PRODPROC - Product and Production Process Modeling and Configuration *

Dario Campagna and Andrea Formisano

Dipartimento di Matematica e Informatica, Università di Perugia, Italy
(dario.campagna|formis)@dmi.unipg.it

Abstract. Software product configurators are an emerging technology that supports companies in deploying mass customization strategies. Such strategies need to cover the management of the whole customizable product cycle. Adding process modeling and configuration features to a product configurator may improve its ability to assist mass customization development. In this paper, we describe a modeling framework that allows one to model both a product and its production process. We first introduce our framework focusing on its process modeling capabilities. Then, we outline a possible implementation based on Constraint Logic Programming of such product/process configuration system. A comparison with some of the existing systems for product configuration and process modeling concludes the paper.

1 Introduction

In the past years many companies started to operate according to *mass customization* strategies. Such strategies aim at selling products that satisfy customer's needs, preserving as much as possible the advantages of *mass production* in terms of efficiency and productivity. The products offered by such companies, usually called *configurable products*, have a predefined basic structure that can be customized by combining a series of available components and options (modules, accessories, etc.) or by specifying suitable parameters (lengths, tensions, etc.). Actually, a configurable product does not correspond to a specific physical object, but identify sets of (physical) objects that a company can realize. A *configured product* is a single variant of a configurable product, obtained by specifying each of its customizable attributes, which corresponds to a fully-specified physical object. The *configuration process* consists of a series of activities and operations ranging from the acquisition of information about the variant of the product requested by the customer, to the generation of data for its realization.

The mass customization operating mode involves a series of difficulties that companies struggle to resolve by using traditional software tools, designed for repetitive productions. As more companies started offering configurable products, different systems designed for supporting them in deploying mass customization strategies appeared. These systems are called *software product configurators* and allow one to effectively and efficiently deal with the configuration process [21]. They offer functionality for the

* Research partially founded by GNCS-2011 and MIUR-PRIN-2008 projects, and grants 2009.010.0336 and 2010.011.0403.

representation of configurable products through *product models*, and for organizing and managing the acquisition of information about the product variants to be realized.

Mass customization strategies need to cover the management of the whole customization product cycle, from customer order to final manufacturing. Current software product configurators focus only on the support to product configuration, and do not cover aspects related to the production process planning. Extending the use of configuration techniques from products to processes, may avoid or reduce planning impossibilities due to constraints introduced in the product configuration phase, as well as configuration impossibilities due to production planning requirements. Existing languages/tools for process modeling, such as BPMN [28] and YAWL [24], do not offer suitable features for specifying production processes and process configuration. Moreover, they lack the capability of modeling, in a single uniform setting, product models and their corresponding process models. The framework we propose, called PRODPROC, intends to overcome these limitations and act as a core for a full-fledged configuration system, covering the whole customization product cycle.

2 A Framework for Product/Production Modeling

In this section we present the PRODPROC framework by exploiting a working example that will be used throughout the paper (cf., Sections 2.1 and 2.2). We also provide a brief description of PRODPROC semantics in term of model instances (Sect. 2.3). See [6] for a description of PRODPROC graphical modeling language.

A PRODPROC *model* consists of a description of a product, a description of a process, and a set of constraints coupling the two. In order to introduce the PRODPROC features let us consider a rectangular base prefabricated component multi-story building, together with its construction process. More specifically, a building is composed by the followings parts: story, roof, heating service, ventilation service, sanitary service, electrical/lighting service, suspended ceiling, floor, partition wall system. For the purposes of this paper, we consider two types of building:

Warehouse: it is a single story building, it has no mandatory service except for the electrical/lighting service, it has no partition wall system and no suspended ceiling, it may have a basement.

Office building: it may have a basement and up to three stories, all services except ventilation are mandatory, suspended ceiling and floor are mandatory for each story, each story may have a partition wall system.

The building construction process can be split in four main phases: preparation and development of the building site; building shell and building envelope works; building services equipment; finishing works. (For a detailed description of such phases see [23].)

2.1 Product Description

A product is modeled as a multi-graph, called *product model graph*, and a set of constraints. The nodes of the graph represent the components of the product. The edges represent the *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

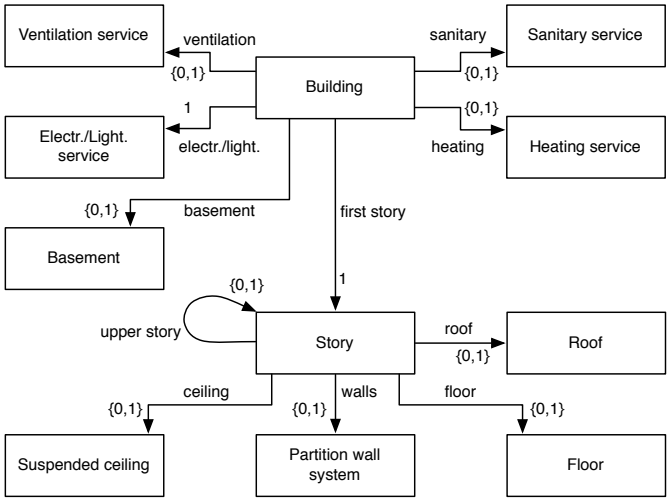

Fig. 1. Building product model graph.

node *root node*. Such a product description will represent a configurable product whose configuration can lead to the definition of different (producible) variants that 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. Fig. 1 shows the product model graph for our example. Edges are labeled with names describing the *has-part* relations and numbers indicating the admitted values for the cardinalities.

