A Graphical Framework for Supporting Mass Customization *

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

Abstract

Many companies deploying mass customization strategies adopt product configuration systems to support their activities. While such systems focus mainly on configuration process support, mass customization needs to cover the management of the whole customizable product cycle. In this paper, we describe a graphical modeling framework that allows one to model both a product and its production process. We first introduce our framework. Then, we outline a possible implementation based on Constraint Logic Programming of such product/process configuration system. A comparison with some existing product configuration systems and process modeling tools concludes the paper.

1 Introduction

Product configuration systems are software of interest for companies deploying mass customization strategies, since they can support them in the management of configuration processes. In the past years many research studies have been conducted on this topic (see, e.g., [Sabin and Weigel, 1998]), and different software product configurators have been proposed (see, e.g., [Fleischanderl *et al.*, 1998; Junker, 2003; Configit A/S, 2009; Myllärniemi *et al.*, 2005]).

Process modeling tools, instead, allows one to effectively deal with (business) process management. In general, they allow the user to define a description of a process, and guide she/he through the process execution. Also within this field it is possible to find tools and scientific works (see, e.g, [White and Miers, 2008; ter Hofstede *et al.*, 2010; Pesic *et al.*, 2007]).

Mass customization needs to cover the management of the whole customizable product cycle, from product configuration to product production. Current product configuration systems and researches on product configuration, focus only on product modeling and on techniques for configuration process support. They do not cover product production process problematics, despite the advantages that coupling of product with process modeling and configuration could give.

*This work is partially supported by GNCS and MIUR projects.

Inspired by the works of Aldanondo et al. (see, e.g., [Aldanondo and Vareilles, 2008]), we devised a graphical framework for modeling configurable products, whose producible variants can be represented as trees, and their production processes. The intent of our framework is to allow the propagation of consequences of product configuration decision toward the planning of its production process, and the propagation of consequences of process planning decision toward the product configuration.

The paper is organized as follows. First, we introduce our framework in Sect. 2. Then, in Sect. 3 we show how a configuration system based on Constraint Logic Programming can be implemented on top of it. A comparison with some of the existing product configuration systems and process modeling tools is outlined in Sect. 4. An assessment of the work done and of future research directions is given in Sect. 5.

2 A Graphical Framework for Product/Process Modeling

In this section, we present the PRODPROC graphical framework (cf. Sections 2.1 and 2.2). Moreover, we provide a brief description of PRODPROC semantics in terms of model instances (Sect. 2.3).

A PRODPROC *model* consists of a product description, a process description, and a set of constraints coupling the two. To better present the different modeling features offered by our framework, we will exploit a working example. In particular, we will consider a bicycle with its production process.

2.1 Product Modeling Features

We are interested in modeling configurable products whose corresponding (producible) variants can be represented as trees. Nodes of these trees correspond to physical components, whose characteristics are all determined. The tree structure describes how the single components taken together define a configured product.

Hence, we model a configurable product as a multi-graph, called *product model graph*, and a set of constraints. Nodes of the multi-graph represent well-defined components of a product (e.g., the frame of a bicycle). While edges model *has-part/is-part-of* relations between product components. We require the presence of a node without entering edges in the product model graph. We call this node *root node*. A prod-

uct model represents a configurable product. Its configuration can lead to the definition of different (producible) product variants.

Each node/component consists of a name, a set of variables modeling configurable characteristics of the component, and a set of constraints (called *node constraints*) involving variables of the node and variables of its ancestors in the graph. Each variable is endowed with a finite domain (typically, a finite set of integers or strings), i.e., the set of its possible values. Constraints define compatibility relations between configurable characteristics of a node and of its ancestors. The graphical representation of a node (cf. Fig. 1) consists of a box with three sections, each containing one of the elements constituting a node.

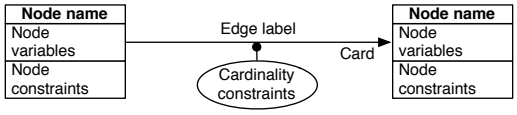

Figure 1: Graphical representation of nodes and edges.

In the description of a configured product, physical components are represented as instances of nodes in the product model graph. An instance of a node $NodeName$ consists of the name $NodeName$, a unique id, and a set of variables equals to the one of $NodeName$. Each variable has a value assigned. The *root node* will have only one instance, such instance will be the root of the configured product tree.

Let us consider, for example, the node *Frame* of the (fragment of) product model graph for a bicycle, depicted in Fig. 2.[1] The section *Frame variables* may contain the following couples of variables and domains:

$$\langle FrameType, \{\text{Racing bike}, \text{Citybike}\}\rangle,$$
$$\langle FrameMaterial, \{\text{Steel}, \text{Aluminum}, \text{Carbon}\}\rangle.$$

While in *Frame constraints* we may have the constraint

$$FrameType = \text{Racing} \Rightarrow FrameMaterial \neq \text{Steel}.$$

This constraint states that a frame of type racing can not be made of steel. An example of instance of $Frame$ is the triple $\langle Frame, 1, \{FrameType = \text{Racing}, FrameMaterial = \text{Carbon}\}\rangle$. Note that values assigned to node instance variables have to satisfy all the node constraints. For the node *Wheel* (that also appears in Fig. 2) we may have the variables

$$\langle WheelType, \{\text{Racing bike}, \text{City bike}\}\rangle,$$
$$\langle SpokeNumber, [18, 28]\rangle,$$

and the constraints

$$WheelType = \langle FrameType, Frame, [\_]\rangle, \tag{1}$$

$$\langle FrameType, Frame, [rear\ wheel]\rangle = \text{Racing bike} \Rightarrow \Rightarrow SpokeNumber > 20. \tag{2}$$

These constraints involve features belonging to an ancestor of the node *Wheel*, i.e., the node *Frame*. We refer to variables

[1]The depicted product model graph is one of the possible graphs we can define with our framework. We chose this one for presentation purposes only.

