Morphos Configuration Engine: the core of a commercial configuration system in CLP(FD)*

Dario Campagna
Dept. of Mathematics and Computer Science
University of Perugia
dario.campagna@dipmat.unipg.it

Christian De Rosa
Acritas s.r.l.
Martignacco (UD), Italy
derosa@acritas.it

Agostino Dovier
Dept. of Mathematics and Computer Science
University of Udine
agostino.dovier@uniud.it

Angelo Montanari
Dept. of Mathematics and Computer Science
University of Udine
angelo.montanari@uniud.it

Carla Piazza
Dept. of Mathematics and Computer Science
University of Udine
carla.piazza@uniud.it

October 4, 2010

Abstract

Product configuration systems are an emerging software technology that supports companies in deploying mass customization strategies. In this paper, we describe a CLP-based reasoning engine that we developed for a commercial configuration system. We first illustrate the advantages of the CLP approach to product configuration over other ones. Then, we describe the actual encoding of the considered product configuration problem as a constraint satisfaction problem. We devote a special attention to the key issues of constraint propagation and optimization as well as to the relevant process of assignment revision. A comparison with existing systems for product configuration concludes the paper.

1 Introduction

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 with a predefined basic structure 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 machineries, 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 preliminary version of this paper has been published in the Proceedings of the 24th Italian Conference on Computational Logic (CILC 2009) [7].

a configurable product, that is, by specifying the value of each customizable attribute of the configurable product. The *configuration process* consists of a series of activities and operations ranging from the acquisition of information about the specific variant of the product requested by the customer to the generation of data for its realization.

Companies offering customizable products face a series of management difficulties involving different functional areas. In particular, a lot of problems are related to an inadequate flow of information among the different areas. Traditional solutions adopted for the acquisition of customizable product orders are not, in general, satisfactory remedies 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 core of a software product configurator is the *product model*. It is a logical structure that formally represents the features of the types of product offered by a company; moreover, it specifies a number of constraints on the relationships among these features. The main modules of a product configurators are the *modeler* and the *configuration engine*. The modeler supports the modeling process. It allows one to create and to modify product models by defining features and constraints of configurable products. The configuration engine supports the configuration process. 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 consistency of inserted data.

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 [30]. In particular, significant approaches based on computational logic [14, 35] and constraint programming [11, 24, 29] have emerged. In this paper, we describe the main achievements of our research on product configuration based on Constraint Logic Programming (CLP). We focus our attention on the proposal of a new general CLP-based (product) configuration engine, called MCE (Morphos Configuration Engine), for the existing commercial configuration platform Morphos. Morphos has been developed by Acritas S.r.l., and successfully used by various local factories, whose names are omitted for privacy reasons. It has been designed to be simple to use and easily integrable with the information system of a company. The user is guided in the creation of a new configured product through a wizard, and he/she can use a graphical interface to provide a detailed definition of the components of the product. Every answer from the user can be constrained through previously-defined rules that force or restrict the set of possible choices. Before the integration of MCE in Morphos, configuration rules were implemented using JavaScript scrips. Such scripts were used by the configuration engine to infer information on product parameters after user's choices during the configuration process. On the one hand, the use of scripts gave a high degree of freedom in rule definition and it allowed the management of computational procedures. On the other hand, maintainance of configuration rules was extremely difficult, it was not possible to distinguish between rules and computational procedures, and checking rule consistency was practically unfeasible. Moreover, there were not specific techniques to deal with complex rules. Finally, definition of rules was quite difficult for people having no specific skills.

In the proposed framework, the product model consists of a tree, whose nodes represent the components of a product, and a set of constraints. Each node is paired with a set of variables, which express configurable features of the component it represents. Each variable 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 constraints 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 expressiveness and efficiency degrees are available, e.g., SICStus Prolog [39], ECL^{i}PS^e [4], and SWI Prolog [42]. Moreover, the translation of a partial configuration, that is, a partially configured product consisting of a set of nodes, with the associated variables, and a set of constraints about them, into a CLP program does not present any major 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 relevant information about consequences of user's choices.

MCE has been implemented using the C# language. It takes advantage of a model of the product and of user's choices about property values to create a program in SWI Prolog [42], which encodes a Constraint Satisfaction Problem (CSP). Each variable of the CSP represents a variable of the product being configured, while the constraints of the CSP encode the constraints that specify the relationships among product variables. Once created, the program is executed using the finite domain constraint solver of SWI Prolog. If all constraints are satisfiable, the execution of the solver 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 (successfully) ends, for each variable whose domain has been restricted, MCE communicates to Morphos the

restricted domain computed by the solver. Using these pieces of information, Morphos is able to prevent user's choices that violate the given constraints. In case of a failure, MCE supports the user in the process of assignment revision. In the design, development, and implementation of MCE, we paid a special attention to fundamental capabilities, such as constraint propagation and optimization, and assignment revision, for which existing solutions are not fully satisfactory (in the conclusions, we briefly contrast the solution we propose with alternative ones, pointing out its distinctive features).

The paper is organized as follows. In Section 2, we summarize the state of the art of research about methodologies and techniques for product configuration. We first introduce and briefly discuss rule-based, case-based, and model-based approaches to product configuration. Then, we restrict our attention to the latter approach and analyze systems based on description logics, answer set programming, binary decision diagrams, and constraint programming. A special emphasis is devoted to the last one. In Section 3, we first describe the main features of MCE product models and then we illustrate the structure of the MCE configuration process. Section 4 presents the encoding of the product configuration problem as a constrain satisfaction problem. Constraint propagation and optimization are the subject of Section 5. Section 6 is devoted to the process of assignment revision. In Section 7 we briefly compare MCE with the most important existing configuration tools. An assessment of the work done and an outline of future research direction are given in Section 8. In Appendix A, we report some screenshots of a preliminary version of the proposed configuration system.

2 Related Work

The problem of product configuration has been addressed in a number of different ways. According to [18], three main approaches can be identified: rule-based reasoning [22], case-based reasoning, and model-based reasoning [26].

2.1 Approaches to Product Configuration

Rule-based approaches represent user's requirements about the desired configured product in terms of assertions in the working memory and map them to product components through suitable *if-then* rules. A rule for selecting a component is fired if there is a requirement that expresses a need for such a component, that is, if its *if* condition is satisfied. When executed, a rule can generate new components, customize existing one, or add sub-components. The rule conditions may need to check possible interactions with other components. In these cases, the structure of conditions turns out to be very complex. Moreover, whenever compatibility constraints among components change, the conditions of several rules must be revised. This lack of modularity leads to difficulties in the management of rule-based configurators. As a concrete example, it caused a severe maintenance problem in the R1/XCON configurator of DEC computers [6, 22]. As a matter of fact, rule-based systems can be successfully exploited to deal with very simple configuration problems involving a small number of choices only.

Case-based reasoning (CBR) is a problem solving paradigm proposed in AI that differs from standard AI paradigms in many respects. Rather than relying on general knowledge about a problem domain or establishing associations between problem descriptions and solutions by means of suitable general relationships, CBR takes advantage of specific knowledge about concrete problem instances it already dealt with in the past (cases). A new problem instance is addressed and possibly solved by finding a similar past case and reusing the machinery that has been used to solve it. Moreover, additional knowledge provided by the solution of a new instance of the problem becomes immediately available for the management of future problem instances. The application of case-based reasoning to the configuration problem has been explored in various contributions. An interesting example can be found in [41]. The main weaknesses of such an approach are connected to the lack of a complete model of the application domain, the complexity of the knowledge acquisition problem, the low quality of the proposed solutions, and various inefficiencies at execution time. To overcome these weaknesses, hybrid approaches, combining case-based reasoning with other reasoning techniques, such as rule-based reasoning and constraint satisfaction techniques, have been proposed.

Model-based reasoning approaches address the deficiencies of rule-based and case-based reasoning ones. Firstly, they separate the description of the problem from the algorithm that solves it, thus allowing an analysis of the problem which actually does not depend on the chosen approach. Secondly, the description of the problem is based on a model of the system to be configured. Such a model consists of a set of decomposable entities and a specification of the interactions between them. Modularity and compositionality are well-supported since components can easily be added or removed without changing the whole model. Thirdly, they require the

description of the problem to be complete by defining a closed space of possible configurations. If no element of this space satisfies the given requirements, then the problem has no solution.

Different proposals can be situated in the setting of model-based reasoning, including approaches based on description logic [23], constraint programming [24, 36], and resource models [16]. Additional solutions have been recently proposed based on SAT [33], binary decision diagrams [15], integer programming [40], and answer-set programming [32]. Finally, there have been various attempts to combine different solutions in order to enhance the modeling and solving capabilities of model-based reasoning configurators.

2.2 The Model-based Approach to Product Configuration

In the following, we restrict our attention to model-based reasoning approaches. First, we present some of the most interesting existing systems respectively based on description logic, answer-set programming, and binary decision diagrams; then, we focus our attention on configuration systems based on constraint programming.

Description logic (DL) is well-suited to describe component types and their relations. It organizes component types in a taxonomy and it allows one to define complex types out of primitive ones. Moreover, DL is able to express specialization relations among complex types and to test type consistency by a reasoning task called classification. Classification can actually be exploited to solve configuration problems provided that knowledge about configuration can be completely expressed in DL. Unfortunately, in many cases the expressiveness of DL is not sufficient and a rule-based or constraint-based engine must be used to cope with complex compatibility and numerical constraints. A significant example of such a hybrid approach is given by the CLASSICS project [23].

Answer-set programming (ASP) [32] expresses configuration knowledge by means of suitable default rules, while ensuring well-justified configurations through groundedness conditions [34]. A significant representative of the ASP approach to configuration is Kumbang Configurator [27], a tool supporting the configuration task for configurable software product families. In Kumbang Configurator, a configurable software product family is modeled from two independent, yet mutually related, points of view. *Features* are abstractions from the set of requirements of the product family. A feature may include a number of sub-features as its constituent elements and it can be characterized by a number of attributes. The technical aspects of the product family are captured by its architecture, which is defined in terms of *components*. A component may have a number of other components as its parts and it can be be characterized by a number of attributes. Furthermore, the interaction possibilities for a component are specified by its interfaces and bindings between them. In addition, Kumbang Configurator supports the user in the configuration task by providing a graphical user interface through which the user can enter his/her specific requirements for a product individual. Moreover, it checks the configuration for consistency and completeness after each step and it derives the consequences of the selections made so far. The derivation process is implemented using *smodels* [32], a general-purpose inference tool based on the stable model semantics for logic programs.