Each node/component of a product model graph is characterized by a name, a set of variables representing configurable features of the component, and a set of constraints that may involve variables of the node as well as 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. In the description of a configured product, physical components will be 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 will have a value assigned. The instance of the *root node* will be the root of the configured product tree. For example, the node *Building* in Fig. 1, which is the root node of the product model graph, is defined as the triple $\langle Building, \mathcal{V}_{Building}, \mathcal{C}_{Building} \rangle$, where the involved variables and the set of constraint are as follows:

$$\mathcal{V}_{Building} = \{\langle BuildingType, \{\text{Warehouse}, \text{Office building}\}\rangle, \\ \langle StoryNum, [1,3]\rangle, \langle Width, [7,90]\rangle, \langle Length, [7,90]\rangle\},$$

$$\mathcal{C}_{Building} = \{BuildingType = \text{Warehouse} \Rightarrow StoryNum = 1\}.$$

Hence, a building is described by four features/variables, each one with a set of possible values. Note that the single constraint associated with the node imposes that if the building is a warehouse, then it must have exactly one story. The node representing a

story of the building is defined as $\langle Story, \mathcal{V}_{Story}, \mathcal{C}_{Story}\rangle$, where:

$$\mathcal{V}_{Story} = \{\langle FloorNum, [1,3]\rangle, \langle Height, [3,15]\rangle\},$$
$$\begin{aligned}\mathcal{C}_{Story} = \{&FloorNum = \langle FloorNum, Story, [upper\ story]\rangle + 1,\\ &FloorNum \leq \langle StoryNum, Building, [first\ story, \star]\rangle,\\ &\langle BuildingType, Building, [first\ story, \star]\rangle = \text{Office building} \Rightarrow\\ &\quad \Rightarrow Height \geq 4 \wedge Height \leq 5\}.\end{aligned}$$

In this case we have two variables associated with the node *Story*, whose values are controlled by three constraints. Note that these constraints involve features/variables associated with ancestors of the node *Story*. To refer to specific variables in the ancestors of a node, we introduce the notion of *meta-variable*, i.e., a triple of the form $\langle VarName, AncestorName, MetaPath\rangle$. This triple denotes a variable $VarName$ in an ancestor node $AncestorName$ (e.g., $BuildingType$ in the node $Building$). The third component of a meta-variable, $MetaPath$, is a list of edge labels (see below) and describes 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 first constraint in $\mathcal{C}_{Story}$ will have to hold only for those instances of node $Story$ which are connected to another instance of node $Story$ through an edge labeled $upper\ story$. Intuitively, a node constraint for the node N will have 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 is defined by: a name, two node names indicating the parent and the child nodes in the *has-part* relation, the cardinality of such relation (expressed as either an integer number or a variable), and a set of constraints. Such constraints may involve the cardinality variable (if any) as well as the variables of the parent node and of any of its ancestors (referred to by using meta-variables). An instance of an edge labeled $label$ connecting a node N with a node M, will be an edge labeled $label$, connecting an instance of N and an instance of M. Let us consider the edges *first story* and *upper story* of our sample model. The former is the edge that relates the building and its first story. It is defined as $\langle first\ story, Building, Story, 1, \emptyset\rangle$. Note that the cardinality is imposed to be 1 and there is no constraint. The edge *upper story* represents the *has-part* relation over two adjacent stories of the building. It is defined as $\langle upper\ story, Story, Story, Card, \mathcal{CC}\rangle$, where the variable $Card$ is defined as $\langle Card, [0,1]\rangle$, while the set of constraints is defined as follows:

$$\begin{aligned}\mathcal{CC} = \{&FloorNum = \langle StoryNum, Building, [first\ story, \star]\rangle \Rightarrow Card = 0,\\ &FloorNum < \langle StoryNum, Building, [first\ story, \star]\rangle \Rightarrow Card = 1\}.\end{aligned}$$

The two constraints in $\mathcal{CC}$ control the number of instances of the node $Story$. An instance of the node $Story$ will have as child another instance of node $Story$, if and only if its floor number is not equal to the number of stories of the building. Intuitively, a cardinality constraint for and edge e will have to hold for each instance of the parent node P in e, such that P has ancestors connected with it through paths matching with $MetaPath$s occurring in the constraint.

As mentioned, a product description consists of a product model together with a set of global 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*). Also, global constraints like `alldifferent` [27] and aggregation constraints can be used to define node model constraints. Intuitively, a node model constraint will have to hold for all the tuples of node instances reached by paths matching with $MetaPath$s occurring in the constraint. The following is an example of cardinality model constraint:

$$\langle upper\ story, Story, Story, Card\rangle \neq \langle roof, Story, Roof, Card\rangle.$$

This constraint states that, given an instance of the node $Story$ the cardinality of the edge $upper\ story$ and $roof$ exiting from it must be different, i.e., an instance of the node $Story$ can not have both an upper story and a roof.

2.2 Process Description

PRODPROC allows one to model a process in terms of activities and temporal relations between them. Moreover, PRODPROC makes it possible to model process resource production and consumption, and to intermix the product and the process modeling phases.

In general, a process consists of: 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 on activity durations.

There are three kinds of activity: *atomic activities*, *composite activities*, and *multiple instance activities*. An *atomic* activity A is an event that happens 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 time and end time of the activity. They define the time interval $[t^{start}, t^{end}]$, subject to the implicit requirement that $t^{end} \geq t^{start} \geq 0$.
- a decision variable $d = t^{end} - t^{start}$ denoting the duration of the activity.
- a flag $exec \in \{0, 1\}$.

When $d = 0$ we say that A is an *instantaneous activity*. If $exec = 1$ holds, A *is executed*, otherwise (namely, if $exec = 0$) A is *not executed*. A composite activity is an event described in terms of a process. Hence, it has associated four variables analogously to an atomic activity, as explained earlier. Moreover, it is associated with 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 four variables (and possibly the sub-process model), a multiple instance activity has associated a decision variables (named $inst$) representing the number of times the activity can be executed.