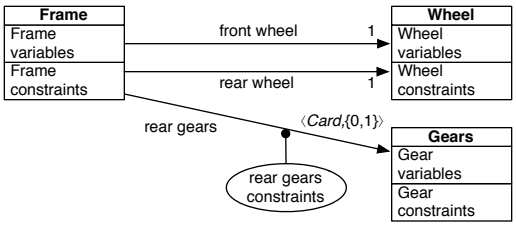

Figure 2: Fragment of bicycle product model graph.

in ancestors of a node using *meta-variables*, i.e., triples of the form $\langle VarName, AncestorName, MetaPath\rangle$. This writing denotes a variable $VarName$ in an ancestor node $AncestorName$ (e.g., $FrameType$ in $Frame$). The third component of a meta-variable, $MetaPath$, is a list of edge labels (see below) describing a path connecting the two nodes in the graph (wildcards '_' and '$\star$' can be used to represent arbitrary labels and a sequence of arbitrary labels respectively). $MetaPath$s are used to define constraints that will have effect only on particular instances of a node. For example, the constraint (2) for the node $Wheel$ has to hold only for those instances of node $Wheel$ which are connected to an instance of node $Frame$ through an edge labeled $rear\ wheel$. Intuitively, a node constraint for the node N has to hold for each instance of N, such that it has ancestors connected with it through paths matching with the $MetaPath$s occurring in the constraint.

An edge $e = \langle label, N, M, Card, \mathcal{CC}\rangle$ of the product model graph is characterized by: a name ($label$), two node names denoting the parent (N) and the child node (M) in the *has-part* relation, the cardinality ($Card$) of such relation (expressed as either an integer number or an integer variable), and a set ($\mathcal{CC}$) of constraints (called *cardinality constraints*). Such constraints may involve the cardinality variables (if any) as well as variables of the parent node and of its ancestors (referred to by means of meta-variables). An edge is graphically represented by an arrow connecting the parent node to the child node (cf. Fig. 1). Such an arrow is labeled with the edge name and cardinality, and may have attached an ellipse containing cardinality constraints.

An instance of an edge labeled $label$ connecting a node N with a node M, is an edge connecting an instance of N and an instance of M. It is labeled $label$ too.

Let us consider the edges *front wheel* and *rear gear* depicted in Fig. 2. The former is the edge relating the frame with the front wheel, its cardinality is imposed to be 1, and there is no cardinality constraint. Hence, there must be (only) one instance of the node *Wheel* connected to an instance of the node *Frame* through an edge labeled *front wheel*. The latter edge, *rear gears*, represents the *has-part* relation over the frame and the rear gears of a bicycle. Its cardinality is a variable named $Card$, taking values in the domain $\{0, 1\}$. Hence, we may have an instance of the node *Gears* connected to an instance of the node *Frame* through an edge labeled *rear gears*. Among the cardinality constraints of the edge *rear gears* we may have the following one:

$$FrameType = \text{Racing} \Rightarrow Card = 1.$$

Intuitively, a cardinality constraint for and edge e has to hold

for each instance of the parent node N in e, such that N has ancestors connected with it through paths matching with $MetaPath$s occurring in the constraint.

As mentioned, we model a product as a graph and a set of constraints. Such constraints, called *model constraints*, involve variables of nodes not necessary related by *has-part* relations (*node model constraints*), as well as cardinalities of different edges exiting from a node (*cardinality model constraints*). Moreover, global constraints like `alldifferent` [van Hoeve, 2001] can be used to define node model constraints. In node model constraints, variables are referred to by means meta-variables. A $MetaPath$ in a node model constraint represents a path connecting a node to one of its ancestors in the graph. $MetaPath$s are used to limit the effect of a node model constraint to particular tuples of node instances. An example of node model constraint for the bicycle is the following one:

$$\langle GearType, Gears, [rear\ gears]\rangle = \text{Special} \Rightarrow \\ \Rightarrow \langle SpokeNumber, Wheel, [rear\ wheel]\rangle = 26. \quad (3)$$

This constraint states that if the type of rear gears chosen is "Special", then the rear wheel must have 26 spokes. Intuitively, a node model constraint has to hold for all the tuples of node variables of node instances reached by paths matching with the $MetaPath$s occurring in the constraint.