Binary decision diagrams (BDDs) have been successfully exploited to support configuration systems for interactive sales. These systems must be able to simultaneously solve the configuration problem for a number of customers, guaranteeing short response times. As the considered problem instances only differ in the users' requirements, and not in the product model, preprocessing techniques can be used to compute configurations in advance. For problems involving a fixed number of parts with small finite domains, it may be possible to represent the whole configuration space compactly by a BDD [15]. A well-known representative of this class of configuration systems is the commercial system Configit Product Modeler. This system implements a two-phase approach to interactive product configuration. In the first phase, the product model is compiled into a reduced ordered BDD representing the set of valid configurations (the solution space). In the second phase, a fast specialized BDD operation is used to prune the variable domains. The worst-case response time only grows polynomially with the size of the BDD. Since the size of the compiled BDD is small, the computationally hard part of the configuration problem is fully confined in the offline phase.

Constraint programming (CP) is a well-known and successful technique for knowledge representation and problem solving. The adequacy of CP as a tool for product configuration is witnessed by a significant number of contributions (see, for instance, [12, 25]). A configuration problem can be naturally formulated as a Constraint Satisfaction Problem (CSP) whose solutions represent the possible configured products. An enlightening example of such an encoding can be found in [13], where the authors take advantage of the classical problem of the N queens to illustrate the distinctive features of a product configurator based on CP with preferences. Two representative configuration systems based on CP are the ILOG (J)configuration system [17] and the Lava system [11]. A fundamental issue in CP-based configuration systems is that of computational efficiency. Various techniques and heuristics have been proposed in the literature to improve the efficiency of constraint solving in the configuration process. Extensions to the basic CP paradigm have been investigated to support

advanced desirable features (of configuration systems), such as the ability to organize system 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 some of these features, a number of variants of the standard CSP model have been developed. An extension of CSP, called *Conditional CSP*, that makes it possible to manage optional components during the modeling phase in an effective way, has been proposed in [24]. Another variant of CSP, close to Conditional CSP, called *Generative CSP*, has been developed in [11]. The handling of CSP at various levels of abstraction, called *Composite CSP*, has been illustrated in [29]. A configurator that uses CSP and soft constraints to implement an iterative approach to the configuration problem has been described in [13]. Finally, the integration of the CSP technology with that of *Truth Maintenance Systems* [20], which results in the so-called *assumption-based CSP*, has been explored in [3] (as a matter of fact, such a paper contains an interesting discussions of advantages and limitations of configurators based on CSP).

3 MCE Product Model and Configuration Process

In this section, we first present the structure of MCE product models (Section 3.1) and then we describe the organization of the configuration process in MCE (Section 3.2).

3.1 MCE Product Model

MCE supports two basic sorts: the sort int for integer numbers and the sort string for ASCII strings (the possible addition of the sort real for real numbers will be discussed in the conclusions). We assume that there is a denumerable set of variables $\mathcal{V}_{\mathsf{int}}$ of sort int and a denumerable set of variables $\mathcal{V}_{\mathsf{string}}$ of sort string. Moreover, we assume that a domain $D(V)$ is associated with each variable V, which contains the set of admissible values for V. We consider as domains sets of integers and sets of ASCII strings.
Both integer and string domains are defined by explicitly enumerating their elements; in addition, whenever possible, an integer domain is compactly represented as an interval:

- $D(V) = \{i_1, i_2, \ldots, i_n\}$, with $i_1 \in \mathbb{Z}, \ldots, i_n \in \mathbb{Z}$, for $V \in \mathcal{V}_{\mathsf{int}}$;
- $D(V) = [\min, \max]$ $(= \{i \in \mathbb{Z} \mid \min \leq i \leq \max\})$, with $\min \in \mathbb{Z}$ and $\max \in \mathbb{Z}$, for $V \in \mathcal{V}_{\mathsf{int}}$;
- $D(V) = \{s_1, \ldots, s_m\}$, where, for $i = 1, \ldots, m$, s_i is an ASCII string, for $V \in \mathcal{V}_{\mathsf{string}}$.

The notion of *sorted term* can be recursively defined in the usual way:

- a variable of $\mathcal{V}_{\mathsf{int}}$ is a term of sort int and a variable of $\mathcal{V}_{\mathsf{string}}$ is a term of sort string;
- an integer number is a (ground) term of sort int and an ASCII string is a (ground) term of sort string;
- if t_1 and t_2 are terms of sort int, then $t_1 \oplus t_2$ is a term of sort int, where $\oplus \in \{+, -, *, /, \mathsf{mod}\}$;
- if t_1 and t_2 are ground terms of sort string, then $t_1 \circ t_2$ is a ground term of sort string.

Terms can be used to build constraints as follows:

- if t_1, t_2 are terms of sort int and $op \in \{<, \leq, =, \geq, >, \neq\}$, then $t_1\ op\ t_2$ is a *primitive constraint*;
- if t_1, t_2 are terms of sort string and $op \in \{=, \neq\}$, then $t_1\ op\ t_2$ is a *primitive constraint*;
- primitive constraints are *constraints*; any Boolean combination of constraints is a *constraint*, that is, if C_1 and C_2 are constraints, then $\neg C_1$, $C_1 \vee C_2$, $C_1 \wedge C_2$, and $C_1 \Rightarrow C_2$ are *constraints*.

In the following, we will focus our attention on three syntactical subclasses of the set of constraints that turn out to be particularly useful in product modeling:

- *Rule constraints.* A rule constraint is a constraint of the form $C_1 \Rightarrow C_2$, where C_1 and C_2 are constraints. C_1 is called *condition*, C_2 is called *action*.
- *Integrity constraints.* An integrity constraint is a constraint C, whose outermost connective (if any) is not $\Rightarrow$.

- *Disjunctive Normal Form (DNF) constraints.* A DNF constraint over a set of variables $\mathcal{V} = \{X_1, X_2, \ldots, X_k\}$ is a disjunction of the form $C_1 \vee \cdots \vee C_h$, where each disjunct C_i is a conjunction of the form:

$$X_{i,1} = v_{i,1} \wedge \cdots \wedge X_{i,k_i} = v_{i,k_i},$$

with $k_i \leq k$ and $X_{i,j} \neq X_{i,j'}$ for $j \neq j'$, where $X_{i,j}$ is a variable belonging to $\{X_1, X_2, \ldots, X_k\}$ and $v_{i,j}$ is an integer number or an ASCII string (depending on the sort of $X_{i,j}$).

A DNF constraint is *complete* if it is of the form:

$$(X_1 = v_{1,1} \wedge X_2 = v_{1,2} \wedge \ldots \wedge X_k = v_{1,k}) \vee \ldots \vee (X_1 = v_{h,1} \wedge X_2 = v_{h,2} \wedge \ldots \wedge X_k = v_{h,k})$$

It is worth pointing out that the distinction between rule and integrity constraints is merely syntactic as $A \Rightarrow B$ (rule constraint) is the same as $\neg A \vee B$ (integrity constraint). The introduction of the class of rule constraints is motivated by the fact that they make it possible to adopt a rule-based style in the specification of the dependencies among product features, which is quite common in existing configuration systems (Morphos included). Notice also that DNF constraints are a proper subclass of integrity constraints, and that complete DNF constraints provides a way of defining precisely the set of admissible tuples of values for the variables in $\mathcal{V}$.

We are now ready to introduce the notion of *Morphos product model*, which consists of the following components:

1. a tree, called a *product model tree*, whose nodes represent well-defined components of a product; it basically defines the *has part/is part of* relation over product components;
2. for each node N of the product model tree, a set of *node variables* $\mathcal{V}_N$ that define the configurable characteristics of the corresponding component; each node variable $V \in \mathcal{V}_N$ is endowed with a domain $D(V)$, which specifies the set of values V can assume;
3. a set of constraints on node variables, called *compatibility relations pool*, that define the compatibility relations between configurable characteristics of product components, e.g., some configurations of components are incompatible, other ones depend on each other.

In the rest of the paper, we will refer to Morphos product models simply as product models. The following example shows a simple product model for a bicycle. It will be used throughout the paper as a source of exemplification.

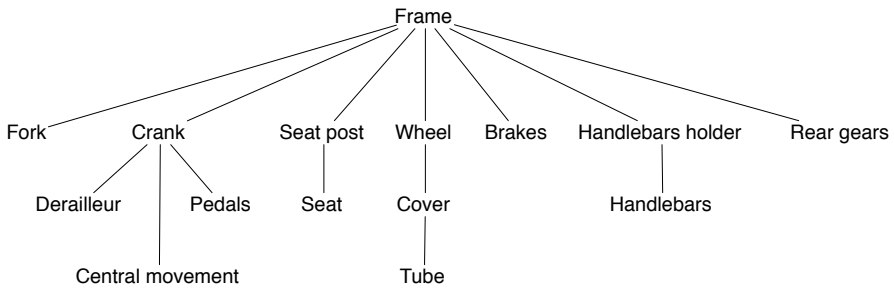

Figure 1: The product model tree of a bicycle.

Example 1 *Let us (partially) define a product model for a bicycle. The structure of the product model tree is depicted in Figure 1. The set of variables for the nodes* `Fork`*,* `Crank`*,* `Wheel`*, and* `Cover` *are the following (we omit the set of variables for the other nodes as they are not relevant to the purpose of this example):*

$$\begin{aligned}
\mathcal{V}_{\texttt{Fork}} = \{&\texttt{Type} \in \{Racing, MTB\},\\
&\texttt{Material} \in \{Aluminium, Carbon\},\\
&\texttt{ShockAbsorber} \in \{80mm, 100mm, \textit{85–135mm}, SingleChassis\}\}\\
\mathcal{V}_{\texttt{Crank}} = \{&\texttt{Type} \in \{Racing, MTB\},\\
&\texttt{CentralMovement} \in \{Integrated, None\},\\
&\texttt{Gears} \in \{46/36, 48/38, 50/34, 44/32/33\}\}
\end{aligned}$$

$$\mathcal{V}_{\texttt{Wheel}} = \{\texttt{Type} \in \{Racing, MTB\},$$
$$\texttt{Material} \in \{Aluminium, Carbon\},$$
$$\texttt{Diameter} \in \{22, 24, 26, 28, 29\},$$
$$\texttt{SpokesNumber} \in [18, 24],$$
$$\texttt{Cover} \in \{Traditional, Tubeless\}\}$$
$$\mathcal{V}_{\texttt{Cover}} = \{\texttt{Type} \in \{Racing, MTB\},$$
$$\texttt{Typology} \in \{Traditional, Tubeless\},$$
$$\texttt{Diameter} \in \{22, 24, 26, 28, 29\}\}$$

The following are examples of a rule constraint, an integrity constraint, a complete DNF constraint, and a partially unspecified DNF constraint, respectively, that define compatibility relations between node variables of the bicycle product model. We use two tables as shorthands for DNF constraints and we denote the variable `Variable` *of the node* `Node` *by the notation* `Node.Variable`*.*

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

Integrity constraint : **Condition** : `Wheel.Cover` = `Cover.Typology`

Complete DNF constraint :

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

(**Partially unspecified**) **DNF constraint** :

Wheel.Type	*Wheel.Diameter*
MTB	
Racing	*24*
Racing	*26*

As the attentive reader may have noticed, in our (simple) bicycle example cardinality constraints are hard coded in the product model. In general, the encoding of cardinality constraints between components or in the relation *has part/is part of* is one of the major problems to cope with in product configuration. In this paper, we do not address such a problem in its full generality. However, as we will show in Section 4, we use the finite domain constraint solver of SWI Prolog for constraint propagation and solving, and SWI Prolog, as well as other comparable solvers, makes it possible to deal with a number of *global* constraints, such as `alldifferent`, `cardinality`, and `cumulative`. Hence, we can easily extend the syntax of MCE integrity constraints to accommodate built-in global constraints of SWI Prolog. Needless to say, this only partially solves the problem of encoding cardinality constraints, as it does not allow the nesting of global constraints within local ones, that is, only conditions of the form $IC \wedge GC_1 \wedge GC_2 \ldots$, where IC (resp., GC) denote an integrity (resp., a global) constraint, can be expressed. Nevertheless, such a partial solution turns out to be sufficient in many practical cases.