Temporal constraints between activities are inductively defined starting from *atomic temporal constraints*. Let A and B be to activities. We consider as atomic temporal constraints all the thirteen mutually exclusive binary relations which capture all the possible ways in which two intervals might overlap or not (as introduced by Allen in [3]), and some further constraints inspired by the constraint templates of the language ConDec [19]. The following are some examples of atomic temporal constraints (for lack of space we avoid listing all the possibilities):

1. $A\ before\ B$ to express that A is executed before B.

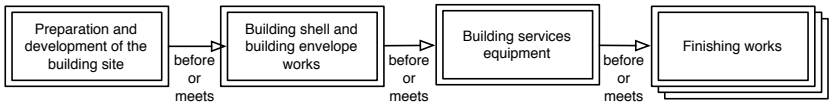

Fig. 2. Temporal constraint network for the building construction process.

2. $A\ meets\ B$ to express that the execution of A ends at time point in which the execution of B starts.
3. $A\ must{-}be{-}executed$ to express that A must be executed.
4. $A\ is{-}absent$ to express that A can never be executed.
5. $A\ not{-}co{-}existent{-}with\ B$ to express that either A or B can be executed (i.e., it is not possible to execute both A and B).
6. $A\ succeeded{-}by\ B$ to express that when A is executed than B has to be executed after A.

The constraints 1 and 2 are two of the binary relations of [3]. The constraints 3–6 have been inspired by the templates used in the language ConDec [19]. 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.

Plainly, the truth of the 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 activities in $\mathcal{A}$. Fig. 2 shows the temporal constraint network for the building construction process. Fig. 3 shows the temporal constraint network for the sub-process represented by the composite activity called "Building services equipment". In the figures, atomic activities are depicted as rectangles, composite activities as nested rectangles, multiple instance activities as overlapped rectangles. Binary temporal constraints are represented as edges whose labels describe the temporal relations. If an activity is involved in a $must\ be\ executed$ or in a $is\ absent$ constraint, it is depicted as a dashed line rectangle or a dotted line rectangle, respectively. A conditional temporal constraints is depicted together with its activation condition.

PRODPROC allows one to specify constraints on resource amounts [15] and activity durations. A resource constraint is a quadruple $\langle A, R, q, TE \rangle$, where A is an activity; R is an 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 the execution of activity A. The possibilities for TE are: $FromStartToEnd$, $AfterStart$, $AfterEnd$, $BeforeStart$, $BeforeEnd$, $Always$, with the obvious meaning. The following is an

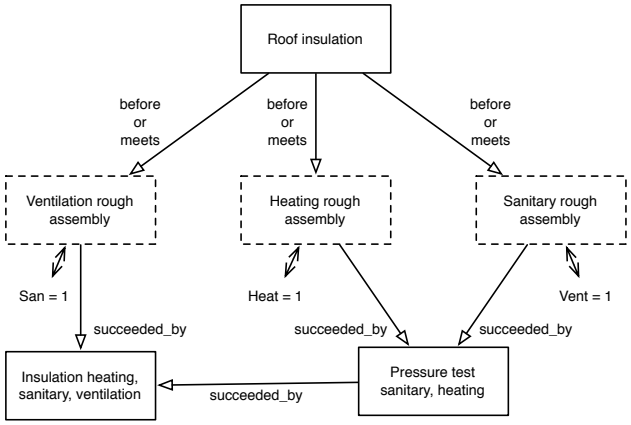

Fig. 3. Temporal constraint network for the composite activity "Building services equipment".

example of resource constraints for the third phase of the building construction process.

$$\langle \text{Roof insulation}, GeneralWorkers, \langle q_{GW}, [-10, -4] \rangle, FromStartToEnd \rangle.$$

This constraint specifies that the number of $GeneralWorkers$ available is reduced of an amount between 4 and 10 during the execution of the activity Roof insulation. All the workers will return available as soon as the activity ends. Note that resource constraints may (implicitly) imply constraints on the number of instances of multiple instance activities. Another form of resource constraints establishes initial level constraints, i.e., expressions defining 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 $iv \in \mathbb{N}$.

An activity duration constraint has the form $\langle A, Constraint \rangle$, where A is the name of an activity, and $Constraint$ may involve the duration of A, process variables, and quantity variables for resource related to A. This is an example of activity duration constraint for the third phase of the building construction process (where $BuildingArea$ is a process variable, and q_T, q_C are quantity variables):

$$\Big\langle \text{Roof insulation}, d = \frac{BuildingArea}{2 \cdot |q_{GW}| + 2 \cdot |q_T| + 3 \cdot |q_C|} \Big\rangle.$$

PRODPROC also allows one to couple elements for modeling a process and elements for modeling a product through constraints involving process variables and product variables. The following are examples in our sample model:

$$\langle Building, sanitary, Card \rangle = San \ ,$$
$$\langle StoryNum, Building, [] \rangle = inst_{Finishing\ works}.$$

For instance, the last one states that the number of stories of a building has to be equal to the value of $inst_{Finishing\ works}$ (i.e., number of times the event $Finishing\ works$ is executed). In general, constraints involving both product and process variables may help to detect/avoid planning impossibilities due to product configuration, and configuration impossibilities due to product configuration, during the configuration of a product.

