22nd International Joint Conference on Artificial Intelligence
IJCAI 2011

Proceedings

Workshop on Configuration

Saturday, July 16, 2011
Barcelona, Spain

Kostyantyn Shchekotykhin

Dietmar Jannach

Markus Zanker

Table of contents

Workshop organization

Workshop co-chairs

Kostyantyn Shchekotykhin, *AAU, Austria*
Dietmar Jannach, *TU Dortmund, Germany*
Markus Zanker, *AAU, Austria*

Program committee

Patrick Albert*, IBM, France*
Tomas Axling*, Tacton Systems AB, Sweden*
Claire Bagley*, Oracle Corporation, USA*
Conrad Drescher*, University of Oxford, UK*
Alexander Felfernig*, TU Graz, Austria*
Albert Haag*, SAP AG, Germany*
Alois Haselböck*, Siemens AG, Austria*
Lothar Hotz*, Universität Hamburg, Germany*
Ulrich Junker
Tomi Männistö*, Aalto University, Finland*
Klas Orsvarn*, Tacton System AB, Sweden*
Markus Stumptner*, University of South Australia, Australia*
Barry O'Sullivan*, Cork Constraint Computation Centre, Ireland*
Juha Tiihonen*, Aalto University, Finland*

Preface

In many cases competitiveness of modern products is defined by the degree of customization, i.e. the ability of a manufacturer to adapt a product according to customer requirements. Knowledge-based configuration methods support the composition of complex systems from a set of adjustable components. However, there are two important prerequisites for a successful application of knowledge-based configuration in practice: (a) expressive knowledge representation languages, which are able to capture the complexity of various models of configurable products and (b) powerful reasoning methods which are capable of providoing services such as solution search, optimization, diagnosis, etc. The Configuration Workshop aims to bring together industry representatives and researchers from various areas of AI to identify important configuration scenarios found in practice, exchange ideas and experiences and present original methods developed to solve configuration problems.

The workshop continues the series of successful Configuration Workshops started at the AAAI'96 Fall Symposium and continued on IJCAI, AAAI, and ECAI since 1999. During this time the focus of the events broadened from configuration approaches applied to traditional products such as cars, digital cameras, PC, telecommunication switches or railway interlock systems to configuration of software and services available on the Web. In parallel, research in the field of constraint programming, description logic, non-monotonic reasoning, fundamentals of configuration modeling and so forth pushed the limits of configuration systems even further.

The papers selected this year for presentation on the Configuration Workshop continue a recent trend in the research community and focus on modeling and solving of configuration and reconfiguration problems. The papers of the workshops strongly correlate with the topics of the two invited talks, which discuss application and evaluation of different knowledge representation and reasoning methods as well as the necessity of adequate information support for an end-user involved in the configuration process.

Kostyantyn Shchekotykhin, Dietmar Jannach and Markus Zanker
July 2011

Invited talks

Fabrizio Salvador
Instituto de Empresa Business School, Madrid, Spain

Understanding configuration process proficiency: Interactive effects of information provisioning and learning

Anecdotal evidence and case-based research point to the importance of the effective management of information on feasible product configurations in order to achieve good responsiveness. However, no empirical, large-sample test of this contention has been done as yet. This study begins to close this research gap by testing a theory-derived model of how information relating to product configuration determines the responsiveness in serving customers. We find that availability of information supporting the product configuration task indeed allows companies to serve their clients faster and more accurately. We also find these benefits to be mediated by the availability to learn from past product configurations.

Gerhard Friedrich
Alpen-Adria Universität, Klagenfurt, Austria

Configuration – a reality check for knowledge representation and reasoning

It is more than 30 years that knowledge representation and reasoning (KRR) methods have been applied to solve configuration problems. Therefore, it is not surprising that the history of knowledge based configuration reflects almost all tops and flops of KRR. Indeed many successful applications based on KRR methods were deployed showing the utility of KRR to solve practically highly relevant problems. However, this success may mislead to the conclusion that configuration is solved. In this talk I will show by a very simple though relevant configuration example that on the one hand current reasoning methods discovered valuable yet unknown solutions but on the other hand many interesting research questions have to be answered in order to expand the applicability of KRR for practical configuration problems. In particular, I will focus on the effects of different knowledge representation formalisms, different models, variants of symmetry breaking constraints and local versus complete search. Interestingly, it turned out that folklore knowledge may be a false friend.

A Graphical Framework for Supporting Mass Customization [*]

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

Abstract

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

1 Introduction

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

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

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

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

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

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

2 A Graphical Framework for Product/Process Modeling

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

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

2.1 Product Modeling Features

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

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

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

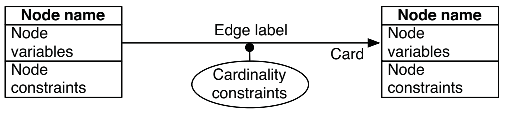


Figure 1: Graphical representation of nodes and edges.

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

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

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

While in *Frame constraints* we may have the constraint

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

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

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

and the constraints

$$\mathit{WheelType} = \langle \mathit{FrameType}, \mathit{Frame}, [\_]\rangle, \quad (1)$$

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

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

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

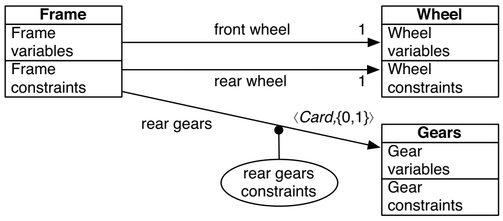


Figure 2: Fragment of bicycle product model graph.

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

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

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

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

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

Intuitively, a cardinality constraint for and edge e has to hold for each instance of the parent node N in e, such that N has ancestors connected with it through paths matching with $MetaPath$s occurring in the constraint.

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

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

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

2.2 Process Modeling Features

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

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

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

We say that A is an *instantaneous activity* if $d = 0$. *A is executed* if $exec = 1$ holds, otherwise (i.e., if $exec = 0$) A is *not executed*. A *composite* activity is an event described in terms of a process. It has associated the same parameters of an atomic activity, and a model of the process it represents. A *multiple instance* (atomic or composite) activity is an event that may occur multiple times. Together with the

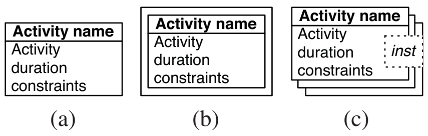


Figure 3: Graphical representation of activities.

usual parameters (and possibly the process model), a multiple instance activity has associated an integer decision variable (named $inst$) representing the number of times the activity can be executed. Note that the execution/non-execution of activities determines different instances of a configurable process. Figures 3a, 3b, and 3c, show the graphical representation of atomic activities, composite activities, and multiple instance activities, respectively.

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

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

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

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

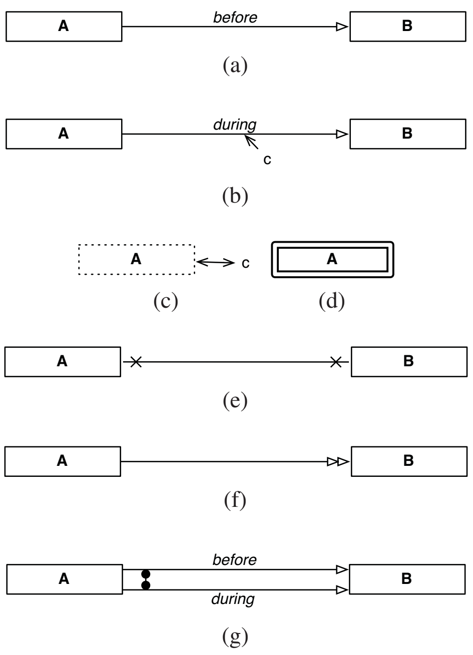


Figure 4: Graphical representation of temporal constraints.

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

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

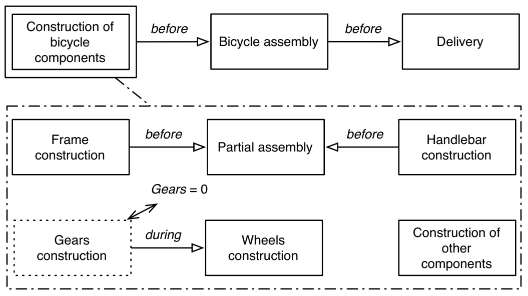


Figure 5: Temporal constraint network of the bicycle production process.

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

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

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

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

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

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

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

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

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

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

Product related constraints are another type of constraints coupling product elements with process elements. They make it possible to define resource constraints where resources are product components. More precisely, a product related constraint is a constraint on activities and product nodes that implicitly defines resource constraints, and constrains on process and product variables. A product related constraint has the form $A\ produces\ n\ N\ for\ B$, where A and B are activities, $n \in \mathbb{N}^+$, and N is the name of a node in the product model graph, having (at least) one incoming edge having associated a cardinality variable. Such a product related constraint corresponds to the following PRODPROC constraints:

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

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

$$\text{Wheel construction } produces\\ 2\ Wheel\ for \text{ Bicycle assembly.}$$

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

2.3 PRODPROC Instances

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

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

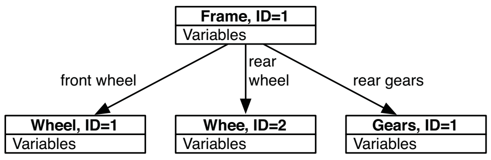


Figure 6: Instance tree for a bicycle.

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

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

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

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

$$\langle WheelType, Wheel_1^I\rangle = \langle FrameType, Frame_1^I\rangle, \tag{4}$$

$$\langle WheelType, Wheel_2^I\rangle = \langle FrameType, Frame_1^I\rangle, \tag{5}$$

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

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

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

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

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

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

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

3 CLP-based Product/Process Configuration

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

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

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

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

In it, `Vars` is the list of variables in $\mathcal{V}$, `DOMS` is the conjunction of domain constraints for domains in $\mathcal{D}$, and `CONSTRS` is the conjunction of constraints in $\mathcal{C}$. Given a program with the above-described characteristics, a finite domain solver can be used to reduce the domains associated with variables, preserving satisfiability, or to detect the inconsistency of the encoded CSP (due to user's assignments that violate the set of constraints or to inconsistencies of the original product model). Moreover, it can be used to determine that further node instances are needed, or that there are too many nodes in the instance tree.

4 Comparison with Related Work

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

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

Product configuration systems based on binary decision diagrams (BDDs), e.g., Configit Product Modeler [Configit A/S, 2009], trade the complexity of the construction of the BDD, for the simplicity and efficiency of the configuration process. Despite their various attracting features, BDD-based systems suffer from some significant limitations. First, they basically support flat models only, even though some work has been done on the introduction of modules (see, e.g., [van der Meer and Andersen, 2004]). Second, they find it difficult to cope with global constraints. In [Nørgaard *et al.*, 2009] the authors combine BDD and CSP to tackle `alldifferent` constraints. However, they consider flat models only. We are not aware of any BDD system that deals with global constraints in a general and satisfactory way.

Unlike ASP-based and BDD-based systems, CSP-based product configuration systems are (usually) capable of dealing with non-flat models and global constraints. Unfortunately, the modeling expressiveness of CSP-based systems has a cost, i.e., backtrack-free configuration algorithms for CSP-based systems are often inefficient, while non backtrack-free ones need to explicitly deal with dead ends. Moreover, most CSP-based systems do not offer high-level modeling languages (product models must be specified at the CSP level). Some well-known CSP-based configuration systems, such as ILOG Configurator [Junker, 2003], which features various interesting modeling facilities, and Lava [Fleischanderl *et al.*, 1998], which is based on Generative-CSP, seem to be no longer supported. A recent CSP-based configuration system is Morphos Configuration Engine (MCE) [Campagna *et al.*, 2010]. As other CSP-based systems, it makes it possible to define non-flat models. Its configuration algorithm is not backtrack-free, but it exploits back-jumping capabilities, to cope with dead ends, and branch-and-prune capabilities, to improve domain reduction. From the point of view of process modeling, PRODPROC can be viewed as an extension of MCE modeling language. In particular, it extends MCE modeling language with the following features: (1) *cardinality variables*, i.e., *has-part/is-part-of* relations can have non-fixed cardinalities; (2) *product model graph*, i.e., nodes and relations can define a graph, not only a tree; (3) *cardinality constraints* and *cardinality model constraints*, i.e., constraints can involve cardinalities of relations; (4) $MetaPath$s, i.e., a mechanism to refer to particular node instance variables in constraints.

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

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

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

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

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

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

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

5 Conclusions

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Modelling Configuration Knowledge in Heterogeneous Product Families

Matthieu Quéva[1] and **Tomi Männistö**[2] and **Laurent Ricci**[3] and **Christian W. Probst**[1]
[1]DTU Informatics, {mq,probst}@imm.dtu.dk
[2]Aalto University, tomi.mannisto@aalto.fi
[3]Microsoft Development Center Copenhagen, lricci@microsoft.com

Abstract

Product configuration systems play an important role in the development of Mass Customisation. The configuration of complex product families may nowadays involve multiple design disciplines, e.g. hardware, software and services. In this paper, we present a conceptual approach for modelling the variability in such heterogeneous product families. Our approach is based on a framework that aims to cater for the different stakeholders involved in the modelling and management of the product family. The modelling approach is centred around the concepts of views, types and constraints and is illustrated by a motivation example. Furthermore, as a proof of concept, a prototype has been implemented for configuring a non-trivial heterogeneous product family.

1 Introduction

In many companies, there has been an increasing need to reduce the costs while offering highly customised products. Indeed, today's customers demand products with lower prices, higher quality and faster delivery, but they also want products customised to match their unique needs. Product configuration systems (or configurators) have allowed the manufacturers to adapt their business model to Mass Customisation [Pine, 1993] and propose products with hundreds of product features and options for a competitive price.

Model-based configuration is based on a strict separation between the product knowledge (i.e. the data representing the characteristics of the products) and the problem solving knowledge (i.e. the mechanisms used to ensure the consistency of the customised product). As the solving process is independent from the product knowledge, this separation provides a good robustness, compositionality and reusability, making model-based systems the prime choice for configuring large and more complex models [Sabin and Weigel, 1998].

Most of the research on product knowledge modelling has concentrated on manufactured product families [Hvam *et al.*, 2008]. Moreover, configuration techniques have recently been applied to other types of products, such as software variability [Asikainen *et al.*, 2007] or configurable services [Heiskala *et al.*, 2005]. However, many products nowadays are heterogeneous, i.e. different design disciplines are taken into account within the *same* product family. Modelling such products raises two main issues. One must first consider how to structure the different kind of knowledge that needs to be modelled. A second issue concerns the evolution of the product family. The set of features provided by a product family varies according to where and when it is distributed.

In this paper, we present a framework for modelling heterogeneous product families, based on *modelling views*. This framework synthesizes, unifies and extends different approaches to modelling configuration in the different design disciplines, e.g. physical products, software or services. The different views used in the approach are described using UML metamodels, together with different types of constraints that govern the dependencies both within and between views.

Section 2 and Section 3 introduce the necessary background and the research problem behind our approach. Section 4, 5 and Section 6 present concepts and constraints involved in our framework. Section 7 provides a brief overview of the proof of concept for our work, while Section 8 discusses our results and related work. Finally, Section 9 concludes the paper.

2 Background and Previous Work

This section provides a brief overview of different research areas on which this work is based.

2.1 Product Configuration

Product configuration modelling is widely based on concepts such as *components*, *ports*, *resources* and *functions* [Soininen *et al.*, 1998]. A *configurable product* is composed by components that are connected together via ports to form a hierarchical partonomy structure. Specialisation relations also permits to create a taxonomy structure in the model. Resources are balanced entities that can be produced or consumed by components, while functions can be used to define the product from the point of view of what functionalities it provides. The model also contains *constraints* that limit the number of possible variants, e.g. by restricting the combinations of values allowed for the different attributes of the product. The traditional method for modelling products is the type-instance approach: the model defines a number of *types*, that are then instantiated as *individuals* during the configuration process to store the final data.

Several high-level modelling languages tailored for product configuration have been proposed, including PCML [Tiihonen *et al.*, 2002]. Other languages such as UML have also been studied for modelling product configuration [Felfernig *et al.*, 2002; Hvam *et al.*, 2008]. Finally, product configuration has also been successful in industry [Haag, 1998].

2.2 Software Product Lines

Software product lines (SPL), also known as software product families, is a set of software systems sharing a common set of features that "satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements and Northrop, 2001].

Some approaches consider software product lines from an architectural point of view. *Architecture description languages (ADLs)* have been proposed to describe the SPLs in terms of their structure, including their components, interfaces, or communication protocols; but few can handle variability in SPLs. A few exceptions exist: Koalish [Asikainen *et al.*, 2003] for example extends Koala, an ADL based on components and interfaces, by adding variability elements such as attribute *domains* and *constraints*.

A more common method to model SPL is using features. Feature modelling approaches are based on the concept of *features*, that usually represent the visible characteristics of the system, from an end-user point of view. Well-known feature modelling methods, such as FODA (Feature Oriented Domain Analysis) [Kang *et al.*, 1990] or FORM (Feature Oriented Reuse Method) [Kang *et al.*, 1998] use a *feature model*, which represents a feature tree using different relations between features and subfeatures, including *mandatory*, *optional* or *alternative* relations. Feature models have been extended to support shared subfeatures, feature attributes and cardinalities or feature groups [Czarnecki *et al.*, 2005b].

Finally, Asikainen et al. [2007] recently proposed Kumbang. Kumbang combines advanced feature modelling concepts with the approach from Koalish, and adds support for advanced constraint relations compared to traditional feature modelling approaches.

2.3 Service Configuration

Configurable services represent services that can be customised from a set of pre-defined options, in order to fit the needs of individual customers. Research on configurable services and how to model them is a relatively recent topic: several authors have been discussing the configuration of different types of services, e.g. IT services [Böhmann *et al.*, 2003].

Other researchers [Akkermans *et al.*, 2004; Heiskala *et al.*, 2005] propose more detailed conceptualisations for modelling services. The most similar to our work, Heiskala et al. [2005], presents a conceptual model following a type/instance approach using four viewpoints, called *worlds*: the *needs world*, representing the customer's needs; the *service solutions world*, for the service's specifications; the *process world*, related to the service delivery; and finally the *object-of-services world*, that is used to describe the service recipient and the environment in which the service will be supplied.

3 Research Problem

In this section, we define the Research Problem motivating our work, and describe the example that illustrates our approach.

The Research Problem considered in this paper is: *how to uniformly support the configuration and management of heterogeneous product families?* What we refer to as a heterogeneous product family is a family of products integrating separate design disciplines interacting with each others. In the rest of the paper, we refer to these design disciplines as the *dimensions* of such a product family.

Modelling variability in heterogeneous product families yields various issues, due to the diversity of the product knowledge necessary to address these different dimensions. Indeed, the model of a heterogeneous product family can be very complex, and involves several types of users with different skills and objectives, making the need for uniformity of prime importance. The different dimensions in such a product family are rarely independent, and it is primordial to take the interactions between dimensions into account.

With this, we specify the main research problem in more detail with the following Research Questions:

RQ1 *What are the needs of the users of the model to be supported?*

RQ2 *What modelling constructs support addressing the heterogeneity?*

RQ3 *How to integrate together the different dimensions of heterogeneity in the models?*

RQ4 *How to support the management of such product family over time and for different market situations?*

As a motivation example, we present in Figure 1 a simplified version of a product family consisting of netbooks, smartphones and tablet computers. The example represents a configurable family of products consisting of: a set of physical elements (a motherboard with hardware chips, a screen, ...); the configurable software running on the devices (applications, libraries, ...); and the services associated with the devices (subscriptions, synchronisation services, ...).

This running example illustrates how complex the modelling of a heterogeneous product family can be. Indeed, the engineers responsible for modelling the variability in the physical parts of the system often possess a knowledge different from the ones responsible for managing the software configuration model, or creating the service model. As can be seen in the previous section, although the modelling approaches often use the same basis (types, partonomy, etc.), the high-level concepts behind each type of modelling are different, and thus require different mindsets. One can then assume that the task of managing the variability of the hardware, software and service parts for large products is delegated to separate groups of knowledge engineers.

Configuring such a product family can be quite complex, due to the amount of technical details represented in each different aspects of the products. Those details are often not very accessible to salespersons and end-customers, who prefer viewing the features (or functions) of the product families, as described in [Soininen *et al.*, 1998]. Defining the feature set of the product family may be enough in some cases.

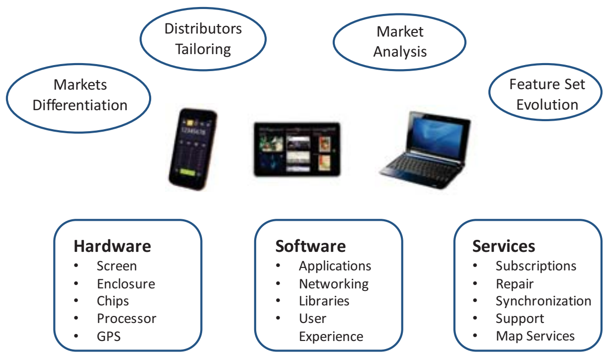


Figure 1: Running example. The product family represents mobile devices and is configured according to three dimensions, hardware, software and services, and need to be adapted to different scenarios, e.g. tailored to distributors and markets.

However, we identified several scenarios that illustrate different situations where this feature set may need refinement:

- *Market differentiation*: The company selling the products proposes different feature sets for different markets. In our example, different markets, e.g. Europe and United States, means different data signals to be handled by the phones, as well as different regulations. The possible combinations of features may just be restricted on those different markets.
- *Feature set evolution*: The product family's feature set is evolving with time. Devices may not arrive fully featured on the market, due to time constraints or strategic decisions. A refined feature set may be needed for a specific time, with additional constraints that may disappear (or be modified) in future evolution of the product family.
- *Distributors tailoring*: The producing company is distributing the products to different intermediary vendors. Products as our example may not be distributed directly by the manufacturer. This producer may propose a feature set to vendors that can adapt it in order to forbid specific combinations, or to create a more simple feature set for the end-customer. For example, the example products may be sold by distributors by letting the customer choose between different feature packages, limiting the choices in configuration.
- *Market analysis*: The final customers can also be considered first (instead of the product family). A market study identifies the different needs of the final customers (or needs that the company wants to introduce in the market) and build different feature sets to satisfy these needs, aiming at creating a product family to fit those. On the contrary to the first three scenarios, this scenario considers the market needs as the basis for designing the product family.