3.2 MCE Configuration Process

The configuration task consists of finding a configured product (*configuration*), that is, a set of customized components together with a description of their relationships, that satisfies all user's preferences on product characteristics and all the relations defined by the product model. This task is accomplished by the user through the *configuration process*. During this interactive process, the user (i) selects the components that will compose the configured product and (ii) chooses suitable values for their configurable characteristics. The configuration system checks the validity of user's choices with respect to the product model and reacts to user's inputs by propagating their effects.

Given the product model defined in Section 3.1, a configuration can be viewed as a set of node instances, whose variables have been assigned a value belonging to their domain, and a tree, called *instance tree*, that specifies the pairs of node instances in the relation *has part/is part of*. In the following, we will use the term *partial configuration* to indicate the set of node instances and the instance tree at an intermediate state of the configuration process, that is, to refer to a configuration under construction. Figure 2 shows the instance tree

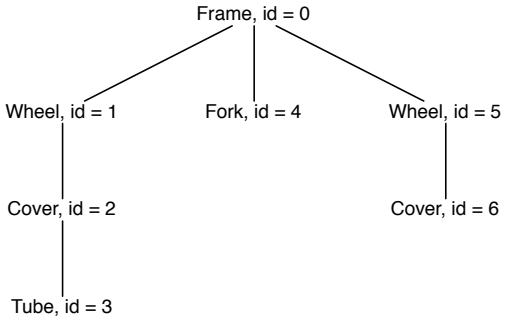

Figure 2: Bicycle: an example of instance tree.

of a partial configuration of a bicycle, where different instances of the same node are identified by a different *id*.

The general structure of the configuration process in Morphos, with a special attention to the role of MCE, is pictorially described in Figure 3. First, Morphos initializes MCE (1) by sending to it information about the

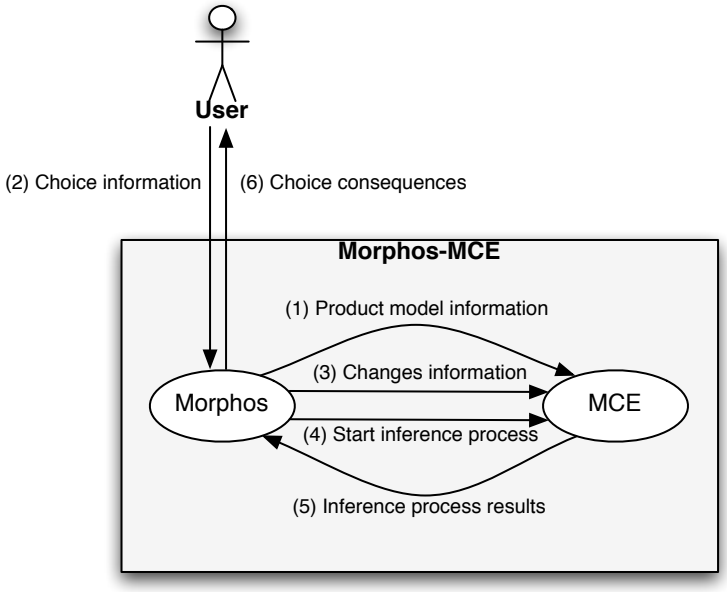

Figure 3: Morphos MCE: the configuration process.

model of the product to be configured, that is, nodes, variables, and constraints defined by the product model. After such an initialization phase, the interaction with the user starts. The user makes her/his choices using the Morphos interface (2). She/he can add or remove node instances (product components) and set the values of node variables (product component characteristics). When adding an instance n_i of a node N_i having a node N_j as its father in the product model tree, the user specifies the instance n_j of N_j that will be the father of n_i in the instance tree. Morphos communicates to MCE each data variation specified by the user (3) and MCE updates the current partial configuration accordingly. Whenever an update of the partial configuration takes place, the user, through the Morphos interface, can activate the MCE inference process (4). MCE encodes the product configuration problem in a CLP program (encoding a CSP), and it uses the finite domain solver of SWI Prolog to propagate the logical effects of user's choices. Once the inference process ends, MCE returns to Morphos the results of its computation (5). In its turns, Morphos communicates to the user the consequences of her/his choices on the (partial) configuration (6).

Figure 5 shows the current interface of the Morphos MCE system. It consists of a series of controls for the configuration and the assignment revision processes (the assignment revision process will be analyzed in Section 6), together with areas for showing data of the current partial configuration. The *Product Model* text box and the *Select* button allow one to select and load the file encoding the product model of the product to be configured. Buttons *Add Node*, *Remove Node*, *Set Value*, and *Unset*, together with the associated text boxes,

allow one to add a node instance, to remove a node instance, to assign a value to a node instance variable, and to remove a variable assignment, respectively. The button *Run* activates the MCE inference process. Buttons *Allowed Values*, *Backjump*, and *Recompute*, together with the associated text boxes, allow one to perform the assignment revision process. The list box *Nodes* shows the node instance created, the list box *Variables* shows the instance variables that have a value assigned. The main box is used to show the results of the inference process, that is, the restricted domains of node instance variables. Product models are encoded in XML files. The interface and the methods realizing the MCE configuration and assignment revision processes have been implemented using the C# language. XML messages are used for data exchanges between the interface and the MCE methods. Files are used for communication purposes between MCE and SWI-Prolog.

The way in which the configuration process takes place makes it clear that the proposed product configuration system is a software tool *supporting* the product configuration activity and not a system for *automatic* product configuration. Hence, it goes without saying that solving a configuration problem requires a constant human-machine interaction.

4 The Encoding of the Product Configuration Problem

Given the product model and a partial configuration (node instances, identified by a node name N and an id i, user's choices for the values of the variables associated with them, and the instance tree), MCE generates an SWI Prolog program, encoding a CSP, to compute the consequences of user's choices on the (partial) configuration. Basically, it applies a transformation function τ that maps (variables, terms, and) constraints into the corresponding entities of a CSP formulated in Constraint Logic Programming.

Constraints on finite domains are formulated according to the concrete syntax of SWI Prolog, which is close to the one provided by similar systems. In Table 1, we show the way in which basic arithmetic operators are denoted in SWI Prolog. Sets defining variable domains are represented using the union operator `\/`, e.g., the set $\{1, 2, 5\}$ is represented as `1 \/ 2 \/ 5`; intervals $[a, b]$ are compactly represented as `a..b`.

Relational symbols:	
Abstract syntax	SWI Concrete Syntax
$=$	`#=`
$\neq$	`#\=`
$<$ $(>)$	`#<` $(>)$
$\leq$ $(\geq)$	`#=<` $(\geq)$

Boolean constructors:	
Abstract syntax	SWI Concrete Syntax
$\neg$	`#\`
$\vee$	`#\/`
$\wedge$	`#/\`
$\Rightarrow$	`#==>`

Table 1: SWI concrete syntax for finite domains operators.

Variables of the CSP correspond to variables of nodes instances belonging to the partial configuration. They are identified by the concatenation of three elements, namely, the variable p of the instance with id i of a node N:

$$\tau(p) = \texttt{N\_i\_p}$$

The Prolog variable `N_i_p` is used as a FD variable. Its domain is the domain $D(V)$ of the original variable V, denoted by $\langle N, i, p\rangle$. With a slight abuse of notation, we substitute $D(N, i, p)$ for $D(V)$. Due to the lack of a tool for string handling in the finite domain solver of SWI Prolog, we define a bijection $B : \mathsf{strings} \rightarrow \mathbb{Z}$ that associates an integer with each string belonging to the domain of a variable of sort string. The domain constraint for `N_i_p` is encoded as follows:

- $\tau(D(N, i, p) = \{i_1, i_2, \ldots, i_n\}) =$ `N_i_p in` i_1 `\/` i_2 `\/ ... \/` i_n, if $i_1, \ldots, i_n$ are integers;
- $\tau(D(N, i, p) = [\min, \max]) =$ `N_i_p in` min `..` max;
- $\tau(D(N, i, p) = \{s_1, s_2 \ldots, s_n\}) =$ `N_i_p in` $B(s_1)$ `\/` $B(s_2)$ `\/ ... \/` $B(s_n)$, if $s_1, \ldots, s_n$ are ASCII strings.

The result of the application of the translation function τ to (terms and) constraints is inductively defined as follows:

- if t is a constant of sort int, then $\tau(t) = t$;
- if t is a constant of sort string, then $\tau(t) = B(t)$;

- if t_1, t_2 are terms of sort int and $op \in \{+, -, *, /, \texttt{mod}\}$, then $\tau(t_1\, op\, t_2) = \tau(t_1)\, op\, \tau(t_2)$ (there is obviously a slight abuse of notation here, as the arithmetical function symbol + denotes both the sum on integers and the sum on finite domains; the same holds for the other operators);
- if $t_1, t_2, \ldots, t_n$ are ground terms of sort string, then $\tau(t_1 \circ t_2 \circ \cdots \circ t_n) = B(t)$, where t is the string concatenation of $t_1, t_2, \ldots, t_n$;
- if $t_1\ op\ t_2$ is a primitive constraint, then $\tau(t_1\ op\ t_2) = \tau(t_1)\, \mathsf{FDop}\, \tau(t_2)$, where FDop is the finite domain version of the binary operator op, that is, `#=` for $=$, `#<` for $<$, and so on, as in Table 1;
- if C_1 is a constraint, then $\tau(\neg C_1) = \texttt{\#\textbackslash}\tau(C_1)$;
- if C_1 and C_2 are constraints and $Boolop$ is the binary Boolean operator $\wedge$ (resp., $\vee, \Rightarrow$), then $\tau(C_1\, Boolop\, C_2) = \tau(C_1)\, \mathsf{FDBoolop}\, \tau(C_2)$, where FDBoolop is the finite domain binary Boolean operator `#/\` (resp, `#\/`,`#==>`).