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* represent one of such variant 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 $\mathcal{N}$ of *node instances*, i.e., tuples of the form $n = \langle N, i, \mathcal{V}_N \rangle$ where N is a node in the product model graph, $i \in \mathbb{N}$ is an index (different for each instance of a node), $\mathcal{V}_N$ 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 node instance n 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 $IT = \langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{E}$ is a set of tuples $f = \langle label, n, m \rangle$ such that there exists an edge $e = \langle label, N, M, Card, \mathcal{CC} \rangle$ in the product model graph, n is an instance of N and m is an instance of M.
- A set $\mathscr{A}_{\mathsf{Cards}}$ of *assignments* for all the instance cardinality variables, i.e., expressions of the form $IC_n^e = k$ where n is an instance of a node N, e is a quintuple $\langle label, N, M, Card, \mathcal{CC} \rangle$, $IC_n^e \equiv Card$, and k is the number of the edges $\langle label, n, m \rangle$, such that m is an instance of M, in the instance tree.
- A set $\mathcal{A}$ of *activity instances*, i.e., pairs $a = \langle A, i \rangle$ where A is the name of an activity such that $exec_A = 1$ and $i \in \mathbb{N}$ is a unique id for instances of A.
- A set $\mathscr{E}$ of flags $exec_A$, one for each activity A such that $exec_A \neq 1$.
- 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.

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, edges constraints, temporal constraints, resource constraints, etc.), appropriately instantiated with variables of node instances and activity instances in the candidate instance.

The (constraint) instantiation mechanism produces a set of constraints on candidate instance variables from each constraint in the PRODPROC model. A candidate instance must satisfy all these constraints to qualify as an instance. We give here an intuitive explanation 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_{node} as the set of k-tuple of node instances $\langle n, n_1, \ldots, n_k \rangle$ where: n is an instance of N; for $i = 1, \ldots, k$ n_i is an instance of N_i, connected with n through a path q_i in the instance tree such that $\texttt{match}(q_i, p_i) = true$ holds. `match` is defined as follows.[1]

[1] Given two lists l_1 and l_2, $l_1 \circ l_2$ denotes their concatenation. We denote with $[x|l]$ the list obtained by prepending the element x to the list l.

$$\texttt{match}(q,p) = \begin{cases} true & \text{if } q = p \\ \texttt{match}(ps, mps) & \text{if } q = [label|ps] \wedge (p = [label|mps] \vee p = [\_|mps]) \\ true & \text{if } p = [\star, label|ps] \wedge \\ & \wedge \exists s.(q = s \circ [label|ps] \wedge \texttt{match}(ps, mps)) \\ false & \text{otherwise} \end{cases}$$

For each k-tuple $t \in L_{node}$, we obtain a constraint on instance variables appropriately substituting variables in c with variables of node instances in t. If c is a constraint for e, given a k-tuple $\langle n, n_1, \ldots, n_k \rangle$ on which to instantiate it, the cardinality occurring in it is substituted with the cardinality variable IC_n^e.

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 $\langle n_1, \ldots, n_k \rangle$, where for $i = 1, \ldots, k$ n_i is an instance of N_i connected by a path q_i with one of its ancestors in the instance tree, such that $\texttt{match}(q_i, p_i) = true$ holds. 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 aggregation or an `alldifferent` constraint, then we define an equivalent constraint on the list consisting of all the node instances of $N_1, \ldots, N_k$ reached by a path matching with the corresponding $MetaPath$.

The instantiation of cardinality model constraint 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 $n_1, \ldots, n_h$ 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 $IC_{n_1}^{e_1}, \ldots, IC_{n_k}^{e_k}$.

Let us now consider process constraints. Let A be an activity, let $a_1, \ldots, a_k$ 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 $\langle a_i, R, q_i, TE \rangle$ for each $i = 1, \ldots, k$, where $q_i = q$ is a fresh variable. 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}, and each quantity variable q with the corresponding variable q_i. Finally, let B an activity, let $b_1, \ldots, b_h$ 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, and of B with b_j. This mechanism can be easily extended to temporal constraints involving more than two activities.

3 Product and Process Configuration

On top of the framework we described in Sect. 2 it is possible to implement a configuration system based on Constraint Logic Programming (CLP) [13]. In this section, we first explain how such a system can support a user through the configuration of a product and its production process. Then, we show how we can generate a CLP program from a model and a (partial) candidate instance.

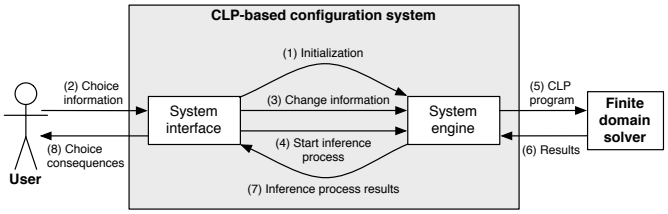

Fig. 4. Configuration process supported by a CLP-based system.

A possible general structure of a configuration process supported by a CLP-based system is pictorially described in Fig. 4. First, the user initializes the system (1) selecting the model of the product/process 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 on the current (partial) candidate instance, and encodes the product/process configuration problem in a CLP program (encoding a Constraint Satisfaction Problem, abbreviated to CSP). Then, it 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).

In the following, we briefly explain how it is possible to obtain a CLP program from a PRODPROC model and a (partial) candidate instance (a candidate instance is partial when there are variables with no value assigned to) corresponding to it. We do this considering only the process side of a model, the operations necessary to obtain CLP variables and constraints for the product side are similar.

Given a PRODPROC model and a corresponding (partial) candidate instance defined by a user, we can easily obtain a CSP $\langle \mathcal{VAR}, \mathcal{DOM}, \mathcal{CONSTR} \rangle$, where $\mathcal{VAR}$ is a set of variables, $\mathcal{DOM}$ is a set of finite domain for variables in $\mathcal{VAR}$, and $\mathcal{CONSTR}$ is a set of constraints on variables in $\mathcal{VAR}$. $\mathcal{VAR}$ will contain a variable for each node instance variable, cardinality variable, activity parameter, process characteristic, resource, and quantity resource variable. $\mathcal{DOM}$ will contain a domain, obtained form the PRODPROC model, for each variable in $\mathcal{V}$. $\mathcal{CONSTR}$ will contain all the constraints that the (partial) candidate instance should satisfy. As we explained in Sect. 2.3, such constraints are determined by an instantiation mechanism. We give here a formalization of such mechanism for the process side of a model. We define a function μ that, given the set of activity instances $\mathcal{A}$, the set $\mathcal{RDC} = \mathcal{R} \cup \mathcal{D} \cup \mathcal{C}$, where $\mathcal{R}$ is the set of resource constraints, $\mathcal{D}$ is the set of activity duration constraints, $\mathcal{C}$ is the set of temporal constraints, generates a set of constraints instantiated on activity instances. To define μ we preliminary need to introduce some basic notions. If c is a temporal constraint $\texttt{acts}(c)$