These scenarios provide a more concrete characterisation of how the functionalities of the product family may need to evolve depending on its use and distribution, as introduced in Research Question 4.

4 Modelling Framework

Our approach is based on the concept of *modelling views*. Those views are used to model different aspects of the product family, according to the different roles of the modellers. The main assumption is that each product family considered consists of different dimensions, and that all those parts need (and benefit from) configuration. Models are created and maintained by knowledge engineers from various informations given by domain experts. However, heterogeneous product families with multiple dimensions may require different kind of domain experts with different roles and sets of skills, according to the degree of technicality or the dimension considered.

In this section, we thus define three different types of views: the feature views, the structure views and the realisation views, depending on their intended audience and how they contribute to the model of the product family through different levels of abstraction. The views are characterised by a set of concepts with a specific organisation. Most of the concepts presented here are not new in themselves, but how they interact between each others within and between views is of importance.

4.1 Feature Views

Feature views provide a view of a product family from a high level of abstraction. These views are targeted at sales persons or end-customers that need to have an understanding of what the product individuals can do, instead of how they can do it. In our conceptual approach, feature views are not separated according to the different dimensions of the product family. The relations between the concepts described in feature views are related to the product individuals as a whole, and as such should not be dimension-specific. Product individuals can indeed be characterised by the features (or functions) they provide, independently from the way they are structured.

A feature view is composed of *feature types*, organise in partonomy (*subfeatures*) and taxonomy (*subtypes*) structures, as shown in the UML metamodel in Figure 2(a). Variability is defined in each feature type using *attributes* that can take different values. A feature subtype inherits all the properties of its supertype, i.e. its attributes, subfeatures, and constraints. Two types of constraints can be added to each type. *Compatibility constraints* model dependencies between the feature view, i.e. it specifies conditions that must hold in a valid configuration. *Implementation constraints* model the dependencies between different types of views, and will be detailed in Section 5.

Example: Consider our motivation scenario (Figure 3). Feature types such as *Input* or *Localisation* can be used to define the input type (touch input, keyboard features) or if GPS localisation should be available on the device.

4.2 Structure Views

Feature views are implemented by *structure views*, which define the different design components that realise the described features of the product family, and the relations between them. Structure-based approaches for configuration are widely used [Soininen *et al.*, 1998], as the compositional structure of the product families is often used to represent the product data knowledge. The structure views communicate the aspects of the architecture of interest to those involved in designing the

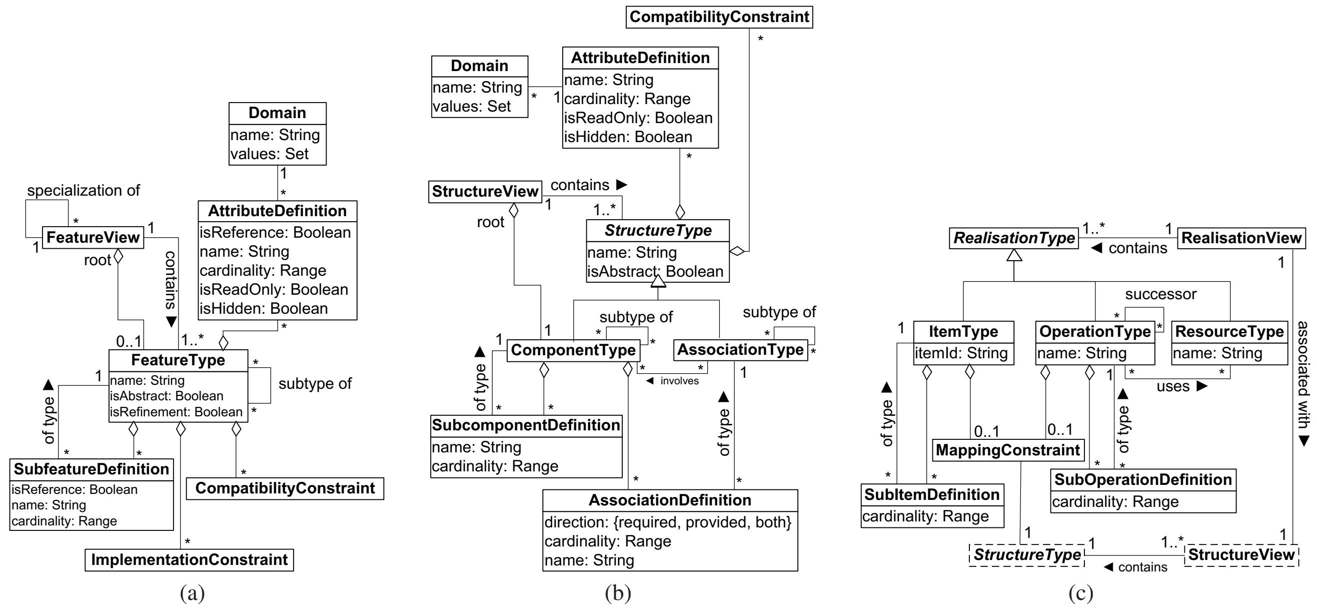


Figure 2: UML metamodels for: (a) Feature views (b) Structure views (c) Realisation views

system. They provide more concrete models of a product family, as it represents the specifications of the components of the system. Structure views are thus mainly aimed at design or maintenance and are for example targeted at product design engineers, software architects or service contractors.

A structure view is composed of *structure types*, that can be either *component types* or *association types*. As for feature views, structure views are organised in partonomy and taxonomy structures (Figure 2(b)). Types can have attributes and compatibility constraints as well.

The different concepts used in structure views have a specific meaning according to which dimension each view refers to. For example, a *physical structure view* represents the physical structure of the product family. Component types are entities whose individuals are physical components involved in the physical design, while association types are used to model non-directional physical links between two components. A *software structure view* describes the architecture of the software system involved in the product family. Instances of component types represent software components, and association can be defined to model interfaces, whether they provide software functions or require some. Also, a *service structure view* describes the specifications of the service to be delivered. Component types are service element types, and describes contractual agreements of what to be delivered, similar to what is modelled in the service solutions world of Heiskala et al. [2005].

Example: In our motivation scenario (Figure 3), the physical structure view contains a *Screen* and a *TouchScreen* component types with a *size* attribute, while the software view handles the *User Experience (UX)* framework and software interfaces to the *Middleware* libraries. The service structure view declares *RepairCoverage* or *PhoneSubscription* as types.

4.3 Realisation Views

Realisation views offer a detailed technical view of how the product individuals are realised. Compared to structure views, whose purpose is to represent the design of a specific dimension of the product family, realisation views are aimed at describing the elements necessary for the concrete realisation of the system for that dimension. They are thus targeted at highly specialised engineers, e.g. product engineers, software developers or service deliverers, and represent the lowest abstraction level in our conceptual modelling framework. Each realisation view is associated with a dimension, which defines its proper meaning: physical products use this view to represent manufacturing data, while software involve the solution deployment, and services the delivery process.

The building blocks of a realisation view are *realisation types*. There are three possible realisation types: *item types*, *operation types* and *resource types*. Item types represent the production components used to realise the products. It can be a BOM item for manufactured parts, a software package when dealing with software, or an object to be produced when delivering a service (e.g. a contract or a bill). Operation types are used to specify a set of operations needed during the production of individuals (e.g. manufacturing operations, software deployment, service processes). Resource types may describe a machine, an operator, an information or anything that may be necessary to complete the operations.

Contrary to structure views, realisation views are not starting with a single root type. Instead, each realisation view is associated with a structure view (from the same dimension), and each item or operation type may be associated to a relevant structure type via a *mapping constraint*. Types mapped to a structure type defines their own tree of *subitems*, providing a more detailed breakdown of the production components.

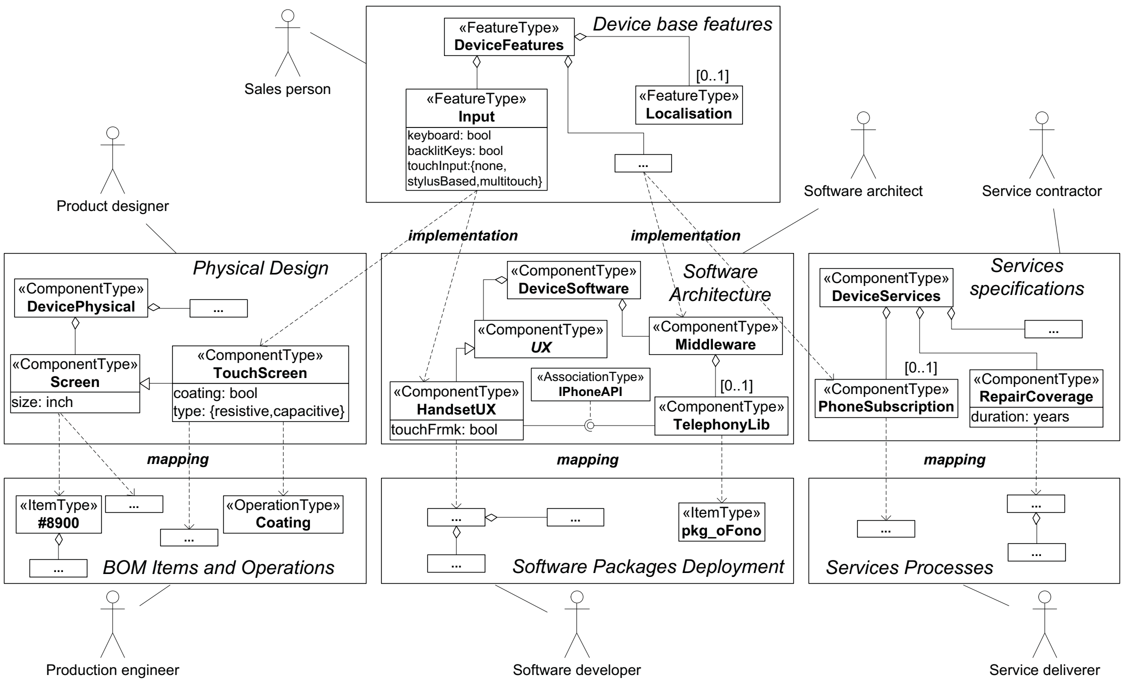


Figure 3: Overview of the motivation example model and the different modelling views, depending on the three dimensions (physical, software and services). Partonomy relations are shown using UML Aggregations, taxonomy relations using UML Generalizations. For the sake of brevity, only a subset of the types and the base feature view in the hierarchy are shown.

Example: The *Screen* component type in the physical structure view of our motivation example can be mapped to different manufacturing items. A specific mapping can be made if *TouchScreen* is chosen instead, or when a specific configuration is made (e.g. changing the value of the *size* attribute).

5 Dependencies and Constraints

Constraints can be used to specify dependencies within or between views when other modelling mechanisms are not sufficient to capture them. Constraints are written in a constraint language, and involve types attributes and predicates using pre-defined functions, such as *Count(...)* that returns the actual cardinality in a partonomy relation. The full description of the constraint language is out of the scope of this paper.

5.1 Compatibility Constraints

A compatibility constraint is specific to a particular view, and can only involve properties of this view. The evaluation of a constraint occurs during configuration, when types are instantiated to individuals. Each instance of the context type in which the constraint is declared must satisfy it.

Example: The following constraint, declared in the *Input* component type, specifies that the physical keyboard feature must be selected if one wants a backlit keyboard:

$$backlitKeys \Rightarrow keyboard$$

Constraints may also contain references to properties that are not always present in the product individual being configured, e.g. if a constraint accesses a subpart whose cardinality is not fixed, or an attribute from a subtype that may not be chosen (the property is said *inactive*). Each compatibility constraint containing at least one inactive term is evaluated to *true* during configuration.

5.2 Implementation Constraints

Implementation constraints are essential to our framework, as they model the interaction between the base feature view and the structure views (and in the feature views hierarchy, see Section 6). They are composed of a left-hand side expression *L* and a right-hand side expression *R*, related by an implication or an equivalence operator. The expression *L* represents the features to be implemented by the constraint, in a similar way as in the compatibility constraints. On the other hand, the expression *R* represents what is needed in the stucture view(s) to implement the features specified by *L*.

Example: Consider the following constraint in the *Input* feature type from the base feature view:

$$\begin{aligned}&touchInput = \text{“}multitouch\text{''} \Leftrightarrow\\ &\quad(Physical :: TouchScreen.type = \text{“}capacitive\text{''}\\ &\quad\wedge Software :: HandsetUX.touchFrmk = true)\end{aligned}$$

This implementation constraint specifies that a device has a multitouch input if *there exists* a capacitive touchscreen and a touch framework is implemented in the software. Existential quantifiers are implicitly used in the semantics of the expression *R*, as the feature may exist if there is at least one combination of structural elements implementing it. Universal quantifiers can also be explicitly used in some specific cases.

5.3 Mapping Constraints

Mapping constraints are defined in realisation views to specify under which conditions a realisation type should be included in the configuration results. There exists indeed a mapping between structure types and item and operation types, and the latters should only be part of the final configuration if certain conditions are met. Mapping constraints are declared in item and operation types, and refers to attributes from the structural type defined as context.

Example: The following mapping constraint is declared in the Coating operation type and takes as context the TouchScreen component type from the physical structure view:

$$c_{map}(TouchScreen, Coating) : oleophobicCoating = true$$

A valid configuration thus ensures that the latter constraint is true for each instance of the *TouchScreen* type, i.e. an instance of the *Coating* operation type is present for each instance of *TouchScreen* where the attribute *oleophobicCoating* is *true*.

6 Feature View Hierarchy

To address the management and evolution of the product family (Research Question 4) and the scenarios discussed in the Section 3, several feature views can be defined and organised in a *feature view hierarchy*. A model defines a *base feature view*, which will contain all the features available for the modelled product family, and should be implemented by the structural views. This base feature view may then be specialised, as different versions or evolutions of the product family may require special restrictions to the set of available features (*Market differentiation* and *Feature set evolution* scenarios), or even more abstract feature views in order to be presented to final customers (*Distributors tailoring* and *Market analysis*).

The feature view hierarchy defines a specialisation tree, rooted by the base feature view. A feature view $\mathcal{F}'$ is the child of another feature view $\mathcal{F}$ if $\mathcal{F}'$ is a *specialisation* of $\mathcal{F}$. This specialisation is done through different concepts:

- **Implementation**: A feature view $\mathcal{F}'$ can declare new feature types and attributes, for example to define more abstract feature groups and properties. As for the base feature view, the types in $\mathcal{F}'$ must use *implementation constraints* to associate their properties to the feature view $\mathcal{F}$, parent of $\mathcal{F}'$.
- **Refinement and reference**: Apart from defining new feature types, feature views can refine feature types from their parent view. A *refined feature type* can transform the original type by: defining new attributes or subfeatures; refining *referenced* attribute or subfeature definitions by restricting its cardinality, its domain or visibility (attribute) or change it to one of its subtypes (subfeature); changing the type from concrete to abstract to force the use of its subtypes; or by adding compatibility constraints to constrain the model even more.

Figure 4 shows the mechanism of feature types refinement. Type $F1$ is refined: the attribute $a1$ in $F1$ is declared as hidden, and a new attribute $a5$ is declared. The $feat3$ subfeature cardinality is also refined to $[1..2]$. Finally, even though $F1'$ is not directly modified, the type $F4$ is also refined: the domain of $a4$ is reduced and a new subfeature is defined.

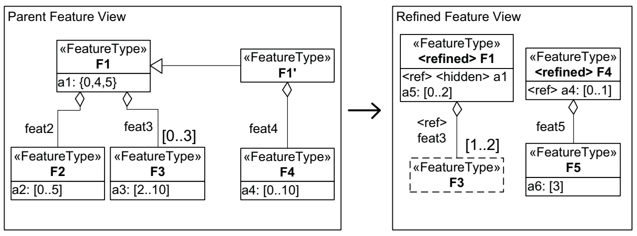


Figure 4: Feature types refinement. Refined types are characterised by the **<refined>** tag, while referenced definitions are tagged with **<ref>**. The type $F3$ in the Refined Feature View is shown with a dashed outline, as only the $feat3$ definition is part of the refined view, while the feature type is not and is just shown here for illustrative purpose.

7 Proof of Concept

A textual modelling language named ProCoLa has been defined to support our modelling framework. The language syntax is closely mapped to the different concepts described in this paper. A language service has been implemented in Visual Studio to support the ProCoLa language, providing an important number of features for tool support, such as syntax highlighting, syntax and semantic checks, automatic word completion among others. The language is supported by a C# compiler, and resembles that of an object-oriented programming language. A formalism of the framework and model analyses are currently being worked on in order to provide additional tool support, such as the ability to see model-wide dependencies of any change that may happen in a view, e.g. the deletion of an attribute or type.

The semantics behind our modelling approach (and ProCoLa) have been defined by implementing the translation of models to Dynamic CSPs (DCSPs) [Bartak and Surynek, 2005] and Conditional CSPs (CondCSPs) [Mittal and Falkenhainer, 1990] formalisms. DCSPs are used to handle the dynamic addition and removal of value assignments to attributes during interactive configuration, while CondCSPs are used to handle the notion of activity involved when dealing with dynamic cardinality or taxonomy structures for example, or the existential (or universal) quantifiers implied by implementation constraints. During the configuration process, an end-user first chooses which feature view he wants to use (if multiple feature views exist in the hierarchy), and then can enter his requirements through a user-interface by assigning values to attributes, or connecting associations. A single CSP model is usually used for all views, allowing a full propagation of the choices to the other views. However, the user may also consider configuring only a single view (using only compatibility constraints). More details can be found in [Quéva, 2011].

A larger mobile device product family based on the motivation example presented in this paper has been modelled using our conceptual framework and ProCoLa. The model is split into 13 views, including three realisation views and three structure views (one for each dimension), and a feature views hierarchy of 7 views. It contains around 250 types, 200 attributes and over 300 constraints. During the modelling of the product family, ProCoLa has provided a sufficient level of support to capture the different part of the family and their dependencies, in a reasonable amount of time. The translation into the CSP formalisms is very fast, while the consistency checks at runtime are done within a few seconds at most.

8 Discussion and Comparison with Related Work

The different views provides a modelling framework as a contribution to address the Research Questions (RQs) exposed in Section 3. The clear separation of concerns in the structural and realisation data for each dimension is motivated by RQ1 (*What are the needs of the users of the model to be supported?*) and previous work on modelling each dimension (Section 2). Each view is targeted at a different audience: the structural model of the software is handled by a software architect, while a production engineer may be more adequate to handle bill-of-materials and manufacturing operations. Moreover, we argue that structural and realisation views from each dimension should be considered independently from each others, and unified in the feature models they contribute to implement, defined in feature views. In Figure 3, the sales persons working on the device features model the types of input that the end-user may be interested in. How this feature is implemented is dependent on several structural elements from different parts of the system: the touch screen hardware and a touch framework component in the user experience software. Those two elements can however be chosen independently from each others, but will only provide the feature if they are both present in the final product.

The UML metamodels (Figure 2) provide a good basis in order to address the problem of modelling the different dimensions of an heterogeneous product family, as raised in RQ2 (*What modelling constructs support addressing the heterogeneity?*). Uniform modelling constructs and the different types of inter-views constraints defined in the framework also contribute to the issue posed in RQ3 (*How to integrate together the different dimensions of heterogeneity in the models?*): the implementation and mapping constraints permit to model the interdependencies between the views, allowing a tight integration of the different dimensions of the product family. Modelling these constraints requires communication between the different stakeholders. The sales person responsible for the touch input feature inquires the product designer in order to assess what hardware components are needed for the requested feature. On the other hand, product designers and production engineers need to confer on which items are available to realise the structural design of the hardware.

Our modelling approach also extends the concept of feature model to a feature view hierarchy, as a contribution to RQ4 (*How to support the management of such product family over time and for different market situations?*). The refinement of feature type's attributes can be used to model scenarios such as *Market differentiation* (by adding constraints for specific markets), *Feature set evolution* (by creating multiple feature views depending on the current capabilities of the product) or *Distributors tailoring* (by allowing them to create their own specialised views). In a *Market analysis* scenario, several specific feature views are created in order to match the product feature sets to introduce in the market. These views may then be joined into one base feature view, by gathering common elements or creating more abstract features that can be specialised to fit the original views, via refinement or implementation. The feature view hierarchy thus enables a unification of the product family management and evolution at the feature level, independently from the heterogeneity of the family, while each dimension may have its own separate mechanism for coping with this issue (e.g. product data management, ...).

Modelling concepts from our approach are based on previous work, mainly in product configuration [Soininen *et al.*, 1998]. The four worlds from Heiskala et al. [2005] can also be compared with the modelling views of our framework: the *needs world* concerns the customer's needs (in an abstract way), and is thus close to our feature views, which describes the abstract features that the customer may require; the *service solution world* denotes the set of elements used to establish the service's specifications, as the structure views; the *process world* describes how the service will be delivered, or realised, as in our realisation views. Note that there is nothing in our conceptual approach that is similar to the *object-of-services world* from [Heiskala *et al.*, 2005], which specifies the service recipients or the environment relevant to those recipient. From a modelling point of view, all these worlds are based on the same metamodel, using different types and attributes, as well as taxonomy and partonomy structures, as in our approach. However, dependencies between types of different worlds are simply modelled using classical constraints, while we use implementation and mapping constraints. Also, our framework is centered on the configured product, and thus the services described in the services dimension are seen from the configured product's point of view, while the external environment is not considered. Another type of view may thus be necessary in our framework to define externally controlled elements (such as, in the running example, access to company specific services or credentials, data transfer from an old device, etc.).