According to the above definition, constraints defining compatibility relations among properties of product components are mapped into SWI Prolog constraints as follows:

- a rule constraint $C_1 \Rightarrow C_2$ is translated into the SWI Prolog constraint $\tau(C_1)$ `#==>` $\tau(C_2)$;
- an integrity constraint C is translated into the SWI Prolog constraint $\tau(C)$;
- a partially unspecified DNF constraint $C_1 \vee \cdots \vee C_h$ is translated into the SWI Prolog constraint $\tau(C_1)\texttt{\#\textbackslash/}\cdots\texttt{\#\textbackslash/}\tau(C_h)$;
- a complete DNF constraint on variables p_1 of node N_1, p_2 of node N_2, ..., p_k of node N_k ($N_1, \ldots, N_k$ need not to be pairwise distinct), which defines the set of admissible tuples $\{\langle t_{1,1}, \ldots, t_{1,k}\rangle, \ldots, \langle t_{h,1}, \ldots, t_{h,k}\rangle\}$, is translated into the following SWI Prolog DNF constraint:

  `tuples_in([[N1_j1_p1,...,Nk_jk_pk]],[[`$t_{1,1}$`,...,`$t_{1,k}$`],..,[`$t_{h,1}$`,...,`$t_{h,k}$`]])`,

  where `tuples_in/2` is the built-in combinatorial global constraint of SWI Prolog and j_i, for $i = 1, \ldots, k$, is the id of an instance of node N_i.

As the proposed encoding clearly preserves the semantics of the operators, its soundness and completeness immediately follows.

The constraints of the CSP are the elements of the compatibility relation pool, appropriately instantiated with variables of node instances in the partial configuration (as a matter of fact, the choices of the values for the instance variables can be viewed as primitive constraints and thus they are constraints of the CSP as well).

The (constraint) instantiation mechanism works as follows. Let c be an element of the compatibility relation pool, involving variables of different nodes. If the nodes whose variables are involved in c all belong to the same path from the root to a leaf of the product model tree, then c must hold for all the tuples of instance variables that belong to the same path from the root to a leaf of the instance tree (notice that multiple paths in the instance tree may correspond to a single path in product model tree). Otherwise, c must hold for all the ordered tuples of instance variables it involves.

Formally, we define a function μ that, given the instance tree IT and a constraint c of the compatibility relation pool, instantiates c by generating a set of constraints on variables of node instances. To define μ, we preliminarily need to introduce some basic notions. For any variable V, let N_V be the node such that $V \in \mathcal{V}_{N_V}$. Moreover, for any constraint c, let $\texttt{vars}(c)$ be the list of variables occurring in c. Finally, given a list L and an integer i, let $el(L, i)$ be the i-th element of L. The set of nodes whose variables belong to $\texttt{vars}(c)$ is $\texttt{nodes}(c) = \{N_V | V \in \texttt{vars}(c)\}$. A *root path* $rp(T)$ in a (product model or instance) tree T is a sequence of nodes $[N_1, \ldots, N_k]$ such that N_1 is the root of T, N_k is a leaf in T, and, for $1 < i \leq k$, there is an edge from N_{i-1} to N_i. Given a node N of a tree T and a root path $rp(T)$, we say that $N \in rp(T)$ if and only if there exists $i \in \{1, \ldots, k\}$ such that $N = N_i$. Given a node N of the product model tree and a node M of the instance tree, we say that $M \leftrightarrow_{Node} N$ if and only if M is an instance of N (correspondence between nodes and node instances). Given a variable $p \in \mathcal{V}_N$ and a variable $q \in \mathcal{V}_M$, we say that $q \leftrightarrow_{Var} p$ if and only if $M \leftrightarrow_{Node} N$ (correspondence between node variables and node instance variables).

Let PMT and IT be a product model tree and an instance tree, respectively. The function μ is defined as follows:

$$\mu(IT, c) = \begin{cases} \emptyset & \neg(\varphi_{inst}) \\ \{c_I \mid c_I = c[V_1/U_1, \ldots, V_k/U_k], & \\ \quad [V_1, \ldots, V_k] = \texttt{vars}(c), [U_1, \ldots, U_k] \in L_1\} & \varphi_{inst} \wedge \varphi_{rootPath} \\ \{c_I \mid c_I = c[V_1/U_1, \ldots, V_k/U_k], & \\ \quad [V_1, \ldots, V_k] = \texttt{vars}(c), [U_1, \ldots, U_k] \in L_2\} & \varphi_{inst} \wedge \neg(\varphi_{rootPath}) \end{cases}$$

where

$$\begin{aligned}
\varphi_{inst} &\equiv \forall N \in \texttt{nodes}(c)\ \exists M \in IT\ M \leftrightarrow_{Node} N \\
\varphi_{rootPath} &\equiv \exists rp(PMT)\ \texttt{nodes}(c) \subseteq rp(PMT) \\
L_1 &= \left\{ [U_1, \ldots, U_k] \mid \exists rp(IT) \bigwedge_{i=1}^{k} (V_i = el(\texttt{vars}(c), i) \wedge U_i \leftrightarrow_{Var} V_i \wedge M_{U_i} \in rp(IT)) \right\} \\
L_2 &= \left\{ [U_1, \ldots, U_k] \mid \bigwedge_{i=1}^{k} (V_i = el(\texttt{vars}(c), i) \wedge U_i \leftrightarrow_{Var} V_i \wedge M_{U_i} \in IT) \right\}
\end{aligned}$$

The union of the set of instantiated constraints and the set of user's choices on instance variables (abbreviated UC), called *instance constraint pool* (ICP), can be defined in terms of μ as follows:

$$ICP = \{\mu(IT, c) \mid c \in CRP\} \cup UC,$$

where CRP is the compatibility relation pool.

The constraints of the CSP are the elements of the set:

$$\{\tau(c) \mid c \in ICP\}$$

Example 2 *As an example of the application of the function μ, let us consider the following pair of integrity constraints for the bicycle product model and the corresponding instance tree depicted in Figure 1 and Figure 2, respectively.*

Integrity constraint c_1: ***Condition:*** `Wheel.Cover = Cover.Typology`

To compute $\mu(IT, c_1)$, we first check the truth value of φ_{inst} and $\varphi_{rootPath}$. φ_{inst} holds for c_1, as $\texttt{nodes}(c_1) = \{\texttt{Wheel}, \texttt{Cover}\}$ *and, for example, both* $\texttt{Wheel} \leftrightarrow_{Node} \texttt{Wheel}, \texttt{id} = 1$ *and* $\texttt{Cover} \leftrightarrow_{Node} \texttt{Cover}, \texttt{id} = 2$. *$\varphi_{rootPath}$ holds for c_1 as well, as* $[\texttt{Frame}, \texttt{Wheel}, \texttt{Cover}, \texttt{Tube}]$ *is a root path. Hence, to compute $\mu(IT, c_1)$, we must determine the set L_1. Since the instance tree IT features the following two root paths:*

$$[\texttt{Frame}, \texttt{id} = 0, \texttt{Wheel}, \texttt{id} = 1, \texttt{Cover}, \texttt{id} = 2, \texttt{Tube}, \texttt{id} = 3];$$

$$[\texttt{Frame}, \texttt{id} = 0, \texttt{Wheel}, \texttt{id} = 5, \texttt{Cover}, \texttt{id} = 6],$$

we have that:

$$\begin{aligned}
L_1 = \{&[\texttt{Wheel}, \texttt{id} = 1.\texttt{Cover}, \texttt{Cover}, \texttt{id} = 2.\texttt{Typology}], \\
&[\texttt{Wheel}, \texttt{id} = 5.\texttt{Cover}, \texttt{Cover}, \texttt{id} = 6.\texttt{Typology}]\}
\end{aligned}$$

Hence, by definition of μ,

$$\begin{aligned}
\mu(IT, c_1) = \{&\texttt{Wheel}, \texttt{id} = 1.\texttt{Cover} = \texttt{Cover}, \texttt{id} = 2.\texttt{Typology}, \\
&\texttt{Wheel}, \texttt{id} = 5.\texttt{Cover} = \texttt{Cover}, \texttt{id} = 6.\texttt{Typology}\}
\end{aligned}$$

Let us consider now another constraint for the bicycle product model.

Integrity constraint c_2: ***Condition:*** `Fork.Material = Wheel.Material`

The formula φ_{inst} holds, because $\texttt{nodes}(c_2) = \{\texttt{Fork}, \texttt{Wheel}\}$ *and, for example, both* $\texttt{Fork} \leftrightarrow_{Node} \texttt{Fork}, \texttt{id} = 4$ *and* $\texttt{Wheel} \leftrightarrow_{Node} \texttt{Wheel}, \texttt{id} = 1$. *On the contrary, the formula $\varphi_{rootPath}$ does not hold, as the product model tree features no root path containing both node* `Fork` *and node* `Wheel`. *Hence, to compute $\mu(IT, c_2)$, we must determine the set L_2.*

$$\begin{aligned}
L_2 = \{&[\texttt{Fork}, \texttt{id} = 4.\texttt{Material}, \texttt{Wheel}, \texttt{id} = 1.\texttt{Material}], \\
&[\texttt{Fork}, \texttt{id} = 4.\texttt{Material}, \texttt{Wheel}, \texttt{id} = 5.\texttt{Material}]\}
\end{aligned}$$

Hence, by definition of μ,

$$\begin{aligned}
\mu(IT, c_2) = \{&\texttt{Fork}, \texttt{id} = 4.\texttt{Material} = \texttt{Wheel}, \texttt{id} = 1.\texttt{Material}, \\
&\texttt{Fork}, \texttt{id} = 4.\texttt{Material} = \texttt{Wheel}, \texttt{id} = 5.\texttt{Material}\}
\end{aligned}$$

Given a program with the above-described characteristics, the finite domain solver of SWI Prolog can be used to reduce the domains associated with variables, preserving satisfiability, or to detect the inconsistency of the encoded CSP (due to user's assignments that violate the set of constraints or to inconsistencies of the original product model). When the computation of the solver ends, MCE communicates to Morphos the reduced domains and, by exploiting information about (restricted) domains, Morphos can prevent user's choices that would violate the constraints, that is, subsequent user's choices incompatible with previous ones.

5 Propagation Issues and Constraint Optimizations

One of the problems of interactive configurators based on constraint programming resides in the fact that constraint propagation may not remove all inconsistent values from variable domains. This may cause the configuration process to reach a dead end. A number of choices of different nature, including the choice of the solver, the definition of the translation rules, and the specification of the constraint optimization strategies, have been made in order to tackle and at least partially solve this problem. In the following, we describe in some detail the choices we made and we discuss their (positive) effects.

5.1 Binary constraints and DNF constraints

In the first implementation of MCE [7], the inference process was based on the use of the `clpfd` solver of SICStus Prolog. To improve the propagation of binary constraints, we replaced SICStus Prolog by SWI Prolog. The finite domain solver of the latter, indeed, exploits a notion of local consistency which is stronger than the one used by the former. The benefits of such a replacement are illustrated by the following simple example.