2.2 Process Modeling Features

Processes can be modeled in PRODPROC in terms of activities and temporal relations between them. More precisely, a process is characterized by: a set of activities, a set of variables (as before, endowed with a finite domain of strings or of integers) representing process characteristics and involved resources; a set of temporal constraints between activities; a set of resource constraints; a set of constraints involving product elements; a set of constraints on activity durations. A process model does not represent a single production process. Instead, it represents a configurable production process, whose configuration can lead to the definition of different executable processes.

PRODPROC defines three kinds of activities: *atomic activities*, *composite activities*, and *multiple instance activities*. An *atomic* activity A is an event occurring in a time interval. It has associated a name and the following parameters.

- Two integer decision variables, t^{start} and t^{end}, denoting the start and end time of the activity. They define the tine interval $[t^{start}, t^{end}]$, and are subject to the implicit condition $t^{end} \geq t^{start} \geq 0$.
- A decision variables $d = t^{end} - t^{start}$ denoting the duration of the activity.
- A flag $exec \in \{0, 1\}$.

We say that A is an *instantaneous activity* if $d = 0$. *A is executed* if $exec = 1$ holds, otherwise (i.e., if $exec = 0$) A is *not executed*. A *composite* activity is an event described in terms of a process. It has associated the same parameters of an atomic activity, and a model of the process it represents. A *multiple instance* (atomic or composite) activity is an event that may occur multiple times. Together with the usual parameters (and possibly the process model), a multiple instance activity has associated an integer decision variable (named $inst$) representing the number of times the activity can be executed. Note that the execution/non-execution of activities determines different instances of a configurable process. Figures 3a, 3b, and 3c, show the graphical representation of atomic activities, composite activities, and multiple instance activities, respectively.

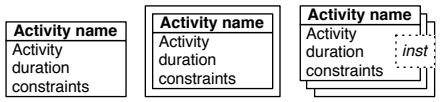

Figure 3: Graphical representation of activities.

Temporal constraints between activities are inductively defined starting from *atomic temporal constraints*. We consider as atomic temporal constraints all the thirteen mutually exclusive relations on time intervals introduced by Allen in [Allen, 1983] (they capture all the possible ways in which two intervals might overlap or not), and some other constraints inspired by constraint templates of the language ConDec [Pesic *et al.*, 2007]. Some examples of atomic temporal constraints are listed as follows (for lack of space we avoid listing all of them), where A and B are two activities. Fig. 4 shows their graphical representations (we used a slightly different graphical notation for activities, i.e., we omitted the activity duration constraint sections).

1. $A\ before\ B$ to express that A is executed before B (cf. Fig. 4a);
2. $A\ during\ B$ to express that A is executed during the execution of B (cf. Fig. 4b);
3. $A\ is{-}absent$ to express that A can never be executed (cf. Fig. 4c);
4. $A\ must{-}be{-}executed$ to express that A must be executed (cf. Fig. 4d);
5. $A\ not{-}co{-}existent{-}with\ B$ to express that it is not possible to executed both A and B (cf. Fig. 4e);
6. $A\ succeeded{-}by\ B$ to express that when A is executed then B has to be executed after A (cf. Fig. 4f).

The constraints 1 and 2 are two of the relations presented in [Allen, 1983]. The constraints 3-6 have been inspired by the templates used in ConDec [Pesic *et al.*, 2007]. A *temporal constraint* is inductively defined as follows.

- An atomic temporal constraint is a constraint.
- If φ and ϑ are temporal constraint, then $\varphi\ and\ \vartheta$ and $\varphi\ or\ \vartheta$ are temporal constraints.
- If φ is a temporal constraint and c is a constraint on process variables, then $c \rightarrow \varphi$ is an *if-conditional* temporal constraint, stating that φ has to hold whenever c holds. Also, $c \leftrightarrow \varphi$ is an *iff-conditional* temporal constraint, stating that φ has to hold if and only if c holds.

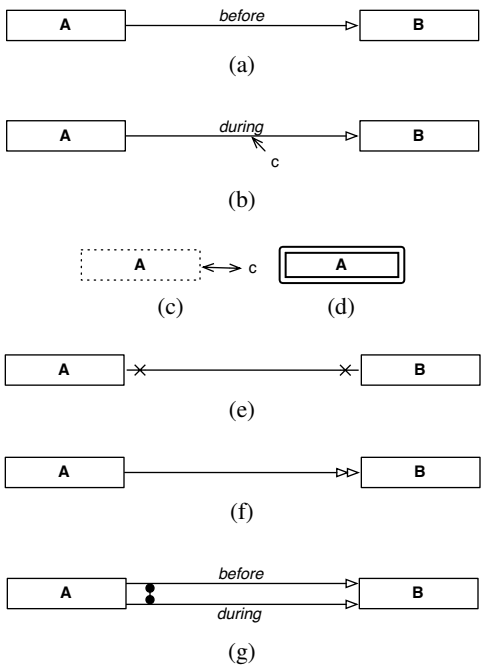

Figure 4: Graphical representation of temporal constraints.

A conjunction of atomic constraints between two activities can be depicted by representing each constraint of the conjunction. Fig. 4g shows the graphical representation for a disjunction of atomic temporal constraints between two activities (i.e., for the constraint $A\ before\ B\ or\ A\ during\ B$). An if-conditional and an iff-conditional temporal constraint with condition *c* are depicted in Fig. 4b and 4c, respectively. Finally, a non-atomic temporal constraint can be depicted as an hyper-edge connecting the activities involved in it, and labeled with the constraint itself.