Feature modelling approaches such as cardinality-based models [Czarnecki *et al.*, 2005a] also have similarities to our feature views, although the richness of constraints and the partonomy/taxonomy structures used in our models is somewhat more complex than with the classic feature-oriented relations. Multi-view models in feature modelling have also been studied. Czarnecki et al. [2006] sketches a model where different levels of customisation are modelled (including feature and design view). Reiser and Weber [2006] and work from Zaid et al. [2010] propose feature models with different perspectives, although they are all centered on software variability and feature modelling techniques only, and the lack of specialisation hierarchy may make the task of implementing the unification with different structured views difficult.

Kumbang [Asikainen *et al.*, 2007] is the closest to our work on the software variability side, including their type-instance approach. We consider our work to be an extension of Kumbang, as we use implementation constraints to unify structure views from the different dimensions (including manufactured products and services), as well as we model realisation data. Thus the main contribution of our work is to provide conceptual and practical mechanisms to bring the different dimensions together and unify them under feature models.

9 Conclusion

In this paper, we present an approach to help with the issue of modelling a product family consisting of different design disciplines (or dimensions). The presented framework has been motivated by a four research questions and illustrated by several scenarios.

Our framework is based on modelling views and synthesizes the concepts from different approaches from product configuration, software variability and service configuration, and unify them around feature views using implementation mechanisms. We also describe a feature view hierarchy and refinement mechanisms to cope with the evolution and adaptation of the product family, which remains an important issue [Krebs, 2008].

The approach has been motivated by the use case of a mobile devices product family, and has been implemented in a language prototype as a proof of concept. However, we have yet to perform an in-depth case study with industrial data in order to test the feasibility of implementing a real-life product family with our framework, as well as completing the formalism and tool support for the language, which is planned as future work.

References

[Akkermans *et al.*, 2004] H. Akkermans, Z. Baida, J. Gordijn, N. Peña, A. Altuna, and I. Laresgoiti. Value webs: Using ontologies to bundle real-world services. *IEEE Intelligent Systems*, 19(4):57–66, 2004.

[Asikainen *et al.*, 2003] Timo Asikainen, Timo Soininen, and Tomi Mannisto. A koala-based approach for modelling and deploying configurable software product families. *Proc. International Workshop on Product Family Engineering*, pages 225–249, Jan 2003.

[Asikainen *et al.*, 2007] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.

[Bartak and Surynek, 2005] R. Bartak and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. *Proc. FLAIRS'05*, pages 161–166, 2005.

[Böhmann *et al.*, 2003] T. Böhmann, M. Junginger, and H. Kremar. Modular service architectures: a concept and method for engineering it services. In *Proc. International Conference on System Sciences*, 2003.

[Clements and Northrop, 2001] P. Clements and L. Northrop. *Software Product Lines — Practices and Patterns*. Addison-Wesley, 2001.

[Czarnecki *et al.*, 2005a] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practices*, 10(1):7–29, 2005.

[Czarnecki *et al.*, 2005b] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practices*, 10(2):143–169, 2005.

[Czarnecki *et al.*, 2006] K. Czarnecki, M. Antkiewicz, Chang Hwan, and Peter Kim. Multi-level customization in application engineering. *Communications of the ACM - Software product line*, 49(12):61–65, 2006.

[Felfernig *et al.*, 2002] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Configuration knowledge representation using uml/ocl. *Lecture notes in computer science*, pages 49–62, Jan 2002.

[Haag, 1998] A. Haag. Sales configuration in business processes. *IEEE Intelligent Systems*, 13(4):78–85, 1998.

[Heiskala *et al.*, 2005] M. Heiskala, J. Tiihonen, and T. Soininen. A conceptual model for configurable services. In *IJCAI Workshop on Configuration*, Scotland, 2005.

[Hvam *et al.*, 2008] L. Hvam, N. H. Mortensen, and J. Riis. *Product customization*, volume XII. Springer, 2008.

[Kang *et al.*, 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S .A. Peterson. Feature-oriented domain analysis (foda) — feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Kang *et al.*, 1998] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[Krebs, 2008] Thorsten Krebs. Knowledge management for evolving products. In Max Bramer, Frans Coenen, and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXIV*, pages 307–320. 2008.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. 1990.

[Pine, 1993] B. J. Pine. *Mass Customization - The New Frontier in Business Competition*. Harvard Business School Press, 1993.

[Quéva, 2011] Matthieu Quéva. *A Framework for Constraint-Programming based Configuration*. PhD thesis, DTU Informatics, 2011. Submitted in May 2011.

[Reiser and Weber, 2006] Mark-Oliver Reiser and Matthias Weber. Managing highly-complex product families with multi-level feature trees. In *Proc. of the 14th IEEE International Requirements Engineering Conference*, 2006.

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

[Soininen *et al.*, 1998] T Soininen, J Tiihonen, T Männistö, and R Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[Tiihonen *et al.*, 2002] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen. Empirical testing of a weight constraint rule based configurator. In *Workshop on Configuration, ECAI*, pages 17–22, 2002.

[Zaid *et al.*, 2010] L.A. Zaid, F. Kleinermann, and O. De Troyer. Feature assembly: a new feature modeling technique. In J. Parsons et al., editor, *ER 2010, LNCS*, volume 6412, pages 233–246, 2010.

(Re)configuration using Answer Set Programming[*]

Gerhard Friedrich and
Anna Ryabokon
Universitaet Klagenfurt, Austria
firstname.lastname@aau.at

Andreas A. Falkner, Alois Haselböck,
Gottfried Schenner and **Herwig Schreiner**
Siemens AG Österreich, Vienna, Austria
firstname.{middleinitial.}lastname@siemens.com

Abstract

Reconfiguration is an important activity for companies selling configurable products or services which have a long life time. However, identification of a set of required changes in a legacy configuration is a hard problem, since even small changes in the requirements might imply significant modifications. In this paper we show a solution based on answer set programming, which is a logic-based knowledge representation formalism well suited for a compact description of (re)configuration problems. Its applicability is demonstrated on simple abstractions of several real-world scenarios. The evaluation of our solution on a set of benchmark instances derived from commercial (re)configuration problems shows practical applicability.

1 Introduction

Reconfiguration is an important task in the after-sale lifecycle of configurable products and services, because requirements for these products and services are changing in parallel with the customers' business [6; 2]. In order to keep a product or a service up-to-date a re-engineering organization has to decide which modifications should be introduced to an existing configuration such that the new requirements are satisfied but change costs are minimized.

Following the knowledge based configuration approach, we formulate reconfiguration problem instances as extensions of declaratively defined configuration problem instances where configurations are represented by facts and requirements are expressed by logical descriptions. These requirements may be partitioned into customer requirements and system specific configuration requirements. A configuration is simply defined as a subset of a logical model of the requirements. Informally, a reconfiguration problem instance is generated by an adaption of the requirements resulting in a new set of requirements and therefore a new instance of a configuration problem is formulated. Subsequently, given legacy configurations have to be adapted to configurations for the new requirements. In our approach, the knowledge base comprises two parts, the description of the new configuration problem instance and transformation knowledge regarding reuse and deletion of parts of a legacy configuration. The first part is a usual instance of a configuration problem where all valid configurations are specified by the set of adapted requirements. The second part describes a mapping from the pieces of the legacy configuration to the ontology of the new configuration problem instance. Technically speaking this is a mapping from facts describing the legacy configuration to facts in the ontology of the new configuration problem instance. For generating a reconfiguration the problem solver has to decide which parts of the legacy configuration are either reused or deleted and which new parts have to be created.

We introduce general definitions for (re)configuration problems employing Herbrand-models of logical descriptions. Based on these definitions it is simple to see that configuration and reconfiguration problems fall into the same complexity classes. Because of the remarkable advances of answer set programming (ASP) [8; 5] we base our implementation on this reasoning framework. ASP was first applied to configuration problems by [9]. In particular, we provide modeling patterns for configuration and reconfiguration which allow the generation of optimized reconfigurations exploiting standard ASP solvers. Finally, our evaluation shows that the proposed method solves reconfiguration problem instances which are practically interesting for industrial applications.

In Section 2 we present an introductory example of a configuration problem and some reconfiguration scenarios. Then, configuration problems are defined in Section 3. In Section 4 a review of the basic concepts of ASP is given followed by an exemplification of modeling in Section 5. Section 6 provides the definition of reconfiguration problems. Subsequently, modeling patters and an example of their application are provided in Section 7. Finally, we show the results of an evaluation in Section 8 and conclude in Section 9.

2 Example

Let us exemplify different configuration and reconfiguration scenarios on a problem which is a simple abstraction of several configuration problems occurring in practice, i.e. entities may be contained in other entities but some restrictions must be fulfilled. We employ the ontology comprising the concepts *person*, *thing*, *cabinet*, and *room* where persons are related to things, things are related to cabinets, cabinets are related to

[*]This work has been developed within the scope of the project RECONCILE (reconciling legacy instances with changed ontologies) and was funded by FFG FIT-IT (grant number 825071).

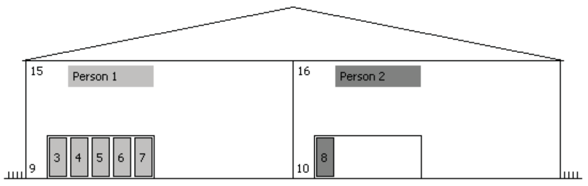


Figure 1: Solution of the sample house configuration problem. The house configuration includes rooms 15 and 16, two cabinets 9 and 10, and six things numbered from 3 to 8

rooms, and rooms are related to persons. These relations are modeled either by roles, associations, or predicate symbols depending on the modeling language (e.g. description logic, UML, or predicate logic).

As input to the configuration problem an ownership relation between persons and things is provided. We call this input a customer requirement since it reflects the individual needs of a customer using a configuration system whereas configuration requirements specify the properties of the system to be configured. Each person can own any number of things but each thing belongs to only one person. The problem is to place these things into cabinets and the cabinets into rooms of a house such that the following configuration requirements are fulfilled:

- each thing must be stored in a cabinet;
- a cabinet can contain at most 5 things;
- every cabinet must be placed in a room;
- a room can contain at most 4 cabinets;
- a person can own any number of rooms;
- each room belongs to a person;
- and a room may only contain cabinets storing things of the owner of the room.

In order to keep the example simple we only consider configuration of one house and represent all individuals using unique integer identifiers.

Informally, a configuration is every instantiation of the relations which satisfies all requirements.

Let a sample house problem instance include two persons such that the first person owns five things numbered 3 to 7 and the second person owns one thing 8. A solution for this house configuration problem instance is shown in Figure 1.

Reconfiguration is necessary, whenever the customer requirements or configuration requirements are changed. For instance, it becomes necessary to differentiate between long and short things with the following new requirements:

- a cabinet is either small or high;
- a long thing can only be put into a high cabinet;
- a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room;
- all legacy cabinets are small.

The customer requirements, in this case, define for each thing if it is long or short. For instance, the customer provides information that the things 3 and 8 are long; all others are short.

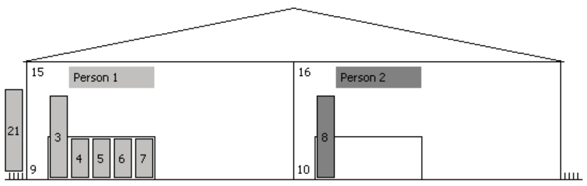


Figure 2: Reconfiguration initial state

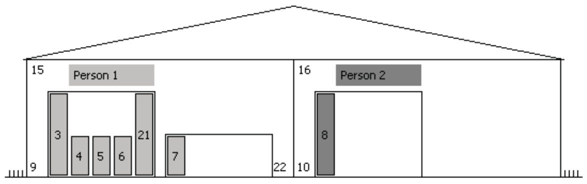


Figure 3: Reconfiguration solution 1

Moreover, the first person gets an additional long thing 21. The changes to the legacy configuration are summarized in Figure 2 showing an inconsistent configuration, where thing 21 is not placed in any of the cabinets, and cabinets 9 and 10 are too small for things 3 and 8.

To obtain a solution which is shown in Figure 3 the reconfiguration process changes the size of cabinets 9 and 10 to high and puts the new thing 21 into cabinet 9. A new small cabinet 22 is created for thing 7.

In our reconfiguration process every modification to the existing configuration, i.e. reusing/deleting/creating individuals and their relations, is associated with some cost. Therefore the reconfiguration problem is to find a consistent configuration by removing the inconsistencies and minimizing the costs involved. Different solutions will be found depending on the given modification costs. If, for example, the costs for adding a new high cabinet are less than the cost for changing an existing small cabinet into a high cabinet, then the previous solution should be rejected as its costs are too high. One of the solutions with less reconfiguration costs (see Figure 4) includes two new cabinets 22 and 23, because this is cheaper than converting the existing small cabinets into high cabinets. Also it contains the empty cabinet 10 because it's cheaper to keep the cabinet than to delete it. Note, this behavior can be controlled by the domain specific costs.

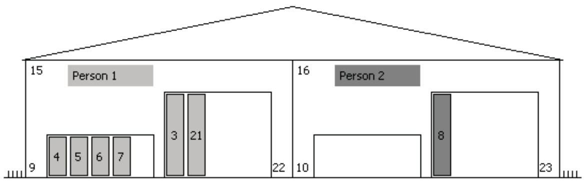


Figure 4: Reconfiguration solution 2

3 Configuration problems

We employ a definition of configuration problems based on logical descriptions [9; 3]. The basic idea is that every finite Herbrand-model contains the description of exactly one configuration.

The description of a configuration is defined by relations expressed by a set of predicates $\mathbf{P_S}$. This set of predicates is called the *solution schema*. For our example the solution schema consists of the four unary predicates `thing/1`, `person/1`, `cabinet/1` and `room/1` representing the individuals and the four binary predicates, namely `personTOthing/2`, `personTOroom/2`, `roomTOcabinet/2` and `cabinetTOthing/2` representing the relations. An instantiation of this solution schema corresponds to a configuration. A fragment of this instantiation is presented below.

```
{person(1), thing(3), room(15), cabinet(9),
cabinetTOthing(9,3), personTOthing(1,3),
roomTOcabinet(15,9), personTOroom(1,15),...}
```

Note, this description of a configuration generalizes the component/port models or variable/value based descriptions of a configuration.

We assume that every predicate symbol is unique in a logical theory and has a unique arity. The set of Herbrand-models is specified by a set of logical sentences $\mathbf{REQ}$, which usually comprises the individual *customer requirements* and the *configuration requirements*. Configuration requirements reflect the set of all allowed configurations for an artifact, whereas customer requirements may comprise facts and logical sentences specifying the individual needs of customers. The same configuration requirements are a basis for different sets of customer requirements. E.g. the component library of a technical system is stable for some time.

Definition 1 (Instances of configuration problems) *A configuration problem instance $\langle \mathbf{REQ}, \mathbf{P_S} \rangle$ is defined by a set of logical sentences $\mathbf{REQ}$ representing requirements and $\mathbf{P_S}$ a set of predicate symbols representing the solution schema. For optimization purposes an objective function $f(\mathbf{S}) \mapsto \mathbb{N}$ maps any set of atoms $\mathbf{S}$ to positive integers where $\mathbf{S}$ contains only atoms whose predicate symbols are in $\mathbf{P_S}$.*

Let $\mathcal{HM}(\mathbf{L})$ denote the set of Herbrand-models of a set of logical sentences $\mathbf{L}$ for a given semantics.

Definition 2 (Configuration) $\mathbf{S}$ *is a configuration for a configuration problem instance* CPI $= \langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathcal{HM}(\mathbf{REQ})$ *and* $\mathbf{S}$ *is the set of* all *the elements of* $\mathbf{M}$ *whose predicate symbols are in* $\mathbf{P_S}$ *and* $\mathbf{S}$ *is finite, i.e.* $\mathbf{S} = \{\mathrm{p}(\bar{\mathrm{t}}) | \mathrm{p} \in \mathbf{P_S} \text{ and } \mathrm{p}(\bar{\mathrm{t}}) \in \mathbf{M})\}$. *By* $\mathrm{p}(\bar{\mathrm{t}})$ *we denote a ground instance of* p *with a term vector* $\bar{\mathrm{t}}$.

$\mathbf{S}$ *is an optimal configuration for* CPI *iff* $\mathbf{S}$ *is a configuration for* CPI *and there is no configuration* $\mathbf{S}'$ *of* CPI *s.t.* $f(\mathbf{S}') < f(\mathbf{S})$.

Definition 3 (Configuration problems) *Let the instances of configuration problems be defined by $\langle \mathbf{REQ}, \mathbf{P_S} \rangle$ and objective functions $f(\cdot)$.*

Decision problem: *Given a set of atoms* $\mathbf{S}$. *Decide if* $\mathbf{S}$ *is a configuration for a configuration problem instance.*

Generation (optimization) problem: *Generate a set of atoms* $\mathbf{S}$ *s.t.* $\mathbf{S}$ *is a configuration (an optimal configuration) for a configuration problem instance.*

The set of Herbrand-models depends on the semantics of the employed logic. In this paper, we apply answer set programming and a stable model semantics for knowledge representation and reasoning because this approach allows a concise and modular specification, assures decidability, and avoids the inclusion of unjustified atoms (e.g. unjustified components) in configurations [9].

4 Overview on answer set programming

ASP is based on a decidable fragment of first-order logic enhanced with default negation and aggregation. We give a brief summary of the employed ASP variant and language constructs as needed. A detailed discussion of ASP can be found in [5; 4].

We start our introduction with rules without variables and introduce logical variables afterwards. A rule has the structure $C_0 \leftarrow C_1, \ldots, C_n$. Elements $C_1, \ldots, C_n$ on the right-hand-side (the body of a rule) are either literals or *weight* constraints. A literal is either an atom or a default negated atom. Default negation is expressed by not. C_0 (head of the rule) is either an atom or a weight constraint. We do not consider default negation on the left-hand-side and in weight constraints. If all C_i are literals then such a rule is called a *normal* rule.

To be able to express the requirements of our example domain we introduce a simplified version of weight constraints and their special case – *cardinality* constraints [9; 8; 4]. Weight constraints are of the form $l \leq \{a_1 = w_1, \ldots, a_n = w_m\} \leq u$ where a_i are atoms, w_j are integers representing weights of corresponding atoms and l, u are integers specifying lower and upper bounds. Given a set of atoms $\mathbf{M}$ representing a Herbrand-interpretation, the interpretation of a weight constraints evaluates to *true* iff the sum of weights of literals $a_1, \ldots, a_n$ which are contained in $\mathbf{M}$ is between l and u. E.g. $0 \leq \{a = 1, b = 2\} \leq 2$ is satisfied by $\emptyset$, $\{a\}$ or $\{b\}$ but not by $\{a, b\}$. Missing lower or upper bounds express the fact that there are no limits. Cardinality constraints are of the form $l \leq \{a_1, \ldots, a_n\} \leq u$ where each weight is considered to be equal 1. As usually (negated) atoms in the body of the rule are *true* if they are (not) in $\mathbf{M}$.

The semantics of a set of rules is defined by a stable model semantics. We give a brief informal description of this semantics for the restricted version employed in this paper and refer the reader to [8] for an in-depth exposition. A set of ground atoms $\mathbf{M}$ is a stable model for a set of rules $\mathbf{RU}$ iff two properties are fullfiled: (1) $\mathbf{M}$ *satisfies* all rules in $\mathbf{RU}$ and (2) every atom in $\mathbf{M}$ is *justified* by a reduced rule set $\mathbf{RU^M}$. A rule is satisfied by a set of ground atoms $\mathbf{M}$ iff $\mathbf{M}$ satisfies C_0 or there exists a literal $C_1, \ldots, C_n$ which is not satisfied by $\mathbf{M}$. An empty body of a rule is always satisfied. A rule with empty head is satisfied iff one literal in the body is not satisfied. The precise semantics of justification is expressed by a reduction of the rule set $\mathbf{RU}$. Given $\mathbf{RU}$ and depending on the set of atoms $\mathbf{M}$, the reduct $\mathbf{RU^M}$ is generated as follows. In our simplified version, default negated atoms are replaced in the rules $\mathbf{RU}$ according to their truth value w.r.t. $\mathbf{M}$, i.e. $not\ a$ is *true* iff $a \notin \mathbf{M}$. Rules in $\mathbf{RU}$ are deleted if the head does not include an atom of $\mathbf{M}$ or some of the upper bounds are violated. Note, weight constraints in the head of a rule may comprise several atoms. Roughly speaking an atom in $\mathbf{M}$ is justified iff it is contained in the head of a rule and all atoms and weight constraints of this rule are justified. *True* is always and *false* is never justified. A weight/cardinality constraint in the body of a rule is justified if enough atoms contained in the weight/cardinality constraint are justified s.t. the lower bound is met. Facts are rules with *true* as body. Justifications must be acyclic. For instance, $0 \leq \{a, b\} \leq 1 \leftarrow c$ is satisfied by $\{a\}$ but $\{a\}$ is not justified. However, if we add the fact c to the knowledge base, $\{c\}$, $\{c, a\}$, and $\{c, b\}$ are stable models.

In order to allow logical variables and functional symbols but to guarantee decidability the set of allowed rules is restricted. Potassco [4] requires level-restricted programs. The basic idea is that for each variable V in a rule there is an unnegated atom a in the body s.t. the potentially derivable ground instances of a are limited. If such an atom is available the ground terms to which V needs to be instantiated are known a-priori. I.e. every variable in a rule must be bound to a finite set of ground terms via a predicate that is not subject to a positive recursion (recursion over unnegated atoms) through that rule.

For a succinct specification of facts in our example we use so-called intervals, e.g. $\texttt{person}(1..2)$. corresponds to the facts $\texttt{person}(1)$. $\texttt{person}(2)$. To exemplify the application of cardinality constrains, let an ASP program contain the facts:

$\texttt{thing}(3..4)$. $\texttt{cabinetDomain}(9..10)$.