Example 3 *Let us consider the goal:*

```
?- X in 0 .. 2 \/ 4 \/ 7 \/ 10 .. 12,
   Y in 0 .. 4 \/ 7 .. 16 \/ 18 \/ 21 \/ 23 \/ 24,
   Y #= 2 * X.
```

The execution of this goal with the finite domain solver of SWI Prolog generates the following result:

```
X in 0..2 \/ 4 \/ 7 \/ 12,
Y in 0..4 \/ 8 \/ 14 \/ 24
```

On the contrary, the execution of an equivalent goal with the `clpfd` *solver of SICStus-Prolog does not produce any reduction in domain variables.*

As for DNF constraints, there exists a straightforward translation of them in SWI Prolog as disjunctions of conjunctions of primitive constraints of the form `X #= v`, where `X` is a variable and `v` is an integer. Unfortunately, the propagation of such constraints may not produce the desired reduction in variable domains due to the presence of disjunctions (the problems with disjunctive constraints are discussed in detail in the next section). This is the reason why we chose to encode complete DNF constraints by means of the constraint `tuples_in/2`, whose propagation reduces variable domains forcing arc consistency.

5.2 Disjunctive constraints

As illustrated in Section 4, node constraints of the form $C_1 \vee C_2$ are mapped into SWI Prolog constraints of the form `C_1 #\/ C_2`. The same translation applies to partially unspecified DNF constraints. For efficiency reasons, constraint solvers do not fully exploit disjunction for domain pruning during constraint propagation. Consequently, the presence of the operator `#\/` in SWI Prolog constraints amplifies the problem of lack of propagation, as shown by the following example.

Example 4 *Let us consider the following goal*

```
?- A in 1..3,
   B in 4..6,
   A #= 1 #==> B #= 5 #\/ B #= 6,
   A #= 1.
```

The execution of this goal with the finite domain solver of SWI Prolog produces the following result:

```
A = 1,
B in 4..6
```

Hence, constraint propagation has not removed the value 4 from the domain of `B`.

However, if we replace the constraint `A #= 1 #==> B #= 5 #\/ B #= 6` by the equivalent constraint `A #= 1 #==> B in 5..6` in the goal of Example 4, the execution returns `B in 5..6`, that is, the expected reduction of the domain for the variable `B`. Such an observation led us to formulate a strategy for manipulating constraints and automatically adding (possibly redundant) implications that may improve the effect of constraint propagation on variable domains. In the following, let us denote by $f^{\wedge}, g^{\wedge}$ and $f^{\vee}, g^{\vee}$ conjunctions and disjunctions of primitive constraints, respectively. The proposed strategy can be described as follows.

Let $F \Rightarrow G$ be a rule constraint belonging to the product model. We proceed as follows:

1. first, we rewrite F in disjunctive normal form (DNF): $f_1^{\wedge} \vee f_2^{\wedge} \vee \ldots \vee f_n^{\wedge} \equiv F$;
   then, we rewrite G in conjunctive normal form (CNF): $g_1^{\vee} \wedge g_2^{\vee} \wedge \ldots \wedge g_m^{\vee} \equiv G$;
   finally, we replace $F \Rightarrow G$ by $f_1^{\wedge} \vee f_2^{\wedge} \vee \ldots \vee f_n^{\wedge} \Rightarrow g_1^{\vee} \wedge g_2^{\vee} \wedge \ldots \wedge g_m^{\vee}$. Such a rule constraint is in its turn equivalent to the set of rule constraints:
   $$R_{F \Rightarrow G} = \{f_i^{\wedge} \Rightarrow g_j^{\vee} \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}$$

2. We do the same with its counterpositive $\neg G \Rightarrow \neg F$. Let $\bar{G} \equiv \neg G$ and $\bar{F} \equiv \neg F$. We proceed as follows:
   we rewrite $\bar{G}$ in DNF as $\bar{g}_1^{\wedge} \vee \bar{g}_2^{\wedge} \vee \ldots \vee \bar{g}_h^{\wedge}$;
   we rewrite $\bar{F}$ in CNF as $\bar{f}_1^{\vee} \wedge \bar{f}_2^{\vee} \wedge \ldots \wedge \bar{f}_k^{\vee}$;
   we accordingly rewrite $\bar{G} \Rightarrow \bar{F}$ as $\bar{g}_1^{\wedge} \vee \bar{g}_2^{\wedge} \vee \ldots \vee \bar{g}_h^{\wedge} \Rightarrow \bar{f}_1^{\vee} \wedge \bar{f}_2^{\vee} \wedge \ldots \wedge \bar{f}_k^{\vee}$. As before, such a rule constraint is equivalent to the set of rule constraints:
   $$R_{\bar{G} \Rightarrow \bar{F}} = \{\bar{g}_i^{\wedge} \Rightarrow \bar{f}_j^{\vee} \mid 1 \leq i \leq h,\ 1 \leq j \leq k\}$$

3. Let us consider now the rule constraints belonging to the set $R_{F \Rightarrow G}$ or to the set $R_{\bar{G} \Rightarrow \bar{F}}$. The right-hand-side of each of them is a disjunction of the form:
   $$\ell_1 \vee \ell_2 \vee \ldots \vee \ell_w,$$
   where ℓ_i, with $1 \leq i \leq w$, is a primitive constraint. To each of these disjunctions, we apply simplification rules like the following ones:
   - if there exists a pair ℓ_i, ℓ_j such that ℓ_i is equal to $V = n$ and ℓ_j is equal to $V \neq n$, where n is an element of the domain of the variable V, then we replace the disjunction by the constraint `true` which is always satisfied;
   - if there exists a pair ℓ_i, ℓ_j such that ℓ_i is equal to $V \neq u$ and ℓ_j is equal to $V \neq v$, where u and v are two distinct elements of the domain of the variable V, then we replace the disjunction by a constraint `true` which is always satisfied;
   - if for all $i \in I \subseteq \{1, \ldots, w\}$, with $|I| > 2$, ℓ_i is equal to $V = n_i$, where n_i is an element of the domain of the variable V, we rewrite $\bigvee_{i \in I} V = n_i$ as the equivalent constraint $V \in \{n_i \mid i \in I\}$ (such a constraint can be translated into an SWI Prolog constraint that exploits the membership constraint `in`) and replace the disjunction by the constraint $\bigvee_{i \notin I} \ell_i \vee V \in \{n_i \mid i \in I\}$.

   The outcome of the process consists of two simplified sets of rule constraints $R^S_{F \Rightarrow G}$ and $R^S_{\bar{G} \Rightarrow \bar{F}}$, which are paired with the original rule constraint $F \Rightarrow G$.

Such a constraint optimization strategy is applied to all rule constraints. Together with the other constraints defined by the product model, the resulting sets of rule constraints contribute to the definition of the CSP encoding the configuration problem under consideration. We experimentally verified that their presence often considerably improves the effects of constraint propagation on domains of variables involved in rule constraints of the form $F \Rightarrow G$. Consider, for instance, the following example.

Example 5 *Let P be a Prolog program encoding a CSP and let $V = \{X_i | 1 \leq i \leq n\} \cup \{Y_i | 1 \leq i \leq n\}$ be its set of variables. Moreover, let the integer interval [1,1000] be the domain of each variable in V. Finally, let each pair of variables X_i, Y_i, with $1 \leq i \leq n$, satisfy the following rule constraints:*

$$\left(\begin{array}{l} X_i \leq 100 \vee (X_i > 200 \wedge X_i < 300) \vee \\ (X_i > 400 \wedge X_i < 500) \vee (X_i > 600 \wedge X_i < 700) \vee \\ (X_i > 800 \wedge X_i < 900) \end{array}\right) \Rightarrow \left(\begin{array}{l} (Y_i > 150 \wedge Y_i \leq 200) \vee \\ (Y_i \geq 350 \wedge Y_i \leq 400) \vee \\ (Y_i \geq 750 \wedge Y_i \leq 800) \end{array}\right)$$

$$\left(\begin{array}{l}(X_i > 100 \wedge X_i \leq 200) \vee (X_i \geq 300 \wedge X_i \leq 400) \vee \\ (X_i \geq 500 \wedge X_i \leq 600) \vee (X_i \geq 700 \wedge X_i \leq 800) \vee \\ X_i \geq 900\end{array}\right) \Rightarrow \left(\begin{array}{l}Y_i \leq 50 \vee Y_i > 850 \vee \\ (Y_i > 250 \wedge Y_i < 300) \vee \\ (Y_i > 450 \wedge Y_i < 500)\end{array}\right)$$

If we apply the proposed optimization strategy to P, we obtain a new program P_{opt} that includes 95 additional rule constraints for each pair of variables X_i, Y_i (60 of them are obtained from the first constraint, the other 35 are obtained from the second one). These new rule constraints considerably improve the effects of constraint propagation on variable domains. Let $n = 400$. To find a solution to the CSP, we execute a labeling procedure on the variables in V. Computing a solution with the program P requires about 11 seconds, with the program P_{opt} only 4 seconds (tests are performed with SWI-Prolog 5.8.3 on a MacBook with a 2.4 GHz Intel Core 2 Duo, 4 GB of RAM, running Mac OS X version 10.6.3). The reduction in computation time originates from a more consistent pruning of variable domains and, consequently, of the search tree that the additional constraints make it possible.

Obviously, the proposed optimization strategy generates a new set of constraints which is equivalent to the set of constraints defined by the product model. For any rule constraint $F \Rightarrow G$, indeed, the proposed strategy introduces two sets of new rules which, by construction, are equivalent to the original constraint $F \Rightarrow G$, and it does not remove any constraint from the set of constraints defined by the product model. Moreover, such an optimization strategy can be directly applied to the rule constraints of the product model in an off-line phase, thus without affecting the performance of the configuration process.

6 MCE Assignment Revision Process

In this section, we describe the process of assignment revision that plays an important role in the Morphos MCE system. This process copes with two situations that may occur during the configuration process: (i) the user changes her/his preferences; (ii) a dead end is reached because constraint propagation has not removed all inconsistent values from variable domains. To deal with such situations, Morphos MCE allows the user to modify past choices about property values. Such a functionality is supported by the assignment revision process that allows the user to *jump back* from a dead end or a (partial) configuration that no longer satisfies her/his preferences. Figure 4 shows how Morphos MCE manages the assignment revision process. We illustrate such

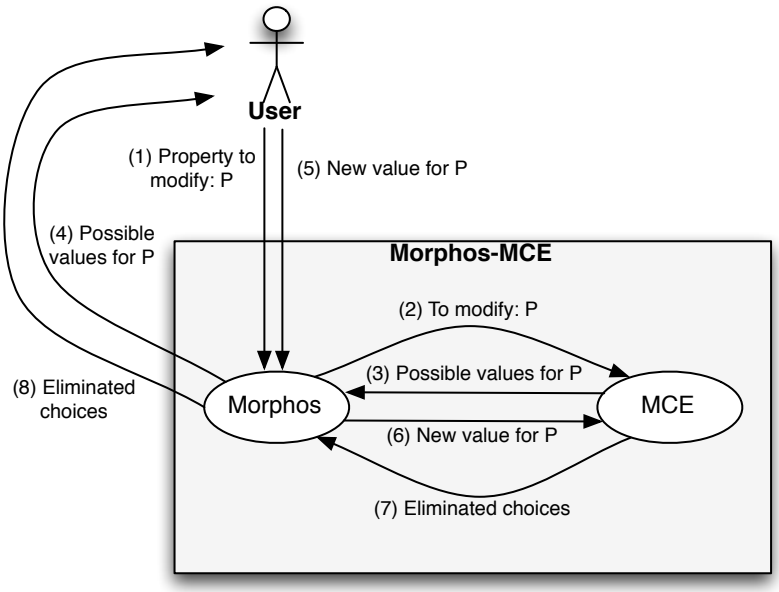

Figure 4: Morphos MCE: the assignment revision process.

a capability of the system by means of a simple example. Let us assume that the user would like to modify the value he/she previously assigned to variable `P`. The user selects variable `P` through the Morphos interface (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 generating any conflict with constraints and choices made before the one concerning variable `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 introducing 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 her/his 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.