The truth of atomic temporal constraints is related with the execution of the activities they involve. For instance, whenever for two activities A and B it holds that $exec_A = 1 \wedge exec_B = 1$, then the atomic formulas of the forms 1 and 2 must hold. A *temporal constraint network* $\mathcal{CN}$ is a pair $\langle \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{A}$ is a set of activities and $\mathcal{C}$ is a set of temporal constraints on $\mathcal{A}$. Fig. 5 shows the temporal constraint network of the bicycle production process.

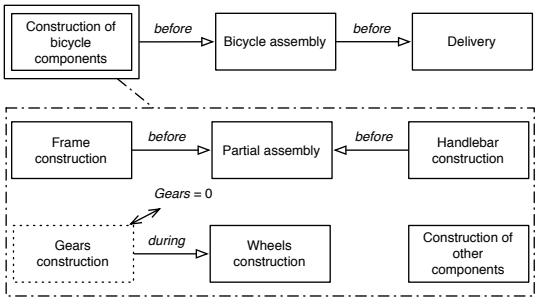

Figure 5: Temporal constraint network of the bicycle production process.

Resource constraints [Laborie, 2003] are quadruple $\langle A, R, q, TE \rangle$, where A is an activity; R is a variable endowed with a finite integer domain; q is an integer or a variable endowed with a finite integer domain, defining the quantity of resource R consumed (if $q < 0$) or produced (if $q > 0$) by executing A; TE is a time extent that defines the time interval where the availability of resource R is affected by A. The possible values for TE are: $FromStartToEnd$, $AfterStart$, $AfterEnd$, $BeforeStart$, $BeforeEnd$, $Always$, with the obvious meaning. Another form of resource constraint defines initial level constraints, i.e., expressions determining the quantity of a resource available at the time origin of a process. The basic form is $initialLevel(R, iv)$, where R is a resource and $iv \in \mathbb{N}$. Examples of resource constraints for the bicycle production process are:

$$\langle \text{Wheel construction}, Aluminum, -4, AfterStart \rangle,$$
$$\langle \text{Frame construction}, Workers, q_W \in [-1, -2], FromStartToEnd \rangle.$$

The first constraint states that activity "Wheel construction" consumes 4 unit of aluminum once its execution starts. The second constraints states that activity "Frame construction" needs 1 or 2 workers during its execution. As for temporal constraint, we can define *if-conditional* (i.e., $c \rightarrow \langle A, R, q, TE \rangle$) and *iff-conditional* (i.e., $c \leftrightarrow \langle A, R, q, TE \rangle$) resource constraints. Their meaning are similar to the ones defined above for temporal constraints.

An *activity duration constraint* for an activity A, is a constraint involving the duration of A, process variables, and quantity variables for resources related to A. The following is an example of an activity duration constraint for the activity "Frame construction" in the bicycle production process (where $FrameMult$ is a process variable)

$$d = \frac{2 \cdot FrameMult}{|q_W|}$$

PRODPROC also allows one to mix elements for modeling a process with elements for modeling a product, through constraints involving process variables and product variables. This is an example for the bicycle model

$$\langle FrameType, Frame, [] \rangle = Racing \Rightarrow FrameMult = 4.$$

It relates the variable $FrameType$ of the node $Frame$ with the process variable $FrameMult$. Another example is the following:

$$\langle rear\ gears, Frame, Gears, Card \rangle = Gears.$$

This constraint relates the process variable $Gears$ with the cardinality of the edge $rear\ gears$ of the bicycle product model graph. The cardinality is represented by the quadruple $\langle rear\ gears, Frame, Gears, Card \rangle$, where the first element is an edge label, the second one is the name of the parent node of the edge, the third one is the name of the child node of the edge, and the last one is the name of the cardinality.

Product related constraints are another type of constraints coupling product elements with process elements. They make it possible to define resource constraints where resources are product components. More precisely, a product related constraint is a constraint on activities and product nodes that implicitly defines resource constraints, and constrains on process and product variables. A product related constraint has

the form A *produces* n N *for* B, where A and B are activities, $n \in \mathbb{N}^+$, and N is the name of a node in the product model graph, having (at least) one incoming edge having associated a cardinality variable. Such a product related constraint corresponds to the following PRODPROC constraints:

$\langle A, R_N, q_A \in D_{R_N}, AfterEnd\rangle$, $\langle B, R_N, -n, AfterStart\rangle$, $initialLevel(R_N, 0)$, $aggConstraint(sum, CE_N, =, R_N)$,

where R_N is a resource variable whose domain D_{R_N} is defined as $D_{R_N} = \left[0, \sum_{C \in CE_N} max(D_C)\right]$ (D_C denotes the domain of C, and CE_N is the list of cardinality variables of edges entering in N). The constraint $aggConstraint(sum, CE_N, =, R_N)$ is a global constraint stating that $\sum_{C \in CE_N} = R_N$ has to hold. An example of product related constraint for the bicycle is

Wheel construction *produces*
2 $Wheel$ *for* Bicycle assembly.