In order to formulate weight constraints concisely, so called conditional literals are supported. The basic idea is that conditional literals serve as a generator for producing a set of atoms. The constraint

$$1\{\texttt{cabinetTOthing}(\texttt{X},\texttt{Y}) : \texttt{cabinetDomain}(\texttt{X})\}1 \leftarrow \texttt{thing}(\texttt{Y}).$$

where $\texttt{cabinetTOthing}(\texttt{X},\texttt{Y}) : \texttt{cabinetDomain}(\texttt{X})$ is a conditional literal, which is expanded to

$$1\{\texttt{cabinetTOthing}(9,3), \texttt{cabinetTOthing}(10,3)\}1 \leftarrow \texttt{thing}(3).$$
$$1\{\texttt{cabinetTOthing}(9,4), \texttt{cabinetTOthing}(10,4)\}1 \leftarrow \texttt{thing}(4).$$

expressing that things 3 and 4 must be connected to exactly one of the cabinets 9 and 10. Conditional literals can be used in weight constraints in place of atoms, where the *conditional part* (e.g. $\texttt{cabinetDomain}(\texttt{X})$) is a (conjunction of) *domain predicate*(s) preceded by the *main part*. As usual, strings starting with upper case letters are logical variables. The instantiation of domain predicates is defined by non-recursive normal rules and ground facts. For instantiating conditional literals we have to distinguish between *local* and *global* variables. A variable is local iff it appears only in a conditional literal, e.g. X is local in our example. All other variables are global, e.g. Y. During grounding of the rules, global variables are instantiated first. Then the main part of the conditional literal is expanded for the instantiations of the local variables where the conditional part is fulfilled.

Note, in Potassco [4] weight constraints are declared by square brackets $\texttt{l} \leq [\texttt{L}_1 = \texttt{w}_1, \ldots, \texttt{L}_n = \texttt{w}_n] \leq \texttt{u}$, where $\texttt{L}_\texttt{i}$ is a literal and $\texttt{w}_\texttt{i}$ is a numerical value representing its weight. Literals $\texttt{L}_\texttt{i}$ could be equal. Curly brackets are employed to define cardinality constraints where duplicated literals are removed.

Answer set programming solvers like [8; 5; 4] offer optimization services. In particular, the statement $\#\texttt{minimize}[\texttt{L}_1 = \texttt{w}_1@\texttt{p}_1, \ldots, \texttt{L}_n = \texttt{w}_n@\texttt{p}_n]$. allows minimization. The minimization statement is similar to the weight constraints with a possibility to assign a priority level $\texttt{p}_\texttt{i}$ to each weighted literal. Instead of $\#\texttt{minimize}$ also $\#\texttt{maximize}$ could be used. An answer set is optimal iff the sum of the weights of literals which are satisfied in this answer set is minimal (maximal) among all answer sets of a given program. Optimization is performed in the order of priorities starting from the highest priority value.

5 Defining configuration problem instances

In [9] various modeling patterns based on weight constraints were introduced. A fixed set of ground facts define the individuals which are employed for a configuration. This fixed set of ground facts in conjunction with the level-restriction place an upper bound on the size of the number of grounded rules and therefore decidability is guaranteed. At the current state of research such an upper bound on the number of individuals is necessary for many applications. In particular, it is well known from database theory that so called tuple generating dependencies lead to undecidability even under rather strict syntactical restrictions [1]. A tuple generating dependency is $\forall\overline{X}\forall\overline{Y}\phi(\overline{X},\overline{Y}) \rightarrow \exists\overline{Z}\psi(\overline{X},\overline{Z})$ where $\phi(\overline{X},\overline{Y})$ and $\psi(\overline{X},\overline{Z})$ are conjunctions of atoms and $\overline{X}, \overline{Y}$, and $\overline{Z}$ are representing vectors of logical variables. Unfortunately, such rules may occur in configuration problem instances. E.g. if a condition holds, a specific individual of some type must exist and this individual must be connected to some other individuals.

However, in many cases it is undesirable to consider only a fixed number of individuals employed in a configuration. Guessing the right number is for configuration generation problems or optimization problems quite hard and often impossible. Therefore we apply the following modeling pattern.

Let $\texttt{pLower}$ and $\texttt{pUpper}$ represent the upper and lower number of individuals of type $\texttt{p}$. Such a type is called *bounded*. We require each individual of a configuration, represented by its unique identifier, to be a member of exactly one bounded type. To each bounded type a domain $\texttt{pDomain}$ is associated, representing the set of possible individuals of the bounded type. We employ numbers as identifiers, starting from some offset. For every bounded type $\texttt{p}$ we add the following axioms:

$$\texttt{pDomain}(\texttt{pOffset} + 1\ ..\ \texttt{pOffset} + \texttt{pUpper}).$$
$$\texttt{pLower}\{\texttt{p}(\texttt{X}) : \texttt{pDomain}(\texttt{X})\}\texttt{pUpper}.$$
$$\texttt{p}(\texttt{X}) \leftarrow \texttt{pDomain}(\texttt{X}), \texttt{pDomain}(\texttt{Y}), \texttt{p}(\texttt{Y}), \texttt{X} < \texttt{Y}.$$

The first rule instantiates the maximal required number of unique individuals of $\texttt{p}$ in $\texttt{pDomain}$. The second rule makes sure that at least $\texttt{pLower}$, but at most $\texttt{pUpper}$ individuals of $\texttt{p}$ are asserted. The third rule breaks the symmetry of assertions. By these rules the required number of $\texttt{p}$ individuals are asserted, in order to find a configuration within the given upper and lower bounds.

For some bounded types, e.g. `person/1` and `thing/1` the bounds `pLower` and `pUpper` coincide because the exact number of individuals employed in any configuration is known. In this case the fixed set of `p` facts can be asserted without using the rules presented above.

In our example the customer provides a number of requirements for a configuration that include definitions of person and thing individuals as well as their relations.

```
person(1..2). thing(3..8).
personTOthing(1,3). personTOthing(1,4).
personTOthing(1,5). personTOthing(1,6).
personTOthing(1,7). personTOthing(2,8).
```

For the bounded type cabinet we add the following rules. The upper and lower numbers of cabinets are computed based on the number of things and persons. The rules for rooms are defined accordingly.

```
cabinetDomain(9..14).
2{cabinet(X):cabinetDomain(X)}6.
cabinet(X) :- cabinetDomain(X), cabinetDomain(Y),
                                cabinet(Y), X<Y.
```

Cardinality restrictions given in Section 2 are encoded with cardinality constraints, where one direction of an association is encoded as a generation rule (see Section 4) and the other direction as a constraint. Such encoding corresponds to Guess/Check/Optimize pattern [5]. Note, the cardinality constraints just as the weight constraints require that logical variables appear in domain predicates. Therefore, we have to use `pDomain` predicates rather than `p` predicates, e.g. `cabinetDomain(X)` instead of `cabinet(X)`. However, individuals employed in relations must also be contained in the corresponding types (see the last four rules of the next sequence of rules). By these rules we avoid situations where an individual is used in a relation but not included in the bounded type. In our example, if the program asserts `cabinetTOthing(14,1)` then `cabinet(14)` is also asserted.

```
1{cabinetTOthing(X,Y):cabinetDomain(X)}1 :- thing(Y).
:- 6 {cabinetTOthing(X,Y):thing(Y)}, cabinet(X).
1{roomTOcabinet(X,Y):roomDomain(X)}1 :- cabinet(Y).
:- 5 {roomTOcabinet(X,Y):cabinetDomain(Y)}, room(X).
room(X) :- roomTOcabinet(X,Y).
room(Y) :- personTOroom(X,Y).
cabinet(X) :- cabinetTOthing(X,Y).
cabinet(Y) :- roomTOcabinet(X,Y).
```

The next rules describe the fact that a room may contain things of its owner only.

```
personTOroom(P,R) :- personTOthing(P,X),
          cabinetTOthing(C,X), roomTOcabinet(R,C).
:- personTOroom(P1,R), personTOroom(P2,R), P1!=P2.
```

In addition, optimization can be applied to generate optimal configurations which minimize the overall configuration costs depending on the objective function. We model the objective function by assigning to each atom in $\mathbf{S}$ some costs. This can be achieved with the following modeling pattern. By the atom $\mathtt{cost}(\mathtt{create}(\mathtt{a},\mathtt{w}))$, where a is an element of $\mathbf{S}$ and $\mathtt{w}$ is an integer, the costs of creating an element a in a configuration are defined. We employ the conjunction of atoms $\alpha(\overline{\mathtt{X}},\overline{\mathtt{Y}},\mathtt{W})$ to allow case specific determination of costs. For each $\mathtt{p} \in \mathbf{P_S}$ include axioms of the following form in $\mathbf{REQ}$:

$$\mathtt{cost}(\mathtt{create}(\mathtt{p}(\overline{\mathtt{X}})),\mathtt{W}) \leftarrow \mathtt{p}(\overline{\mathtt{X}}), \alpha(\overline{\mathtt{X}},\overline{\mathtt{Y}},\mathtt{W}).$$

such that for each atom $\mathtt{p}(\overline{\mathtt{t}})$ in $\mathbf{S}$ the answer set contains an atom $\mathtt{cost}(\mathtt{create}(\mathtt{p}(\overline{\mathtt{t}}),\mathtt{w}))$ where $\mathtt{w}$ is an integer. E.g.:

```
roomCost(5). personTOroomCost(1).
cost(create(room(X)),W) :- room(X), roomCost(W).
cost(create(personTOroom(X,Y)), W) :-
        personTOroom(X,Y), personTOroomCost(W).
```

All other creation costs are expressed in the same way. We minimize the sum of all costs by means of the following optimization statement:

```
#minimize[cost(X,W)=W@1].
```

For the given example the solver finds the optimal configuration including two cabinets and two rooms with the overall cost 40 (depicted in Figure 1).

```
{cabinet(10), cabinet(9), room(16), room(15), ...,
roomTOcabinet(15,9), roomTOcabinet(16,10),
cabinetTOthing(10,8) cabinetTOthing(9,7), ...,
cabinetTOthing(9,3), personTOroom(1,15),
personTOroom(2,16)}
```

6 Reconfiguration problems

We view reconfiguration as a new configuration-generation problem where parts of a *legacy configuration* are possibly reused. The conditions under which some parts of the legacy configuration can be reused and what the consequences of a reuse are, is expressed by a set of logical sentences $\mathbf{T}$ which relate the legacy configuration $\mathbf{S}$ and the new configuration problem instance $\langle \mathbf{REQ_R}, \mathbf{P_R} \rangle$.

Definition 4 (Instances of reconfiguration problems) *A reconfiguration problem instance* $\langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *is defined by:* $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$ *an instance of a configuration problem,* $\mathbf{S}$ *a legacy configuration and* $\mathbf{T}$ *a set of logical sentences representing the transformation constraints regarding the legacy configuration.*

For optimization purposes an objective function $g(\mathbf{S},\mathbf{R}) \mapsto \mathbb{N}$ *maps legacy configurations* $\mathbf{S}$ *and configurations* $\mathbf{R}$ *of* $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$ *to positive integers.*

Note, the two-placed objective function expresses the fact that the costs of an reconfiguration depend not only on the elements contained in a reconfiguration but also on the reuse or deletion of elements of the legacy configuration.

In order to avoid name conflicts between the entities of the legacy configuration $\mathbf{S}$ and instances of new configuration problems $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$, we usually formulate $\mathbf{P_R}$ and $\mathbf{REQ_R}$ using constants not employed in $\mathbf{S}$. In particular, we use different name spaces for terms referencing individuals. Together with the unique name assumption this implies that individuals of the legacy configuration and new individuals introduced by the reconfiguration problem are disjunct.

Reconfigurations are defined analog to configurations as a finite subset of Herbrand-models.

Definition 5 (Reconfiguration) $\mathbf{R}$ *is a reconfiguration for a reconfiguration problem instance* RCI $= \langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathcal{HM}(\mathbf{REQ_R} \cup$

$\mathbf{S} \cup \mathbf{T}$) *and* $\mathbf{R}$ *is the set of all the elements of* $\mathbf{M}$ *whose predicate symbols are in* $\mathbf{P_R}$ *and* $\mathbf{R}$ *is finite.*

$\mathbf{R}$ *is an optimal reconfiguration for* RCI *iff* $\mathbf{R}$ *is a reconfiguration for* RCI *and there is no reconfiguration* $\mathbf{R}'$ *of* RCI *s.t.* $g(\mathbf{S}, \mathbf{R}') < g(\mathbf{S}, \mathbf{R})$.

Reconfiguration problems are formulated analog to configuration problems.

Definition 6 (Reconfiguration problems) *The instances of reconfiguration problems are defined by a tuple* $\langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *and objective functions* $g(\cdot,\cdot)$.

Decision problem: *Given a set of atoms* $\mathbf{R}$. *Decide if* $\mathbf{R}$ *is a reconfiguration for a reconfiguration problem instance.*

Generation (optimization) problem: *Generate a set of atoms* $\mathbf{R}$ *s.t.* $\mathbf{R}$ *is a reconfiguration (an optimal reconfiguration) for a reconfiguration problem instance.*

Because we can reduce configuration problems to reconfiguration problems and vice versa the following property follows trivially.

Property 1 *Employing a logical representation language for representing instances of configuration problems and reconfiguration problems whose satisfiability problem is at least NP-complete, generating a(n optimal) reconfiguration is as hard as generating a(n optimal) configuration w.r.t. computational complexity.*

7 Defining reconfiguration problem instances

In the following we show typical formalization patterns and apply them to our example. The set of atoms $\{\texttt{legacyConfig}(a) | a \in \mathbf{S}\}$ describes the atoms of the legacy configuration $\mathbf{S}$. Note, the definition of reconfiguration problems does not employ first-order logic constructs in order to avoid unnecessary restrictions. However, to facilitate a concise description of the problem we introduce the predicate `legacyConfig/1` to allow quantification over the elements of the legacy configuration. Note, we could rewrite all shown axioms to propositional logic.

For the transformation sentences $\mathbf{T}$ we employ the following general patterns. For reusing parts of the legacy configuration the problem solver has to make the decision either to *reuse* or to *delete*. This is expressed by `reuse(a)` and `delete(a)` atoms where a is an element of $\mathbf{S}$. For each atom a in $\mathbf{S}$ either `reuse(a)` or `delete(a)` must hold. Based on these atoms additional configuration constraints can be defined which describe the proper reuse or deletion of a part of the legacy configuration represented by atom a. In our case, reusing an atom a of the legacy configuration implies the assertion of this atom, whereas deletion requires that the atom is not asserted. In addition, costs are associated to each `reuse(a)` or `delete(a)` operation. This is expressed by the atom `cost(reuse(a), w)` or `cost(delete(a), w)` where `a` is an element of $\mathbf{S}$ and `w` is an integer specifying the corresponding costs. Furthermore, we require that in each model which contains `reuse(a)` or `delete(a)` also `cost(reuse(a), w)` or `cost(delete(a), w)` is contained in order to have defined reuse or deletion costs. The conjunctions $\beta(\overline{X}, \overline{Y}, W)$ and $\gamma(\overline{X}, \overline{Y}, W)$ are employed to define case specific costs.

For each $\texttt{p} \in \mathbf{P_S}$ include the following axioms in $\mathbf{T}$:

$\texttt{1\{reuse(p(}\overline{\texttt{X}}\texttt{)), delete(p(}\overline{\texttt{X}}\texttt{))\}1} \leftarrow \texttt{legacyConfig(p(}\overline{\texttt{X}}\texttt{))}.$

$\texttt{p(}\overline{\texttt{X}}\texttt{)} \leftarrow \texttt{reuse(p(}\overline{\texttt{X}}\texttt{))}.$

$\leftarrow \texttt{p(}\overline{\texttt{X}}\texttt{), delete(p(}\overline{\texttt{X}}\texttt{))}.$

$\texttt{cost(reuse(p(}\overline{\texttt{X}}\texttt{)), W)} \leftarrow \texttt{reuse(p(}\overline{\texttt{X}}\texttt{))}, \beta(\overline{\texttt{X}}, \overline{\texttt{Y}}, \texttt{W}).$

$\texttt{cost(delete(p(}\overline{\texttt{X}}\texttt{)), W)} \leftarrow \texttt{delete(p(}\overline{\texttt{X}}\texttt{))}, \gamma(\overline{\texttt{X}}, \overline{\texttt{Y}}, \texttt{W}).$

Analog to configuration problems, we require each individual contained in a reconfiguration to be a member of exactly one bounded type. Consequently, individuals of the legacy configuration have to be a member of the domain `pDomain(X)` of a bounded type `p` of $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$, because these individuals can be part of a reconfiguration through reuse. I.e. there are rules of the form

$\texttt{pDomain(X)} \leftarrow \texttt{legacyConfig(q(}\ldots\texttt{, X, }\ldots\texttt{))}.$

where `q` is predicate symbol of the solution schema of the legacy configuration.

As for configuration problems, the number of individuals of a bounded type `p` is limited. For every bounded type `p` we add the following axioms:

$\texttt{pLower\{p(X) : pDomain(X)\}pUpper}.$

However, the two other rules for bounded types are changed. In particular, we have to adapt the symmetry breaking pattern of configurations. The reason is that there are two different types of individuals contained in `pDomain`, those which are reused and those which are newly generated. Symmetry breaking does not apply to the reused individuals because they may be linked to other reused individuals. Therefore, exchanging these individuals potentially leads to different configurations. However, the newly generated individuals are interchangeable. We describe them by `pDomainNew/1` for the bounded type `p`. We use `pNewOffset` to generate new identifiers. I.e. the pattern is

$\texttt{pDomainNew(pNewOffset} + \texttt{1 .. pNewOffset} + \texttt{pUpper)}.$

$\texttt{pDomain(X)} \leftarrow \texttt{pDomainNew(X)}.$

$\texttt{p(X)} \leftarrow \texttt{pDomainNew(X), pDomainNew(Y), p(Y), X} < \texttt{Y}.$

In our example, the reconfiguration problem consists of additional customer and configuration requirements described in Section 2. The solution schema for the reconfiguration problem is an extension of the solution schema of the original configuration problem by `cabinetHigh/1`, `cabinetSmall/1`, `thingLong/1` and `thingShort/1` predicates. The additional requirements of the customer are expressed by:

```
thingLong(3).   thingShort(4). thingShort(5).
thingShort(6). thingShort(7). thingLong(8).
thing(21).      thingLong(21). personTOthing(1,21).
```

The legacy configuration presented in Section 3 is encoded using `legacyConfig` predicate as described above.

```
legacyConfig(cabinet(9)). legacyConfig(cabinet(10)).
legacyConfig(cabinetTOthing(10,8)).
legacyConfig(roomTOcabinet(16,10)). ...
```

To implement the configuration requirements of the modified problem we add rules defining the subtypes of cabinets as well as that long things have to be stored in high cabinets. Note, only some of the usual rules for expressing subtypes are needed. Regarding subtypes of thing, no rules are needed at all because for every `thing` fact either a `thingLong` fact or a `thingShort` fact is contained in the customer requirements and none of these predicates appear in the head of a rule.

```
1{cabinetHigh(X), cabinetSmall(X)}1 :- cabinet(X).
cabinetHigh(C) :- thingLong(X), cabinetTOthing(C,X).
```

Moreover, each high cabinet requires more space in a room. Such a cabinet occupies two of the four available slots in a room, whereas a small cabinet uses only one slot. Note, the last constraint does not allow an answer set where the sum of occupied slots in a room is 5 or more.

```
cabinetSize(X,1) :- cabinet(X), cabinetSmall(X).
cabinetSize(X,2) :- cabinet(X), cabinetHigh(X).
roomTOcabinetSlot(R,C,S) :- roomTOcabinet(R,C),
                                   cabinetSize(C,S).
:- 5 [roomTOcabinetSlot(X,Y,S):
            cabinetDomain(Y)=S], room(X).
```

The domains of cabinets and rooms are extended with additional individuals that might be required in a new configuration. The number of new elements in the cabinet and room domains corresponds to the number of things in the modified problem. The upper number `pUpper` of both cabinet and room individuals is set to 7 because 7 things must be stored in the house.

```
cabinetDomainNew(22..28).
cabinetDomain(X) :- cabinetDomainNew(X).
2{cabinet(X):cabinetDomain(X)}7.
cabinet(X) :- cabinetDomainNew(X), cabinet(Y), X<Y,
            cabinetDomainNew(Y).
```

The modeling of new rooms is done in the same way.

The transformation rules are implemented as described above. E.g.

```
1{reuse(cabinet(X)), delete(cabinet(X))}1 :-
                          legacyConfig(cabinet(X)).
cabinetDomain(X) :- legacyConfig(cabinet(X)).
```

However, the transformation rules for `legacyConfig(person(X))`, `legacyConfig(thing(X))` and `legacyConfig(personTOthing(X,Y))` could be deleted because facts about persons, things and their relations are given as requirements. Deleting such an atom results in a contradiction.

Given the reconfiguration program the solver identifies a reconfiguration as well as a set of actions required to transform the legacy configuration into a new one.

For generating optimal reconfigurations we formulate a cost model. The minimization statement in the reconfiguration problem is the same as in the configuration. In our reconfiguration example the costs for creation of new high/small cabinets and rooms `cost(create(a), w)` correspond to the costs definition of the configuration problem. To obtain a reconfiguration scenario with the minimal costs of required actions we extend the costs rules described above with costs for creation of new high/small cabinet and room individuals as well as with costs for newly created relations. E.g.

```
cost(create(cabinetHigh(X)),W) :- cabinetHigh(X),
        cabinetHighCost(W), cabinetDomainNew(X).
```

Rules for deducing the costs of reuse and deletion are formulated as described above.

For our example let us assume that the customer sets all deletion costs to 2, whereas reusing has no costs except for cabinets, which could be altered to high in a reconfiguration. The costs of this alteration is set to 3. Creation costs of new high and small cabinets are set to 10 and 5 respectively. Finally, the costs of a new room is set to 5. Creation of relations between individuals is for free. Given these costs assignments the solver is able to find a set of optimal reconfigurations including the one presented in Figure 3.