is the list of activities involved in c. In the following we will denote with a an instance of an activity, and with $\texttt{pInsts}(a)$ the set of instances of the process associated to a composite activity instance a. We say that $a \leftrightarrow_{Act} A$ if and only if a is an instance of A. The function μ is defined as follows:

$$\mu(\mathcal{A}, \mathcal{RCP}, \mathcal{I}) = \bigcup_{a \in \mathcal{A}} \alpha(a) \cup \bigcup_{c \in \mathcal{RCP}} \gamma(c, \mathcal{A}).$$

The function α generates the set of default constraints on duration, start time, and finishing time for an activity instance a:

$$\alpha(a) = \begin{cases} t^{\mathsf{Comp}}(a) \text{ if } a \text{ is a composite activity instance} \\ t(a) \qquad \text{otherwise} \end{cases},$$

$$t^{\mathsf{Comp}}(a) = \{t_a^{start} = \min_{b \in \texttt{pInsts}(a)} t_b^{start}, t_a^{end} = \max_{b \in \texttt{pInsts}(a)} t_b^{end}, \\ t_a^{end} \geq t_a^{start}, d_a = t_a^{end} - t_a^{start}, exec_A = 1\},$$

$$t(a) = \{t_a^{start} \geq 0, t_a^{end} \geq t_a^{start}, d_a = t_a^{end} - t_a^{start}, exec_A = 1\}.$$

The function γ instantiate a constraint c on activity instances in $\mathcal{A}$.

$$\gamma(c, \mathcal{A}) = \begin{cases} \{\langle a, R, q_a, TE\rangle | a \in \mathcal{A} \wedge & \text{if } c \in \mathcal{R} \wedge \\ \quad \wedge\, a \leftrightarrow_{Act} A \wedge q_a = q_A\} & \quad \wedge\, c \equiv \langle A, R, q_A, TE\rangle \\ c & \text{if } c \in \mathcal{R} \wedge \\ & \quad \wedge\, c \equiv initialLevel(R, iv) \\ \{c[d_A/d_a, q_A/q_a] \mid a \in \mathcal{A} \wedge a \leftrightarrow_{Act} A\} & \text{if } c \in \mathcal{D} \\ \{c[A_1/a_1, \ldots, A_k/a_k] \mid & \text{if } c \in \mathcal{C} \\ \quad [A_1, \ldots, A_k] = \texttt{acts}(c) \wedge & \\ \quad \wedge\, [a_1, \ldots, a_k] \in L_{\mathsf{act}}(c, [A_1, \ldots, A_k], \mathcal{A})\} & \end{cases}$$

The function $L_{\mathsf{act}}(c, [A_1, \ldots, A_k], \mathcal{A})$ generates all the k-tuple of activity instances that are instances of activities involved in a constraint c:

$$L_{\mathsf{act}}(c, [A_1, \ldots, A_k], \mathcal{A}) = \{[a_1, \ldots, a_k] \mid \textstyle\bigwedge_{j=1}^{k} (a_j \in \mathcal{A} \wedge a_j \leftrightarrow_{Act} A_j)\}$$

From instantiated resource constraints and CSP variables for resources it is possible to generate a cumulative constraint [1,4]. To obtain CSP constraints from all other constraints it is sufficient to substitute the instance variables with the corresponding CSP variables. Temporal constraints are defined on activities, but it is possible to compile them into propositional formulas on activity durations, starting times, and finishing times. Table 1 shows the translation for some of the atomic temporal constraints.

Let φ and ϑ be temporal constraints, let $\varphi^{\mathcal{P}}$ and $\vartheta^{\mathcal{P}}$ the corresponding propositional formulas. Then $\varphi \; and \; \vartheta$, $\varphi \; or \; \vartheta$, $c \rightarrow \varphi$ and $c \leftrightarrow \varphi$ correspond to $\varphi^{\mathcal{P}} \wedge \vartheta^{\mathcal{P}}$, $\varphi^{\mathcal{P}} \vee \vartheta^{\mathcal{P}}$, $c \Rightarrow \varphi^{\mathcal{P}}$ and $c \Leftrightarrow \varphi^{\mathcal{P}}$, respectively.

Given the constraint satisfaction problem $\mathcal{CSP}$ it is straightforward to obtain a CLP program encoding it, once a specific CLP system has been chosen, e.g., SICStus Prolog, SWI Prolog, or ECLiPse.

4 A Comparison with Existing Product/Process Modeling Tools

In this section, we briefly compare the PRODPROC framework with some of the most important product configuration systems and process modeling tools to put in evidence its strength and limitations.

Atomic temporal constraint	Propositional formula
$A\ before\ B$	$t_A^{start} < t_B^{start} \wedge t_A^{end} < t_B^{start}$
$A\ meets\ B$	$t_A^{start} < t_B^{start} \wedge t_A^{end} = t_B^{start}$
$A\ must{-}be{-}executed$	$exec_A = 1$
$A\ is-absent$	$exec_A = 0$
$A\ not{-}co{-}existent{-}with\ B$	$exec_A + exec_B \leq 1$
$A\ succeeded{-}by\ B$	$exec_A = 1 \Rightarrow exec_B = 1 \wedge t_B^{start} \geq t_A^{end}$

Table 1. Atomic temporal constraints and corresponding propositional formulas.

