Product and Production Process Modeling and Configuration

Dario Campagna*
Università di Perugia
dario.campagna@dmi.unipg.it

Andrea Formisano
Università di Perugia
formis@dmi.unipg.it

Abstract. Product configuration systems 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, PRODPROC, that allows one to model both a product and its production process. We first introduce our framework by describing how configurable products are modeled. Then, we describe the main features and capabilities offered to model production processes and to link them with the corresponding products models. The configuration task (namely, the procedure that, from a configurable object/activity generates a configured product/process) is then analyzed. We also outline a possible CSP-based implementation of a configurator. A comparison with some of the existing systems for product configuration and process modeling emphasizes that none of the considered system/tools offers the complete set of features supported by PRODPROC for interdependent product and process modeling/configuration.

Key words: Product and process modeling languages, Configuration of product/process models.

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

*Address for correspondence: Università di Perugia
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 identifies a set 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 product configuration systems and allow one to effectively and efficiently deal with the configuration process [33]. They offer functionality for the 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. Most of the current product configuration systems focus only on product configuration and do not cover aspects related to the production process planning (exceptions exist, e.g., the SAP-ERP system). 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 [40] and YAWL [36], do not offer suitable features for specifying production processes and process configuration (e.g., they lack specific constructs for modeling resource production and consumption). Moreover, they lack the capability of modeling, within 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 tailored at covering the whole customization product cycle. As we will see, PRODPROC offers, in a homogeneous setting, all the crucial features that are needed to model complex products together with their production processes. Then, the configuration of a product and its process can be developed simultaneously. Interdependencies between product design and process configuration can be easily rendered through what we call coupling constraints. By means of them, the configuration task can proceed in such a way that the choices made to configure a product (component) can directly offer guidance in configuring its production process, and vice versa.

The paper is organized as follows. Sect. 2 introduces the product/process modeling features of PRODPROC. Sect. 3 describes how PRODPROC instances can be obtained from PRODPROC models through a configuration task. Sect. 4 outlines the main traits of a CSP-based configuration system that relies on commonly available CSP (or CLP) solvers to accomplish the instance generation task. A comparison with some of the existing product configuration systems and process modeling tools is summarized in Sect. 5. An assessment of the work done and of future research direction is given in Sect. 6.

2. A Framework for Product/Production Modeling

In this section we introduce the modeling features offered by PRODPROC that enable the constraint-based modeling of products and processes. We first briefly introduce the underlying constraint language, then
we describe how to model products and processes. Throughout this paper, in order to illustrate PRODPROC’s features, we exploit a working example concerning a rectangular base prefabricated component multi-story building, together with its construction process. Such a case-study is inspired by a real-world complex scenery described in [35] and can be considered as paradigmatic of a class of rather non-trivial “configuration” tasks. A completely detailed treatment of such specific problem can be found in [9]. In this paper, we focus on those aspects of the construction of a building that better illustrate the relevant features of PRODPROC. In our setting, a building is composed by the following parts: story, roof, heating service, ventilation service, sanitary service, electrical/lighting service, suspended ceiling, floor, 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 the purposes of this paper, we consider two types of building:

WAREHOUSE: it is a single story building, it has no mandatory service except 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 and suspended ceiling and floor for each story are mandatory, each story may have a partition wall system.

The characteristics of products and processes will be modeled by means of variables subject to constraints. The constraint language we use involves (constrained) variables of two sorts: each variable $V$ is assigned a finite domain $D(V)$ of admissible values that can be a set of either integer numbers or ASCII strings. We denote a variable $V$ together with its domain as the pair $(V, D(V))$, where $D(V)$ is often specified by enumerating its elements. (Customarily, we denote a set of integers $\{x \in \mathbb{Z} : i \leq x \leq j\}$ as $[i, j]$.) Often, in what follows, the specification of $D(V)$ will be irrelevant or determinable from the context. In such cases, we refer to a variable simply by its name $V$, leaving its domain implicit. A basic term (of sort integer or string) is a variable or value. An arithmetic expression involving integer basic terms (built-up by using the usual operators $+, -, *, /, \mod$) is an integer term. A string value is a string term. If $t_1, t_2$ are terms of the same sort, then $t_1 op t_2$ is a primitive constraint, where $op$ is a comparison operator (e.g., $<, \leq, =, \geq, >, \neq$). Among primitive constraints, we also consider some forms of global constraints. In particular, the allDifferent constraint [39] and the aggregation constraint. The former is defined as follows. Given a list $L$ of basic terms (usually, constrained variables), $\text{allDifferent}(L)$ is satisfied if all elements in $L$ assume a different value. An aggregation constraint has the form $\text{aggConstraint}(f, L, op, val)$ where $f \in \{\text{sum}, \text{avg}\}$, $op \in \{<, \leq, =, \geq, >, \neq\}$, and $val \in \mathbb{Z}$. It is satisfied if $f(L) op val$ holds. A constraint is a propositional combination through Boolean connectives (denoted as usual, by $\lor, \land, \Rightarrow$, etc.) of primitive constraints.

2.1. Product Description

A configurable product is modeled as a directed multi-graph, called product model graph, and a set of model constraints. The nodes of the graph represent the well-defined components of the product (in terms of their configurable characteristics and the constraints they must satisfy). The configurable features are represented by means of variables subject to a set of node constraints. Edges’ labels model the has-part relations between components and constraints are imposed to describe the admissible cardinalities for these relations. (Fig. 1 shows the product model graph for our example.) Such a product description represents a configurable product whose configuration can lead to the definition of its different producible variants, referred to as product instances. Each variant is represented by an instance tree. The nodes of an instance tree correspond to physical components, whose characteristics are fully determined. The tree
Definition 2.1. A PRODUCT MODELING (ProMo) node is a tuple \( \langle N, V, C \rangle \), where \( V \) is a set of node variables, i.e., constrained variables defining the configurable characteristics of the component \( N \). \( C \) is a set of node constraints (to be defined later). A ProMo (directed) edge is a tuple \( \langle \ell, N, M, \text{Card}, CC \rangle \), where the label \( \ell \) is a string representing a description of the relation, \( N \) and \( M \) are the parent and the child nodes in the relation, respectively, \( \text{Card} \) is either an integer or a cardinality variable (it models the cardinality of the relation) and \( CC \) is a set of cardinality constraints (to be defined later).

Definition 2.2. A product model graph is a multi-graph \( G = \langle N, E \rangle \), where \( N \) is a set of ProMo nodes, \( E \) is a set of ProMo edges between nodes in \( N \), and there exists a specific node \( R \in N \), called root node, such that no edges enter it.

Multiple edges enable uniform and succinct modeling of different aspects of the same model component. Indeed, different instances of the same node may implement different functions in the configured product.

Example 2.3. Considering the configuration of a bike, among the components we might want to model and configure we have: the front wheel, the rear wheel, the gears, and the frame. This configurable product can be modeled by a product model graph having two edges between the same pair of nodes—see Fig. 2, where these two edges, connecting the nodes Frame and Wheel, are labeled rear wheel and front wheel, resp. In this case, two different parts of each configured product, the front wheel and the rear wheel, actually are instances of the same configurable component Wheel and play different roles in the configured product. Moreover, they might require different production processes.

Before defining node and cardinality constraints as variants of the notion of constraint seen earlier, we introduce some ancillary notions. A path between two nodes \( N \) and \( M \) in a graph \( G \) is represented by the list of the labels of the edges composing the directed path exiting \( N \) and reaching \( M \). In specifying constraints for a node (say, \( N \)) we use paths to refer to (constrained) variables of other nodes connected to \( N \) in the product model graph \( G \). The set of all paths from \( N \) to \( M \) is denoted by \( \text{path}_G(N, M) \). A node \( N \) is an ancestor of \( M \) if \( \text{path}_G(N, M) \neq \emptyset \). We also define \( \text{ancestors}_G(N) \) as the set of ancestors of \( N \). Note that \( N \in \text{ancestors}_G(N) \). (For simplicity, whenever it does not introduce ambiguity, we will drop the subscript \( G \) in these notations.) The syntactical notion of meta-path allows us to denote collections of paths by using the special symbols _ and * as wild-cards for labels (see Example 2.7):
Definition 2.4. Given a list \( p = [\ell_1, \ldots, \ell_k] \) of \( k \geq 0 \) labels, each list \( p' = [h_1, \ldots, h_k] \), s.t. \( \forall i h_i \in \{\ell_1, \ldots, \ell_k\} \), is a meta-path. Moreover, if \( p = [\ell_1, \ldots, \ell_i, \ell_{i+1}, \ldots, \ell_k] \), is a meta-path, then each list \( p' \) of the forms \( [h_1, \ldots, h_{i-1}, *, h_{i+2}, \ldots, h_k] \) and \( [h_1, \ldots, h_i, *, h_{i+1}, \ldots, h_k] \), s.t. \( \forall i h_i \in \{\ell_1, \ldots, \ell_k\} \), is a meta-path. In all these cases we say that the meta-path \( p' \) matches the list \( p \) and write \( \text{match}(p', p) \). We also say that \( \text{match}(p_1, p_2) \) if there exists \( p_3 \) such that \( \text{match}(p_1, p_3) \) and \( \text{match}(p_3, p_2) \).