Modification of the costs results into different optimal reconfigurations. Let us assume the sales-department changes both the costs of deletion of a cabinet and the costs of increasing the height of a cabinet to 10, and decreases the creation costs of new high and small cabinets to 2 and 1 respectively. In this case the solutions returned by a solver will include the one presented in Figure 4. Given their simplicity, the presented optimal solutions were found in milliseconds.

8 Evaluation

The evaluation of our approach was done on a set of test cases derived from four reconfiguration scenarios encountered by us in practice. Each scenario can be represented as an instance of the (re)configuration problem presented in Section 2. In the *empty* reconfiguration scenario the legacy configuration is empty and the customer requirements contain sets of things and persons owning 5 things each. Every thing is labeled as short. The reconfiguration process should create missing cabinets, rooms as well as all required relations.

The customer requirements of the *long* scenario specify that each given person owns 15 things. The legacy configuration contains a set of relations that indicate placement of these things into cabinets, s.t. all things of one person are stored in three cabinets that are placed in one room. The customer also requires 5 things of each person to be labeled as long whereas the remaining 10 as short. The goal of the reconfiguration is to find a valid rearrangement of long things to reused or newly created high cabinets.

The next *new room* scenario models a situation when new rooms have to be created and some of the cabinets reallocated. In this scenario each person owns 12 things. These things are stored in 3 cabinets placed in one room as indicated by the legacy configuration. In the reconfiguration problem the customer requirements declare 6 of the 12 things as long.

The last scenario, *swap*, describes a situation when the customer requirements include only one person, who owns 35 things. In the legacy configuration the things are placed in 3 cabinets in the first room and in 4 cabinets in the second room. Moreover, one of the things in the second room is labeled as high in the customer requirements. Given the costs schema presented above, the solution corresponds to a rearrangement of the cabinets in the rooms such that a high cabinet can be placed in one of these rooms. All these scenarios can be easily scaled by increasing the number of things. The number of persons in the empty, long and new room scenarios can always be computed given the number of things.

Experiments were performed using Potassco 3.0.3 on Core2 Duo 3Ghz with 4Gb RAM. In our experiments we con-

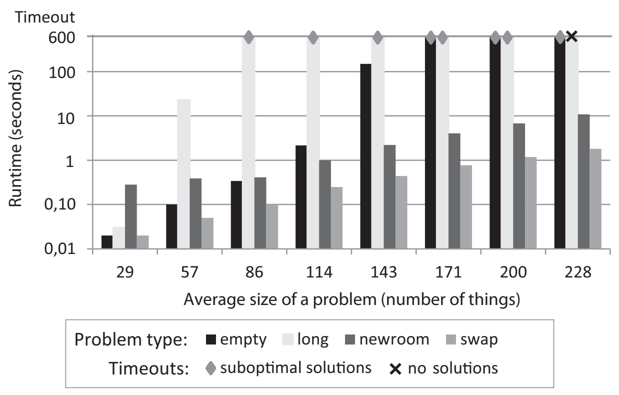


Figure 5: Evaluation results

sidered only creation costs for newly generated cabinets and rooms because these are the dominant costs for our application domain. The performance of the reconfiguration process is presented in Figure 5. Potassco was able to find optimal solutions within 600 seconds for all instances of the new room and swap scenarios. Optimal solutions were also found for small and mid-size instances of the empty scenario. For all other instances at least one suboptimal solution was found. The long scenario included the hardest problems. The solver did not find any solutions for one of them in 600 seconds. This was the only unsolved problem instance in the whole experiment. Because the solved instances are comparable to real world applications based on our experiences, we consider the proposed reconfiguration method as feasible for a practically interesting set of reconfiguration problem instances.

9 Conclusions and related work

The existing approaches for reconfiguration can be separated into revision-based [7; 10] and model-based [11]. The revision-based approaches employ a knowledge base describing “fixes”, i.e. reconfiguration operations and configuration invariants [7]. A solution requires that there is a *sequence* of operations which transform the legacy configuration into a new configuration. The approach of [11] views reconfiguration as a consistency-maintenance (diagnosis) problem, where a solution corresponds to a consistent set of assumptions s.t. requirements are implied. Similarly, our approach can be seen as searching for a consistent (optimal) set of assumptions regarding reuse or deletion of parts of the legacy configuration and creation of new parts. This search is provided by the ASP reasoning system, implementing a *correct and complete* problem solving method. No additional diagnosis component is required. Regarding the revision-based approach, our domains do not need the computation of sequences of operations, because if a reconfiguration is found, a sequence of real-world change operations can be easily derived. Thus, we can avoid the additional combinatorial explosion introduced by permutations of change operations. However, we can view our approach as a form of the revision-based method assuming that all change operations are executed simultaneously. The effects of these operations and the combination of allowed operations are described by the transformation knowledge. Thus we can model complex “fix” operations which involve the reuse of several parts of the legacy configuration and which have multiple effects such as creating new parts or deleting existing ones.

To sum up, we have developed a method which allows the modeling of reconfiguration problems based on legacy configurations, transformation knowledge, and a new configuration problem instance. We showed various modeling patterns and implemented the approach based on ASP. Evaluation results show the feasibility for practical applications.

References

[1] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 70–80. AAAI Press, 2008.

[2] A. Falkner and A. Haselböck. Challenges of Knowledge Evolution in Practice. In *Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET 2010)*, pages 1–5, 2010.

[3] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004.

[4] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user’s guide to gringo, clasp, clingo and iclingo, 2010.

[5] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

[6] P. Manhart. Reconfiguration - A problem in search of solutions. In D. Jannach and A. Felfernig, editors, *IJCAI’05 Configuration Workshop*, pages 64–67, 2005.

[7] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen. Framework and conceptual model for reconfiguration. In B. Faltings, E. C. Freuder, and G. Friedrich, editors, *AAAI’99 Workshop on Configuration*, volume 99, pages 59–64, 1999.

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

[9] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pages 195–201, 2001.

[10] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. In *2nd International Conference on Knowledge Capture*, pages 162—-171, New York, NY, USA, 2003. ACM Press.

[11] M. Stumptner and F. Wotawa. Model-based reconfiguration. In J. S. Gero and F. Sudweeks, editors, *5th International Conference on Artificial Intelligence in Design*, pages 45–64, 1998.

Enumeration of valid partial configurations

Alexey Voronov, Knut Åkesson, Fredrik Ekstedt
Chalmers University of Technology, Göteborg, Sweden
`{voronov, knut}@chalmers.se`
Fraunhofer-Chalmers Research Centre for Industrial Mathematics, Göteborg, Sweden
`fredrik.ekstedt@fcc.chalmers.se`

Abstract

Models of configurable products can have hundreds of variables and thousands of configuration constraints. A product engineer usually has a limited responsibility area, and thus is interested in only a small subset of the variables that are relevant to the responsibility area. It is important for the engineer to have an overview of possible products with respect to the responsibility area, with all irrelevant information omitted. Configurations with some variables omitted we will call *partial configurations*, and we will call a partial configuration *valid* if it can be extended to a complete configuration satisfying all configuration constraints. In this paper we consider exact ways to compute valid partial configurations: we present two new algorithms based on Boolean satisfiability solvers, as well as ways to use knowledge compilation methods (Binary Decision Diagrams and Decomposable Negation Normal Form) to compute valid partial configurations. We also show that the proposed methods are feasible on configuration data from two automotive companies.

1 Introduction

Within the automotive industry it is common to have a few general platforms where each platform are highly configurable to adapt to the needs on different markets but also to satisfy the needs of individual customers. Having highly configurable products put high demands on the engineering systems to support the engineers during the development of the platforms. While having a configurable product or platform it is inevitable to also have constraints that specify what can be allowed together and what is not allowed together. These constraints can, in many situations, be defined as a set of Boolean formulas defined over finite domain variables.

Development of complex products—like in the automotive industry—is done in large teams, where each engineer is working with only a limited part of the product. For the individual engineer only a small subset of the variables from those that describe the full product, are of immediate interest. However, an important problem for the engineer is to know which combinations of variable assignments for a subset of the variables might result in a product that satisfies all constraints, that is to know *valid partial configurations*. This is important because it helps the engineer to develop solutions only for those combinations that can actually be built and sold. Overestimation of these solutions will lead to engineer doing unnecessary designs. Underestimation can lead to costly delays if an overlooked configuration is requested by a customer afterwards.

Computing the exact set of valid partial configurations is generally hard. Simply taking configurations of the complete products and projecting them on relevant variables is not feasible, as practical problems can have 10^{120} and more buildable complete products.

One of the ways to get exact set of valid partial configurations is to existentially quantify all irrelevant variables from the formula that represents the conjunction of all configuration constraints, for example using resolution inference rule [Robinson, 1965; Davis and Putnam, 1960], and then use a standard algorithm to enumerate all (complete) assignments of the new simplified formula. A simple enumeration of complete assignments searches for a satisfying assignment, adds it to the result, and also adds it to the current set of constraints as a blocking constraint, forbidding future search from returning it again. The disadvantage of this approach is that the formula size can grow significantly after quantification. In this paper we present a modification of this enumeration algorithm that does not require existential quantification of variables to enumerate partial configurations (Section 4.1). Similar algorithm can be found in [Gebser *et al.*, 2009].

The problem of partial configurations can also be tackled by the widely used interactive configurators. In an interactive configurator a user selects values for variables one by one. The configurator should guide the user so that at any point there exist at least one way to complete the configuration without changing any of the earlier decisions, in this case a configurator is *backtrack-free*. Configurator should also be *complete* meaning that if a configuration is allowed according to the constraints, configurator should allow it. Having such complete and backtrack-free configurator, it is possible to automatically check all (partial) assignments of values to the relevant variables, and the configurator will show which of them are valid. If a configurator is not backtrack-free, it can overestimate allowed partial configurations. If it is not complete, it will underestimate them.

Previous work on methods for building interactive configurators started as extensions of Constraint Satisfaction Problem (CSP) with conditional and dynamic formulations [Dechter and Dechter, 1988; Mittal and Falkenhainer, 1990; Soininen and Gelle, 1999; Sabin and Freuder, 1998; Gottlob *et al.*, 2007]. However, supported implementations of these algorithms are not readily available. Binary Decision Diagrams (BDDs) [Bryant, 1986] is a knowledge compilation method [Darwiche and Marquis, 2002] successfully used for configuration [Hadzic *et al.*, 2004], especially for real-time interactive configuration. However, BDDs suffer from memory explosion for many datasets of practical size. Other knowledge compilation methods used for configuration include automata representation [Amilhastre *et al.*, 2002], Tree-of-BDDs [Subbarayan, 2005], Joint Matched CSP [Subbarayan and Andersen, 2005], Decomposable Negation Normal Form (DNNF) [Darwiche and Marquis, 2002] (especially Deterministic DNNF for model counting [Kübler *et al.*, 2010]), as well as combinations of search and BDDs [Norgaard *et al.*, 2009]. Recently, Boolean Satisfiability Solvers (SAT-solvers) emerged as an alternative to work with configurations [Sinz *et al.*, 2003; Küchlin and Sinz, 2000; Sinz *et al.*, 2006; Janota, 2008], including interactive configurators [Janota, 2010].

In this paper we show how a SAT-solver can be used to enumerate partial configurations based on modification of standard enumeration algorithm, and based on checking every partial assignment inspired by interactive configuration. We also show how existing DNNF algorithms can be used to enumerate partial configurations. We show feasibility of a SAT-solver based implementation on configuration data from two automotive companies.

The paper is organized as following. Section 2 covers formal preliminaries, Section 3 gives a motivating example, Section 4 presents the algorithms. Section 5 provides experimental results and discussion, and Section 6 concludes the paper.

2 Preliminaries

The configuration problem is a triple $\mathcal{P} = \langle X, D, C \rangle$, where $X = \{x_1, x_2, \ldots, x_K\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_K\}$ is a set of corresponding finite domains, and $C = \{C_1, C_2, \ldots, C_J\}$ is a set of propositional formulas over atomic propositions $x_k = v$ where $v \in D_k$, specifying conditions that the variable assignments have to satisfy.

A *complete assignment* to a configuration problem $\mathcal{P}$ is a function $A : X \to D$ which is defined for all $x_k \in X$. A *valid* complete assignment (or solution) to $\mathcal{P}$ is a complete assignment A for which each C_j is satisfied. A *partial* assignment to $\mathcal{P}$ is a partial function $B : X \to D$ defined for variables $x_k \in Y \subseteq X$. We will write $\mathit{vars}(B) = Y \subseteq X$ to denote the set of variables of B, or the *scope* of B. We will call a partial assignment *valid* iff it can be extended to a valid complete assignment. We will use $\mathcal{P}[B]$ to denote the simplified problem obtained by setting the variables defined in B.

3 Motivating example

Configuration problems describing complete products can have thousands of variables and hundreds of thousands constraints. An engineer, or a group of engineers, is usually responsible only for a subset of the variables. It could be that it is a requirement to design all possible solutions within the responsibility area, in case someone will order such a product. In such a case it can be expensive to have overestimated set of valid configurations, since engineers will spend time designing forbidden ones. Underestimations are also bad, since they lead to delays for designing a solution for ordered, but missed configuration.

The problem can be illustrated by the following example of a simple car configuration. Let $X = \{\mathit{body}, \mathit{engine}, \mathit{transmission}\}$ be the set of variables, and $D = \{\{mini, sedan, suv\}, \{gasoline, diesel, electric\}, \{manual, auto, evt\}\}$ be the set of corresponding domains. Let the following be the set of constraints C:

- $\neg((body = mini) \wedge (engine = gasoline))$
- $\neg((body = mini) \wedge (engine = diesel))$
- $\neg((body = sedan) \wedge (engine = electric))$
- $\neg((body = suv) \wedge (engine = gasoline))$
- $(engine = electric) \to (transmission = evt)$
- $(transmission = evt) \to (engine = electric)$

Valid assignments of $\mathcal{P} = \langle X, D, C \rangle$ can be presented, for example, in a tabular form, as shown in Table 1. Each row in the table corresponds to an assignment, and each column corresponds to a variable. Each cell contains a value assigned to the corresponding variable.

Let us suppose that there is a group of engineers that are interested only in connection between *body* and *transmission*, and they would like to disregard all information about *engine*. So they define the limited scope to be $S = \{body, transmission\}$. Valid partial assignments for S are presented in Table 2.

One way to get the partial assignments is to enumerate all complete assignments, project them onto the relevant variables (remove the irrelevant columns from the table), and remove duplicate partial assignments (rows). This approach is infeasible in practice, as some industrial examples from automotive industry have 10^{120} allowed complete assignments. However, scopes of interest for the engineers may have less

Table 1: Valid complete assignments

body	engine	transmission
mini	electric	evt
sedan	gasoline	manual
sedan	gasoline	automatic
sedan	diesel	manual
sedan	diesel	automatic
suv	diesel	manual
suv	diesel	automatic
suv	electric	evt

Table 2: Valid partial assignments

body	transmission
mini	evt
sedan	manual
sedan	automatic
suv	manual
suv	automatic
suv	evt

than a thousand valid partial assignments, which is computable using methods presented below. Approximations of this approach are found in practice, where instead of all valid assignments, only the ones that correspond to the products built during the last year (for example) are considered, which gives an underestimation of the answer. Clearly, there is a need for a better method.

4 Enumerating valid partial assignments

This sections presents two algorithms adopted to solve the problem based on *satisfiability solvers*. By a satisfiability solver we mean a tool that is able to answer whether an instance of a configuration problem has at least one valid complete assignment, and provides one if such exists. For example, tools for solving Constraint Satisfaction Problems (CSP-solvers) and Boolean Satisfiability Problems (SAT-solvers) can be used for this purpose. This section also shows how DNNF algorithms can be used to enumerate valid partial configurations.

4.1 Searching for complete, then forbidding partial

One way to enumerate all valid complete assignments is to iteratively search for any valid complete assignment, and in addition to adding it to the result, add a negation of it as a blocking constraint to the existing set of constraints. This algorithm can be modified to enumerate valid partial assignments, as shown in Algorithm 1. When a solver returns the first complete assignment, the assignment is projected onto the relevant scope. This partial assignment is returned as the first element of the result, and also added as a blocking constraint, ensuring that the solver will not return any (complete) assignment that will contain the partial one. Then this process is repeated.

The ability of the solver to incrementally add blocking constraints, while still keeping previously inferred information, is very important for the good performance of this algorithm. This is supported by, for example, Minisat-like solvers [Een and Sörensson, 2004; Een and Sörensson, 2003].

Algorithm 1 Search for complete, then forbid partial

```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
$C' \leftarrow C$
$\mathcal{P}' \leftarrow \mathcal{P}$
result $\leftarrow \{\}$
while sat($\mathcal{P}$): /* ask solver */
    $A \leftarrow$ assignment($\mathcal{P}$) /* assignment from solver */
    $B \leftarrow A$ projected on $S$
    result $\leftarrow$ result $\cup B$
    $C' \leftarrow C' \wedge \neg(B$ as constraint)
    $\mathcal{P}' \leftarrow \langle X, D, C' \rangle)$
return result
```

4.2 Enumerating partial, then extending

In this approach it is necessary to enumerate *all* partial assignments, and try to extend each of them to a valid complete assignment using a solver. Just checking a partial assignment against each of the constraints in isolation is not enough, because there could be dependencies between variables that are not visible within the local scope, but are only visible within the complete scope. The solver can be used as following: each partial assignment is added as an extra constraint to the set of configuration constraints, and removed after the solver has returned an answer. The key to the good performance in this method is in the ability of the solver to cheaply add and retract constraints consisting of atomic propositions; again, Minisat-like solvers have this feature.

Algorithm 2 Enumerate partial, then extend

```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
output: valid partial assignments
result $\leftarrow \{\}$
for $B$ in allAssignments($S$):
    if sat( $\mathcal{P}[B]$ ):
        result $\leftarrow$ result $\cup B$
return result
```

An example illustrating this method is presented in Table 3. The columns for *body* and *transmission* contain all possible (not only valid) partial assignments for S. The table must be extended with the columns for variables $(X \setminus S)$, in this case it is only one, *engine*. If there is at least one valid complete assignment that contains the partial one for the row, the values for all variables are written in the row. Otherwise, a "—" indicates that there is no such valid complete assignment, and the partial assignment is not valid.

Table 3: Illustration of Algorithm 2 (Enumerate partial, then extend).

body	transmission	*engine*
mini	manual	—
mini	autmatic	—
mini	evt	*electric*
sedan	manual	*gasoline*
sedan	automatic	*gasoline*
sedan	evt	—
suv	manual	*diesel*
suv	automatic	*diesel*
suv	evt	*electric*

An advantage of Algorithm 1 compared to Algorithm 2 is that it builds upon heuristics of an underlying solver to skip checking many of the non-allowed assignments. A disadvantage is that to find the next partial assignment, it is necessary to process a larger (increased by one) set of constraints; this could be a problem when it is necessary to produce millions of partial assignments. But when there is a small number of valid partial assignments, or a user specifically asked for the first hundred of assignments as a sample, and there are many non-allowed configurations, Algorithm 1 can be beneficial.

4.3 Knowledge compilation: DNNF

Knowledge compilation is a family of approaches that addresses intractability of many Artificial Intelligence problems. A propositional model is compiled in an off-line phase in order to support some queries in polytime [Darwiche and Marquis, 2002]. Binary Decision Diagrams (BDDs) [Bryant, 1986] belong to this family and received substantial attention as a tool for configuration problems [Hadzic *et al.*, 2004]. Decomposable Negation Normal Form (DNNF) [Darwiche, 2001] is a data structure used in knowledge compilation for which BDD is a special case. It is more succinct than BDDs and its compilation time is often shorter than that of BDDs [Subbarayan *et al.*, 2007]. DNNF supports smaller number of tractable operations than BDD, while still allowing polytime existential quantification (forgetting) and assignments enumeration, which together allow polytime partial assignments enumeration once DNNF is compiled.

Formally, a propositional formula a is in negation normal form (NNF) if and only if a is either a positive or negative atomic proposition (a *literal*); a conjunction $\wedge_i a_i$; or a disjunction $\vee_i a_i$ where each a_i is in negation normal form. A formula in NNF f is *decomposable* (DNNF) if and only if for any conjunction $a = a_1 \wedge \cdots \wedge a_n$ no atomic propositions are shared by any conjuncts in a: $\text{ATOMS}(a_i) \cap \text{ATOMS}(a_j) = \emptyset$ for every $i \neq j$. A formula in NNF is *smooth* if for every disjunction $a = a_1 \vee \cdots \vee a_n$, $\text{ATOMS}(a) = \text{ATOMS}(a_i)$ for every i.

Existential quantification of variables from DNNF is presented in Algorithm 3 [Darwiche, 2001]. Every occurence of irrelevant variable is replaced by *true*, and the formula is simplified accordingly. This procedure preserves decomposability, and its running time is linear in the DNNF size.

Algorithm 3 FORGET – existential quantification on DNNF

```
input: relevant variables $S \subseteq X$, DNNF $f$
output: DNNF with variables $X \setminus S$ existentially quantified
if $f$ is a Literal $l$:
    if $\text{VAR}(l) \in S$:
        return $f$
    else
        return true
else if $f$ is a conjunction $a_1 \wedge \cdots \wedge a_n$:
    return $\text{FORGET}(a_1) \wedge \ldots \wedge \text{FORGET}(a_n)$
else if $f$ is a disjunction $a_1 \vee \cdots \vee a_n$:
    return $\text{FORGET}(a_1) \vee \ldots \vee \text{FORGET}(a_n)$
```

Enumeration of assignments of DNNFs is shown in Algorithm 4 [Darwiche, 2000], where each assignment is represented as a set of literals, and $\times$ is a Cartesian product on them:

$$\{N_1, \ldots, N_n\} \times \{M_1, \ldots, M_m\} = \{N_1 \cup M_1, \ldots, N_n \cup M_m\}.$$

The complexity of enumerating the models of a smooth DNNF f is $O(mn^2)$, where m is the size of f and $n = |\text{MODELS}(f)|$ [Darwiche, 1998].