Product configuration systems based on Answer Set Programming (ASP) [11], e.g., Kumbang Configurator [16], 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 range 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 do not support global constraints, and they encounter some problems in the management of arithmetic constraints related to the so called grounding stage [16].

Systems based on binary decision diagrams (BDDs) for product configuration, e.g., Configit Product Modeler [8], trade the complexity of the construction of the BDD, that basically provides an encoding of all possible configurations [12], for the simplicity and efficiency of the configuration process. 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 [25,26], they basically support flat models only. Moreover, they find it difficult to cope with global constraints. Some attempts at combining BDD with CSP to tackle `alldifferent` constraints have been recently done [17]; 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 general and satisfactory way.

Unlike ASP-based and BDD-based product configuration systems, CSP-based systems allow the user to define non-flat models and to deal with 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. Some well-known CSP-based configuration systems, such as ILOG Configurator [14] and Lava [10], seem to be no longer supported. A recent CSP-based configuration system is Morphos Configuration Engine (MCE) [7]. From the point of view of process modeling, PRODPROC can be viewed as an extension of the 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 node instance variables in constraints.

In [22] the authors present an ontology representing a synthesis of resource-based, connection-based, function-based and structure-based product configuration approches. The PRODPROC framework covers only a subset of these concepts. However, it is not limited to product modeling and it defines a rich (numeric) constraint language, while

it remains unclear to what extent the language used in [22] supports the formulation of configuration-domain specific constraints.

PRODPROC can be viewed as the source code representation of a configuration system with respect to the MDA abstraction levels presented in [9]. 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 an explicit context as OCL constraint must have.

In the past years, different formalisms have been proposed for process modeling. Among them we have: the Business Process Modeling Notation (BPMN) [28], Yet Another Workflow Language (YAWL) [24], DECLARE [19]. Languages like BPMN and YAWL model a process as a detailed specification of step-by-step procedures that should be followed during the execution. They 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, sub-process, and multiple instance activity. The effect of BPMN joins and splits on the process flow can be obtained by 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. As opposed to traditional imperative approaches to process modeling, DECLARE uses a constraint-based declarative approach. DECLARE 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 combine 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., [20], 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.

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. [2]. 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 have separated models for products and processes and, constraints for coupling them too. However, our modeling language is far more complex and expressive than the one presented in [2].

5 Conclusions

In this paper we focused on the problem of product and process modeling and configuration. In particular, we pointed out the lack of a tool covering both physical and production aspects of configurable products. To overcome this absence, we proposed a framework called PRODPROC, that allows one to model a configurable products and its production process. Moreover, we showed how it is possible to build a CLP-based configuration systems on top of this framework, and compared it to existing product configuration systems and process modeling tools.

We have 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. PROPROC and SysML [18] have various commonalities in terms of modeling features, despite the fact that their purposes are different. We plan to further investigate the relations that exists between the two modeling languages. We also plan to experiment our configuration system on different real-world application domains, and to compare it with commercial products, e.g., [5].

References

1. A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.
2. 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.
3. J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26:832–843, 1983.
4. N. Beldiceanu and M. Carlsson. A New Multi-resource cumulatives Constraint with Negative Heights. In P. Van Hentenryck, editor, *CP 2002*, volume 2470 of *LNCS*, pages 63–79. Springer Berlin / Heidelberg, 2006.
5. U. Blumöhr, M. Münch, and M. Ukalovic. *Variant Configuration with SAP*. SAP Press, 2009.

6. D. Campagna. A Graphical Framework for Supporting Mass Customization. In *Proc. of the IJCAI'11 Workshop on Configuration*, pages 1–8, 2011.
7. D. Campagna, C. D. 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.
8. Configit A/S. Configit Product Modeler. `http://www.configit.com`.
9. A. Felfernig. Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization. *IEEE Trans. on Engineering Management*, 54(1):41–56, 2007.
10. 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.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.
12. T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Moller, and H. Hulgaard. 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*, pages 131–138. 2004.
13. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
14. 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.
15. P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artif. Intell.*, 143:151–188, February 2003.
16. 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.
17. 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 *Proc. of the IJCAI'09 Workshop on Configuration*, 2009.
18. OMG. OMG Systems Modeling Language. `http://www.omgsysml.org`.
19. M. Pesic, H. Schonenberg, and W. van der Aalst. DECLARE: Full support for loosely-structured processes. In *EDOC'07*, pages 287–287, 2007.
20. M. L. Rosa. *Managing Variability in Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2009.
21. D. Sabin and R. Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13:42–49, 1998.
22. T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *Artif. Intell. Eng. Des. Anal. Manuf.*, 12:357–372, September 1998.
23. H. Sommer. *Project Management for Building Construction*. Springer, 2010.
24. A. H. M. ter Hofstede, W. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation - YAWL and its Support Environment*. Springer, 2010.
25. 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.
26. E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *Proc. of the 2006 ACM symposium on Applied computing*, SAC '06, pages 409–414. ACM, 2006.
27. W. J. van Hoeve. The alldifferent Constraint: A Survey, 2001.
28. S. A. White and D. Miers. *BPMN modeling and reference guide: understanding and using BPMN*. Lighthouse Point, 2008.

A Model Instantiation and CSP creation

In this section, exploiting the building model introduced in Sect. 2, we show an example of PRODPROC partial candidate instance, and describe the CSP we obtain from it. The purpose of the example is twofold: first, to show how multiple instances of a node affect the constraint instantiation and the CSP corresponding to a model instance; second, to better describe the encoding of the process description into a CSP, in particular the generation of a cumulative constraint from resources and instantiated resource constraints.

Fig. 5 shows the instance tree of the partial candidate instance we consider. It consists of one instance of the root node (i.e., the node *Building*) of the product model graph depicted in Fig. 1, one instance of the node *Electr./Light. service*, two instances of the node *Story*, and one instance of the node *Roof*.