In what follows let \( G = \langle N, E \rangle \) be a product model graph. Given a meta-path \( mp \), the set of paths in \( G \) matched by \( mp \) is denoted by \( \text{paths}_G(mp) \).\(^1\) A meta-path \( mp \) connects two nodes \( N, M \) in \( G \) if there exists a path \( p \) from \( N \) to \( M \) in \( \text{paths}_G(mp) \). In such a case we say that \( mp \) leaves \( N \) and reaches \( M \).

Definition 2.5. A meta-variable is a tuple \( \langle V, M, p \rangle \) such that
- \( V \) is a node variable of the node \( M \) and \( p \) is a meta-path that leaves or reaches \( M \); or
- \( V \) is a cardinality variable of an edge labeled \( \ell \) that either leaves or reaches \( M \) and \( p = [\ell] \).

We are now ready to introduce node/cardinality constraints and model constraints. The node and cardinality constraints model compatibility relations that are “local” with respect to has-part relations. Model constraints describe “global” compatibility relations, i.e., relations between configurable characteristics of various components not necessarily related by has-part relations. We proceed by extending the syntax of constraints so as to admit meta-variables as basic terms (together with constants and variables). With slight abuse of notation we will often identify a variable \( V \) in a node \( M \) with the meta-variable \( \langle V, M, [] \rangle \).

Definition 2.6. A node constraint for a node \( N \) in \( G \) is a constraint on (meta-)variables, say \( \langle V, M, p \rangle \), occurring in nodes in \( \text{ancs}_G(N) \), such that \( p \) leaves \( M \) and reaches \( N \).

A cardinality constraint for an edge \( \langle \ell, N_1, N_2, \text{Card}, \text{CC} \rangle \) in \( G \), is a constraint on the variable \( \text{Card} \) and the (meta-)variables \( \langle V, M, p \rangle \) occurring in \( \text{ancs}_G(N_1) \), such that \( p \) leaves \( M \) and reaches \( N_1 \).

Some comments on the previous definition are due. As we will see, constraints involving meta-variables can be seen as “templates” in a product model graph \( G \), representing a collection of “specialized” constraints to be imposed on the configured product characteristics (see, Sect. 3). Since there may be more edges entering the same node in \( G \), in the instance tree (i.e., in the configured product) there might be instances of the same node reached by different paths. Hence, when specifying a node constraint in \( G \) we may need/want to refer to variables of specific ancestors of a node. We may also want to impose specific constraints only on specific instances of nodes. This is done by exploiting meta-paths. Intuitively, a node constraint for a node \( N \) must hold for each instance of \( N \) such that it has ancestors connected with it through paths matching with meta-paths occurring in the constraint. Similarly, a cardinality constraint specified in \( G \) for an edge \( E \) may induce constraints on several instances of \( E \) in the instance tree.

Note that, the requirement that only variables of ancestor nodes can occur in a constraint for a node \( N \) does not represent a real restriction. In fact, the constraints involving variables of descendants of \( N \) can be specified as constraints of such descendants. On the other hand, this requirement might make it easier, for a user, to design or understand a product model, since it reflects the directionality of the has-part relation. It also simplifies the instance generation process (cf., Sect. 3).

Example 2.7. Consider the product model graph in Fig. 1 and its root node \( \langle \text{Building}, \mathcal{V}_{\text{Building}}, C_{\text{Building}} \rangle \). Let \( \mathcal{V}_{\text{Building}} = \{ \langle \text{Type}, \{\text{Warehouse, Office building}\} \rangle, \langle \text{StoryNum}, [1, 3] \rangle, \langle \text{Width}, [7, 90] \rangle, \langle \text{Len},\)

\(^1\)For the sake of simplicity, we are overloading the symbol \( \text{paths} \). This will not be a source of ambiguity.
The constraint in $C_{\text{Building}}$ imposes that a warehouse must have exactly one story. The node $\langle \text{Story}, V_{\text{Story}}, C_{\text{Story}} \rangle$, models a story, with $V_{\text{Story}} = \{(\text{Height}, [3, 15]), (\text{FloorNum}, [1, 3])\}$ and $C_{\text{Story}} = \{\text{FloorNum} = (\text{FloorNum}, \text{Story}, [\text{upper story}]) + 1, \text{FloorNum} \leq \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle, (\text{Type}, \text{Building}, [\text{first story}, \star]) = \text{Office building} \Rightarrow (\text{Height} \geq 4 \land \text{Height} \leq 5)\}$. Note that constraints in $C_{\text{Story}}$ involve, through meta-paths, variables associated with ancestors of the node Story. For example, the first of these constraints is imposed on those instances of node Story which are connected to another instance of node Story through an edge labeled upper story. Let us consider the edges first story and upper story. The former relates the building and its first story and is defined as $\langle \text{first story}, \text{Building}, \text{Story}, 1, \emptyset \rangle$. Hence the cardinality is 1 and there are no constraints. The edge upper story represents the has-part relation over two adjacent stories of the building and is defined as $\langle \text{upper story}, \text{Story}, \text{Story}, \text{Card}, CC \rangle$, where $D(\text{Card}) = \{0, 1\}$. The cardinality constraints in $CC$ control the number of instances of the node Story in the configured product/building. An instance of Story has, as a child, another instance Story if and only if its floor number is not equal to the number of stories of the building; $CC = \{\text{FloorNum} = \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 0, \text{FloorNum} < \langle \text{StoryNum}, \text{Building}, [\text{first story}, \star] \rangle \Rightarrow \text{Card} = 1\}.$

Before stating the definition of ProMo model, we introduce two forms of model constraints: the node model constraints, imposed on variables of nodes not necessarily related by the has-part, and cardinality model constraints involving only cardinalities of edges exiting the same node. Recall that, a node constraint (Def. 2.6) models a property/requirement that is “local” to a specific node $N$: all the involved variables are referred by paths reaching $N$ (i.e., the has-part relation is exploited with respect to the component modeled by $N$). This is not the case for node model constraints: they relate variables of nodes which are not related by the has-part relation. (This difference is reflected in the instantiation mechanism to be seen in Sect. 3.) Thanks to meta-paths, a given node model constraint $c$ of $G$ may impose a constraint on each tuple of node instances (in the instance tree) reached by paths matching the meta-paths occurring in $c$. Similarly, a cardinality model constraints may be used to impose conditions on the set of edges exiting a node (for example, to express mutual exclusion between has-part relations).

**Definition 2.8.** A node model constraint for $G$ is a constraint on (meta-)variables, say $\langle V, M, p \rangle$, in $\mathcal{N}$, such that $p$ reaches $M$ (and leaves anyone of the nodes in $G$). A cardinality model constraint, for a node $N_1$ in $G$, is a constraint involving only the cardinality variables of the edges exiting $N_1$, i.e., edges of the form $\langle t, N_1, N_2, \text{Card}, CC \rangle$ (for any $N_2$). Each cardinality variable in a cardinality model constraint is represented as a meta-variable of the form $\langle \text{Card}, N_1, [t] \rangle$. □

**Example 2.9.** The cardinality model constraint $\langle \text{Card}, \text{Story}, [\text{upper story}] \rangle \neq \langle \text{Card}, \text{Story}, [\text{roof}] \rangle$ imposes that for each instance of Story, the cardinalities of the edges upper story and roof exiting from it (both have $\{0, 1\}$ as domain) cannot be equal, i.e., a story cannot have both an upper story and a roof.

**Definition 2.10.** A ProMo model is a pair $\langle G, \mathcal{MC} \rangle$ such that $G$ is a product model graph and $\mathcal{MC} = \mathcal{NMC} \cup \mathcal{CMC}$, where $\mathcal{NMC}$ (resp., $\mathcal{CMC}$) is set of node (resp., cardinality) model constraints. The compatibility relation pool, $\mathcal{CRP}_G$, is the set of all node, cardinality, and model constraints in $\langle G, \mathcal{MC} \rangle$. □
Table 1. Atomic temporal constraints and corresponding propositional constraints.

2.2. Process Description

Processes are modeled in terms of activities and temporal relations between them, considering resource production/consumption and interdependencies between process executions and product productions. In general, a process is characterized by: a set of activities, a set of constrained variables representing process characteristics and involved resources, a set of temporal constraints between activities, a set of resource constraints, and a set of constraints on activity durations. A MART (acronym for Modeling Activities, Resources, Time) model represents configurable processes that, through a configuration phase, can lead to the definition of different (configured) executable processes. Such a phase generates configured activities as instances of the activities in the model. Intuitively, a model describes a mixed planning and scheduling problem whose solutions represent executable processes. Note that, also because of interactions with the product configuration phase, there may be activities whose execution is enabled/required or not and activities that may or may not produce/consume resources. Activities are of three kinds: atomic, multiple-instance, and composite. Let us start by defining atomic and multiple-instance activities.