Algorithm 4 MODELS – enumerating assignments of DNNF

```
input: smooth DNNF $f$
output: (complete) valid assignments of $f$, as sets of literals
if $f$ is a Literal $l$:
    return $\{\{l\}\}$;
else if $f$ is a conjunction $a_1 \wedge \cdots \wedge a_n$:
    return $\text{MODELS}(a_1) \times \cdots \times \text{MODELS}(a_n)$;
else if $f$ is a disjunction $a_1 \vee \cdots \vee a_n$:
    return $\text{MODELS}(a_1) \cup \cdots \cup \text{MODELS}(a_n)$.
```

The overall process of using DNNF is shown in Algorithm 5. DNNF for the car example is shown on Figure 1a. DNNF with variable *engine* forgotten is shown on Figure 1b, and its valid partial assignments can be found in Table 2.

Algorithm 5 Knowledge compilation based approach

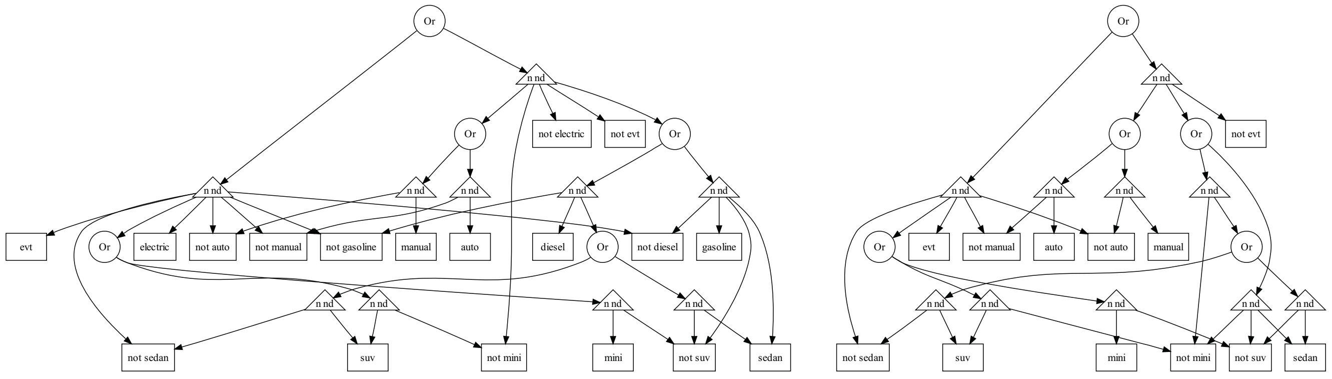
```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
output: valid partial assignments
$f_1 \leftarrow \text{COMPILE}(\mathcal{P})$
$f_2 \leftarrow \text{FORGET}(S, f_1)$ /* see Algorithm 3 */
$m \leftarrow \text{MODELS}(f_2)$ /* see Algorithm 4 */
return $m$
```

Polynomial time guarantee for assignments enumeration operation is an advantage of DNNF. However, the compilation time of arbitrary constraints into DNNF is in general exponential. When the data changes rarely, this time is amortized among multiple queries, but when the data changes very often, this off-line stage does not pays off.

DNNF have an important advantage: if it is smooth and *deterministic*, it can be used to count the assignments [Darwiche, 2000]. An NNF formula a is *deterministic* if for every disjunction $a = a_1 \vee \cdots \vee a_n$, every pair of disjuncts in a is logically inconsistent; that is, $a_i \wedge a_j \models false$ for every $i \neq j$. Unfortunately, operation FORGET does not preserve determinism, and counting in such case will give overestimated answer. However, even overestimated answer can be useful in some cases. It is also possible to recompile the resulting DNNF again into a deterministic one. Some practical applications of counting for configuration using DNNF were considered in [Kübler *et al.*, 2010].

5 Experimental results

We analyzed the data from two automotive companies: three datasets from the first company, and one dataset from the sec-



(a) Full DNNF.

(b) DNNF with variable *engine* existentially quantified.

Figure 1: DNNFs for the car example.

ond, denoted as A, B, C and D, respectively. Details about datasets are presented in Table 4.

We implemented Algorithms 2 and 1 on top of Sat4j solver [Le Berre and Parrain, 2010].

A knowledge compilation tool was developed based on BDD package BuDDy [Lind-Nielsen, 2002]. The pre-ordering algorithms from [Narodytska and Walsh, 2007] were implemented for sorting variables and restrictions, using inflation parameter $r = 1.5$ in the clustering step. A simplified version of the MCL clustering algorithm [van Dongen, 2000] was used, skipping the truncation heuristics and the sparse matrix multiplication tools. No post-ordering of the variables was included. The tool was able to handle only the smallest dataset.

Another attempt to use knowledge compilation involved c2d compiler [Darwiche, 2004] that compiles propositional formulas to deterministic DNNF. Algorithms 3 (FORGET) and 4 (MODELS) were implemented to work with the DNNF output of c2d. Using another DNNF compiler sharpSAT [Muise *et al.*, 2010] resulted in segmentation faults on some of the datasets, and its debugging is underway.

Sat4j and c2d require the input to be in Conjunctive Normal Form (CNF), that is it have to be a conjunction of clauses, and each clause is a disjunction of literals. Each literal is either a positive or negative atomic proposition. Constraints were converted to CNF using Tseitin encoding [Tseitin, 1968].

Two times were measured. The first time measured was preprocessing or off-line time. This included, for example, DNNF compilation, and initial constraint propagation. The results are presented in Table 5. BDD-based implementation was not able to complete the compilation process of larger instances. c2d compiler ran out of 2 GB memory limit (it is available only as a 32 bit application) compiling the largest dataset A.

The second time measured was the on-line time, or the time to actually compute the valid partial configurations for one given scope, while utilizing results from the off-line phase. The results are presented in Table 6. SAT-based solution is very robust on the datasets, even without having theoretical guarantees on running times. The BDD-based solution, when the BDD was successfully built, is the fastest. The reason why DNNF-based method appears to be slow could be a non-optimal implementation of Algorithms 3 and 4.

Table 4: Problem properties.

	A	B	C	D
Variables	511	446	92	217
Domain size, average	6.3	2.4	6.1	3.3
Domain size, max	108	82	75	59
# of assignments	10^{150}	10^{87}	10^{55}	10^{85}
# of valid assignments	10^{124}	10^{57}	10^{49}	10^{33}
CNF clauses	65183	1121	341	9010
DNNF nodes	n/a	5071	5009	528583
Partial scope, variables	6	17	3	8
# of valid partial assignments	200	13770	25	382

6 Conclusions

In this paper we looked at the problem of computing allowed partial combinations, which is important for engineers working with product development. We presented several algorithms, two of which are suitable for SAT-solvers, and one that is based on DNNF. Our experiments showed that SAT-based implementation can handle large datasets from automotive industry quite efficiently. Preliminary experiments with knowledge compilation tools showed that available DNNF compilers cannot handle the largest dataset within the memory and time limits. However, DNNF-based method has

Table 5: Time for compilation/initial clause learning, seconds.

	A	B	C	D
Sat4j, Alg 1	2	2	1	2
BuDDy	timeout	timeout	40	timeout
c2d	out-of-mem	240	2	20

Table 6: Time to compute valid partial assignments (FORGET+MODELS), seconds.

	A	B	C	D
Sat4j, Alg 1	10	29	0.12	3
BuDDy	n/a	n/a	0.01	n/a
DNNF from c2d	n/a	681*	0.15*	22*

*Based on own, unoptimized implementation.

the advantages of polynomial time guarantee on the on-line phase, and the ability to count the assignments when DNNF is determenistic.

Acknowledgements

This work was carried out at the Wingquist Laboratory VINN Excellence Centre within the Area of Advance — Production at Chalmers, supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA). The support is gratefully acknowledged.

References

[Amilhastre *et al.*, 2002] Jérôme Amilhastre, Helene Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs — Application to configuration. *Artificial Intelligence*, 135:199–234, 2002.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17(1):229–264, 2002.

[Darwiche, 1998] Adnan Darwiche. Model-Based Diagnosis using Structured System Descriptions. *Journal of Arti cial Intelligence Research*, 8:165–222, 1998.

[Darwiche, 2000] Adnan Darwiche. On the tractable counting of theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, 2000.

[Darwiche, 2001] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, July 2001.

[Darwiche, 2004] Adnan Darwiche. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *ECAI 2004*, 2004.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Dechter and Dechter, 1988] Rina Dechter and Avi Dechter. Belief maintenance in Dynamic Constraint Networks. In *AAAI-88*, pages 37–42, 1988.

[Een and Sörensson, 2003] Niklas Een and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[Een and Sörensson, 2004] Niklas Een and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919/2004:502–518, 2004.

[Gebser *et al.*, 2009] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected Boolean search problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86, 2009.

[Gottlob *et al.*, 2007] Georg Gottlob, Gianluigi Greco, and Toni Mancini. Conditional Constraint Satisfaction: Logical Foundations and Complexity. In Manuela M. Veloso, editor, *IJCAI-2007*, pages 88–93, Hyderabad, India, January 2007.

[Hadzic *et al.*, 2004] Tarik Hadzic, Sathiamoorthy Subbarayan, R.M. Jensen, Henrik Reif Andersen, J. Mø ller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *PETO conference*, 2004.

[Janota, 2008] Mikolas Janota. Do SAT solvers make good configurators? *12th International Software Product Line Conference. First Workshop on Analyses of Software Product Lines*, pages 1–5, 2008.

[Janota, 2010] Mikolas Janota. *SAT Solving in Interactive Configuration (PhD thesis)*. PhD thesis, University College Dublin, 2010.

[Kübler *et al.*, 2010] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. Model Counting in Product Configuration. *Electronic Proceedings in Theoretical Computer Science*, 29(LoCoCo):44–53, July 2010.

[Küchlin and Sinz, 2000] Wolfgang Küchlin and Carsten Sinz. Proving Consistency Assertions for Automotive Product Data Management. *Journal of Automated Reasoning*, 24(1):145–163, February 2000.

[Le Berre and Parrain, 2010] Daniel Le Berre and Anne Parrain. The Sat4j library , release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[Lind-Nielsen, 2002] Jø rn Lind-Nielsen. BuDDy: A BDD package. http://buddy.sourceforge.net, 2002.

[Mittal and Falkenhainer, 1990] Sanjay Mittal and Brian Falkenhainer. Dynamic Constraint Satisfaction Problems. In *AAAI-90*, pages 25–32, 1990.

[Muise *et al.*, 2010] Christian Muise, Sheila McIlraith, J.Christopher Beck, and Eric Hsu. Fast d-DNNF Compilation with sharpSAT. In *AAAI 2010*, pages 54–60, 2010.

[Narodytska and Walsh, 2007] Nina Narodytska and Toby Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 149–154, Hyderabad, India, 2007. Morgan Kaufmann Publishers Inc.

[Norgaard *et al.*, 2009] Andreas Hau Norgaard, Morten Riiskjaer Boysen, Rune Moller Jensen, and Peter Tiedemann. Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration. In *Proceedings of the 21st International Joint Conferences on Artificial Intelligence(IJCAI-09) Workshop on Configuration*, 2009.

[Robinson, 1965] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sabin and Freuder, 1998] Mihaela Sabin and Eugene C. Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Proceeding of Constraint Programming (CP-98)*, 1998.

[Sinz *et al.*, 2003] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(01):75–97, August 2003.

[Sinz *et al.*, 2006] Carsten Sinz, Wolfgang Küchlin, Dieter Feichtinger, and Georg Görtler. Checking Consistency and Completeness of On-Line Product Manuals. *Journal of Automated Reasoning*, 37(1):45–66, August 2006.

[Soininen and Gelle, 1999] Timo Soininen and Esther Gelle. Dynamic Constraint Satisfaction in Configuration. In *AAAI-99, Workshop on Configuration*, pages 95–100, 1999.

[Subbarayan and Andersen, 2005] Sathiamoorthy Subbarayan and Henrik Reif Andersen. Linear Functions for Interactive Configuration Using Join Matching and CSP Tree Decomposition. In *Configuration Workshop at IJCAI'05*, pages 7–12, 2005.

[Subbarayan *et al.*, 2007] Sathiamoorthy Subbarayan, Lucas Bordeaux, and Youssef Hamadi. Knowledge Compilation Properties of Tree-of-BDDs. In *AAAI 2007*, pages 502–507, 2007.

[Subbarayan, 2005] Sathiamoorthy Subbarayan. Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 3524/2005:351–365, 2005.

[Tseitin, 1968] Gregory S. Tseitin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*, pages 234–259. Steklov Mathematical Institute, 1968.

[van Dongen, 2000] Stijn van Dongen. A cluster algorithm for graphs. Technical Report INS-R0010., 2000.

Incremental prediction of configurator input values based on association rules – A case study

Dietmar Jannach and Lukas Kalabis
TU Dortmund
Dortmund, Germany
dietmar.jannach@tu-dortmund.de, lukas.kalabis@tu-dortmund.de

Abstract

In many configurator applications, the user is required to specify a multitude of configuration options interactively. One way of making the configuration process more convenient for the user is to pre-fill the input or selection fields of the user interface with appropriate defaults. Possible strategies to determine default values for example include the selection based on domain knowledge or the usage of statistics.

In this paper we analyze whether or not the dynamic selection of defaults based on automatically determined association rules can help to predict the most probable next input value in an interactive and incremental configuration process. We base our analysis on the data obtained with a real-world configurator application. The value of choosing defaults more intelligently is determined by measuring the number of correctly predicted inputs in the configuration process and by comparing this number to a default strategy based on simple value frequencies.

1 Introduction

In many industrial sectors, the products on the market can be customized to a customer's individual needs in a variety of ways. Accordingly, the interactive preference elicitation and product configuration process can become time-consuming and cumbersome, because the user of the configurator application is often required to enter input or make selections for several dozens of parameters.

Providing suitable default inputs or default selections for the individual options represents one common way to make the configuration process more convenient for the user. How the system chooses the pre-set default value can be based on different strategies. One simple strategy for input fields with a predefined set of options could be simply selecting the first value of the list. Alternatively, one could use the value that was chosen most frequently in the past (also by other users). In some systems – such as the ADVISOR SUITE sales advisory framework [Felfernig *et al.*, 2006] – the selection of the default values is based on domain knowledge. Beside the static definition of defaults, this framework also supports the definition of *rules* that determine the proposed default value for a field based on the inputs already made in the current session.

Our own anecdotal experiences with using predefined default values provided by the domain expert in the domain of interactive advisory systems however showed that such static rules can have different limitations. First, domain experts often define what should be the default selection for an input field merely based on gut feeling and intuition. Second, in some domains, the most appropriate default value changes over time, for example due to technological advances or a changing marketing strategy. The rules determining the defaults in the configurator applications are however not always updated accordingly.

In this work we propose to use association rules [Agrawal and Srikant, 1994], which can be automatically learned from past configuration sessions, to dynamically choose the most appropriate default values. We evaluate the value of applying this self-adapting default selection strategy by counting the number of correct and wrong predictions when replaying past interactive configuration sessions and comparing our strategy to a statistics-based baseline strategy. The analysis of the approach is based on a real-world configuration data base.

2 Mining input value patterns

Figure 1 shows a schematic but typical fragment of a user interface for a configurator, in which the user of the system – in our case a sales representative – incrementally enters the preferences of the customer. In our real-world database from the roofing industry, on average more than two dozens of such parameters have to be entered during the configuration process. The database comprises more than 9,000 past roofing configurations.

The goal of our work is to try to detect patterns co-occurring input values in these past configurations and exploit these patterns to predict the next input values in the interactive configuration process.

Association rules have been traditionally used in data mining scenarios and in particular for shopping basket analysis. With the help of algorithms such as Apriori [Agrawal and Srikant, 1994], rules of the following form can be extracted from past buying transactions:

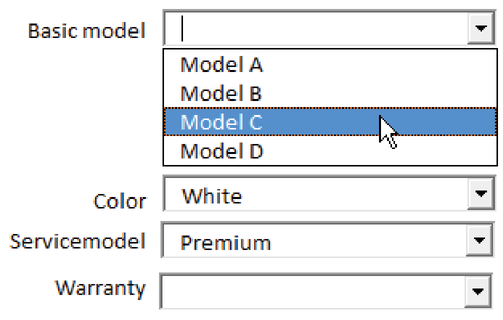


Figure 1: Schematic user interface fragment.

$\{milk, bread\} \Rightarrow \{butter\}$
[support=60%, confidence=80%]

The rule states that whenever customers purchase milk and bread, they also buy butter. Support and confidence are measures for the strength or quality of a rule. The support metric describes in how many of all existing transactions the *itemset* {milk,bread,butter} appeared. The confidence measure corresponds to the percentage of transactions in which milk and bread appeared and where butter (the right hand side of the rule) was also part of the transaction.

In our problem setting, a completed configuration corresponds to a flat list of assignments of values to the more than two dozen input variables, that is, our configuration problem is relatively simple when compared, for example, to classical component port configuration models [Mittal and Frayman, 1989]. Applying association rule mining algorithms such as Apriori (as used in our work) or FP-Growth [Han *et al.*, 2000] is therefore straightforward and the goal of the mining process is to detect patterns such as

$\{Basic\text{-}Model = A, Color = White\} \Rightarrow \{Warranty = 3yr.\}$

As an overall result of the mining process, a set of such association rules can be determined. Usually, a minimum support value has to be set in order to only take significant patterns into account.

3 Default selection schemes

In our evaluation, we compared three schemes for determining the default value: (A) take a random value (the first in the list of options); (B) the most frequent value from the past transactions is taken as a default; (C) the selection is based on association rules.

For scheme C, a “sliding window” strategy was applied. Note that we assume in our application that we have a strict order in which the inputs are entered (from top to the bottom). In scheme C, the defaults therefore depend on the previous inputs. If we, for example look for a value for the warranty field, we look for rules that have the previous inputs (such as the color) in the left hand side of the rule. The window size describes how many of the last n inputs we take into account. Taking all previous inputs into account might be impractical as the set of detected association rules might not contain rules that have 20 or more inputs on the left hand side. When multiple rules are in principle applicable (but suggest different right hand side values), we choose the rule with the highest confidence value. In combination with the sliding window strategy, we also apply a relaxation strategy in case we cannot find a matching rule. If, for example, no rule with the left hand side {Basic-Model = A, Color = White} can be found in a window of size 2, we calculate all subsets of the left hand side and take the rule with the highest confidence value.

Overall, in contrast to default selection B where all values can be set at the beginning, in scheme C, the defaults are determined dynamically based on the previous inputs.

4 Evaluation

Metric. As an evaluation metric, we count the number of clicks that are required to configure the customized product variant. Note that we have one full default configuration for scheme A and one for scheme B. To measure the number of required clicks, we iterate through all 9,000 past transactions and check for each transactions how many of the input field values are different from this default configuration. The configuration effort thus corresponds to the average number of values that have to be changed.

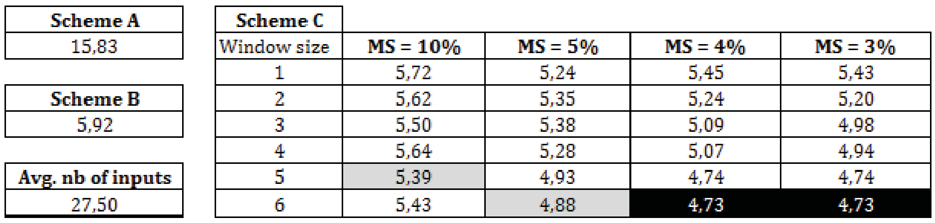
For scheme C, we “replay” the configuration process for each of the past configuration sessions and predict the input field value one after the other based on the association rules. In case the prediction of the next input was correct, we move the sliding window forward and predict the next input. In case the prediction was wrong, we increase the counter of required clicks, correct the input value to the one given in the current past transaction and proceed with predicting the next input field.

Results. Due to the fact that the choice of the basic roof model, which has to be done as a first step, considerably influences the available choice for the rest of the configuration process, we have learned a set of association rules for each of these basic models. In addition, we have experimented with different window sizes as well as minimum support values. Figure 2 exemplarily shows the results for one of the basic models.

On average, 27,5 input values were set for a configuration of this model type. Using the simplistic default selection scheme A (pick the first value in the list), on average a bit more than 15 values have to be set (changed) manually by the user[1]. However, if we apply scheme B (pick the most frequent value), a very strong improvement can be observed and only about 6 of the 27 values were not properly predicted, which strongly indicates that there are some configurations options, which are very popular and that there is a long tail of configurations which is very seldom sold.

Regarding the dynamic prediction scheme C, we experimented with different settings and in particular varied the parameters *window size* and *minimum support (MS)*, see Figure 2. With respect to the window size, we can observe that a larger window size, which in turn means that we learn longer rules, helps to improve the predictive accuracy of our rule-based approach. The best results in our experiments were achieved with window sizes 5 and 6. Further tests showed

[1]The median number of input values is 5.



Scheme A
15,83

Scheme B
5,92

Avg. nb of inputs
27,50

Scheme C				
Window size	MS = 10%	MS = 5%	MS = 4%	MS = 3%
1	5,72	5,24	5,45	5,43
2	5,62	5,35	5,24	5,20
3	5,50	5,38	5,09	4,98
4	5,64	5,28	5,07	4,94
5	5,39	4,93	4,74	4,74
6	5,43	4,88	4,73	4,73

Figure 2: Results for one representative basic model.

that beyond this window size no further improvements could be observed. The time required for the (offline) rule mining process however significantly goes up when the maximum length of the rules to be learned is increased.