We now describe how MCE takes advantage of the finite domain solver of SWI Prolog during the assignment revision process. Let us consider again the case when the user would like to modify the value he/she assigned to variable `P`. To compute the set of alternative values for `P` that generate no conflict with constraints and choices made before the one concerning `P`, MCE defines an SWI Prolog program encoding a CSP as in the configuration process, but without considering the primitive constraints representing choices on variable 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 an SWI Prolog program encoding a CSP as in the configuration process, but with the following additional features. A variable `B_i` with domain `0 \/ 1` is associated with each assignment to variable 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 `B_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`). A maximal subset of the assignments A_i that maintains the configuration consistent is computed using a predicate of the form `labeling([max(B_1 + B_2 + ... + B_m)],[B_1,B_2,...,B_m])`. Such a predicate, through a branch-and-bound algorithm, allows one to compute an assignment of values for variables `B_1,B_2,...,B_m` that maximizes the sum `B_1 + B_2 + ... + B_m`. The computed assignment determines one maximal subset among all possible ones.

We conclude the section with an example of the execution of a backjump during the configuration of a bicycle.

Example 6 *Let us consider the configuration process for a bicycle whose product model is the one given in Example 1. Moreover, let us assume that the user defined a partial configuration whose instance tree is the one depicted in Figure 2 and made the following assignments in the order in which they are presented:*

$$\texttt{Frame}, \texttt{id} = 0.\texttt{Material} = \texttt{Carbon}$$
$$\texttt{Fork}, \texttt{id} = 4.\texttt{Type} = MTB$$
$$\texttt{Fork}, \texttt{id} = 4.\texttt{Material} = Aluminium$$
$$\texttt{Fork}, \texttt{id} = 4.\texttt{ShockAbsorber} = 80mm$$

These assignments result in the partial configuration depicted in Figure 6 (through the Morphos MCE interface). Let us consider now a possible backjump on the variable ***`Fork,id=4.Type`****. It basically consists of the following steps:*

1. *the user communicates to Morphos that he/she wants to change her/his choice for variable* ***`Type`*** *of node* ***`Fork,id=4`****;*
2. *Morphos shows to the user the values he/she can assign to* ***`Fork,id=4.Type`*** *without generating any conflict with constraints and previous choices, that is,* $\{Racing, MTB\}$ *(see Figure 7);*
3. *the user chooses Racing as the new value for* ***`Fork,id=4.Type`****;*
4. *Morphos communicates to the user that the choice on* ***`Fork,id=4.ShockAbsorber`*** *must be withdrawn to keep the configuration consistent (see Figure 8);*
5. *the user accepts the elimination of the assignment* $\texttt{Fork}, \texttt{id} = 4.\texttt{ShockAbsorber} = 80mm$ *and he/she proceeds with the configuration of the bicycle.*

7 A comparison with existing product configuration tools

In this section, we briefly compare MCE with the most important existing configuration tools to put in evidence its strength and limitations. In addition, we evaluate its behavior on some benchmarks for product configuration and we contrast it with those of other systems.

Existing configuration tools can be partitioned in three main classes, namely, ASP-based systems, BDD-based systems, and CSP-based systems (the class MCE belongs to).

ASP-based systems provide a number of features that are specifically tailored to the modeling of software product families. On the one hand, this makes these systems appealing for a relevant set of application domains. On the other hand, it results in a lack of generality, which is probably the major drawback of this class of systems. In particular, they support neither DNF constraints nor global constraints, and they encounter some problems in the management of arithmetic constraints related to the grounding stage. Let us consider the case of Kumbang Configurator, which is probably the most significant system in this class. Kumbang Configurator provides a domain ontology to model variability in software product families, that unifies feature-based and architecture-based approaches. In addition, it allows the user to model the interrelations between these two views. A precise characterization of the main features of Kumbang Configurator has been given in UML [5]. As for its limitations, Kumbang Configurator lacks various natural constructs for product modeling. As an example, in Kumbang Configurator a domain of the form $D(V) = [\min, \max]$, as the one of variable `SpokesNumber` of node `Wheel` in Example 1, can be defined only by explicitly enumerating its elements. Moreover, unlike MCE, Kumbang Configurator does not provide any mechanism for assignment revision. Let us consider the case of a user that would like to change the value he/she assigned to a property `P`. If the new value for `P` makes the configuration inconsistent, the user must find consistent assignments not only for properties to which he/she previously assigned a value, but also for properties that have been instantiated by Kumbang Configurator through constraint propagation.

BDD-based systems, like Configit Product Modeler, CLab, CLab#, and iCoDE, trade the complexity of the construction of the BDD, that basically provides an encoding of all possible configurations, for the simplicity and efficiency of the configuration process. Unlike CLab and iCoDE, Configit Product Modeler also allows one to change previously-assigned values. The mechanism for assignment revision it offers behaves as follows: if the new value chosen for a given product characteristic makes other choices no more valid, then the system asks the user whether or not he/she wants to maintain such a new value and remove invalid choices. Despite their various appealing features, BDD-based systems suffer from some significant limitations. First, even though some work has been done on the introduction of modules, they basically support flat models only. Moreover, they find it difficult to cope with global constraints. As an example, `alldifferent` constraints lead to exponential space complexity for BDD construction. As a matter of fact, some attempts at combining BDD with CSP to tackle `alldifferent` constraints have been recently done; however, they are confined to the case of flat models. We are not aware of any BDD system that deals with global constraints in a sufficiently general and satisfactory way. The most significant configuration system in this class is Configit Product Modeler [9]. Compared to such a system, MCE makes it possible to define a larger set of compatibility relations between product component characteristics. As an example, MCE allows one to compare a characteristic with an expressions, while Configit Product Modeler only allows comparisons with constants. For instance, given two product component characteristics X and Y, MCE allows one to define the integrity constraint **Condition:** $\texttt{X} = 2 \cdot \texttt{Y}$. This is not possible with Configit Product Modeler. Another limitation of Configit Product Modeler concerns the management of multiple instances of a product component. The tool allows one to define product components whose number of instances in a given configuration can vary depending on some constraints by means of the so-called modules. However, such a capability suffers from a severe limitation: it is not possible to define a constraint involving characteristics associated with different modules in the product model. As a consequence, the consistency of a configuration with respect to such a constraint can only be ensured by the runtime application that uses the BDD to support the user in the search for a valid configuration. On the contrary, constraints over arbitrary sets of characteristics can be easily implemented in MCE, thus fully preserving the declarative nature of the modeling process.

Unlike the systems belonging to the other two classes, CSP-based ones allow the user to define non-flat models and to deal with global constraints. Unfortunately, backtrack-free configuration algorithms for CSP-based systems are often inefficient, while non backtrack-free ones need to explicitly deal with dead ends. Moreover, most CSP-based systems do not offer high-level modeling languages (product models must be specified at the CSP level). Some well-known CSP-based configuration systems, such as ILOG Configurator [17], which features various interesting modeling facilities, and Lava [11], which is based on Generative-CSP, seem to be no longer supported. As all CSP-based systems, MCE makes it possible to define non-flat models. Its modeling support turns out to be simple enough and easily extensible (addition of global constraints, definition of specific optimization strategies, etc.). Its configuration algorithm is not backtrack-free, but it exploits back-jumping capabilities, to cope with dead ends, and branch-and-prune capabilities, to improve domain reduction.

As for the experimental evaluation, we compared the computational performances of the current implementation of MCE with three BDD-based configuration systems, namely, CLab, iCoDE and Configit. Four

Benchmark	Variables	Constraints	Solutions
FS	23	26	2.4×10^7
ESVS	26	21	3.5×10^9
PC	45	36	1.1×10^6
Renault	99	113	2.8×10^{12}

Table 2: Configuration benchmarks

Benchm.	Compilation				Query			
	MCE	CLab	iCoDE	Configit	MCE	CLab	iCoDE	Configit
FS	0.06	0.01	0.02	0.25	0.37	0	0	0.0003
ESVS	0.08	0.01	0.03	0.25	0.39	0	0	0.0004
PC	0.06	0.17	0.12	0.89	0.39	0	0	0.0007
Renault	7.62	119	45	460	8.77	0.07	0	0.1273

Table 3: Comparison of compilation (resp., query) times (in seconds), where 0 stands for less than 0.0001.

configuration instances (available from [8]) have been used in the experiments: FS and ESVS (two screw compressor configuration problems), PC (a personal computer configuration problem), and Renault [3] (a car configuration problem). Table 2 illustrates the basic characteristics of these benchmarks.

The outcomes of the experiments are reported in Table 3. The column "Compilation" refers to the time taken by MCE to generate a Prolog program from a product model, and to the time taken by the other three systems to generate a BDD from a product model. The column "Query" refers to the time taken by a system to determine valid choices for model variables on the basis of a user choice. In MCE, this is done by executing the Prolog program, in the other three systems by operating on the BDD. All time measurements are given in seconds. Tests on MCE have been performed on a VMware Fusion virtual machine running Windows 7 Ultimate, with 1 CPU and 2 GB of RAM. Tests on CLab and iCoDE have been performed on a VMware Fusion virtual machine running Ubuntu 10.4, with 1 CPU and 2 GB of RAM. Both virtual machines run on a MacBook with a 2.4 GHz Intel Core 2 Duo, 4 GB of RAM, running Mac OS X version 10.6.4. We used two different virtual machines since MCE only runs under Windows, and CLab and iCoDE only run under Linux.

Timings for Configit have been taken from [38], since the models available in [8] are not compatible with the current version of Configit, which is a proprietary software whose trial version has some significant limitations, e.g., the product model compiler handles product models with at most 50 variables only. Timings for CLab on the Renault car configuration benchmark have been taken from [37], since using the above cited virtual machine we obtained an "out of memory" error at the end of the compilation phase. For the other examples, the running times on the PC we used are very close to those reported in [37, 38] and therefore we decided to report them all together in the same table.

