Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09)

July 11–13, 2009

Pasadena, CA, USA

Markus Stumptner and Patrick Albert, Editors

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Foreword

Representing and solving configuration problems has always been one of the key application and commercialization areas for AI research, and the need for powerful knowledge-representation formalisms to capture the great variety and complexity of configurable product models was a major driver for innovation in areas such as expert systems in the 1980s and constraint satisfaction systems and description logics in the 1990s. Efficient reasoning methods are required to provide intelligent interactive behavior in configurator software, such as solution search, satisfaction of user preferences, personalization, optimization, diagnosis, etc.

From the point of view of configuration as product configuration, different AI approaches are well established as central technologies in industrial configuration systems. This wide-spread industrial use of AI-based configurators makes the field more challenging than ever: the complexity of configurable products still increases, the mass-customization paradigm is extended to fields like service and software configuration, personalized (web-based) user interaction and user preference elicitation are of increasing importance, and finally, the integration of configurators into surrounding IT infrastructure like business information systems or web applications is critical. As a result, configuration technology often finds itself centrally placed in a number of interdisciplinary relationships. In addition to the ones previously listed these include engineering areas such intelligent manufacturing and supply chain solutions, configuration management in software engineering, and the configuration of financial and insurance services.

However, configuration as a computational paradigm traditionally has a wider range of applications. The configuration approach to problem solving, also referred to as model construction, has been used as an underlying technology in areas such as computational biology, service composition, design, image analysis. More recently, configuration emerges as an to Model-Driven Engineering (MDE) techniques for applications such as component-based software construction, where topics of interest would include Model Transformation, Model Satisfaction and Test Case generation, and in the creation and management of Software Product Lines.

The main goal of the workshop is to promote high-quality research in all technical areas related to configuration. As such, the workshop is of interest for researchers working in the various fields within the wide range of applicable AI technologies (e.g. Constraint Programming, Description Logics, Non-monotonic Reasoning, Case-Based Reasoning, Decision Support Systems, Business process and supply chain modeling). It serves as a platform for researchers and industrial participants to exchange needs, ideas, benchmarks, and use cases.

The workshop is the twelfth in a series of successful Configuration Workshops started at the AAAI'96 Fall Symposium and continued on IJCAI, AAAI, and ECAI since 1999. Its best papers, together with those of the 2010 workshop, will be collected in a special issue in AI EDAM in 2011.

Markus Stumptner and Patrick Albert
July 2009

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Workshop Organization

Workshop Co-Chairs

Markus Stumptner	University of South Australia, Adelaide, Australia
Patrick Albert	ILOG S.A., France

Workshop Organizing Committee

Claire Bagley	Oracle Corporation, USA
Alexander Felfernig	Universität Klagenfurt/Configworks, Austria
Albert Haag	SAP AG, Germany
Lothar Hotz	Universität Hamburg, Germany
Thorsten Krebs	encoway, Bremen, Germany
Tomi Männistö	Helsinki University of Technology, Finland
Barry O'Sullivan	Cork Constraint Computation Centre, Ireland
Juha Tiihonen	Helsinki University of Technology, Finland
Markus Zanker	Universität Klagenfurt, Austria

LaTeXmacros for proceedings autoformatting written by Wolfgang Mayer.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Contents

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

A Metamodelling Approach to Configuration Knowledge Representation*

Timo Asikainen and **Tomi Männistö**
Department of Computer Science and Engineering
Helsinki University of Technology
timo.asikainen@tkk.fi, tomi.mannisto@tkk.fi

Abstract

Traditionally, product configuration research has focused on two main areas: first, encoding the configuration task in terms of a knowledge representation language, such as a form of constraint satisfaction or some description logic, and reasoning about the properties of the encoding; second, implementing practical configuration systems. Less attention has been given to the conceptual and semantic foundations of product configuration. In this paper, we show how a novel metamodelling language, NIVEL, can be used to explicitly represent knowledge on three levels of abstraction, e.g., configuration modelling concepts, configuration models and their instances, configurations. In addition to physical products, we also consider the configuration of software product families. Major parts of the semantics for configuration modelling languages is obtained as a by-product of defining a configuration modelling language using NIVEL.

1 Introduction

Traditionally, product configuration research has focused on two main areas: first, encoding the configuration task in terms of a knowledge representation language, such as a form of constraint satisfaction or some description logic, and reasoning about the properties of the encoding; second, implementing practical configuration systems. Less attention has been given to the conceptual and semantic foundations of product configuration. In terms of citations, the most important conceptualisations are those by [Soininen *et al.*, 1998] and [Felfernig *et al.*, 2001]. However, both leave much room for improvement in clarity and level of detail.