With respect to the MS value, lower threshold values lead to better results and the best predictions were achieved with MS values at 3% and 4%. Lower MS values mean that also rules for "rare combinations" are learned and included in the rule base. Again, further decreasing the MS value leads to marginal improvements at the price of a much larger rule base. Overall, we can see that the accuracy consistently increases when the MS value is lowered. For the window sizes, in contrast, we could observe that further increasing this parameter does not always lead to better results.

For the particular basic model for which we show the results in Figure 2, we can see an overall improvement from 5,92 to 4,73 required clicks. At first glance, this might not look too impressive. Note however, that the good results that were achieved with the comparably simple scheme B are due to the very unbalanced distribution of the available input values in the past configurations. Figure 3 shows a typical example for a "yes/no" question. For three out of four user input fields, such a lopsided distribution could be observed.

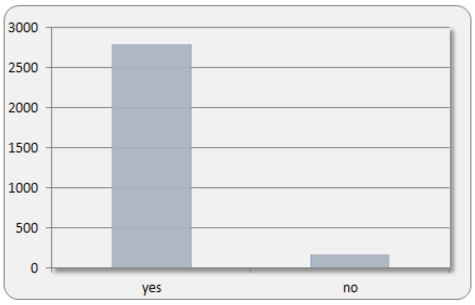


Figure 3: Value distribution for field "standard height (y/n)".

Across all basic models, an average reduction of required clicks of 11.88% was observed. A first analysis showed that the achieved improvements do not so much depend on the number of available training samples but rather on the diversity of the actually configurations.

Running times. The time needed for the generation of input value predictions based on the previously learned association rules can be considered to be suitable for an interactive configuration scenario. In all our experiments, the computation of input values for *all* fields took at most 10 seconds (that is, below 0.5 seconds for each field) even for the largest window sizes considered in the experiments. The times needed for the offline model-building phase strongly depend on the minimum support size. While at the 10% level model-building takes a few seconds, the calculations at the 3% level can take a few hours on a standard desktop PC.

5 Previous works

In [Ardissono *et al.*, 2002], an advanced approach to personalizing the preference elicitation process in a configurator application was proposed. In particular, their system exploits stereotypical user modeling techniques to assess the user's preferences and properties throughout the interaction process. In contrast to our work, in their approach the goal is also to find *personalized* defaults which depend on the individual behavior of the user. Another approach to personalize the interaction process with the goal to reduce the complexity of the overall process for the end user based on user profiles and personalized recommendations was proposed in [Stegmann *et al.*, 2003] and [Stegmann *et al.*, 2006].

Currently, personalization of the default proposal process is beyond the scope of our work but could be relatively easy implemented by learning user-specific rule models for situations, in which enough data is available for personalization.

Recently, in [Tiihonen and Felfernig, 2010] and [Felfernig *et al.*, 2010] a knowledge-based approach to personalizing the user interaction process for configurator applications was presented. Beside the automatic generation of "repair proposals" for situations in which the customer requirements are inconsistent (as also discussed in [Felfernig *et al.*, 2001]), the authors propose different (probabilistic and statistics-based) techniques to determine suitable feature values as proposed earlier already in [Cöster *et al.*, 2002]. The goal of their work is similar to ours, although different techniques are employed and for example metrics for measuring the "distance" between configurations as well as feature weights are exploited. Unfortunately, no empirical evaluation of their proposal was yet done. However, [Felfernig *et al.*, 2010] also consider the question that the proposed feature values have to be consistent with the configuration knowledge base and the current, partial configuration. In our current work, this aspect was not considered yet. One strategy to deal with this aspect could be to systematically try to apply different association rules (ordered by their confidence) and check whether the configuration is still consistent after rule application.

In [Geneste and Ruet, 2001], finally, a Case-Based Reasoning (CBR) approach to reduce the complexity of the interaction process was proposed. The main idea is not to start configurations from scratch, but to reuse and adapt past configurations. The main tasks are therefore to find past similar cases based on a similarity metric and a search algorithm, determine possible adaptations (the adaptation domain) and then guide the user through the adaptation process using constraint satisfaction techniques. Beside the goal of making the interaction process more efficient, one similarity between our work and the work of [Geneste and Ruet, 2001] is that we rely on past configurations to steer the interaction process. However, in our work we assume an incremental process in which configurations are developed and refined step by step. The consideration of the consistency checks before the default proposal process as done in [Geneste and Ruet, 2001] is currently not supported in our approach.

6 Summary and outlook

In this work, we have analyzed how association rules mined from past configurations can be exploited to predict input values for interactive configuration processes and can thus help to make the usage of such systems more comfortable in case the user has to configure a multitude of options. The evaluation on a real-world data set showed that a measurable reduction in required interactions can be achieved when compared to a simpler statistics-based approach or even in cases when we have a market which is dominated by a few very popular configurations.

Among other aspects, our future work includes the analysis of the algorithm on other real-world datasets and the application of other techniques from data mining and machine learning for input value prediction in the configuration domain. Beside that, our goal is to evaluate prediction strategies in which the elements contained in the sliding window not only depend on the chronological order of the inputs but on some relevance weight, assuming that individual inputs are more predictive than others in the configuration process. In addition, future work could also consider the question if there are characteristics of the configuration problem (such as the domain sizes of the variables) which can help us to automatically determine appropriate values for the minimum support threshold.

References

[Agrawal and Srikant, 1994] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of 20th Intl. Conference on Very Large Data Bases*, pages 487–499, Santiago de Chile, Chile, 1994.

[Ardissono *et al.*, 2002] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, M. Meyer, G. Petrone, R. Schäfer, W. Schütz, and M. Zanker. Personalising on-line configuration of products and services. In *Proc. of the 15th European Conference on Artificial Intelligence*, pages 225–229, Lyon, France, 2002.

[Cöster *et al.*, 2002] Rickard Cöster, Andreas Gustavsson, Tomas Olsson, sa Rudstrm, and Asa Rudstrm. Enhancing web-based configuration with recommendations and cluster-based help. In *In Proceedings of the AH'2002 Workshop on Recommendation and Personalization in eCommerce*, pages 30–39, Malaga, Spain, 2002.

[Felfernig *et al.*, 2001] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Intelligent support for interactive configuration of mass-customized products. In *Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 746–756, 2001.

[Felfernig *et al.*, 2006] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. An integrated environment for the development of knowledge-based recommender applications. *International Journal of Electronic Commerce*, 11:11–34, 2006.

[Felfernig *et al.*, 2010] Alexander Felfernig, Monika Mandl, Juha Tiihonen, Monika Schubert, and Gerhard Leitner. Personalized user interfaces for product configuration. In *Proceedings of the 15th International Conference on Intelligent User Interfaces*, pages 317–320, Hong Kong, China, 2010.

[Geneste and Ruet, 2001] L. Geneste and M. Ruet. Experience based configuration. In *Proceedings of the Configuration Workshop at IJCAI'01*, Seattle, USA, 2001.

[Han *et al.*, 2000] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12, Dallas, Texas, 2000.

[Mittal and Frayman, 1989] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proc. 11th Intl. Joint Conference on Artificial Intelligence*, pages 1395–1401, Detroit, Michigan, 1989.

[Stegmann *et al.*, 2003] Rosmary Stegmann, Michael Koch, Martin Lacher, Thomas Leckner, and Volker Renneberg. Generating personalized recommendations in a model-based product configurator system. In *Workshop on Configuration at IJCAI'03*, Acapulco, Mexico, 2003.

[Stegmann *et al.*, 2006] Rosmary Stegmann, Thomas Leckner, Michael Koch, and Johann Schlichter. Customer support for the web-based configuration of individualised products. *International Journal of Mass Customization*, 1(2):195–217, 2006.

[Tiihonen and Felfernig, 2010] Juha Tiihonen and Alexander Felfernig. Towards recommending configurable offerings. *International Journal of Mass Customization*, 3(4):389–406, 2010.

When to use what: Structuralization, Specialization, Instantiation, Metaization - Some Hints for Knowledge Engineers

Lothar Hotz, Stephanie von Riegen
Hamburger Informatik Technology Center, Department Informatik, University of Hamburg, Germany
e-mail: {hotz, svriegen}@informatik.uni-hamburg.de.

Abstract

In knowledge engineering, ontology creation, and especially in knowledge-based configuration often used relations are: aggregate relations (`has-parts`, here called structural relations), specialization relation (`is-a`), and instantiation (`instance-of`). A combination of the later is called metaization, which denotes the use of multiple instantiation layers. In this paper, we give examples and use-hints for these relations especially from the configuration point of view.

1 Introduction

For configuration-based inference tasks, like constructing a description of a specific car periphery system [Hotz *et al.*, 2006] or drive systems [Ranze *et al.*, 2002], the knowledge of a certain domain is represented with a *knowledge-modeling language* (KML) which again is interpreted, because of a defined semantic, through a *knowledge-based system* or *configurator* [Arlt *et al.*, 1999; Günter and Hotz, 1999]. Examples for those KMLs are the Web-Ontology Language (OWL) and the Component Description Language (CDL); further languages are e.g. described in [van Harmelen *et al.*, 2007]. KMLs typically provide *concepts* or *classes* gathering all properties, a certain set of domain objects has, under a unique name. With concepts and instances a strict separation into two layers is made: a *domain model* (or *ontology*) which covers the knowledge of a certain domain (abbr. $layer^D$) and a *system model* (or *configuration*) which covers the knowledge of a concrete system or product of the domain (abbr. $layer^S$).

Properties of a concept that map to primitive data types, like intervals, values sets (enumerations), or constant values, are called *parameters*. Properties that map to other concepts or to instances are called *relations*. KMLs provide structural, specialization, and instantiation as typical relations. A specialization relation relates a *superconcept* to a *subconcept*, where the later inherits the properties of the first. This relation (also called `is-a` relation) forms a *specialization hierarchy* or *lattice*, if a concept has more than one superconcept. The structural relation is given between a concept c and several other concepts r, which are called *relative concepts*. With structural relations a compositional hierarchy based on the `has-parts` relation can be modeled as well as other structural relationships. *Instances* are instantiations of concepts and represent concrete domain objects (`instance-of`).

Additionally to concepts, instances, and their relations, constraints provide model facilities to express n-ary relationships between properties of concepts [John, 2002; Gelle and Faltings, 2003]. Constraints can represent restrictions between properties like arithmetic relations or restrictions on structural relations (e.g. ensuring existence of certain instances). Especially constraints on structural relations extend typical constraint technology, which is based on primitive data types like numbers or strings [Hotz, 2009b].

In this paper, the use of structuralization, specialization, and instantiation are discussed. Even those relations are quite well-known they are sometimes confounded. Furthermore, when used with more than the two mentioned domain and system layers (see [Asikainen and Männistö, 2010; Hotz, 2009a]) the instantiation relation is multiply applied, which leads to new modeling layers and thus, probably to modeling difficulties. The creation of such multiple layers is called *metaization* [Strahringer, 1998].

In the following, we first consider all relations in more depth and give examples of their use (Section 2 and Section 3). Afterward, we discuss metaization and its use for configuration (Section 4). We end with a short discussion on related work and a conclusion.

2 Structuralization

As already elaborated in [Hotz, 2009a] configuration can be considered as model construction, because a description of a certain system (a configuration) is constructed by a configurator. Furthermore, [Hotz, 2009a] emphasizes to consider the `has-parts` relation as a `has` relation that may be used for diverse aspects like `has-Realizations` or `has-Features` in software-intensive systems. For the typical use, a structural relation represents a compositional relation. In this case, between c and its relatives r, c denotes the aggregates and r denotes the parts. The underlying structural relation is used by configurators to construct the description and thus are the motor of configuration. Depending on what instances (of c or r) exist first, instances of the related concepts are created; e.g. this enables reasoning from the aggregate to the parts or contrariwise, from the parts to the aggregate. This semantic holds for every structural relation. Thus, introducing several

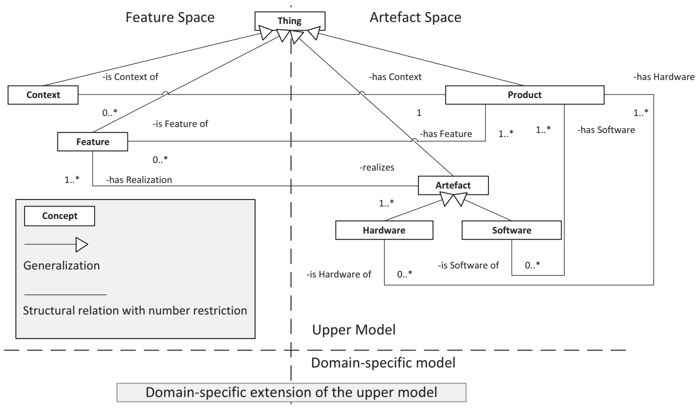


Figure 1: Extract from an upper-model for modeling software-intensive systems.

structural relations enables the use of adequate domain names like `has-Features` or `has-Realizations`, and thus to facilitate maintenance.

Figure 1 pictures an *upper-model for software-intensive systems* (UMSiS, [Hotz *et al.*, 2006]). It defines four asset types (features, context, hardware and software artefacts) which are common to most application domains of software-intensive systems (SiS). A product, i.e. the result of the product derivation, contains software and hardware artefacts as parts, these together realize particular features. Several structural relations are depicted, like `has-Realizations` and `has-Feature`. When using the upper-model for a specific domain, the UMSiS is extended with domain-specific knowledge about hardware and software artefacts, the existing features, relevant context aspects, etc. In the example above, the concepts are organized in different *spaces*. Each space represents a specific aspect of the domain and thus each configured product should have those aspects. Figure 1 provides the example of the feature and artefact aspects in the domain of software-intensive systems. Thus, *spaces* separate concepts of one layer. Through this grouping of concepts of one layer the configuration model is easier to manage for a knowledge engineer. Furthermore, concepts of different spaces are connected by a structural relation. This ensures that a configured product finally contains all modeled aspects. In contrast to this, in Section 3 we will see, how the instantiation relation separates concepts and instances on different *layers*.

3 Specialization vs. Instantiation

A concept describes a set of instances. The specialization relation (or subsumption or `is-a` relation) between two concepts c and s describes a subset relation, i.e. the set of instances of concept c is a subset of the set of instances of its superconcept s (see also [Brachman, 1983]). Or, as defined in `ontogenesis.knowledgeblog.org/699`: "c is-a s iff: given any i that instantiates c, i instantiates s". An instance of a class c is always an instance of each superclass s of c. We consider this aspect as the hinting characteristic for knowledge engineers: During knowledge modeling one can try to make a specialization between two domain aspects and test this characteristic. Thus, it is tested if an instance of c is also reasonably an instance of s. If it is false the knowledge engineer must not use a specialization but e.g. instantiation, because c and s are probably on different layers.

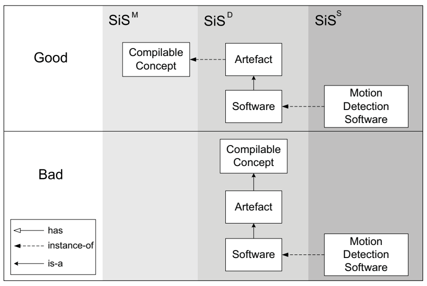


Figure 2: Good and bad use of specialization and instantiation in software-intensive systems.

An example for this situation is shown in Figure 2; it presents the confounded usage of specialization and instantiation relations in the aforesaid modeling of software-intensive systems domain (SiS) (Section 2). The system model layer (SiS^S) covers specific individuals, here the `Motion Detection Software`. This object is an instance of `Software` (SiS^D) but no instance of `Compilable Concept`. `Compilable Concept` denotes a specific kind of concept (thus a specific description of instances) that can be compiled. Thus, in the "bad" use, `Motion Detection Software` is incorrectly considered as a concept, i.e. as a *description* of instances. Instead it is an *instance* (here of `Software`), thus a specific domain object.

When a concept s is specialized to c all properties of s are inherited by c. Furthermore, the properties defined in c that are also defined in s must have more special property values than those in s. For checking this strict specialization, the subset semantic is defined for all primitive data types and the structural relation [Hotz, 2009b]. Thus, the specialization relation is used for structuring the space of needed concepts for representing domain knowledge.

By the time a concept is instantiated, properties of the created instance are initialized by values or value ranges specified in the concept. Thus, the concept determines the structure of the instance (i.e. the properties). In this sense, a concept says something about its instances, i.e. a concept is on a different layer than its instances. By reducing the value ranges according to user decisions or constraint computations the configurator subsequently creates a specific description consisting of instances.

4 Metaization

For structuralization and specialization, the involved concepts are on one layer. However, for instantiation and metaization they are on different layers. By instantiating a concept one instance is created, i.e. a step from a set of instances to an individual element of this set is performed. If this step is cascadized, a concept c can be considered as an instance of another concept c_m, i.e. a step from a set of concepts to one specific concept is performed. The concept c_m

is on a further layer. Figure 3 demonstrates this situation. The concept `Feature` is an instance of `Abstract Concept` which is a specialization of `concept-m`. All concepts on the metalayer CDL^M represent the modeling facilities of CDL, describing the concepts and relations of CDL. Concept `Artefact` is a typical CDL concept (it is an instance of `concept-m`) and the relation `has-Realization` is a structural relation (represented by instantiating the CDL^M concept `relation-descriptor-m`) ([Hotz, 2009a] for more on modeling CDL with CDL). CDL^M represents all what is known about CDL^D, i.e. concepts and relations.

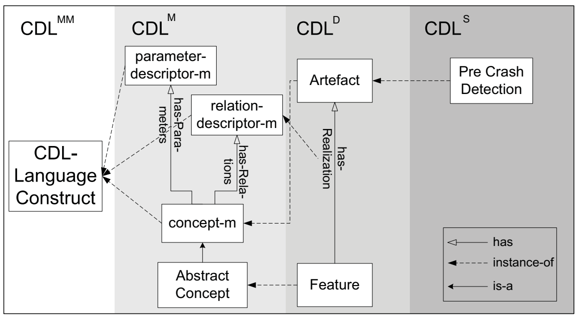


Figure 3: Modeling the Component Description Language.

Figure 4 presents the enhancement of Figure 1 by the additional layer SiS^M. SiS^M describes the SiS^D layer concepts `Feature`, `Software`, and `Hardware` as `Abstract Concept`, `Compilable Concept`, and `Manufacturable Concept`, respectively. Thus, it is a domain dependent extensions of CDL^M.

By doing so, constraints on concepts of SiS^D can be expressed. For example a constraint represents that each feature should be realizable by an artefact. A constraint can check that each feature (a subconcept of `Feature`) should have a structural relation `has-Realization` to a subconcept of `Artefact`. These kinds of constraints may be hard to define, because typically they are not related to one specific concept but to several. Still, such constraints are usually part of some modeling guidelines.[1]

In [Hotz and von Riegen, 2010b; 2010a], we introduce the Reasoning Driven Architecture (RDA) that allows the implementation of metalayers by using a configuration system on each layer. By doing so, each layer can be seen as a knowledge-based system that says something about the layer below. In the case of RDA, SiS^D contains the knowledge of domain objects, which again are represented on SiS^S. By introducing the metalayer SiS^M, knowledge about knowledge is made explicit, i.e. knowledge about the knowledge of domain objects. This enables the use of reasoning techniques for each layer, not only for the domain and system layers as it is typically the case in knowledge-based systems. The central point of such an implementation is a mapping between instances on one layer to concepts on the next lower layer (see Figure 5 and [Hotz and von Riegen, 2010a] for a mapping for CDL and [Tran *et al.*, 2008] for mapping for OWL or [Bateman *et al.*, 2009]). Metalayers allow for handling (meta) tasks and services. For example, [Tran *et al.*, 2008] proposes to provide statistics about the model (e.g. retrieve all knowledge elements about Pre Crash Detection). With a metalayer like provided in Figure 4, during configuration of a software-intensive system one can call different external mechanisms for each specific metaconcept. For example, if an instance of an instance of `Compilable Concept` (e.g. an instance of `Software`) is configured, an external compiler mechanism can be called to realize the software. If an instance of an instance of `Manufacturable Concept` is configured, the warehouse can be contacted to check if the needed parts for the manufacturing are present. Thus, through the metalayer the actual configuration of a product can be monitored and reasoning on the configuration process can be processed.

[1]The SiS^{MM} layer has been omitted because no modeling is required here.

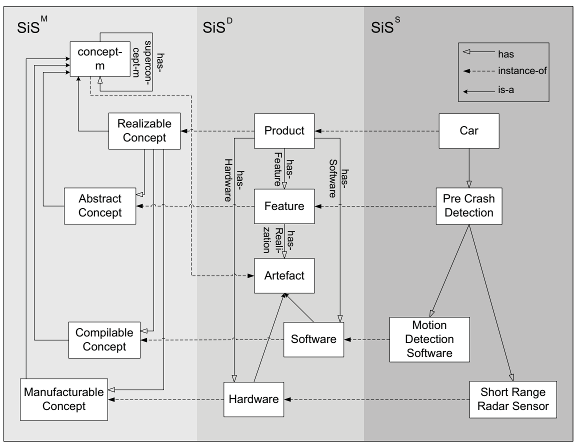


Figure 4: Modeling software-intensive systems.

5 Related Work

The modeling approach, especially metaization [Strahringer, 1998], has similarities to the Model-Driven Architecture [Miller and Mukerji, 2003; Kühne, 2006; Atkinson and Kühne, 2003; Hotz and von Riegen, 2010a], because of the explicitation of several layers. However, the introduction of reasoning systems for each layer allows the direct usage of existing reasoners for inferring on metalayers.

Metaization as such is less considered in knowledge-based configuration. However, especially when learning methods, i.e. automated knowledge engineering, has to be used in changing environments, the automated monitoring of KBs becomes crucial and is conceivable with the presented techniques.