From Table 3, we notice that MCE compilation times are comparable to the ones of the other three systems on benchmarks FS, ESVS, and PC, and much better on the Renault benchmark. On the contrary, MCE query times are always worse than the ones of the other systems, but still acceptable for an interactive configuration tool. As a matter of fact, the current implementation of MCE (re)compiles the product model every time the inference process is activated, and thus query evaluation is always preceded by compilation. However, we expect to release soon a new version of MCE that compiles the product model only once. Moreover, it is worth pointing out that the considered benchmarks do not show the distinctive modeling capabilities of MCE, as all of them are essentially flat. We chose these benchmarks for the sake of comparison: they are all publicly available and they are compatible with the other configuration systems.

8 Conclusions

In this paper, we applied the Constraint Logic Programming (CLP) approach to product configuration to the construction of a configuration engine, called MCE, which has been incorporated in the product configuration system Morphos, developed by Acritas S.r.l. Besides an illustration of the distinctive features of MCE and an intuitive account of its behavior, we provided a detailed analysis of choices and optimization strategies we adopted to tackle a common problem in CP-based configuration systems, namely, the limits of built-in constraint propagation. A simple case of product configuration referring to the domain of bicycles has been used as a source of exemplification throughout the paper.

We are extending the work in two main directions. On the one hand, we are improving MCE functionalities (addition of structured cardinality constraints, that, as we noticed in Section 3.1, are not managed by the current system, explicit introduction of associative relations, which can be currently mimicked through constraints over relevant variables, treatment of real number constraints, formalization of global constraints that can be dealt with by SWI Prolog, computational improvements, speed-up of the communications with Morphos, etc.). On the other hand, we are exploring the possibility of pairing product configuration with process configuration within the same framework/system and of providing an explicit module for the definition of product/process model (to be used, for instance, to guarantee the consistency of the product/process model). We would like to briefly discuss three of these extensions, namely, treatment of real number constraints, computational improvements, and process configuration.

In the current version of MCE, a sort real for real numbers is not included. As a matter of fact, the configuration problems to which Morphos has been applied so far concern products depending on mechanical specifications that define a fixed precision for numerical characteristics. This allowed us to restrict our attention to numbers with a fixed number of decimal digits. Integers clearly suffice for the management of these numbers. If the number of digits after the radix point is low, one can use a single integer for representing a fixed-point number. Such a solution is inappropriate for fixed-point numbers with a high number of decimal digits. In such a case, indeed, by multiplying integer numbers representing fixed-point numbers one can exceed the max-integer of the finite domain solver. To avoid this problem, one can use two integers for representing a fixed-point number, one for the integer part and one for the decimal part of the number, respectively. The only drawback of such a solution is that it requires the re-implementation of all the arithmetic operations (the first solution requires the re-implementation of division and multiplication only). To broaden the range of applications of Morphos, we are considering the possibility of adding the ability of dealing with real numbers to MCE. To this end, we had a look at existing solutions. One of them is provided by the Interval Constraint (IC) library of ECL^{i}PS^e [4], an hybrid integer/real interval arithmetic constraint solver. Unfortunately, the IC implementation of the fundamental algorithm HC3 suffers from too many floating-point approximation errors. Another one is given by the system $\mathcal{TOY}$ [10], that, unlike the finite domain solver of SWI Prolog, allows one to solve CSP involving both finite domain variables and real variables. However, the current version of this system seems to be still very preliminary. As for computational improvements, we plan to improve the CLP program generation phase, by making it possible to reuse already computed data and by avoiding the use of files for the communication with SWI-Prolog. We are also considering the possibility of extending Morphos MCE with the ability of automatically building a complete solution without any user assistance. In its current version, Morphos MCE is a semi-automatic configurator, that is, it supports basic functionalities such as consistency checking, domain-pruning, and assignment revision, but it assigns an active role to the user that progressively completes his/her choices. However, taking advantage of the `labeling` Prolog built-in, it is always possible to automatically complete a partial assignment. We plan to explicitly add such an option, paired with the possibility of expressing preferences among different solutions. As for the extension of Morphos MCE with process configuration facilities, we observe that process configuration and planning have attracted a increasing amount of interest in the last years (see, for example, [19, 28, 31, 43]). In particular, Aldanondo et al. have recently contributed important papers devoted to the coupling of product and process configuration [1, 2]. The addition of process modeling and configuration capabilities to Morphos MCE is at one of the topmost positions in our agenda. One of the requirements we would like to meet is to maintain a clean separation between the concept of product and that of process. To this end, the use of representation techniques inspired by Upper-Level Ontologies seems to be particularly appropriate [21].

Acknowledgments

The work has been partially funded by a grant from the Friuli Venezia Giulia region within the project: "*Techniques and algorithms for the representation and manipulation of knowledge in configuration systems*". Agostino Dovier and Angelo Montanari have also been financially supported by the PRIN project "*Innovative and multi-disciplinary approaches for constraint and preference reasoning*". Agostino Dovier and Dario Campagna are also partially supported by the grant: GNCS-INdAM *Tecniche innovative per la programmazione con vincoli in applicazioni strategiche*. We would like to acknowledge the contribution of Andrea Calligaris to the development and implementation of MCE. We would also like to thank the anonymous reviewers that helped us a lot in improving the paper.

References

[1] Aldanondo, M., Vareilles, E.: Configuration for mass customization: how to extend product configuration towards requirements and process configuration, *Journal of Intelligent Manufacturing*, **19**(5), 2008, 521–535.

[2] Aldanondo, M., Vareilles, E., Djefel, M., Gaborit, P.: Towards an association of product configuration with production planning, in: *Proceedings of ECAI 2008 Workshop on Configuration Systems*, University of Patras, 2008, 41–46.

[3] Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs–Application to configuration, *Artificial Intelligence*, **135**(1-2), 2002, 199 – 234.

[4] Apt, K., Wallace, M.: *Constraint Logic Programming using Eclipse*, Cambridge University Press, 2006.

[5] Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modeling variability in software product families, *Adv. Eng. Inform.*, **21**(1), 2007, 23–40.

[6] Barker, V. E., O'Connor, D. E., Bachant, J., Soloway, E.: Expert systems for configuration at Digital: XCON and beyond, *Communications of the ACM*, **32**(3), 1989, 298–318.

[7] Calligaris, A., Campagna, D., De Rosa, C., Dovier, A., Montanari, A., Piazza, C.: A CLP engine for a general purpose configuration tool, in: *Proc. of the 24th Italian Conference on Computational Logic (CILC)*, University of Ferrara, Italy, 2009.

[8] CLA group, IT-University of Copenhagen: Clib: configuration benchmarks library, `http://www.itu.dk/research/cla/externals/clib/`.

[9] Configit A/S, http://www.configit.com/products/configit_product_modeler.html: *Configit Product Modeler*, 2009.

[10] Estévez-Martín, S., Fernández, A., Hortalá-González, T., Rodríguez-Artalejo, M., Sáenz-Pérez, M., del Vado-Vírseda, R.: A proposal for the cooperation of solvers in constraint functional logic programming, *ENTCS*, **188**, 2007, 37–51.

[11] Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., Stumptner, M.: Configuring Large Systems Using Generative Constraint Satisfaction, *IEEE Intelligent Systems*, **13**(4), 1998, 59–68.

[12] Frayman, F., Mittal, S.: COSSACK: A Constraint-Based Expert System for Configuration Tasks, *Knowledge-Based Expert Systems in Engineering: Planning and Design*, 1987, 143–166.

[13] Freuder, E., Likitvivatanavong, C., Moretti, M., Rossi, F., Wallace, R.: 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.

[14] Friedrich, G., Stumptner, M.: Consistency-based configuration, in: *Proc. of the AAAI'99 Workshop on Configuration*, AAAI Press, 1999, 35–40.

[15] Hadzic, T., Subbarayan, S., Jensen, R. M., Andersen, H. R., Moller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation, in: *Proc. of the International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems*, 2004, 131–138.

[16] Juengst, W. E., Heinrich, M.: Using resource balancing to configure modular systems, *IEEE Intelligent Systems*, **13**(4), 1998, 50–58.

[17] Junker, U.: The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic, in: *Proc. of the IJCAI'03 Workshop on Configuration*, 2003, 13–20.

[18] Junker, U.: *Handbook of Constraint Progrmming*, chapter 24. Configuration, Elsevier, 2006, 837–873.

[19] Kulvatunyou, B., Wysk, R. A., Cho, H., Jones, A.: Integration framework of process planning based on resource independent operation summary to support collaborative manufacturing, *Int. J. Computer Integrated Manufacturing*, **17**(5), 2004, 377–393.

[20] Martins, J. P.: The truth, the whole truth, and nothing but the truth: An indexed bibliography to the literature of truth maintenance systems, *AI Mag.*, **11**(5), 1991, 7–25.

[21] Mascardi, V., Locoro, A., Rosso, P.: Automatic Ontology Matching via Upper Ontologies: A Systematic Evaluation, *IEEE Trans. Knowl. Data Eng.*, **22**(5), 2010, 609–623.

[22] McDermott, J. P.: R1: A rule-based configurer of computer systems, *Artificial Intelligence*, **19**, 1982, 39–88.

[23] McGuinness, D. L., Wright, J. R.: Conceptual modelling for configuration: A description logic based approach, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12**(4), 1998, 333–344.

[24] Mittal, S., Falkenhainer, B.: Dynamic Constraint Satisfaction Problems, in: *Proc. of the 8th National Conference on Artificial Intelligence (AAAI)*, 1990, 25–32.

[25] Mittal, S., Frayman, F.: Making Partial Choices in Constraint Reasoning Problems, *Proc. of the 6th National Conference on Artificial Intelligence (AAAI)*, 1987, 631–636.

[26] Mittal, S., Frayman, F.: Towards a generic model of configuration tasks, in: *Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, 1989, 1395–1401.

[27] Myllärniemi, V., Asikainen, T., Männistö, T., Soininen, T.: Kumbang configurator - a configurator tool for software product families, in: *Proc. of the IJCAI'05 Workshop on Configuration*, 2005, 51–56.

[28] Nielsen, J., Kimura, F.: A Resource Capability Model to Support Product Family Analysis, *JSME International Journal Series C Mechanical Systems, Machine Elements and Manufacturing*, **49**(2), 2006, 568–575.

[29] Sabin, D., Freuder, E. C.: Configuration as Composite Constraint Satisfaction, in: *Proc. of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop* (G. F. Luger, Ed.), AAAI Press, 1996, 153–161.

[30] Sabin, D., Weigel, R.: Product configuration frameworks - a survey, *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, **13**(4), Jul/Aug 1998, 42–49.

[31] Schierholt, K.: Process configuration: Combining the principles of product configuration and process planning, *Artif. Intell. Eng. Des. Anal. Manuf.*, **15**(5), 2001, 411–424.

[32] Simons, P., Niemelä, H., Soininen, T.: Extending and implementing the stable models semantics, *Artificial Intelligence*, **138**(1–2), 2002, 181–234.

[33] Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **17**(1), 2003, 75–97.