**Definition 2.11.** An atomic activity \( A \) is an event occurring in a time interval. It is modeled through:

- The integer decision variables, \( t^\text{start}_A, t^\text{end}_A \), and \( d_A \), denoting the start time, the end time, and the duration, resp., of the activity. These constrained variables define the execution time-interval \( [t^\text{start}_A, t^\text{end}_A] \) for \( A \) and are subject to the implicit requirements \( 0 \leq t^\text{start}_A \leq t^\text{end}_A \) and \( d_A = t^\text{end}_A - t^\text{start}_A \).
- An activity execution flag \( \text{exec}_A \), i.e., a variable with domain \( \{0, 1\} \).

\( A \) is instantaneous if \( d_A = 0 \). The flag \( \text{exec}_A \) models the occurrence of the activity: \( A \) is executed if \( \text{exec}_A = 1 \), and not executed otherwise.

A multiple-instance activity \( B \) models an event that may occur multiple times. It has associated a flag \( \text{exec}_B \) and a constrained variable \( \text{inst}_B \) representing the number of instances of \( B \) that will be part of a (configured) process. If \( \text{inst}_B > 0 \) then \( \text{inst}_B \) instances of \( B \) are executed, otherwise no instances of \( B \) are executed. The implicit constraint \( \text{inst}_B > 0 \iff \text{exec}_B = 1 \) is imposed. A multiple-instance activity \( B \) includes \( \text{inst}_B \) triples of parameters \( t^\text{start}_B, t^\text{end}_B, d_B \), one for each of the \( \text{inst}_B \) admissible instances.

In order to model configurable processes, a MART model might include various forms of constraint on model variables. There are two kinds of variables: resource variables, namely, integer constrained variables intended to model resource amounts (e.g., number of workers, quantity of steel, etc.); and
process variables, namely, constrained variables modeling process characteristics (e.g., type of welding for the fabrication of a bike frame, type of pressure test for building sanitary system, etc.).

The constraints in a model may be: temporal constraints, resource constraints, activity duration constraints, product-related constraints. Temporal constraints between activities are defined starting from the atomic temporal constraints listed in Table 1. These are the 13 mutually exclusive binary relations capturing the possible ways in which two intervals might overlap or not [3], and some constraints inspired by the constraint templates of ConDec [31]. Observe that the truth of an atomic temporal constraint is related with the actual execution of the activities it involves. For example, if a constraint \( A_{before} B \) is specified in a model, then it have to be imposed only when \( \text{exec}_A = 1 \land \text{exec}_B = 1 \) holds.

**Definition 2.12.** An an atomic temporal constraint is a temporal constraint. Moreover, if \( \varphi \) and \( \psi \) are temporal constraint, then \( \varphi \land \psi \) and \( \varphi \lor \psi \) are temporal constraints. If \( c \) is a constraint (on process variables of the model), then \( c \Rightarrow \varphi \) (resp., \( c \Leftrightarrow \varphi \)) is an if-conditional (resp., iff-conditional) temporal constraint, stating that \( \varphi \) must hold whenever (resp., if and only if) \( c \) holds.

**Resource constraints** [27] define how activities require and affect the availability of resources.

**Definition 2.13.** Let \( A \) be an activity and \( R \) a resource variable. A resource constraint is a tuple \( \langle A, R, q, \text{TE} \rangle \), where \( q \) is an integer basic term and \( \text{TE} \) is a time extent. The value of \( q \) defines the quantity of resource \( R \) consumed (if \( q < 0 \)) or produced (if \( q > 0 \)) by \( A \), while \( \text{TE} \) defines the time interval where the availability of resource \( R \) is affected by \( A \). Possible time extents are: \( \text{FromStartToEnd} \), \( \text{AfterStart} \), \( \text{AfterEnd} \), \( \text{BeforeStart} \), \( \text{BeforeEnd} \), \( \text{Always} \), with the obvious intuitive meaning.

An initial level constraint for \( R \) defines the amount of resource that is available at the time the process starts. It has the form \( \text{initialLevel}(R, \text{val}) \), where \( \text{val} \in \mathbb{N} \).

Note that resource constraints may implicitly define temporal constraints between activities. For example, if a resource \( R \) can be produced only by \( A \) and has a null initial level, the temporal constraints \( \langle A, R, 2, \text{AfterEnd} \rangle \) and \( \langle B, R, -2, \text{AfterStart} \rangle \), impose that \( B \) can start only after the end of \( A \).

**Definition 2.14.** An activity duration constraint for an activity \( A \) is a constraint on \( d_A \), process variables, and variables in the set \( Q_A = \{ q | q \text{ is an integer variable in a resource constraint } \langle A, R, q, \text{TE} \rangle \} \).

**Example 2.15.** The following resource constraint specifies that the number of available GeneralWorkers is reduced of an amount \( q_{GW} \) (between 4 and 10) during the execution of the activity “Roof insulation”. All the workers will return available as soon as the activity ends: \( \langle \text{Roof insulation}, \text{GeneralWorkers}, (q_{GW}, [-10, -4]), \text{FromStartToEnd} \rangle \). An example of activity duration constraint concerning the building construction process is: \( d_{\text{Roof insulation}} = \text{BuildingArea} / (2 \cdot |q_{GW}|) \) where \( \text{BuildingArea} \) is a process variable and the variable \( q_{GW} \) is the number of GeneralWorkers involved in the activity “Roof insulation”. This constraint states that the duration of “Roof insulation” depends on the area occupied by the building and on the number of workers assigned to the activity.

In modeling production processes, it might be the case that resources produced by activities actually are components of the configured product (i.e., nodes in the product model) that, in turn, might be used by other activities. Product-related constraints are used to specify such situations and hence they involve both activities and product model nodes:
Definition 2.16. Let $A$ and $B$ be activities. A product-related constraint has one of the forms:

- $A$ produces $kN$ for $B$
- $B$ needs $kN$ from $A$

for $k \in \mathbb{N}^+$ and a node $N$ having (at least) one incoming edge with an associated cardinality variable. □

Remark 2.17. Product-related constraints can be expressed in terms of the other forms of constraints. For example, the constraint $A$ produces $kN$ for $B$ is equivalent to the conjunction:

$$\langle A, R_N, q_A, AfterEnd \rangle \land \langle B, R_N, −k, AfterStart \rangle \land initialLevel(R_N, 0) \land \sum_{C \in CE_N} C \geq \sum_{A \in A, A \text{ produces } N} q_A$$

where $R_N$ is a new resource variable and $CE_N$ is the list of cardinality variables of edges entering $N$. A constraint $B$ needs $kN$ from $A$ is equivalent to the constraint $A$ produces $kN$ for $B$.

Notice also that, if $A$ is a multiple instance activity, each of its instances contributes in producing $R_N$, i.e., each instance of $A$ in the configured process might produce $q_a$ units of $R_N$. Conversely, if $B$ is a multiple-instance activity, each of its instances needs $k$ units of $R_N$. □

Before defining the notion of composed activity we need to introduce MART models, since a composed activity in a configurable process is described in terms of a (nested) sub-process.

Definition 2.18. A MART model is a tuple $\langle A, \mathcal{V}, \mathcal{C}, \mathcal{P}, \mathcal{R}, \mathcal{D} \rangle$, where:

- $A$ is a set of atomic, multiple-instance, and composite activities (to be defined next).
- $\mathcal{V} = \mathcal{V}_\text{Res} \cup \mathcal{V}_\text{Proc}$ is a set of model variables (resource and process variables).
- $\mathcal{C}$ is a set of temporal constraints on activities in $A$.
- $\mathcal{P}$ is a set of product-related constraints.
- $\mathcal{R} = \bigcup_{A \in A} \mathcal{R}_A$, where $\mathcal{R}_A$ is the set of resource constraints on the activity $A$ and resources in $\mathcal{V}_\text{Res}$.
- $\mathcal{D} = \bigcup_{A \in A} \mathcal{D}_A$, where $\mathcal{D}_A$ is the set of duration constraints on $d_A$ and variables in $\mathcal{V}_\text{Proc} \cup \mathcal{Q}_A$. □

Next definition introduces composite activities as events described in terms of sub-processes.

Fig. 3 shows the activities and temporal constraints of the building construction process and the sub-process associated with the composite activity “Building services equipment”.

