicts with constraints, previous choices, and the new choice for `P`, MCE defines a SICStus Prolog program encoding a CSP as in the configuration process, but with the following additional features. A variable `N_i` with domain $\{0, 1\}$ is associated with each assignment to

property values $A_i$ made after the one for `P`. Let $m$ be the number of these assignments. For each of them, a constraint of the form `N_i #=> A_i`, where `A_i` is the translation of the primitive constraint representing $A_i$, is inserted in the program (instead of `A_i`). Moreover, a constraint of the form `N_1 + N_2 + ... + N_m #= T`, where `T` is a variable with domain $\{0, ..., m\}$, is inserted in the program. A maximal subset of the assignment $A_i$ that maintains the configuration consistent is computed using a predicate of the form `maximize(labeling([],[N_1,.
..,N_m,T]),T)`.

## 6 Comparisons with Morphos original engine

MCE has been developed in C# inside the .NET 3.5 framework. It provides several methods to solve the configuration problem. In particular, it supports the following functionalities:

- it communicates with Morphos through XML messages;
- it records and manages both information received from Morphos and solutions returned to Morphos;
- it creates and executes CLP programs.

The execution of CLP programs exploits the *SICStus Runtime Environment* [4]

Compared with the original configuration engine, MCE presents several advantages. We would like to point out a couple of them.

A first advantage of the CLP-based engine of MCE with respect to Morphos script-based one is represented by the simplicity of the translation of rules, constraints, and relations into a SICStus Prolog program. This allows one to easily check whether a partial configuration is satisfiable or not. On the contrary, the script-based engine makes it necessary to define an ad-hoc script for each constraint on properties that must be satisfied. With the CLP-based engine, it is sufficient to define the constraints within the model; the engine automatically inserts all constraints about properties involved in the partial configuration in the SICStus Prolog program.

A second advantage is given by the use of a constraint solver to remove from the domains the values that are incompatible with already made

[4] SICStus Prolog allows one to create a standalone environment where CLP programs can run, without requiring the installation of the SICStus suite on the final user's computer.

choices. Consider, for instance, the following rule for the bicycle product model:

**Condition:** `Fork.Type = Racing`
**Action:** `Fork.ShockAbsorber = SingleChassis`

This rule states that whenever the value `Racing` is assigned to property `Type` of node `Fork`, property `ShockAbsorber` of node `Fork` can only assume the value `SingleChassis`. At the same time, it states that if a value different from `SingleChassis` is assigned to property `ShockAbsorber` of node `Fork`, then property `Type` of node `Fork` cannot assume the value `Racing`. The single SICStus Prolog constraint we obtain from this rule imposes both conditions. On the contrary, in the case of the original script-based engine, Morphos makes use of two different scripts to impose the two conditions. This makes the modeling phase much more complex, since the programmer has both to determine all possible implications of rules and to encode them by means of suitable scripts. Moreover, the complexity of the encoding complicates the maintenance phase.

## 7 Conclusions

In this paper we briefly introduced the important problem of product configuration and we described an approach to it based on Constraint Logic Programming. In particular, we illustrated the distinctive features of a CLP-based configuration engine, called MCE, which has been incorporated in the product configurator system Morphos developed by Acritas S.r.l. Besides an intuitive account of the behavior of MCE, we pointed out its advantages with respect to the original script-based configuration engine of Morphos. A simple case of product configuration referring to the domain of bicycles has been used as a source of exemplification.

Besides experimenting the proposed tool on different real-world application domains, we are currently refining/extending it in several directions. On the one hand, we are refining MCE by adding the possibility of distinguishing between local and global rules, as well as that of dealing with different numerical domains (e.g., integers and reals); on the other hand, we are extending Morphos with model and process configuration capabilities.

*Acknowledgments.* This work has been partially supported by Acritas S.r.l. and Regione Friuli Venezia Giulia. We would like to thank Maurizio Piani, Claudio Buble, Andrea Formisano, and Andrea Schiavinato for useful discussions and suggestions.

## References


1. J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-Application to configuration. *Artificial Intelligence*, 135:199–234(36), February 2002.
2. K.R. Apt and M.G. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.
3. F.Frayman and S. Mittal. COSSACK: A Constraint-Based Expert System for Configuration Tasks. *Knowledge-Based Expert Systems in Engineering: Planning and Design*, pages 143–166, 1987.
4. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.
5. E. Freuder, C. Likitvivatanavong, M. Moretti, F. Rossi, and R. Wallace. Computing explanations and implications in preference-based configurators. In Barry O'Sullivan, editor, Recent Advances in Constraints, volume 2627 of LNAI, pages 76–92., 2003.
6. G. Friedrich and M. Stumptner. Consistency-based configuration. In *AAAI-99 Workshop on Configuration*, pages 35–40. AAAI Press, 1999.
7. U. Junker. The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In *IJCAI-03 Workshop on Configuration*, pages 13–20, 2003.
8. J. P. Martins. The truth, the whole truth, and nothing but the truth: An indexed bibliography to the literature of truth maintenance systems. *AI Mag.*, 11(5):7–25, 1991.
9. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *AAAI*, pages 25–32, 1990.
10. S. Mittal and F. Frayman. Making Partial Choices in Constraint Reasoning Problems. *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 631–636, 1987.
11. V. Myllärniemi, T. Asikainen T. Männistö, and T. Soininen. Kumbang configurator - a configuration tool for software product families. In *IJCAI-05 Workshop on Configuration*, 2005.
12. D. Sabin and E. C. Freuder. Configuration as Composite Constraint Satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996, 1996.
13. D. Sabin and R. Weigel. Product configuration frameworks - a survey. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):42–49, Jul/Aug 1998.
14. T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Unified configuration knowledge representation using weight constraint rules. In *ECAI-00 Configuration Workshop*, pages 79–84, 2000.
15. Swedish Institute of Computer Science, Intelligent Systems Laboratory. *SICStus Prolog User's Manual*, 4.0.3 edition, May 2008.