In general, constraints involving both product and process variables may allow one to detect/avoid planning impossibilities due to product configuration, and configuration impossibilities due to production planning, during the configuration of a product and its production process.

2.3 PRODPROC Instances

A PRODPROC model represents the collection of single (producible) variants of a configurable product and the processes to produce them. A PRODPROC *instance* represents one of such variants and its production process. To precisely define this notion we need to introduce first the notion of *candidate instance*. A PRODPROC candidate instance consists of the following components:

- A set $\mathcal{N}^I$ of *node instances*, i.e., tuples of the form $N_i^I = \langle N, i, \mathcal{V}_{N_i^I}\rangle$ where N is a node in the product model graph, $i \in \mathbb{N}$ is an index (different for each instance of a node), $\mathcal{V}_{N_i^I} = \mathcal{V}_N$ ($\mathcal{V}_N$ is the set of variables of node N).
- a set $\mathscr{A}_{\mathsf{Nodes}}$ of *assignments* for all the node instance variables, i.e., expressions of the form $V = value$ where V is a variable of a node instance and $value$ belongs to the set of values for V.
- A tree, called *instance tree*, that specifies the pairs of node instances in the relation *has-part*. Such a tree is defined as $IT = \langle \mathcal{N}^I, \mathcal{E}^I\rangle$, where $\mathcal{E}^I$ is a set of tuples $e^I = \langle label, N_i^I, M_j^I\rangle$ such that there is an edge $e = \langle label, N, M, Card, \mathcal{CC}\rangle$ in the product model graph, and N_i^I, M_j^I are instances of N and M, respectively.
- A set $\mathscr{A}_{\mathsf{Cards}}$ of *assignments* for all the instance cardinality variables, i.e., expressions of the form $IC^e_{N_i^I} = n$ where N_i^I is an instance of a node N, e is an edge $\langle label, N, M, Card, \mathcal{CC}\rangle$, $IC^e_{N_i^I} = Card$, and n is the number of the edges $\langle label, N_i^I, C_j^I\rangle$ in the instance tree, such that M_j^I is an instance of M.
- A set $\mathcal{A}^I$ of *activity instances*, i.e., pairs $A_i^I = \langle A, i\rangle$ where A is the name of an activity with $exec_A = 1$, and $i \in \mathbb{N}$ is a unique index for instances of A.

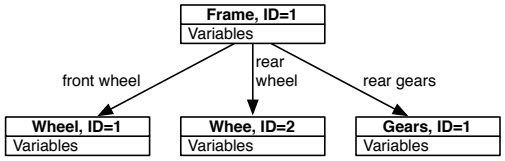

Figure 6: Instance tree for a bicycle.

- A set $\mathscr{E} = \{exec_A \mid A \text{ is an activity} \wedge exec_A \neq 1\}$.
- A set $\mathscr{A}_{\mathsf{Proc}}$ of *assignments* for all model variables and activity parameters (i.e., time instant variables, duration variables, execution flags, quantity resource variables, instance number variables), that is, expressions of the form $P = value$ where P is a model variable or an activity parameter, and $value \in \mathbb{Z}$ or $value$ belongs to the set of values for P.

Fig. 6 depicts a fragment of the instance tree for a bicycle. The tree consists of an instance of node $Frame$, an instance of the node $Gears$, and two instances of node $Wheel$.

A PRODPROC instance is a candidate instance such that the assignments in $\mathscr{A}_{\mathsf{Nodes}} \cup \mathscr{A}_{\mathsf{Cards}} \cup \mathscr{A}_{Proc}$ satisfy all the constraints in the PRODPROC model (node constraints, temporal constraints, etc.), instantiated with variables of node instances and activity instances in the candidate instance.

The constraint instantiation mechanism, given a (partial) candidate instance (a candidate instance is partial when there are variables with no value assigned to), produces a set of constraints on candidate instance variables from each constraint in the corresponding PRODPROC model. A candidate instance has to satisfy all these constraints to qualify as an instance. We give here an intuitive description of how the instantiation mechanism works on different constraint types. Let us begin with node and cardinality constraints. Let c be a constraint belonging to the node N, or a constraint for an edge e between nodes N and M. Let us suppose that $N_1, \ldots, N_k$ are ancestors of N whose variables are involved in c, and let $p_1, \ldots, p_k$ be $MetaPath$s such that, for $i = 1, \ldots, k$, p_i is a $MetaPath$ from N_i to N. We define L_n as the set of k-tuple of node instances $\langle N_j^I, (N_1)^I_{j_1}, \ldots, (N_k)^I_{j_k}\rangle$ where: N_j^I is an instance of N; for $i = 1, \ldots, k$ $(N_i)^I_{j_i}$ is an instance of N_i, connected with N_j^I through a path p_i^I in the instance tree that matches with p_i. For each k-tuple $t \in L_n$, we obtain a constraint on instance variables appropriately substituting variables in c with variables of node instances in t. For example, the constraints (1) and (2) for the node $Wheel$, lead to the following constraints on variables of node instances in Fig. 6 ($\langle V, N^I_{ID}\rangle$ denotes the variable V of the instance with id ID of node N).