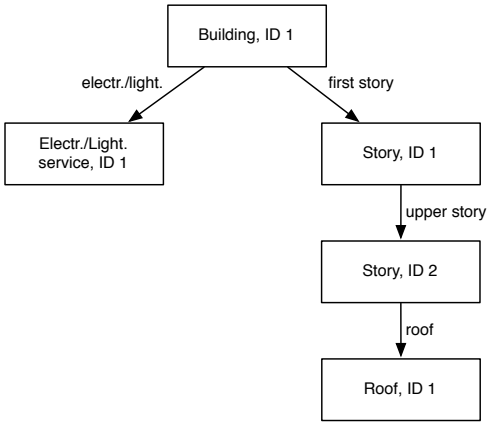

Fig. 5. Instance tree of a building partial candidate instance.

The activity instances and the instantiated temporal constraints of the construction process for the building instance showed in Fig. 5 are depicted in Fig. 6.

In the following we will denote as $\langle Var, Node\text{-}i \rangle$ the variable Var of the instance with id i of the node $Node$. Since we are considering a partial candidate instance, some of the node instance variables and process variables may have a value assigned to. For example, we may have $\langle StoryNum, Building\text{-}1 \rangle = 2$ and $San = 0$.

As mentioned in Sect. 2.3, a candidate instance is an instance if it satisfies all the constraints defined in the model, appropriately instantiated on instance variables. The instantiation of the node constraints for the nodes *Building* and *Story* listed in Sect. 2.1 leads to the following constraints on the variables of node instances in Fig.5.

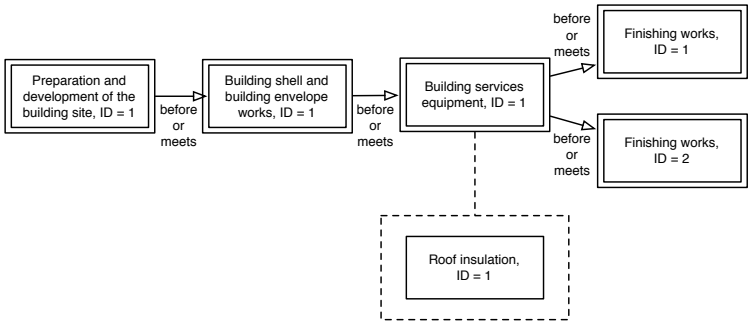

Fig. 6. Activities and temporal constraints for the building partial candidate instance.

$\langle BuildingType, Building\text{-}1\rangle = \text{Warehouse} \Rightarrow \langle StoryNum, Building\text{-}1\rangle = 1,$
$\langle FloorNum, Story\text{-}1\rangle \leq \langle StoryNum, Building\text{-}1\rangle,$
$\langle BuildingType, Building\text{-}1\rangle = \text{Office building} \Rightarrow$
$\quad \Rightarrow \langle Height, Story\text{-}1\rangle \geq 4 \wedge \langle Height, Story\text{-}1\rangle \leq 5,$
$\langle FloorNum, Story\text{-}2\rangle = \langle FloorNum, Story\text{-}1\rangle + 1,$
$\langle FloorNum, Story\text{-}2\rangle \leq \langle StoryNum, Building\text{-}1\rangle,$
$\langle BuildingType, Building\text{-}1\rangle = \text{Office building} \Rightarrow$
$\quad \Rightarrow \langle Height, Story\text{-}2\rangle \geq 4 \wedge \langle Height, Story\text{-}2\rangle \leq 5.$

Instantiating the cardinality constraints for the edge *upper story*, introduced in Sect 2.1, we obtain:

$\langle FloorNum, Story\text{-}1\rangle = \langle StoryNum, Building\text{-}1\rangle \Rightarrow IC^{upper\ story}_{Story\text{-}1} = 0,$
$\langle FloorNum, Story\text{-}1\rangle < \langle StoryNum, Building\text{-}1\rangle \Rightarrow IC^{upper\ story}_{Story\text{-}1} = 1,$
$\langle FloorNum, Story\text{-}2\rangle = \langle StoryNum, Building\text{-}1\rangle \Rightarrow IC^{upper\ story}_{Story\text{-}2} = 0,$
$\langle FloorNum, Story\text{-}2\rangle < \langle StoryNum, Building\text{-}1\rangle \Rightarrow IC^{upper\ story}_{Story\text{-}2} = 1.$

Finally, the instantiation of the cardinality model constraint showed in Sect. 2.1 leads to the constraint:

$$IC^{upper\ story}_{Story\text{-}2} \neq IC^{roof}_{Story\text{-}2}.$$

For each activity instance we have constraints on duration, starting and finishing time. For example, for the composite activity instance "Finishing works" with id 1 we have:

$t^{start}_{\text{Finishing works-1}} = \min_{b \in \texttt{pInsts}(\text{Finishing works-1})} t^{start}_b,$
$t^{end}_{\text{Finishing works-1}} = \max_{b \in \texttt{pInsts}(\text{Finishing works-1})} t^{end}_b,$
$t^{end}_{\text{Finishing works-1}} \geq t^{start}_{\text{Finishing works-1}},$
$d_{\text{Finishing works-1}} = t^{end}_{\text{Finishing works-1}} - t^{start}_{\text{Finishing works-1}}.$

While for the activity instance "Roof insulation" with id 1 we have:

$t^{start}_{\text{Roof insulation-1}} \geq 0, t^{end}_{\text{Roof insulation-1}} \geq 0,$
$t^{end}_{\text{Roof insulation-1}} \geq t^{start}_{\text{Roof insulation-1}},$
$d_{\text{Roof insulation-1}} = t^{end}_{\text{Roof insulation-1}} - t^{start}_{\text{Roof insulation-1}}.$

Instantiating the resource and duration constraints for the activity *Roof insulation* introduced in Sect. 2.2 we obtain:

$\langle$Roof insulation-1, $GeneralWorkers$, $\langle q_{GW}, [-10, -4]\rangle$, $FromStartToEnd\rangle$,

$\big\langle$Roof insulation-1, $d = \dfrac{BuildingArea}{2 \cdot |q_{GW}| + 2 \cdot |q_T| + 3 \cdot |q_C|}\big\rangle$.