Definition 2.19. A composite activity $A$ in a MART model $\langle A, \mathcal{V}, \mathcal{C}, \mathcal{P}, \mathcal{R}, \mathcal{D} \rangle$ consists of:

- A MART (sub-)model $\langle A_A, \mathcal{V}_A \cup \mathcal{V}_C, \mathcal{C}_A, \mathcal{P}_A, \mathcal{R}_A, \mathcal{D}_A \rangle$.
- The variables $t_A^{\text{start}}, t_A^{\text{end}},$ and $d_A$ (start time, end time, and duration, resp., of $A$) subject to the requirements $t_A^{\text{start}} = \min_{B \in A_A} t_B^{\text{start}}, t_A^{\text{end}} = \max_{B \in A_A} t_B^{\text{end}},$ and $d_A = t_A^{\text{end}} - t_A^{\text{start}}$.
- An activity execution flag $\text{exec}_A$.

$A$ is instantaneous if $d_A = 0$. It is executed if $\text{exec}_A = 1$ or not executed otherwise. Composite activities can be multiple. The number of instances of a multiple-instance composite activity $A$ is modeled by a variable $\text{inst}_A$ and each activity instance is characterized by a triple $t_A^{\text{start}}, t_A^{\text{end}},$ and $d_A$ (cf., Def. 2.11). □

2.3. PRODPROC Models

A PRODPROC model is composed of a product model, a process model, and a collection of coupling constraints that defines the coupling of the two models. Coupling constraints involve product model and process model variables. More specifically, let $M_{\text{Prod}}$ be a product model and let $\mathcal{V}_{\text{Nodes}}$ and $\mathcal{V}_{\text{Card}}$ be the node variables and the cardinality variables in $M_{\text{Prod}}$, resp. Let, moreover, $\mathcal{V}_{\text{Proc}}$ be the set of process model variables (including the parameters $\text{inst}_A$ defined for multiple-instance activities). Then, a
Figure 3. Temporal constraints for the building construction process (left) and the composite activity “Building services equipments” (right). Atomic, composite, and multiple-instance activities are here represented by rectangles, nested rectangles, and overlapping rectangles, respectively. Binary temporal constraints are represented as labeled edges. Thick lines and dotted lines denote activities involved in a must be executed or in an is absent constraints, respectively. For conditional temporal constraints the activation conditions are indicated.

coupling constraint is an equality constraint on variables in the set \( V_{\text{Nodes}} \cup V_{\text{Card}} \cup V_{\text{Proc}} \). As done before, we exploit meta-variables in coupling constraints: \( \langle V, M, p \rangle \) identifies the node variable \( V \) of a node \( M \) reached by the meta-path \( p \). Similarly, a cardinality variable \( \text{Card} \) of an edge \( \langle t, N_1, N_2, \text{Card}, CC \rangle \) is identified by the meta-variable \( \langle \text{Card}, N_1, [t] \rangle \). Variables in \( V_{\text{Proc}} \) are simply referred to by their names.

Definition 2.20. A PRODPROC model is a tuple \( \langle M_{\text{Prod}}, M_{\text{Proc}}, C_{\text{Coup}} \rangle \) where \( M_{\text{Prod}} \) is a PROMO model, \( M_{\text{Proc}} \) is a MART model, and \( C_{\text{Coup}} \) is a set of coupling constraint between \( M_{\text{Prod}} \) and \( M_{\text{Proc}} \). 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 process configuration, during the configuration of a product. Two examples follow:

Example 2.21. Consider the models in Figures 1 and 3 and the constraints: \( \langle \text{Card}, \text{Building}, [\text{sani- tary}] \rangle = \text{San} \) and \( \langle \text{StoryNum}, \text{Building}, [] \rangle = \text{inst}_{\text{Finishing works}} \). The former relates the cardinality of the edge sanitary, exiting the node Building, to the process variable \( \langle \text{San}, \{0,1\} \rangle \), that models the presence/absence of the activity “Sanitary rough assembly”. Hence, this activity is executed if and only if there is an instance of the node Sanitary service. The other constraint states that the number of stories of a building has to equal the value of \( \text{inst}_{\text{Finishing works}} \) (i.e., number of times that “Finishing works” is executed).

Example 2.22. Fig. 3 shows that the activities “Ventilation rough assembly” and “Heating rough assembly” are related by an equals constraint. Hence, they must be executed at the same time. Moreover, let us assume that their execution depends on two process variables, \( \langle \text{Vent}, \{0,1\} \rangle \) and \( \langle \text{Heat}, \{0,1\} \rangle \), that occur in the coupling constraints \( \langle \text{Card}_1, \text{Building}, [\text{ventilation}] \rangle = \text{Vent} \) and \( \langle \text{Card}_2, \text{Building}, [\text{heating}] \rangle = \text{Heat} \). Let us suppose that when configuring the building the available resources do not permit to satisfy the equals constraint. That is, it is not possible to realize buildings with ventilation and heating services. If during the product configuration, instances of Ventilation service and Heating service are created, the
coupling constraints help to detect that no plan can be computed, and to determine that the current configuration is not valid. Vice versa, if during the process configuration Heat is imposed to be equal to 0, then also Vent is imposed to be equal to 0, and coupling constraints help to avoid inconsistencies in the product instance by imposing the absence of instances of Heating service and Ventilation service.

3. From PRODPROC Models to Instances

Obtaining a configured product, and an executable process for its production, consists in determining its customized components and their relationships in a way that both user’s preferences and all the constraints defined in the model are satisfied. A PRODPROC instance represents one of the producible variants and its production process. In what follows we describe the instantiation mechanism yielding such instances through a configuration task. This task may be accomplished by using a configuration system, through an interactive process. During this process, the user (i) selects some of the components that will compose the configured product, (ii) chooses suitable values for (some of) their configurable characteristics, (iii) selects activities to be executed, (iv) chooses suitable values for (some of) the process model parameters. The configuration system supports the user checking the validity of his/her choices. In the positive case, the process iterates. Otherwise, the user has to modify some of the choices (s)he made. Alternatively, the entire task may be automatically accomplished by the configuration system, too.

Product Instances. A configured product is represented as an instance tree. In the description of a configured product, physical components are represented as instances of nodes of the product model graph. An instance of a node \( N \) consists of the name \( N \), a unique index, and a set of variables. Each variable has a value assigned. The instance of the root node is the root of the configured product tree.

In what follows let \( \langle G, MC \rangle \) be a ProMo model and \( G = \langle N, E \rangle \) be a product model graph.\(^2\)

**Definition 3.1.** A node instance \( n \) of a node \( \langle N, V, C \rangle \) of \( G \) is a tuple \( \langle N, i, V' \rangle \), where \( i \in \mathbb{N} \) is an index (univocally identifying the node among all instances of \( N \)) each variable in \( V' \) is in one-to-one correspondence with a variable in \( V \) (and the two share the domain of values). An edge instance \( e \) of an edge \( E = \langle \ell, N, M, Card, CC \rangle \) of \( G \) is a tuple \( \langle \ell, n, m \rangle \), where \( n \) and \( m \) are node instances of \( N \) and \( M \), resp. For such an edge \( e \) a cardinality variable \( IC_{n}^{E} \) (having the domain \( D(Card) \)) is defined. \( \square \)

We denote the fact that a node \( n \) (resp., an edge \( e \)) is an instance of \( N \) (resp., \( E \)) by writing \( N \sim n \) (resp., \( E \sim e \)). We extend this notation also to other elements of a model, such as, for example, node variables: we write \( V \sim v \) if a variable \( V \) in \( N \) corresponds to the variable \( v \) of \( n \). Moreover, if \( V \) is a node variable, let \( N_{V} \) (resp., \( n_{V} \)) denote the node (resp., instance node) where \( V \) is defined.

**Definition 3.2.** A ProMo instance is a tuple \( \langle T, \mathcal{A}_{Nodes} \cup \mathcal{A}_{Cards} \rangle \), such that \( T = \langle N, E \rangle \) is a finite tree, called instance tree, where \( N \) (resp., \( E \)) is a set of node (resp., edge) instances of \( G \). Moreover,

- \( \mathcal{A}_{Nodes} \) is a set of assignments for all the variables in \( N \), (i.e., expressions of the form \( V = \text{val} \)).
- \( \mathcal{A}_{Cards} \) is a set of assignments for all the cardinalities of edges in \( E \). Namely, expressions of the form \( IC_{n}^{E} = k \), with \( E = \langle \ell, N, M, Card, CC \rangle \) and \( k \) is the number of the edges \( \langle \ell, n, m \rangle \), in \( E \) (for \( M \sim m \)).