$$\langle WheelType, Wheel_1^I\rangle = \langle FrameType, Frame_1^I\rangle, \quad (4)$$

$$\langle WheelType, Wheel_2^I\rangle = \langle FrameType, Frame_1^I\rangle, \quad (5)$$

$$\langle FrameType, Frame_1^I\rangle = \text{Racing bike} \Rightarrow$$
$$\Rightarrow \langle SpokeNumber, Wheel_2^I\rangle > 20. \quad (6)$$

The instantiation of (1) leads to the constraints (4) and (5), since it can be instantiated on both the couples of node instances appearing in Fig. 6 $\langle Wheel_1^I, Frame_1^I\rangle$ and

$\langle Wheel_2^I, Frame_1^I \rangle$. Instead, the instantiation of (2) leads to only one constraint, i.e. (6), because it can be instantiated only on the couple $\langle Wheel_2^I, Frame_1^I \rangle$.

Node model constraints are instantiated in a slightly different way. Let c be a node model constraint. Let us suppose that $N_1, \ldots, N_k$ are the nodes whose variables are involved in c, let $p_1, \ldots, p_k$ be $MetaPath$s such that, for $i = 1, \ldots, k$, p_i is a $MetaPath$ that ends in N_i. We define L_{nmc} as the set of ordered k-tuples of node instances $\langle (N_1)_{j_1}^I, \ldots, (N_k)_{j_k}^I \rangle$, where for $i = 1, \ldots, k$ $(N_i)_{j_i}^I$ is an instance of N_i connected by a path p_i^I with one of its ancestors in the instance tree, such that p_i^I matches with p_i. For each k-tuple $t \in L_{nmc}$, we obtain a constraint on instance variables appropriately substituting variables in c with variables of node instances in t. If c is an `alldifferent` constraint, then we define an equivalent constraint on the list consisting of all the node instances of $N_1, \ldots, N_k$, connected with one of their ancestors by a path matching with the corresponding $MetaPath$. As an example, let us consider the constraint (3) for the bicycle, it can be instantiated on the couple $\langle Wheel_2^I, Gears_1^I \rangle$ and leads to

$$\langle GearType, Gears_1^I \rangle = \text{Special} \Rightarrow \\ \Rightarrow \langle SpokeNumber, Wheel_2^I \rangle = 26.$$

The instantiation of cardinality model constraints is very simple. Let c be a cardinality model constraint for the cardinalities of the edges with labels $e_1, \ldots, e_k$ exiting from a node N. Let $N_1^I, \ldots, N_h^I$ be instances of N. For all $i \in \{1, \ldots, h\}$, we instantiate c appropriately substituting the cardinality variables occurring in it, with the instance cardinality variables $IC_{N_i^I}^{e_1}, \ldots, IC_{N_i^I}^{e_k}$.

Let us now consider process constraints. Let A be an activity, let $A_1^I, \ldots, A_k^I$ be instances of A. Let r be the resource constraint $\langle A, R, q, TE \rangle$, we instantiate it on each instance of A, i.e., we obtain a constraint $\langle A_i^I, R, q_i, TE \rangle$ for each $i = 1, \ldots, k$, where $q_i = q$ is a fresh variable or an integer. Let c be an activity duration constraint for A, for each $i = 1, \ldots, k$ we obtain a constraint substituting in c d_A with $d_{A_i^I}$, and each quantity variable q with the corresponding variable q_i. Finally, let B an activity, let $B_1^I, \ldots, B_h^I$ be instances of B. If c is a temporal constraint involving A and B, we obtain a constraint on activity instances for each ordered couple $\langle i, j \rangle$, with $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, h\}$, substituting in c each occurrence of A with A_i^I, and of B with B_j^I. This mechanism can be easily extended to non-binary constraints.

3 CLP-based Product/Process Configuration

Constraint Logic Programming (CLP) [Jaffar and Maher, 1994] can be exploited to implement a configuration system that, given a PRODPROC model (cf. Sect. 2), guide a user through the configuration process to obtain a PRODPROC instance (cf. Sect. 2.3). In this section, we first present a possible structure for such a system. Then, we briefly explain how a configuration problem can be encoded in a CLP program.

A CLP-based system can support a configuration process as follows. First, the user initializes the system (1) selecting the model to be configured. After such an initialization phase, the user starts to make her/his choices by using the system interface (2). The interface communicates to the system engine (i.e., the piece of software that maintains a representation of the product/process under configuration, and checks the validity and consistency of user's choices) each data variation specified by the user (3). The system engine updates the current partial configuration accordingly. Whenever an update of the partial configuration takes place, the user, through the system interface, can activate the engine inference process (4). The engine instantiates PRODPROC constraints (cf. Sect. 2.3) on the current (partial) candidate instance defined by user choices, and encodes the product/process configuration problem in a CLP program (encoding a Constraint Satisfaction Problem, abbreviated to CSP). Then, the engine uses a finite domain solver to propagate the logical effects of user's choices (5). Once the inference process ends (6), the engine returns to the interface the results of its computation (7). In its turns, the system interface communicates to the user the consequences of her/his choices on the (partial) configuration (8).