The instantiation of the constraint involving both product and process variables showed in Sect. 2.2 leads to the following constraints:

$$IC^{sanitary}_{Building\text{-}1} = San\ ,$$
$$\langle StoryNum, Building\text{-}1\rangle = inst_{Finishing\ works}.$$

From the PRODPROC partial candidate instance we just described and its instantiated constraints, we can construct a CSP with the following characteristics (we use the SWI-Prolog notation for variables, domains and constraints).

- A finite domain (FD) variable for each node instance variable, e.g., for the variable $\langle StoryNum, Building\text{-}1\rangle$ the FD variable `StoryNum_Building_1`;
- A FD variable for each instance cardinality variable, e.g, for $IC^{roof}_{Story\text{-}2}$ the FD variable `IC_roof_Story_2`;
- FD variables for starting time, ending time, duration of each activity instance, e.g., `T_start_Roof_insulation_1`, `T_end_Roof_insulation_1`, and `D_Roof_insulation_1` for the activity instance "Roof insulation" with id 1;
- FD variables for execution flags of activities with no instance;
- FD variables for process and resource variables, e.g., `BuildingArea` for the process variable $BuildingArea$, `GeneralWorkers` for the resource variable $GeneralWorkers$;
- A domain constraint for each FD variable, e.g., `IC_roof_Story_2 in 0..1`;
- A constraint on an FD variable for each assignments, obtained by substituting each instance variable with the corresponding FD variable;
- A constraint on FD variables for each instantiated constraint, obtained by substituting each instance variable with the corresponding FD variable;
- For each composite activity instance, a `minimum` and a `maximum` constraint on start and end times, e.g., for the instance with id 1 of the activity "Finishing works" the constraint `minimum(T_start_Finishing_works_1,Ts)` and the constraint `maximum(T_end_Finishing_works_1,Te)`, where `Ts`, `Te` are respectively the list of start and end times of the activity in the process related to the instance with id 1 of "Finishing works";
- A constraint on FD variables for each instantiated temporal constraint, obtained by substituting start times, end times, and execution flags with the corresponding FD variables in the propositional formula equivalent to the temporal constraint;
- A constraint on FD variables for each instantiated duration constraint, obtained by substituting duration, process and resource variables with the corresponding FD variables;
- A constraint of the form `cumulatives(Tasks,Machines)` where `Tasks` is a list of `task` predicates, one for each instantiated resource constraint, and `Machines` is a list of `machine` predicates, one for each resource. For example, for the resource constraint showed in Sect. 2.2 and the resource $GeneralWorkers$ we define the predicates

```
task(T_start_Roof_insulation_1,D_Roof_insulation_1,
     T_end_Roof_insulation_1,Q_GW,GeneralWorkers,
     FromStartToEnd)

machine(GeneralWorkers,0..10,10)
```

B CLP-based Configuration System

We are using SWI-Prolog to develop a CLP-based configuration system that exploits the close relation that exists between configuration problems and CSPs.[2] In particular, we are using the SWI-Prolog `pce` library to implement the system graphical user interface, and the `clpfd` library for constraint propagation and labeling purposes. The current version of the system is limited to product modeling. Fig. 7 shows the graphical user interface that allows a user to define a product description using PRODPROC. The interface presents (on the left, from top to bottom) controls for graphical element selection, creation of nodes, creation of edges, and creation of sets of model constraints. Moreover, there is a menu named "Check" with controls for checking model syntactic correctness, and for automatically generate product instances to check model validity.

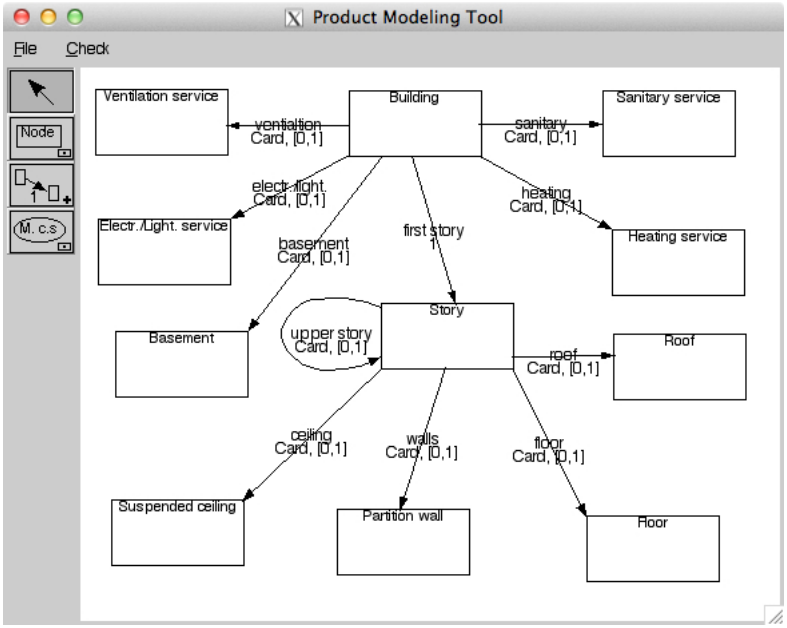

Fig. 7. Graphical user interface for product description creation and checking.

[2] We chose CLP instead of Constraint Programming for the advantages the former gives in terms of rapid software prototyping.