6 Conclusion

In this paper, we state the differences of the main relations for modeling configuration knowledge, i.e. specialization, instantiation, and structuralization. By introducing and clarifying the use of instantiation on several metalayers, we open up a further modeling facility and sketch first usage of this metaization technique for knowledge-based configuration. In

```
(individual :name AbstractEntity-1
  :has-superconcept-m AbstractEntity-2
  :domain-name "Feature")

(individual :name AbstractEntity-2
  :domain-name "PreCrashDetection"
  :is-subconcept-of-m AbstractEntity-1
  :has-relations relation-descriptor-m-1)

(individual :name CompilableEntity-1
  :has-superconcept-m CompilableEntity-2
  :domain-name "Software")

(individual :name CompilableEntity-2
  :domain-name "MotionDetectionSoftware"
  :is-subconcept-of-m CompilableEntity-1
  :has-relations relation-descriptor-m-2)

(individual :name relation-descriptor-m-1
  :domain-name "has-Realization"
  :relation-of-m AbstractEntity-2
  :has-left-side structural-spec-1)

(individual :name structural-spec-1
  :in-relation-left-m relation-descriptor-m-1
  :some-of CompilableEntity-2)

(individual :name relation-descriptor-m-2
  :domain-name "realizes"
  :relation-of-m CompilableEntity-2
  :has-left-side structural-spec-2)

(individual :name structural-spec-2
  :in-relation-left-m relation-descriptor-m-2
  :some-of AbstractEntity-2)
```

Figure 5: CDL Example with instances on CDL^M representing concepts of CDL^D. This representation enables to reason on domain concepts with instance-related reasoning services.

upcoming work, we will apply these techniques in learning environments in the field of robot vision.

References

[Arlt *et al.*, 1999] V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz. EngCon - Engineering & Configuration. In *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, July 19 1999.

[Asikainen and Männistö, 2010] T. Asikainen and T. Männistö. A metamodelling approach to configuration knowledge representation. *International journal of mass customisation*, 3:333–350, 2010.

[Atkinson and Kühne, 2003] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.*, 20(5):36–41, 2003.

[Bateman *et al.*, 2009] J. Bateman, A. Castro, I. Normann, O. Pera, L. Garcia, and J.M. Villaveces. OASIS common hyper-ontological framework (COF), Deliverable D1.2.1. Technical report, University of Bremen, 2009.

[Brachman, 1983] Ronald J. Brachman. What is-a is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16(10):30–36, 1983.

[Gelle and Faltings, 2003] Esther Gelle and Boi Faltings. Solving mixed and conditional constraint satisfaction problems. *Constraints*, 8(2):107–141, 2003.

[Günter and Hotz, 1999] A. Günter and L. Hotz. KONWERK - A Domain Independent Configuration Tool. *Configuration Papers from the AAAI Workshop*, pages 10–19, July 19 1999.

[Hotz and von Riegen, 2010a] L. Hotz and S. von Riegen. A Reasoning-Driven Architecture - a Pragmatic Note on Metareasoning. In J. Sauer, editor, *Proc. of 24. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2010) – KI 2010 Workshop*, Göttingen, Germany, 2010.

[Hotz and von Riegen, 2010b] L. Hotz and S. von Riegen. Knowledge-based Implementation of Metalayers - The Reasoning-Driven Architecture. In Alexander Felfernig and Franz Wotawa, editors, *Proceedings of the ECAI 2010 Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET)*, 2010.

[Hotz *et al.*, 2006] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press, Berlin, 2006.

[Hotz, 2009a] L. Hotz. Construction of Configuration Models. In M. Stumptner and P. Albert, editors, *Configuration Workshop, 2009*, Workshop Proceedings IJCAI, Pasadena, 2009.

[Hotz, 2009b] L. Hotz. *Frame-based Knowledge Representation for Configuration, Analysis, and Diagnoses of technical Systems (in German)*, volume 325 of *DISKI*. Infix, 2009.

[John, 2002] U. John. *Konfiguration and Rekonfiguration mittels Constraint-basierter Modellierung*. Infix, St. Augustin, 2002. In German.

[Kühne, 2006] T. Kühne. Matters of (Meta-)Modeling. *Journal on Software and Systems Modeling*, 5(4):369–385, 2006.

[Miller and Mukerji, 2003] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003.

[Ranze *et al.*, 2002] K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt. A Structure-Based Configuration Tool: Drive Solution Designer DSD. *14. Conf. Innovative Applications of AI*, 2002.

[Strahringer, 1998] S. Strahringer. Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In *Proceedings of the Modellierung 1998*. Astronomical Society of Australia, 1998.

[Tran *et al.*, 2008] Thanh Tran, Peter Haase, Boris Motik, Bernardo Cuenca Grau, and Ian Horrocks. Metalevel Information in Ontology-Based Applications. In Dieter Fox and Carla P. Gomes, editors, *Proc. of the 23rd AAAI Conf. on Artificial Intelligence (AAAI 2008)*, pages 1237–1242, Chicago, IL, USA, July 13–17 2008. AAAI Press.

[van Harmelen *et al.*, 2007] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*. Elsevier Science, 2007.

SiMoL– A Modeling Language for Simulation and (Re-)Configuration[*]

Iulia Nica and **Franz Wotawa**[†]
Technische Universität Graz, Institute for Software Technology
Inffeldgasse 16b/2, Graz, Austria
{inica,wotawa}@ist.tugraz.at

Abstract

Simulation and configuration play an important role in industry. Modeling languages like Matlab/Simulink or Modelica, which are often used to model the dependencies between the components of physical systems, are less suitable for the area of knowledge-based systems. For these languages, the description of knowledge and its connection to a theorem prover for nonmonotonic reasoning (needed for configuration tasks) is, due to technical reasons, almost impossible. In this paper we focus on a language that can be used for both simulation and configuration purposes. SiMoL is an object-oriented language that allows representing systems comprising basic and hierarchical components.

1 Introduction

The adaptation of technical systems after deployment to ensure the desired system's functionality over time is an important task and can never be avoided. Reasons for adaptation are necessary corrections due to faults in system parts, changes in user requirements, or changes of technology among others. All activities necessary for increasing the lifetime of a system and retaining its usefulness are summarized under the general term maintenance.

In our research we focus on system changes due to changes in requirements. For example, consider a cellular network where the base stations are initially configured to ensure current and future needs to some extent. Due to changes in the environment, i.e., new apartment buildings constructed in reach of the base station or an increased use of cellular networks for data communication, the base station or even the local topology of the network has to be adapted. This adaption can more or less be classified as a re-configuration problem where the current system's structure, behavior, and the new requirements are given as input. Changes in the structure and behavior of the system in order to cope with the changes in the requirements are a solution of the re-configuration problem.

In order to provide a method for computing solutions for a given re-configuration problem we need to state the problem in a formal way. Therefore, we require a modeling language for stating systems comprising components and their relationships. In principle, formal languages like first order logic or constraint languages would be sufficient for this purpose. But using such languages usually is not easy and prevents systems based on such languages to be used in practice. Hence, there is a strong need for easy to learn and use modeling languages that are expressive enough to state configuration problems. The SiMoL language we introduce in this paper serves this purpose. The language itself is from a syntactical point of view close to Java. The idea behind SiMoL is to provide a language that can be used for (restricted) simulation and configuration at the same time.

SiMoL is an object-oriented language with multiple inheritance and allows for stating constraints between variables. Beside the basic data types like integer and boolean, SiMoL makes use of component instances. All component instances are statically declared. In this paper we focus on describing the syntax and the semantics of SiMoL.

2 Related research

Over time, the AI community has developed a large variety of configuration tools that fitted the different necessities and goals in each practical area, thus creating a strong foundation for newcomers. As preamble to our approach, we shortly recall three configuration systems, that make use of constraint programming.

ConBaCon [John and Geske, 1999] treats the special case of re-configuration, using the conditional propagation of constraint networks and has its own input language - ConBaConL. In [John and Geske, 1999], the authors present ConBaConL, a "largely declarative specification language", by means of which one can specify the object hierarchy, the context-independent constraints and the context constraints. Furthermore, the constraints are divided into Simple Constraints, Compositional Constraints and Conditional Constraints.

LAVA is another successful automated configurator [Fleischanderl *et al.*, 1998], used in the complex domain of telephone switching systems. It makes use of generative constraints and is the successor of COCOS [Stumptner *et al.*,

[*]Authors are listed in alphabetical orde. The work presented in this paper has been supported by the BRIDGE research project Simulation and configuration of Mobile networks with M2M Applications (SIMOA), which is funded by the FFG.

[†]Corresponding author.

1994], a knowledge-based, domain independent configuration tool. The modeling language is ConTalk, an enhanced version of LCON that follows the Smalltalk notation. A ConTalk constraint is a statement which describes a relationship between components ports or between the attributes values.

A powerful configuration system that combines constraint programming(CP) with a description logic(DL) is the ILOG (J)Configurator [Junker and Mailharro, 2003]. The combined CP-DL language, in which the configuration problem is formulated provides, on the one hand, the constraints, needed in the decision process, and on the other hand, the constructs of the description logic, able to deal with unknown universes. When solving the problem, the constructs of description logic, which are well-suited to model the configuration specific taxonomic and partonomic relations, are mapped on constraints and thus the wide range of constraint solving algorithms may be used.

The other field of interest for our research has been the modeling languages currently used for simulation of technical systems. Matlab/Simulink [4] and Modelica [5] are the most famous ones in the area of dynamic systems modeling and simulation. When working with Simulink, the user is capable of modeling the desired system in the graphical interface, based on the large library of standard components (called blocks). Also making use of predefined building blocks, Modelica, on the other side, is an equation-based object-oriented language with multi-domain modeling capability. Although both of them are complex languages, capable of modeling a great variety of components, neither Simulink or Modelica can be used for re-configuration purposes, as the description of knowledge and its connection to a theorem prover for nonmonotonic reasoning (needed for configuration tasks) is, due to technical reasons, almost impossible.

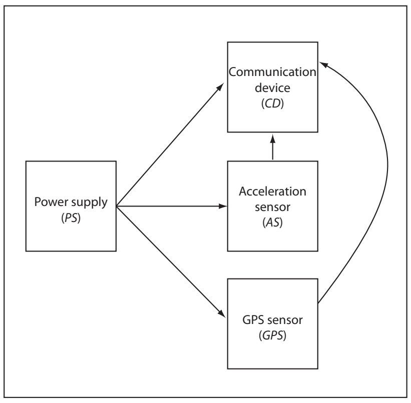


Figure 1: A small sensor systems

Throughout the rest of this paper, we present our modeling language - SiMoL. SiMoL can be applied in both simulation and re-configuration domains, using the powerful mechanism of constraint solving and hence being highly scalable for complex simulation and re-configuration tasks.

3 An example

In this paper we make use of the following small example to discuss SiMoL, as well as re-configuration using SiMoL for modeling systems. Figure 1 depicts a small system comprising 4 components, i.e., a power supply (PS), an acceleration sensor (AS), a GPS sensor (GPS), and a communication device (CD). The communication device is used for sending the measured sensor information to a server. The power supply is for providing electricity to the connected components. All these components have a behavior and provide functionality.

For the purpose of specifying functionality we introduce a function fct that maps a component to a set of attributes, which indicate a certain functionality. For our example, we introduce the attributes **ad**, **gps**, **comm** to state the acceleration sensor functionality, the gps functionality, and the ability for communication respectively.

$$fct(AS) = \{\textbf{ad}\} \;\; fct(GPS) = \{\textbf{gps}\} \; fct(CD) = \{\textbf{comm}\}$$

We now specify additional constraints of the system. The following constraint formally represents the requirement that the power provided by PS must be larger or at least equivalent to the sum of the power consumption of the other components:

$$power(PS) \geq power(AS) + power(GPS) + power(CD)$$

Moreover, we state that the device has to provide at least **ad**, **gps**, **comm** functionality.

$$fct(AS) \cup fct(GPS) \cup fct(CD) \supseteq \{\textbf{ad}, \textbf{gps}, \textbf{comm}\}$$

Finally, we have the requirement that the sum of the cost of each part of the device is not allowed to exceed a certain pre-defined maximum cost.

$$cost(PS) + cost(AS) + cost(GPS) + cost(CD) \leq max\_cost$$

In configuration we are interested in providing specific implementations of the components PS, AS, GPS, and CD such that all requirements are fulfilled and no constraint is violated. Hence, what we do now for our running example, is to introduce specific instances of the generic components with different costs and power consumptions. Table 1 summarizes all the used concrete component implementations.

A valid configuration is now a set of components that fulfills all constraints. For example, when assuming maximum cost of 60, the set $\{PS_1, AS_2, GPS_1, CD_2\}$ is a valid configuration but $\{PS_2, AS_2, GPS_1, CD_2\}$ is not because of violation of the cost constraint.

Throughout this paper we make use of this example and show how SiMoL can be used for modeling such systems.

4 SiMoL definition

In order to define SiMoL we discuss its syntax and semantics as well as its capability to be used for re-configuration purposes.

SiMoL syntax: As already mentioned, SiMoL uses a Java-like syntax and the common conventions compass most of the defined tokens: identifiers for any type of component and attribute, integer and boolean literals, separators, arithmetic and relational operators ($+, -, *, /, =, <, >, <=, >=, != $), special tokens - comments, reserved words and literals.

Additionally, SiMoL offers support for using *units of measurement*, thus creating a more realistic model.

Another feature of the language, that provides direct control over the possible values of a component attribute, is the

[4]www.mathworks.com

[5]www.modelica.com

Generic component	Instance 1	Instance 2
PS	$PS_1 : costs(PS_1) = 10, power(PS_1) = 10$	$PS_2 : costs(PS_2) = 20, power(PS_2) = 15$
AS	$AS_1 : costs(AS_1) = 2, power(AS_1) = 4$	$AS_2 : costs(AS_2) = 20, power(AS_2) = 1$
GPS	$GPS_1 : costs(GPS_1) = 6, power(GPS_1) = 5$	
CD	$CD_1 : costs(CD_1) = 10, power(CD_1) = 10$	$CD_2 : costs(CD_2) = 20, power(CD_2) = 4$

Table 1: The component instances for our small sensor system

```
component CD{
     attribute int power, costs;
     constraints{
       power={4,6,10} W;
       costs={10..30}; } }
```

Figure 2: SiMoL: initialization of attributes with integer valued ranges

initialization of attributes with integer valued ranges, as illustrated in fig. 4.

Basically, a program written in SiMoL comprises 3 sections: *a knowledge base declaration section*, which is optional, *an import declaration section*, which is also optional, and *a component definition section*, that is the main constructing unit of a SiMoL program and it is mandatory. Generally, each component will posses a set of attributes and will introduce constraints in the system. The attributes declaration is marked by the `attribute` keyword, whilst the relations stated between the component attributes and new-component instance-declaration statements appear enclosed in the `constraints{ ... }` block. By convention, an empty component definition section is not allowed, i.e., if the constraints block is missing, we have to declare at least one attribute for the current component. Furthermore, in the case of *derived components*, the opposite holds: even with no attributes declared, we may state constraints over the inherited attributes. For instance:

```
component AS{
     attribute int power, costs;
     constraints{
       power={4,6} W;
       costs={2..30}; }}
component AS1 extends AS{
     constraints{
       power=4;
       costs=2;}}
```

The ability to extend the functionality and behavior of existing components is of great importance for the taxonomic structure of a configuration domain. In any object oriented languages, the taxonomy relations are represented through the inheritance mechanism. We designed SiMoL with multiple inheritance. In order to demonstrate the necessity of this feature, let us consider the following scenario. For our small system described in Section 3, we introduce a new requirement that refers to a specific signal modulation which can be accomplished by a new component - a modem (M). The modem receives the measured sensor information and transmits the modified signal to the communication device. The function fct from Section 3 will similarly depict for M the modulation-demodulation functionality :

$$fct(M) = \{\mathbf{mdm}\}$$

Now the additional constraints of the system become:

$$power(PS) \geq power(AS) + power(GPS) \\ +power(CD) + power(M) \\ fct(AS) \cup fct(GPS) \cup fct(CD) \cup fct(M) \\ \supseteq \{\mathbf{ad}, \mathbf{gps}, \mathbf{comm}, \mathbf{mdm}\} \\ cost(PS) + cost(AS) + cost(GPS) + cost(CD) \\ +cost(M) \leq max\_cost$$

The problem appears if the pre-defined maximum cost is always exceeded, because of the new added component. In other words, we can not afford both a modem and a communication device. Therefore, a new component type - a communication device with integrated modem (MDC)- will solve the case (under the assumption that $cost(MDC) \leq cost(CD) + cost(M)$). In SiMoL, the MDC definition has the following syntax:

```
component MDC extends DC,M {
     constraints{
       power={4,6} W;
       costs={2..30};}}
```

In the constraints section, we may have the following types of statements:

- an empty statement: `;`,
- a component instance declaration: `GPS1 gps1;` Optionally, one can also initialize its attributes: `GPS1 gps1{costs=100};`

  Using this kind of statements, we define the subcomponent hierarchy in our model, i.e., the partonomy relations. The cardinality of these relations (i.e., the number of subcomponents which can be connected to a certain component) is always finite - we cannot have an unlimited number of components in our model.
- an arithmetic or/and boolean expression: `ps1.power>=sum([as1,gps2,cd1],power);`
- a conditional block:

  ```
  if(sum([ps1,as1,gps1,cd1],costs)
  <= max_cost)
       cost=sum([ps1,as1,gps1,cd1],costs);
  else cost=100;
  ```
- a `forall` block:

  ```
  forall(AS1){ power=10 W; costs={1..10};}
  ```
- an `exist` statement, e.g.:

  `exist(at_most(1),GPS1,costs=30);`.

We also mention the built-in functions `min, max, sum, product`, meant to ease the manipulation of large sets of component instances.

Adopting a clear Java-like syntax, SiMoL is a functionality-based, declarative language, creating a good environment for simulation, and, at the same time, it provides many embedded functionalities specially designed for configuration purposes.

Semantics of SiMoL: Because of space reasons we only briefly define the semantics of the language SiMoL where we rely on mathematical equations. In particular, we map every statement to a mathematical equation, and combine these equations for a component, taking care of component inheritance and component instances.

For each component defined in SiMoL we have a set of equations that is defined within the `constraints { ... }` block. Moreover, a component also receives equations from its super components and the instances used in the component definition. For example, when specifying `GPS1 gps;` in the variable declaration a new instance of `GPS1` is generated. All constraints of `GPS1` are added to the constraints of the component. The semantics of SiMoL is now nothing else than the union of all constraints defined including inherited constraints and constraints coming from component instances.

We discuss the expressiveness of the language by classifying its capabilities with respect to the framework offered in the chapter on configuration from [Rossi *et al.*, 2006]. In the context of the successful integration of constraint programming in solving a large variety of configuration problems, the author defines several distinguishing constraint models, each corresponding to a specific type of configuration problem. To set up the constraint model, the appropriate variables and constraints are deduced from the given configuration knowledge. The author states that this knowledge may have three different forms: the component catalogs, the component structure and the component constraints.

The catalog knowledge, as defined in [Rossi *et al.*, 2006], is modeled in SiMoL by means of the generic components (correspondent to the term of technical types in [Rossi *et al.*, 2006]) and the concrete components(derived (extended) from generic component/s or from other concrete component/s, in our case, and correspondent to the term of concrete or functional types in [Rossi *et al.*, 2006]). Both generic and concrete components have a set of attributes, mapped to variables in the constraint model. Based on this kind of knowledge, we build the catalog constraints ([Rossi *et al.*, 2006]), which are stated over the set of variables and formulated by means of C_{attr_val} and C_{attr_attr} constraints.

The structural knowledge of a SiMoL model is determined by the component instances declared in the current model. In this manner, we generate for our system the set of subcomponents, that are either generic or extended components. The logic behind this mechanism has been previously detailed, when presenting the semantics of the language. We recall that the SiMoL model is in fact a component, which describes the configuration problem. The connection ports defined in [Rossi *et al.*, 2006] have no correspondent term in SiMoL yet, but the connection between component instances is possible through C_{attr_attr} constraints. Also the statement in [Rossi *et al.*, 2006] according to which "the sets of direct subtypes of two types are mutually disjoint" does not hold in our approach, because we accept multiple inheritance.

Finally, the configuration constraints are divided into compatibility constraints, requirement constraints and resource constraint. The first ones specify which value combinations are legal for the attributes given in the model and they are modeled in SiMoL through C_{attr_val} and C_{attr_attr} constraints. The requirement constraints describe a relation between two component attributes ([Rossi *et al.*, 2006]), which is best depicted by combining C_{cond} with C_{attr_val} or C_{attr_attr}. Moreover, the resource constraints on numerical attributes were intensively addressed throughout this paper.

Consequently, we find the expressive power of the language sufficient for modeling the discussed configuration knowledge forms. As also stated in [Rossi *et al.*, 2006], the configuration problem complexity may vary from very simple option selection problems to complex cases, but they all appear as combinations of the specified knowledge forms.

5 Conclusion

In this paper, we have presented SiMoL- a new functional-based, declarative modeling language, that serves simulation and re-configuration purposes. The novelty of our approach is designing a language that is easy to learn and capable of modeling large and complex systems. SiMoL can cope with large models and be also efficient with respect to computation time (simulation). Although re-configuration is not fully implemented for the SiMoL language, several ideas are currently analyzed and implemented, such that in the near future a fully working re-configurator can be used for SiMoL models. In future research we mainly focus on providing a sound and complete configuration algorithm that takes SiMoL models and requirements as input and computes valid configurations as output.

References

[Fleischanderl *et al.*, 1998] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. In *IEEE Intelligent Systems & their applications*, pages 59–68, 1998.

[John and Geske, 1999] Ulrich John and Ulrich Geske. Reconfiguration of Technical Products Using ConBaCon. In *Proceedings of WS on Configuration at AAAI99*, Orlando, 1999.

[Junker and Mailharro, 2003] Ulrich Junker and Daniel Mailharro. The logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In *Proceedings of IJCAI-03 Configuration WS*, pages 13–20, 2003.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[Stumptner *et al.*, 1994] Markus Stumptner, Alois Haselböck, and Gerhard Friedrich. COCOS - a tool for constraint-based, dynamic configuration. In *Proceedings of the 10th IEEE Conference on AI Applications (CAIA)*, San Antonio, March 1994.