From a PRODPROC model and a user defined (partial) candidate instance corresponding to it, it is possible to obtain a CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where: $\mathcal{V}$ is the set of all the variables appearing in the (partial) candidate instance; $\mathcal{D}$ is the set of domains for variables in $\mathcal{V}$; $\mathcal{C}$ is the set of constraints in the PRODPROC model instantiated on variables of the (partial) candidate instance. Such a CSP can be easily encoded in a CLP program like the following one.

```
csp_prodProc(Vars) :- DOMS, CONSTRS.
```

In it, `Vars` is the list of variables in $\mathcal{V}$, `DOMS` is the conjunction of domain constraints for domains in $\mathcal{D}$, and `CONSTRS` is the conjunction of constraints in $\mathcal{C}$. Given a program with the above-described characteristics, a finite domain solver 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). Moreover, it can be used to determine that further node instances are needed, or that there are too many nodes in the instance tree.

4 Comparison with Related Work

In order to point out strengths and limitations of the PRODPROC framework, we present in this section a brief comparison with some of the most important product configuration systems and process modeling tools.

Answer Set Programming (ASP) [Gelfond and Lifschitz, 1988] has been used to implement product configuration systems that are specifically tailored to the modeling of software product families, e.g., Kumbang Configurator [Myllärniemi *et al.*, 2005]. Even if these systems result to be appealing for a relevant range of application domains, they lack of generality. In particular, they do not support global constraints, and the so called grounding stage may cause problems in the management of arithmetic constraints [Myllärniemi *et al.*, 2005].

Product configuration systems based on binary decision diagrams (BDDs), e.g., Configit Product Modeler [Configit A/S, 2009], trade the complexity of the construction of the BDD, for the simplicity and efficiency of the configuration process. Despite their various attracting features, BDD-based systems suffer from some significant limitations. First,

they basically support flat models only, even though some work has been done on the introduction of modules (see, e.g., [van der Meer and Andersen, 2004]). Second, they find it difficult to cope with global constraints. In [Nørgaard *et al.*, 2009] the authors combine BDD and CSP to tackle `alldifferent` constraints. However, they consider flat models only. We are not aware of any BDD system that deals with global constraints in a general and satisfactory way.

Unlike ASP-based and BDD-based systems, CSP-based product configuration systems are (usually) capable of dealing with non-flat models and global constraints. Unfortunately, the modeling expressiveness of CSP-based systems has a cost, i.e., 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 [Junker, 2003], which features various interesting modeling facilities, and Lava [Fleischanderl *et al.*, 1998], which is based on Generative-CSP, seem to be no longer supported. A recent CSP-based configuration system is Morphos Configuration Engine (MCE) [Campagna *et al.*, 2010]. As other CSP-based systems, it makes it possible to define non-flat models. 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. From the point of view of process modeling, PRODPROC can be viewed as an extension of MCE modeling language. In particular, it extends MCE modeling language with the following features: (1) *cardinality variables*, i.e., *has-part/is-part-of* relations can have non-fixed cardinalities; (2) *product model graph*, i.e., nodes and relations can define a graph, not only a tree; (3) *cardinality constraints* and *cardinality model constraints*, i.e., constraints can involve cardinalities of relations; (4) $MetaPath$s, i.e., a mechanism to refer to particular node instance variables in constraints.

PRODPROC can be viewed as the source code representation of a configuration system with respect to the MDA abstraction levels presented in [Felfernig, 2007]. PRODPROC product modeling elements can be mapped to UML/OCL in order to obtain platform specific (PSM) and platform independent (PIM) models. The mapping to OCL of $MetaPath$s containing '$\star$' wildcards and of model constraints requires some attention. For example, the latter do not have explicit contexts as OCL constraints must have. Since PRODPROC does not support the definition of taxonomies of product components, there will not be generalization hierarchies in PMSs and PIMs corresponding to PRODPROC models.

In the past years, different formalism have been proposed for process modeling. Among them we have: the Business Process Modeling Notation (BPMN) [White and Miers, 2008]; Yet Another Workflow Language (YAWL) [ter Hofstede *et al.*, 2010]; DECLARE [Pesic *et al.*, 2007].

Languages like BPMN and YAWL model a process as a detailed specification of step-by-step procedures that should be followed during the execution. BPMN and YAWL adopt an *imperative* approach in process modeling, i.e., all possibilities have to be entered into their models by specifying their control-flows. BPMN has been developed under the coordination of the Object Management Group. PRODPROC has in common with BPMN the notion of atomic activity, subprocess, and multiple instance activity. The effect of BPMN joins and splits on the process flow can be obtained using temporal constraints. In PRODPROC there are no notions such as BPMN events, exception flows, and message flows. However, events can be modeled as instantaneous activities and data flowing between activities can be modeled with model variables. YAWL is a process modeling language whose intent is to directly supported all control flow patterns. PRODPROC has in common with YAWL the notion of task, multiple instance task, and composite task. YAWL join and split constructs are not present in PRODPROC, but using temporal constraints it is possible to obtain the same expressivity. The notion of cancellation region is not present in PRODPROC, but our framework could be extended to implement it.