\(^{2}\)We will use upper case letters to denote elements of the PRODPROC models (such as \( N \) for a node or \( A \) for an activity, . . . ) and lower case letters for the elements of the PRODPROC instances (such as \( n \) for a node instance or \( a \) for an activity instance, . . . ).
\(\mathcal{A}_\text{Nodes} \cup \mathcal{A}_\text{Cards}\) does not violate any constraint in \(\mu_{\text{prod}, T}(c)\) for all \(c \in \mathcal{CRP}_G\), where the function \(\mu_{\text{prod}, T}(\cdot)\), to be seen, properly instantiates the constraints in the compatibility relation pool of \(G\). □

Note that, although the tree composing a PrOMO instance has to be finite, it is possible to come up with product model graphs that admit only infinite tree as configurations. This can be done via cycles involving unbounded cardinality variables, or also via more complex interactions between constraints imposed on cardinalities. Consequently, any concrete system implementing an instantiation mechanism has to deal with potentially non-terminating computations (searching for a finite configuration for a model that admits only infinite ones). We will comment further on this point in Sect. 4.

For each constraint \(c\) in the compatibility relation pool of \(G\), the function \(\mu_{\text{prod}, T}\) mentioned in Def. 3.2 generates a set of constraints on variables of \(T\). In what follows, we first give an intuitive description of how the instantiation mechanism works on different constraint types. Then, we provide a formal definition of the instantiation function \(\mu_{\text{prod}, T}\).

Let \(\text{nvars}(c)\) be the list of node meta-variables occurring in a constraint \(c\). For a cardinality model constraint \(c\), let \(\text{cvars}(c)\) be the list of cardinalities meta-variables in \(c\).

Let \(c\) be a node constraint for a node \(N\) or a cardinality constraint for an edge \(E\) between nodes \(N\) and \(M\). Suppose that \((V_1, N_1, p_1), \ldots, (V_k, N_k, p_k)\) are the meta-variables occurring in \(c\) (notice that the \(N_i\)'s are not necessarily pairwise distinct). We first determine the set of tuples of node instances \(n, n_1, \ldots, n_k\) of \(T\) such that \(N \sim n\) and for \(i = 1, \ldots, k\) \(N_i \sim n_i\) and there is a path from \(n_i\) to \(n\) matched by \(p_i\). Each of these tuples identifies a correspondence between variables \(V_1 \sim v_1, \ldots, V_k \sim v_k\). A constraint for the PrOMO instance is obtained by substituting in \(c\) each \(V_i\) with \(v_i\). Node model constraints are instantiated in a slightly different way, the only difference being that each \(p_i\) reaches the node \(N_i\) instead of leaving it (cf. Def 2.8). The translation is slightly different if \(c\) is a global node model constraint (see below). If \(c\) is a cardinality model constraint for the edges \(E_1, \ldots, E_k\) exiting from a node \(N\), let \(n_1, \ldots, n_h\) be the instances of \(N\) in the instance tree. Then \(h\) instances for \(c\) are generated. Each one of them is obtained by substituting its cardinality variables with the variables \(IC_{E_1}, \ldots, IC_{E_k}\).

Let \(M_{\text{prod}} = (G, N, MC \cup CMC)\) be a PrOMO model, and let \(T\) be an instance tree. Since each constraint \(c\) in the model may involve variables defined in various nodes of \(G\), to correctly instantiate \(c\), suitable instances of such nodes must exist in \(T\). We define below some predicates encoding such a precondition to constraint instantiability, for the various kinds of constraint we deal with:

\[
\begin{align*}
ancp_T(n, p) & \iff \exists m, q \ (m \in \text{anc}_T(n) \land q \in \text{paths}_T(m, n) \land \text{match}(p, q)) \\
\text{inst}_\text{nec}_T(c, N) & \iff \exists n \ (N \sim n \land \forall (V, M, p) \in \text{nvars}(c) \exists m, q \ (M \sim m \land q \in \text{paths}_T(m, n) \land \text{match}(p, q))) \\
\text{inst}_\text{nmnc}_T(c) & \iff \forall (V, N, p) \in \text{nvars}(c) \exists m \ (M \sim m \land \text{anc}_T(m, p)) \\
\text{inst}_\text{cmc}_T(c) & \iff \exists n \ (N \sim n \ \text{for} \ (V, N, p) \ \text{in} \ c)
\end{align*}
\]

In particular, \(\text{anc}_T(n, p)\) denotes the property that the node \(n\) has an ancestor \(m\) in \(T\) such that the meta-path \(p\) matches the (unique) path from \(m\) to \(n\). Let \(c\) be a primitive constraint. Then, \(\text{inst}_\text{nec}\) determines the instantiability of node constraints and cardinality constraints, with respect to a node \(N\). Intuitively, \(c\) can be instantiated if there exists an instance \(n\) of \(N\) and, for each meta-variable \((V, M, p)\) occurring in \(c\), there exists an instance \(m\) of \(M\) connected with \(n\) through a path matched by \(p\). Similarly, \(\text{inst}_\text{nmnc}\) determines if a node model constraint \(c\) can be instantiated. It can be instantiated if for each meta-variable \((V, M, p)\) in \(c\), there exists an instance \(m\) of \(M\) reached by any path matched by \(p\). Finally, \(\text{inst}_\text{cmc}\) determines if a cardinality model constraint \(c\) can be instantiated, that is, if there exists an
instance of the node whose exiting edges’ cardinalities are involved in $c$. The function $\mu_{\text{Prod},T}$ is defined as follows for primitive constraints and readily extends to any propositional combination of constraints.

\[
\mu_{\text{Prod},T}(c) = \begin{cases} 
\{\sigma : \sigma \in S_{\text{node}}(c, N)\} & \text{if } c \text{ is a node constraint such that } \text{inst}_{\text{necc}}(c, N) \\
\{c[C/IC_{E}^{\ell}] : \sigma : \sigma \in S_{\text{node}}(c, N)\} & \text{if } c \text{ is a cardinality constraint on the edge } E = (l, N, M, C, CC) \text{ and } \text{inst}_{\text{necc}}(c, N) \\
\{\sigma : \sigma \in S_{\text{nmc}}(c)\} & \text{if } c \text{ is a node model constraint, it is not a global constraint, and } \text{inst}_{\text{nmc}}(c) \\
\{\text{allDifferent}(Lglb_{T}(L))\} & \text{if } c \text{ is a node model constraint allDifferent}(L) \\
\{\text{aggConstraint}(f, Lglb_{T}(L), op, k)\} & \text{if } c \text{ is a node model constraint aggConstraint}(f, L, op, k) \\
\{\sigma : \sigma \in S_{\text{cmc}}(c)\} & \text{if } c \text{ is a cardinality model constraint and } \text{inst}_{\text{cmc}}(c) 
\end{cases}
\]

where $\sigma$ denotes a variable substitution, i.e., a list of pairs of terms (variables, in the case at hand) $[V_{1}/v_{1}, V_{2}/v_{2}, \ldots]$, and $\sigma$ is the constraint obtained from $c$ by substituting each $V_{i}$ with $v_{i}$. $S_{\text{node}}(c), S_{\text{nmc}}(c),$ and $S_{\text{cmc}}(c)$ determine the tuples of node instance variables to be used in instantiating a constraint. Notice that global constraints occurring as node, cardinality, and cardinality model constraints are treated as the other primitive constraint. This is not the case for global node model constraints. Indeed, global node model constraints on the list of variables $L$ is rendered, in the PROMO instance, through a global constraint imposed on the list of all existing instance variables of all variables in $L$. $Lglb_{T}(c)$ determines such a list.

\[
S_{\text{node}}(c, N) = \left\{ \sigma : \sigma \text{ is a substitution } [V_{i}/v_{i}, \ldots, V_{h}/v_{h}] \text{ s.t. } \text{nvars}(c) = (V_{i}, N, p_{1}), \ldots, (V_{h}, N, p_{h}) \wedge \forall i \in \{i, \ldots, h\} \{V_{i} \sim v_{i} \wedge n_{v_{i}} \in \{ t \} \in \text{nituples}_{T}(N, c)\}\right\}
\]

\[
S_{\text{nmc}}(c) = \left\{ \sigma : \sigma \text{ is a substitution } [V_{i}/v_{i}, \ldots, V_{h}/v_{h}] \text{ s.t. } \text{nvars}(c) = (V_{i}, N, p_{1}), \ldots, (V_{h}, N, p_{h}) \wedge \forall i \in \{i, \ldots, h\} \{V_{i} \sim v_{i} \wedge \text{ancp}(n_{v_{i}}, p_{i})\}\right\}
\]

\[
S_{\text{cmc}}(c) = \left\{ \sigma : \sigma \text{ is a substitution } [C_{1}/(IC_{1})_{E_{1}}^{\ell_{1}}, \ldots, C_{h}/(IC_{h})_{E_{h}}^{\ell_{h}}] \text{ s.t. } \text{cvars}(c) = (C_{1}, N, [\ell_{1}]), \ldots, (C_{h}, N, [\ell_{h}]) \wedge N \sim n \wedge \forall i \in \{i, \ldots, h\} \{E_{i} = (\ell_{i}, N, M, C_{i}, CC_{i}) \text{ is an edge in } G\}\right\}
\]