[34] Soininen, T., Gelle, E., Niemelä, I.: A fixpoint definition of dynamic constraint satisfaction, in: *Proc. of the 5th International Conference on Principles and Practice of Constraint Programming*, 1999, 419–433.

[35] Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Unified configuration knowledge representation using weight constraint rules, in: *Proc. of the ECAI'00 Workshop on Configuration*, 2000, 79–84.

[36] Stumptner, M., Friedrich, G., Haselbök, A.: Generative constraint based configuration of large technical systems, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12**(4), 1998, 307–320.

[37] Subbarayan, S.: Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems, in: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (R. Barták, M. Milano, Eds.), vol. 3524 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2005, 351–365.

[38] Subbarayan, S., Jensen, R. M., Hadzic, T., Andersen, H. R., Hulgaard, H., MÃ¸ller, J.: Comparing Two Implementations of a Complete and Backtrack-Free Interactive Configurator, in: *Proc. of CSPIA Workshop, at CP'04*, 2004, 97–111.

[39] Swedish Institute of Computer Science, Intelligent Systems Laboratory: *SICStus Prolog User's Manual*, 4.0.3 edition, May 2008.

[40] Thorsteinsson, E. S., Ottosson, G.: Linear relaxations and reduced-cost based propagation of continuous variable subscripts, *Annals of Operations Research*, **115**, 2002, 15–29.

[41] Tseng, H.-E., Chang, C.-C., Chang, S.-H.: Applying case-based reasoning for product configuration in mass customization environments, *Expert Systems with Applications*, **29**(4), 2005, 913–925.

[42] University of Amsterdam, Human-Computer Studies: *SWI-Prolog 5.6 Reference Manual*, August 2008.

[43] Zheng, L., Dong, H., Vichare, P., Nassehi, A., Newman, S.: Systematic modeling and reusing of process knowledge for rapid process configuration, *Robotics and Computer-Integrated Manufacturing*, **24**(6), 2008, 763 – 772.

A Morphos MCE Interface

In this appendix, we include some screenshots of the current version of the Morphos MCE system.

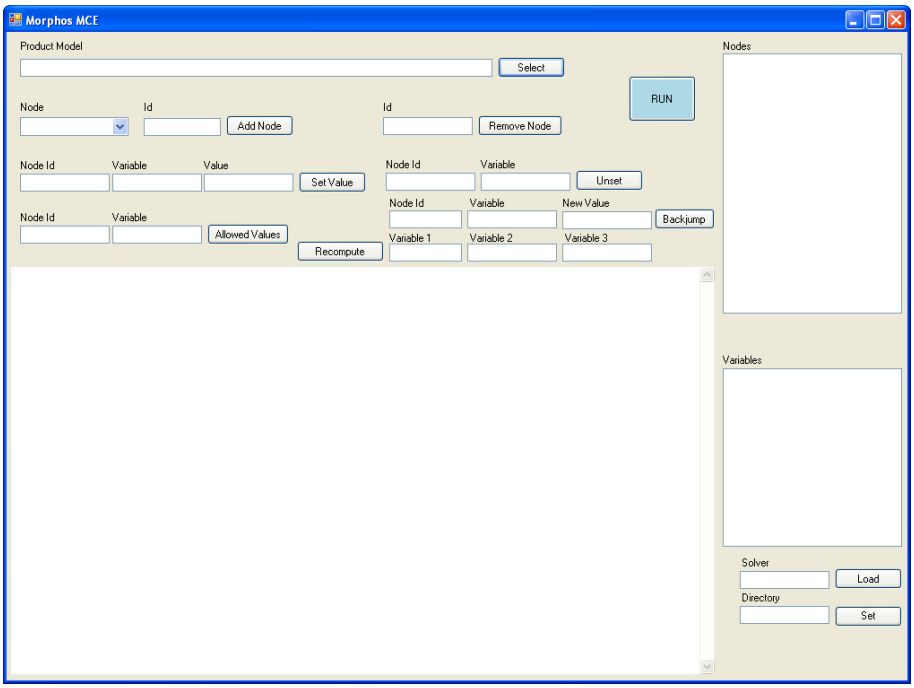

Figure 5: The current interface of the system.

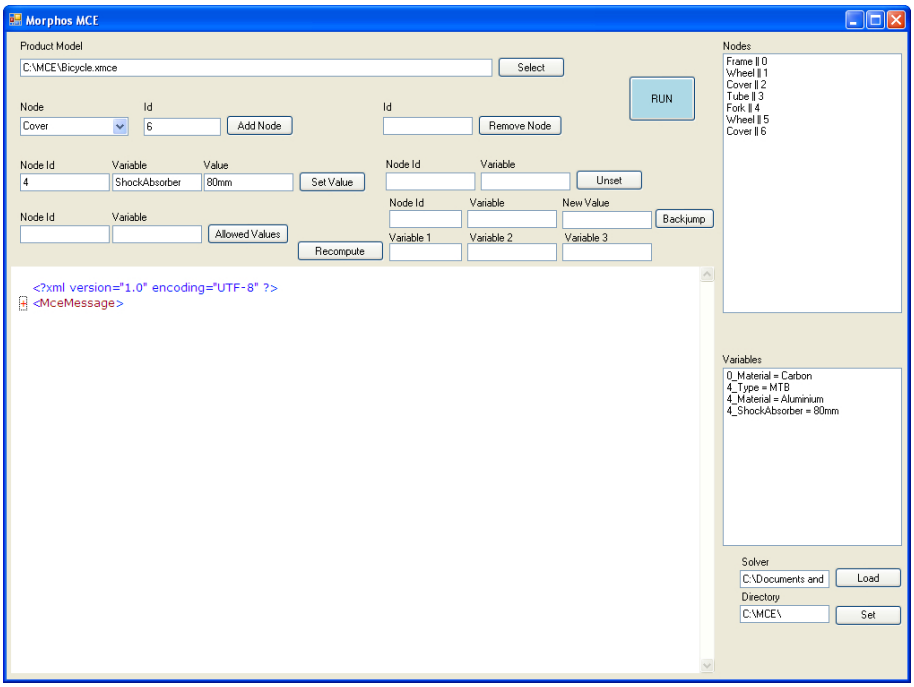

Figure 6: The partial configuration of the bicycle once the assignments listed in Example 6.1 have been taken into consideration. Nodes belonging to the current instance tree are listed in box *Nodes*, while user's assignments to node instance variables are listed in box *Variables*.

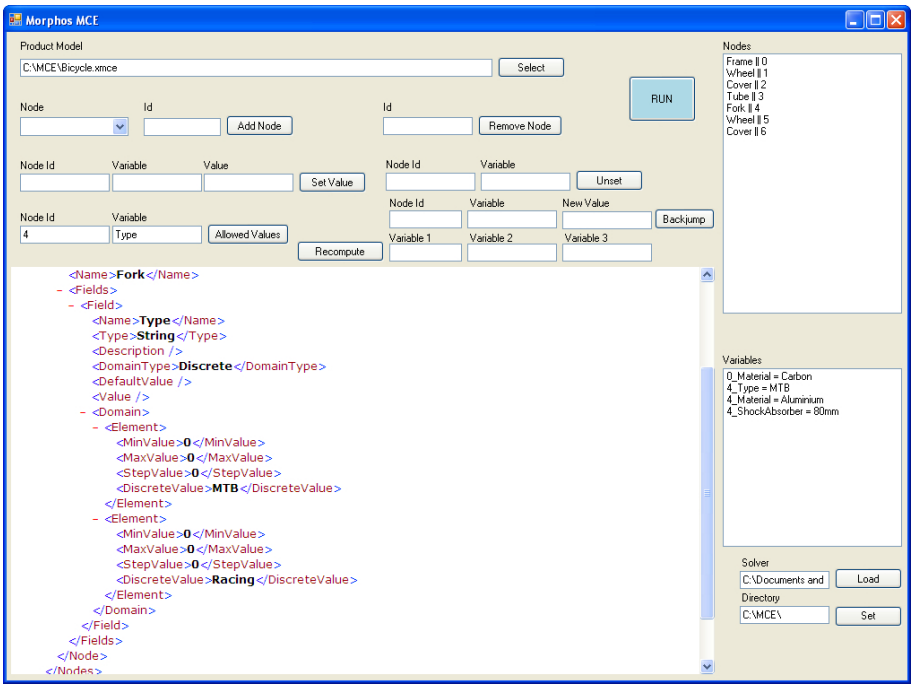

Figure 7: Taking advantage of the *Allowed values* button and the related text fields *Node Id* and *Variable*, the user communicates to the system that he/she wants to change her/his choice for variable `Fork,id=4.Type`. The system computes the allowed values for the variable and shows them in the main box.

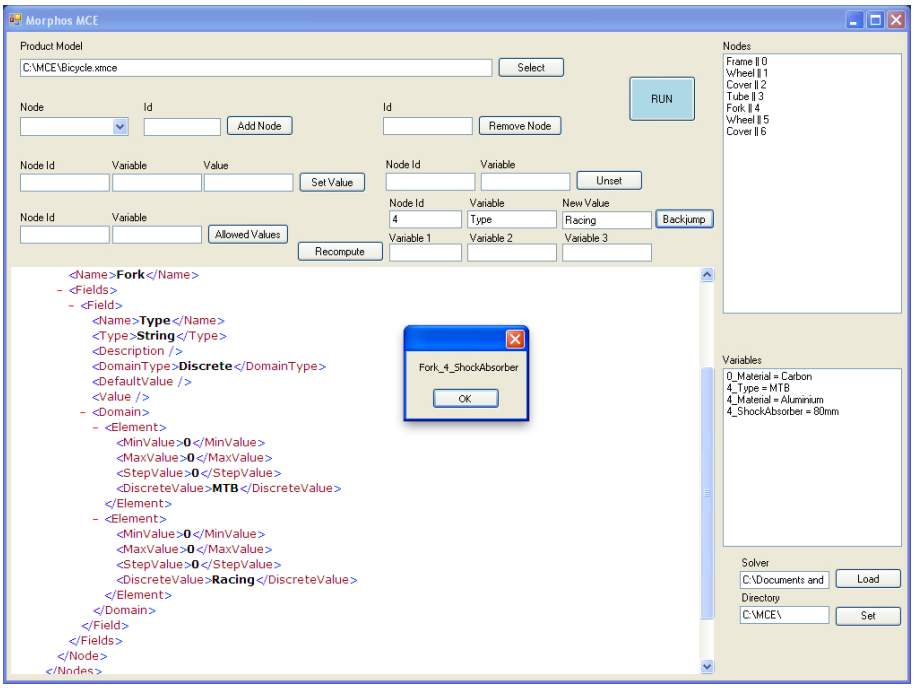

Figure 8: Taking advantage of the *Backjump* button and the related text fields *Node Id*, *Variable*, and *New Value*, the user chooses the new value for `Fork,id=4.Type`. The system computes a set of choices that have to be withdrawn and returns it through a message box.