As opposed to traditional imperative approaches to process modeling, DECLARE uses a constraint-based declarative approach. Its models rely on constraints to implicitly determine the possible ordering of activities (any order that does not violate constraints is allowed). With respect to DECLARE, PRODPROC has in common the notion of activity and the use of temporal constraints to define the control flow of a process. The set of atomic temporal constraints is not as big as the set of template constraints available in DECLARE, however it is possible to easily the available ones so as to define all complex constraints of practical interest. Moreover, in PRODPROC it is possible to define multiple instance and composite activities, features that are not available in DECLARE.

From the point of view of process modeling, PRODPROC combines modeling features of languages like BPMN and YAWL, with a declarative approach for control flow definition. Moreover, it presents features that, to the best of our knowledge, are not presents in other existing process modeling languages. These are: resource variables and resource constraints, activity duration constraints, and product related constraints. Thanks to these features, PRODPROC is suitable for modeling production processes and, in particular, to model mixed scheduling and planning problems related to production processes. Furthermore, a PRODPROC model does not only represent a process ready to be executed as a YAWL (or DECLARE) model does, it also allows one to describe a configurable process. Existing works on process configuration, e.g., [Rosa, 2009], define process models with variation points, and aim at deriving different process model variants from a given model. Instead, we are interested in obtaining process instances, i.e., solutions to the scheduling/planning problem described by a PRODPROC model.

With respect to the works of Mayer et al. on service process composition (e.g. [Mayer *et al.*, 2009]), PRODPROC is more geared toward production process modeling and configuration. However, certain aspects of service composition problems can be modeled using PRODPROC too.

The PRODPROC framework allows one to model products, their production processes, and to couple products with processes using constraints. The only works on the coupling of product and process modeling and configuration we are aware

of are the ones by Aldanondo et al. (see, e.g., [Aldanondo and Vareilles, 2008]). They propose to consider simultaneously product configuration and process planning problems as two constraint satisfaction problems. In order to propagate decision consequences between the two problems, they suggest to link the two constraint based models using coupling constraints. The development of PRODPROC has been inspired by the papers of Aldanondo et al., in fact, we also have separated models for products and processes and, constraints for coupling them. However, our modeling languages are far more complex and expressive than the one presented in [Aldanondo and Vareilles, 2008].

5 Conclusions

In this paper, we considered the problem of product and process modeling and configuration, and pointed out the the lack of a tool covering both physical and production aspects of configurable products. To cope with this absence, we presented a graphical framework called PRODPROC. Furthermore, we shown how it is possible to build a CLP-based configuration systems on top of it, and presented a comparison with some of the existing product configuration systems and process modeling tools.

We already implemented a first prototype of a CLP-based configuration system that uses PRODPROC. It covers only product modeling and configuration, but we are working to add to it process modeling and configuration capabilities. We also plan to experiment our configuration system on different real-world application domains, and to compare it with commercial products, e.g., [Blumöhr *et al.*, 2009].

References

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

[Allen, 1983] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26:832–843, 1983.

[Blumöhr *et al.*, 2009] U. Blumöhr, M. Münch, and M. Ukalovic. *Variant Configuration with SAP*. SAP Press, 2009.

[Campagna *et al.*, 2010] D. Campagna, C. De Rosa, A. Dovier, A. Montanari, and C. Piazza. Morphos Configuration Engine: the Core of a Commercial Configuration System in CLP(FD). *Fundam. Inform.*, 105(1-2):105–133, 2010.

[Configit A/S, 2009] Configit A/S. Configit Product Modeler. http://www.configit.com, 2009.

[Felfernig, 2007] A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Trans. on Engineering Management*, 54(1):41–56, 2007.

[Fleischanderl *et al.*, 1998] 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.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[Jaffar and Maher, 1994] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

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

[Laborie, 2003] P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artif. Intell.*, 143:151–188, 2003.

[Mayer *et al.*, 2009] W. Mayer, R. Thiagarajan, and M. Stumptner. Service composition as generative constraint satisfaction. In *Proc. of the 2009 IEEE Int. Conf. on Web Services*, pages 888–895, 2009.

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

[Nørgaard *et al.*, 2009] A. H. Nørgaard, M. R. Boysen, R. M. Jensen, and P. Tiedemann. Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration. In *Porc. of the IJCAI'09 Workshop on Configuration*, 2009.

[Pesic *et al.*, 2007] M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full support for loosely-structured processes. In *Proc. of EDOC'07*, pages 287–287, 2007.

[Rosa, 2009] M. La Rosa. *Managing Variability in Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2009.

[Sabin and Weigel, 1998] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13:42–49, July 1998.

[ter Hofstede *et al.*, 2010] A. H. M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation - YAWL and its Support Environment*. Springer, 2010.

[van der Meer and Andersen, 2004] E. R. van der Meer and H. R. Andersen. BDD-based Recursive and Conditional Modular Interactive Product Configuration. In *Proc. of Workshop on CSP Techniques with Immediate Application (CP'04)*, pages 112–126, 2004.

[van Hoeve, 2001] W. J. van Hoeve. The alldifferent Constraint: A Survey, 2001.

[White and Miers, 2008] S. A. White and D. Miers. *BPMN modeling and reference guide: understanding and using BPMN*. Lighthouse Point, 2008.