\[
Lglb_{T}(c) = \left\{ v : \langle V, M, p \rangle \text{ in } c \text{ s.t. } \exists q \in \text{mapath}_{T}(n, p, M) \wedge N \sim n \wedge M \sim m \wedge V \sim v\right\}
\]

where $\text{nituples}_{T}(N, c)$ determines all the sets of instance nodes to be considered in generating different instances of a given node constraint $c$ of a node $N$:

\[
\text{nituples}_{T}(N, c) = \left\{ n \cup s : N \sim n \wedge s \text{ is a set of all node instances } \{m_{1}, \ldots, m_{k}\} \text{ s.t. } \wedge \right\}
\]

\[
\exists \langle V, M, p \rangle \in \text{nvars}(c) \{(M \in \text{anc}_{G}(N) \wedge M \sim m_{j}) \wedge q \in \text{path}_{T}(m_{j}, n) \text{ is the shortest path in } \text{mapath}_{T}(n, p, M)\}\}
\]

where $\text{mapath}_{T}(n, p, M) = \{ q : \exists m, q \in \text{paths}_{T}(m, n) \wedge M \sim m \wedge \text{match}(p, q)\}$ is the set of paths in $T$ that are matched by the meta-path $p$ and connect an instance of node $M$ with $n$. Hence, each set $\{n, m_{1}, \ldots, m_{k}\} \in \text{nituples}_{T}(N, c)$ is such that $n$ is an instance of $N$, and the $m_{i}$‘s are ancestors of $n$ connected to it through paths matched by meta-paths occurring in meta-variables in $c$.

**Process Instances.** Let $A$ be the set of all activities in a given MART model $M_{\text{Proc}}$ (including those constituting composed activities) and let $C, R, P,$ and $D$ be the sets of all temporal, resource, product-related, and duration constraints of $M_{\text{Proc}}$, respectively (including those occurring in composed activities).

**Definition 3.3.** A **MART instance** is a tuple $\langle T, F, \mathcal{A} \rangle$ such that:

- $T$ is a finite set of activity instances, i.e., couples of the form $a = \langle A, i \rangle$ where $A$ is an activity in $M_{\text{Proc}}$ such that $\text{exec}_{A} = 1$ and $i \in \mathbb{N}$ is an index univocally identifying $\langle A, i \rangle$ among all instances
of $A$. In particular, for each instance $a$ of $A$ the parameters $t^\text{start}_a$, $t^\text{end}_a$, and $d_a$ are introduced. (Each of such variables shares the domain with the corresponding variable of $A$.)

- $\mathcal{F}$ is a set of flags $\text{exec}_A$, one for each activity $A$ in $M_{\text{Proc}}$ that is not executed (such that $\text{exec}_A = 0$).

- $\mathcal{A}$ is a set of assignments for all model variables and activity parameters (i.e., time instant variables, duration variables, execution flags, resource variables,...), that is, expressions of the form $P = \text{val}$ where $P$ is a model variable or an activity parameter.

- $\mathcal{A}$ does not violate any constraint $c \in \mu_{\text{Proc},I}(C \cup R \cup \mathcal{D})$ where the function $\mu_{\text{Proc},I}(\cdot)$, to be seen, properly instantiates the constraints in $M_{\text{Proc}}$. $\square$

Before defining the instantiation function $\mu_{\text{Proc},I}(\cdot)$, let us briefly describe how the instantiation mechanism works. Let $A$ be an activity and $a_1, \ldots, a_k$ be its instances. If $c$ is the resource constraint $\langle A, R, q, TE \rangle$, then it is instantiated for each $a_i$, obtaining the constraint $\langle a_i, R, q_i, TE \rangle$, where $q_i$ is a fresh variable. If $c$ is an activity duration constraint for $A$, then we introduce a constraint for each $a_i$ by substituting, in $c$, $d_A$ with $d_{a_i}$ (and similarly for the other variables in $c$). Concerning temporal constraints, let us consider the case of a single atomic constraint $c$ involving activities $A$ and $B$. Let $b_1, \ldots, b_h$ be $B$‘s instances. We introduce a constraint on activity instances for each ordered tuple $\langle i, j \rangle$ (for $i \in \{1, \ldots, k\}, j \in \{1, \ldots, h\}$), by substituting in $c$ each occurrence of $A$ (resp., $B$) with $a_i$ (resp., $b_j$). This procedure is generalized to treat temporal constraints involving more than two activities.

As regards product-related constraints, we can preliminarily translate them in terms of constraints of the other forms, as outlined in Remark 2.17. Hence, without loss of generality, we will not consider them in the instantiation phase.

If $c$ is a temporal constraint, let $\text{acts}(c)$ denote the list of activities involved in $c$. Let, moreover, $\text{pInsts}(a)$ be the set of instances of activities in the process associated to the composite activity instance $a$. Given an activity instance $a$ and an activity $A$, we write $A \sim a$ if and only if $a$ in an instance of $A$.

Let $\mathcal{CRD} = C \cup R \cup \mathcal{D}$, then the function $\mu_{\text{Proc},I}$ is defined as follows:

$$\mu_{\text{Proc},I}(\mathcal{CRD}) = \bigcup_{a \in I} \text{deflt}(a) \cup \bigcup_{c \in \mathcal{CRD}} \gamma_I(c).$$

where $\text{deflt}(\cdot)$ generates the implicit constraints on duration, start and finishing time (cf., Def. 2.11):

$$\text{deflt}(a) = \begin{cases} \{t^\text{start}_a = \min_{b \in \text{pInsts}(a)} t^\text{start}_b, & t^\text{end}_a = \max_{b \in \text{pInsts}(a)} t^\text{end}_b, \\
 & t^\text{end}_a \geq t^\text{start}_a, & d_a = t^\text{end}_a - t^\text{start}_a, & \text{exec}_A = 1 \} \quad \text{if } a \text{ is composite} \\
\{t^\text{start}_a \geq 0, & t^\text{end}_a \geq t^\text{start}_a, & d_a = t^\text{end}_a - t^\text{start}_a, & \text{exec}_A = 1 \} \quad \text{otherwise} \end{cases}$$

The function $\gamma_I(\cdot)$ instantiates a constraint $c$ on activity instances. It is defined as follows (where, by slightly abusing the notation, we denote by $\{A_1/a_1, \ldots, A_k/a_k\}$ a substitution of activities instances for activities in temporal constraints):

$$\gamma_I(c) = \begin{cases} \langle a, R, q^R_a, TE \rangle \ : a \in I \land A \sim a \} & \text{if } c = \langle A, R, q^R_A, TE \rangle \\
\{c\} & \text{if } c = \text{initialLevel}(R, \text{val}) \\
\{c[a/a] \ : \sigma \text{ is a substitution } [A_1/a_1, \ldots, A_k/a_k] \text{ s.t.} \\
\text{acts}(c) = [A_1, \ldots, A_k] \land \forall i \in \{1, \ldots, k\} A_i \sim a_i \} & \text{if } c \in \mathcal{D} \end{cases}$$

**PRODPROC Instances.** Let us extend our notation and denote by $\text{nvars}(c)$ (resp., $\text{cvars}(c)$) the list of node variables (resp., cardinality variables) occurring in a coupling constraint $c$. The instantiability condition for coupling constraints is encoded by the predicate:
\[ \text{inst\_coup}_T(c) \Leftrightarrow (\forall (V, N, p) \in \text{nvars}(c) \exists m (M \sim m \land \text{ancp}_T(m, p))) \land (\forall (V, N, p) \in \text{cvars}(c) \exists n (N \sim n)) \]

while \( S_{\text{coup}}(c) \) generates all substitutions \( \sigma \) needed to instantiate a coupling constraint:

\[
S_{\text{coup}}(c) = \{ \sigma : \sigma \text{ is a substitution } [V_1/v_1, \ldots, V_h/v_h, C_1/IC_{m_1}, \ldots, C_k/IC_{m_k}] \text{ s.t.} \\
\text{nvars}(c) = [(V_1, N_1, p_1), \ldots, (V_h, N_h, p_h)] \land \text{cvars}(c) = [(C_1, M_1, [f_1]), \ldots, (C_k, M_k, [f_k])] \land \\
\forall i \in \{i, \ldots, h\} (V_i \sim v_i \land \text{ancp}_T(n_{v_i}, p_i)) \land \\
\forall i \in \{i, \ldots, k\} (M_i \sim m_i \land E_i = \{e_1, M_1, M'_1, C_1, C'_1\} \text{ is an edge in } G) \} \]

(Note that different instantiations of the same coupling constraint differ only by node and cardinality instance variables. They all share the same process variables.) Finally, we have the following definition:

**Definition 3.4.** Given a PRODPROC model \( \langle M_{\text{Prod}}, M_{\text{Proc}}, \mathcal{C}_{\text{Coupl}} \rangle \), a PRODPROC instance consists of a PROMO instance \( I_{\text{Prod}} \) of \( M_{\text{Prod}} \) and a MART instance \( I_{\text{Proc}} \) of \( M_{\text{Proc}} \) such that the assignments for variables in the \( I_{\text{Prod}} \) and \( I_{\text{Proc}} \) do not violate any constraint in

\[
\bigcup_{c \in \mathcal{C}_{\text{Coupl}}} \{ \alpha \sigma : \sigma \in S_{\text{coup}}(c) \}
\]

4. A CSP-based Configuration System

A possible structure of a configuration process supported by a CSP-based system is depicted in Fig. 4. We can imagine a scenery in which the user incrementally develops the configuration task by interacting with the system. In each step of this interaction the user makes some choices about some configurable features of the product/process defined in the PRODPROC model and obtains guidance from the system that provides a validation (or a rejection) of their feasibility. In such a way, a “partial” instance (i.e., an instance that, being “under development”, contains variables with no value yet assigned to) is incrementally refined until a complete configuration is achieved. Each interaction evolves as follows. First, the user initializes the system (1) by selecting the model of the product/process to be configured and starts making her/his choices by using the system interface (2). The interface communicates to the system engine (i.e., the module of the system that maintains a representation of the product/process under configuration and checks the 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 takes place, the user, through the system interface, can activate the engine inference process (4) to make the system validate/refute the update. The engine instantiates PRODPROC constraints on the current instance and encodes the product/process configuration problem in a CSP (clearly, at least to a certain extent, this can be done just by updating the previously generated encoding, accordingly to the last updates). Then, it uses a finite domain solver to validate user’s choices (5) and, once the inference phase ends (6), it returns to the interface the results of the computation (7). In its turn, the interface communicates to the user the consequences of her/his choices on the (partial) configuration (8).

As mentioned before, it is possible to specify model graphs that admit only infinite configurations. Hence, the configuration task may not terminate under certain circumstances. To avoid infinite computations one can develop a checking procedure that detects whether or not a given model admits only (in)finite instances. This represent a non-trivial task and is a challenging subject for future work. A viable possibility we adopted consists in imposing a bound on the size of the instance sought for (e.g., on the number of nodes of the instance tree).
In the following, we briefly outline how to obtain a CSP encoding from a PRODPROC model and a (partial) instance. This encoding provides a CSP-semantics of PRODPROC instances. (An alternative approach would consist in adopting Generative CSP as target framework for our encoding. Exploring this option goes beyond the purposes of this paper. The interested reader is referred to [11].) The CSP encoding can be plainly obtained because PRODPROC constraints, except temporal and resource constraints, are just syntactic variants of the analogous CSP counterparts. Instead, semantics of temporal and resource constraints can be easily given in terms of a cumulatives global constraint [1, 5].

Given a PRODPROC model and a corresponding (possibly partial) instance, we define a CSP \((V, D, C)\), where \(V\) is a set of finite-domain (FD) CSP-variables, \(D\) is a set of domain constraints for variables in \(V\), and \(C\) is a set of CSP-constraints on \(V\). \(C\) contains the encodings of all the constraints that the instance should satisfy. Such constraints are determined by an instantiation mechanism, as explained in Sect. 3. For each distinct variable \(v\) in the PRODPROC instance we introduce a CSP variable whose domain constraint reflects the finite domain \(D(v)\). Terms and primitive constraints are immediately translated by using the standard CSP operators, primitive constraints, and Boolean connectives and by substituting the instance variables with the corresponding CSP variables. Slightly more involved is the translation of temporal and resource constraint on activities. As regards instance activities, for each activity \(A\) of the model we introduce the CSP variables \(\text{exec}_A\) (modeling the flag \(\text{exec}_A\)), and \(t_{\text{start}}^i\), \(t_{\text{end}}^i\), and \(d_{ai}\), for each index \(i\) identifying a different instance of \(A\) (cf., Def. 3.3). In case \(A\) is a multiple instance activity, we also introduce a CSP variable \(\text{inst}_A\) modeling the parameter \(\text{inst}_A\). The encoding of temporal constraints is achieved by using their equivalent propositional formulas as listed in Table 1, taking into account that each one of such constraints has to be imposed only when (instances of) all involved activities are executed. Resource constraints require a more specific treatment. The scheduling problem they implicitly define can be encoded in CSP by means of a cumulatives global constraint of the form \(\text{cumulatives}(\text{Tasks}, \text{Machines})\) where \(\text{Tasks}\) is the list of all elements of the form \(\text{task}(t_{\text{start}}^i, d_{ai}, t_{\text{end}}^i, q, v, TE)\), such that \(a_i\) is an instance of index \(i\) of an activity \(A\), \(\langle a_i, V, q_{ai}, TE\rangle\) is a resource constraint, and \(Q\) and \(V\) are the CSP translations of \(q_{ai}\) and \(V\), resp. The list \(\text{Machines}\) contains all elements of the form \(\text{machine}(v, D(v), k)\) such that \(\text{initialLevel}(V, k)\) is an initial level constraint and \(V\) is the CSP variable corresponding to \(V\) (having \(D(V)\) as domain). Notice that not all available CSP-solvers offer such a general form of cumulatives global constraint. If this is the case, this needed constraint can be implemented, for example, by exploiting simpler (global) constraints or by using the techniques described in [27]. Actually, this is what has been done in [9], where the author concretely implemented in SWI-Prolog an executable version of the above outlined translation.

---

If the CSP solver does not support string domains, these can be easily mapped into integer domains and constraints. This is always possible because, given a PRODPROC model, the set of all strings that may occur in an instance is finite.
Assuming correctness of the CSP solver (and the rendering of the cumulatives constraint, if needed), the soundness of the CSP encoding immediately follows from the fact that CSP constraints are essentially just syntactic variants of \textsc{Pro}d\textsc{Pro}c constraints. As a result we obtain that each solution of the CSP directly encodes an admissible configured product together with its configured production process.

Observe that, during the user-system interactions the CSP solver can be used to reduce the domains associated with variables (and, consequently, to restrain future user’s choices), or to detect the inconsistency of the encoded CSP (due to user’s assignments that violate constraints or to inconsistencies of the original \textsc{Pro}d\textsc{Pro}c model). Also the structure of the product under configuration may be affected. For example, the inferences made by the system may imply that, in order to ensure/recover feasibility of the configuration task, further node instances and/or activity instances have to be added to the \textsc{Pro}d\textsc{Pro}c instance, or, on the contrary, that there are too many components in the instance tree.

The CSP encoding can also be used as core of a automated configurator that, given a \textsc{Pro}d\textsc{Pro}c model, obtains a \textsc{Pro}d\textsc{Pro}c instance by performing a search in a tree, where each node corresponds to a partial instance, and by exploiting the CSP encoding to prune branches of the search tree.

To show the effectiveness of the proposed framework we implemented a CLP-based modeling system. Such a prototype supports, through a graphical interface, the specification of product/process models. The user can also activate basic tasks such as correctness check and product/process instance generation. See [9] for a complete description of the tool.

5. A Comparison with Existing Product/Process Modeling Tools

In this section, we compare \textsc{Pro}d\textsc{Pro}c with some of the available product configurators and process modeling tools. The former can be classified as ASP-based, BDD-based, and CSP-based systems.

Systems based on answer set programming (ASP) demonstrated applicable in a relevant range of application domains. An example of ASP-based product configuration systems is the Kumbang Configurator [29] which is specifically tailored to the modeling of software product families. In general, systems of this kind offer support to fully declarative modeling but offer limited support for features such as global constraints, arithmetic computations, and structured data. These drawbacks are inherited from the underlying ASP-solver and are mainly related to the so called grounding stage. However, it is reasonable that the continuous advances in the ASP-solver technology will circumvent these drawbacks and make ASP-based configurators more competitive. Indeed, work is under way both theoretically and practically in order to avoid the grounding phase and to enhance ASP with modules, preferences, ontologies, resource handling, and integration with CSP systems (see [16, 12, 28, 21, 13] among many).

Systems based on binary decision diagrams (BDDs), such as Configit Product Modeler (see \url{www.configit.com}), trade the complexity of the construction of the BDD, that basically provides an encoding of all possible configurations [19], for the simplicity and efficiency of the configuration process. Such systems suffer from some significant limitations. First, even though some work has been done on the introduction of modules [37, 38], they basically do not support variable cardinalities in product models. Moreover, they find it difficult to cope with global constraints. (As an example, consider that the \texttt{alldifferent} constraint lead to exponential space complexity for BDD construction.) Attempts at combining BDD with CSP to tackle \texttt{alldifferent} constraints have been recently done [30]; however, also in these cases only constant cardinalities for the \texttt{has-part} relations are allowed. We are not aware of any BDD-based system that deals with global constraints in a general and satisfactory way.
<table>
<thead>
<tr>
<th>System/Tool</th>
<th>Mod</th>
<th>N-f</th>
<th>ACs</th>
<th>CCs</th>
<th>GCs</th>
<th>AsTs</th>
<th>CAs</th>
<th>MAs</th>
<th>RCs</th>
<th>DCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kumbang</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Configit</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MCE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Ontologies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Feature Diagrams</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>BPMN</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>YAWL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>DECLARE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>~</td>
<td>x</td>
</tr>
<tr>
<td>SysML</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>C-iEPC [26]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Provop [20]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2. Comparison summary. The signs ✓, x, and ~ denote available, absence, and limited support, respectively, for a feature. The features are: modular structure of product models (Mod), has-part relations with non-fixed cardinalities (N-f), arithmetic constraints (ACs), cardinality constraints (CCs), global constraints (GCs), modeling of activities and temporal relations (AsTs), modeling of composite activities (CAs), modeling of multiple-instance activities (MAs), resource constraints (RCs), duration constraints (DCs).

As regards CSP-based systems, they usually support both variable cardinalities and global constraints, but such higher modeling expressiveness has a cost. In fact, whereas backtrack-free configuration algorithms for CSP-based systems are often inefficient, whereas 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 [22] and Lava [18] (which is based on Generative-CSP), seem to be no longer supported. A recent CSP-based configuration system is Morphos Configuration Engine (MCE) [10]. From the point of view of product modeling, ProDPROC extends the MCE modeling language by introducing graph structured product models (instead of trees); cardinality variables, cardinality constraints and cardinality model constraints; the use of meta-paths and meta-variables.

In [34] the authors present an ontology representing a synthesis of resource-based, connection-based, function-based and structure-based product configuration approaches. 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 [34] supports the formulation of configuration-domain specific constraints. ProDPROC can be also viewed as the source code representation of a configuration system with respect to the MDA abstraction levels presented in [17]. ProDPROC product modeling elements can be mapped to UML/OCL in order to obtain platform specific (PSM) and platform independent (PIM) models.

Feature Diagrams (FDs) [23] are widely used in software product line configuration. A feature is a system property that is relevant to some stakeholder and is used to capture some commonalities or to discriminate among systems in a family. Features are organized in feature diagrams, namely trees of features with the root representing a concept. Feature models are feature diagrams plus additional information such as feature descriptions, binding times, priorities, stakeholders, etc. A number of extensions for FDs have been proposed. Examples are: feature diagrams based on directed acyclic graphs [25]; cardinality-based feature models [15]; feature models with extra functional features [6]. Staged configuration [14] is
a process that allows the incremental configuration of feature models through a step-wise specialization. Despite the existing extensions, feature diagrams lack different features available in PRODPROC, such as the multi-graph structure of the model and the cardinality (model) constraints.

In the past years, different formalisms have been proposed for process modeling. Among them we mention: the Business Process Modeling Notation (BPMN) [40], Yet Another Workflow Language (YAWL) [36], DECLARE [31]. Languages like BPMN and YAWL model a process as a detailed specification of a step-by-step procedure to be followed during the execution. They adopt an imperative approach in process modeling and all possibilities have to be entered into the models by specifying their control-flows. 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 through 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 intended to directly support all control flow patterns. PRODPROC has in common with YAWL the notion of task, multiple-instance task, and composite task. As opposed to traditional imperative approaches, DECLARE uses a constraint-based declarative approach to implicitly determine the possible ordering of activities. DECLARE and PRODPROC share the notion of activity and the use of temporal constraints to specify the control flow. The set of PRODPROC atomic temporal constraints is not as large 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 BPMN and YAWL with a declarative approach for control flow definition. It also presents features that, to the best of our knowledge, are not presents in other existing process modeling languages. Examples are: resource variables, resource constraints, and activity duration constraints. Thanks to these features, PRODPROC is suitable for modeling production processes as well as mixed scheduling and planning problems related to production processes. Moreover, 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 [32, 26, 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.

SysML (see, www.omg.sysml.org) is a general-purpose modeling language for systems engineering applications. Its modeling constructs can be partitioned into static structural constructs and dynamic behavioral constructs. Structural constructs (in particular blocks and constraints) may be used for defining models of configurable products. That is, SysML may be used as a modeling language for a product configurator. However, it does not allow one to model a product as a graph of components. Concerning process modeling, the behavioral constructs of SysML can be used to describe the flow of control and flow of inputs and outputs among process activities. However, they are geared toward modeling system behaviors and not toward (configurable) production processes. Even if SysML has activities and constructs for control flow description, it lacks specific constructs to model resources and resource constraints. Duration constraints are not included in the current version of SysML.

Alternative activities [4] and Extended RCPSP [24] are two formalism dealing with the modeling of mixed scheduling and planning problems, and could be used to model configurable processes. Alternative activities allow to model processes with multiple plans involving activities that may be not
executed. The different plans are modeled using XOR nodes. All the possible plans for a process have to be explicitly represented, this is in contrast with our declarative approach for plans definition. Extended RCPSP is similar to PRODPROC in terms of their process modeling capabilities. For example, it supports the modeling of activities whose execution is optional and the use of constraint on activities executions. However, it lacks some forms of temporal constraints and it offers simpler forms of resource constraints w.r.t. those included in our framework. Anyway, extensions of Extended RCPSP toward a more declarative and richer formalism for modeling configurable products are foreseeable.

In [41] Zhang et al. propose the use of a system of nested colored object-oriented Petri nets with changeable structures (NOPNs-cs) to carry out production configuration modeling. Despite the fact that NOPNs-cs can lead to the definition of highly detailed process models, their imperative nature and complexity make them not completely suitable for modeling just those elements of a production process that are relevant during product configuration (e.g., resources and time constraints).

An important feature of PRODPROC is the possibility of explicitly specifying constraints that couple products with their processes. The only works we are aware of about the coupling of product/process modeling and configuration, are the ones by Aldanondo et al., e.g, [2]. They propose to simultaneously consider product configuration and process planning as two CSPs and suggest to link the two models by using coupling constraints, in order to propagate decision consequences between the two problems. The development of PRODPROC has been inspired by [2], nevertheless, it offers a richer set of constructs and features for product/process modeling. Table 2 summarizes a comparison, considering a set of relevant features, of PRODPROC with the mentioned systems and tools for product and process modeling.

6. Conclusions

In this paper we focused on the problem of product and process (coordinate) modeling and configuration. We pointed out the lack of tools covering both physical and production aspects of configurable products in a satisfactory manner. To overcome this absence, we proposed a framework, called PRODPROC, that allows one to model a configurable product together with its production process. We introduced a constraint-based language to model configurable product and activities and to correlate their characteristics. The instantiation of the resulting model corresponds to the execution of a configuration task. In each step of this task, the user can proceed by mixing the customizations of product’s characteristics and aspects of its production. The coupling constraints act as a bridge between product instances and process instances and propagate, in both directions, the effect of user choices. Consequently, choices made in configuring a product component may affect the set of choices available for the production phase. This can be exploited to offer the user guidance in configuring its production process. In case of failures, due to some unsatisfiable combinations of user’s requirements, the system may exploit coupling constraints to provide the user with justifications and detailed explanations for the failure or even to suggest ways to recover from it by means of a minimal modification of the invalid product+process instance.

We showed how it is possible to realize a CSP-based configuration systems on top of this framework and compared it to existing product configuration systems and process modeling tools. The comparison put in evidence that the PRODPROC framework presents features that, to the best of our knowledge, are not present in other existing systems. These are, for example, graph-structured product models, model constraints, activity duration constraints, resource variables and constraints. The fact that all these crucial
features belong to a single (declarative) framework, allows one to model products, their production processes, and to couple products with processes by using constraints in a single and homogeneous setting.

We illustrated the viability of the proposal by means of a working example concerning a non-trivial, albeit rather unusual, “configuration” task: the modeling and configuration of a building. The adequacy of PRODPROC in supporting such a complex task, witnesses its applicability in a variety of contexts where large-scale configuration problems arise. We have already implemented a first prototype of a CSP-based configuration system, called PRODPROC Modeler, that supports the specification of a product and its production process by using the PRODPROC graphical language [8]. The system also implements basic tasks such as checking model syntactic correctness and performing automated generation of product/process instances. We plan to extend PRODPROC Modeler to a full-fledged configuration system supporting not only modeling (and automated configuration), but interactive configuration too (including, for instance, a sub-system for failure justification). Another extension that is subject of current research is the inclusion of real-valued variables in the modeling framework. This would enable the modeling of product characteristics that may assume value in a continuous domain. Such an extension would require the integration (or even the development) of a solver for mixed domains (to handle constraints involving both integer and real variables). We also plan to experiment our configuration system on various different application domains and to compare it with commercial products, e.g., [7].

References