Also, the conceptual modelling languages, e.g., Ontolingua [Gruber, 1992] applied by [Soininen *et al.*, 1998] or UML [Object Management Group, 2007] applied by [Felfernig *et al.*, 2001], are based on two levels of abstraction and hence do not support representing the three levels of abstraction—*metalevel* containing the definition of configuration modelling concepts, *model level* containing configuration models, and *instance level* containing configurations—needed to represent configuration knowledge.

In this paper, we show how recent results in metamodelling, in particular the metamodelling language NIVEL [Asikainen and Männistö, 2009], can be applied in the context of product configuration. In more detail, we show how configuration knowledge on the three levels of abstraction can be represented in a natural and uniform manner using NIVEL. Major parts of the semantics of configuration modelling languages is obtained as a by-product of defining them using NIVEL. In addition to standard conceptualisations of configuration knowledge [Soininen *et al.*, 1998; Felfernig *et al.*, 2001], we discuss representative solutions to a similar problem in the software engineering domain known under the term *software variability management*. Further, we extend NIVEL with role cardinalities that turn out to be instrumental in representing configuration knowledge.

The remainder of this paper is structured as follows. NIVEL is introduced and extended with role cardinalities in Section 2. Section 3 shows how NIVEL can be used to represent configuration knowledge. Discussion and comparison to previous work follow in Section 4. The paper is rounded up by some concluding remarks in Section 5.

2 Nivel: A metamodelling language

In this section, we discuss NIVEL [Asikainen and Männistö, 2009], a metamodelling language that will be used in the next section for representing configuration knowledge.

2.1 Overview of NIVEL

NIVEL is a metamodelling language that does not commit to a single modelling paradigm, such as object-orientation, and therefore covers a large variety of different modelling purposes. NIVEL is based on a core set of modelling concepts, i.e., class, instantiation, association, generalisation, attribute and value, and incorporates a number of recent ideas including strict metamodelling [Atkinson and Kühne, 2002a], distinction between ontological and linguistic instantiation [Atkinson and Kühne, 2003; Kühne, 2006], unified modelling elements [Atkinson and Kühne, 2002b] and deep instantiation [Atkinson and Kühne, 2007]. NIVEL supports modelling on any number of levels in a uniform manner.

*This is a preliminary version of an article to appear in the *International Journal of Mass Customization*

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

2.2 Abstract syntax

The abstract syntax of NIVEL is illustrated in Figure 1 using the UML class diagram notation. However, the figure is not intended to serve as an exhaustive description of the abstract syntax of NIVEL, let alone as a definition of a semantics for NIVEL. Instead, the figure is intended to give an intuitive account of the concepts of NIVEL to a reader familiar with the UML class diagram notation.

Class is the central language element of NIVEL in the sense that all other language elements are directly related to *class*. A class is an entity with an identity. Figure 2 contains an example that will be used to illustrate the NIVEL language elements throughout this section. A number of classes are present in Figure 2, e.g., *Book*, *Person* and *Novel*.

Instantiation is a binary relation between classes. If the pair (i, t) is in the relation, class i is termed an *instance of* t, and class t a *type of* i. A restriction to *single classification* can be made, in which case a class not on the top level must be an instance of exactly one class; otherwise such a class must be an instance of at least one type. Classes in Figure 2 (b) are instances of classes in Figure 2 (a) with their types given after their names, separated using colon (":"); or alternatively, using a dashed arrow, see Figure 2 (c).

A *model* consists of a set of classes, some of which are *top-level* classes. A top-level class is such a class that is not an instance of another class. The classes in Figure 2 (a) are top-level ones.

A class may simultaneously be an instance and a type. Hence, being an instance or a type are not intrinsic properties of a class but roles played by a class in the $instanceOf$ relation, as illustrated in Figure 2 (c), where *Ivanhoe* is both an instance of *Book* and a type of *LefasIvanhoe*. *LefasIvanhoe* is said to be a *first-order instance* of *Ivanhoe* and a *second-order instance* of *Book*. The term *higher-order instance* is used to refer to instances of order 2 or more.

The *potency* of a class gives the maximum order of its instances. For instance, in Figure 2 (c), the class *Book* is of potency 2 while *LefasIvanhoe* is of potency 0 and can therefore have no instances. The potency is given using a superscript, with 1 as the default value if none is given.

There can be *generalisations* between pairs of classes. A generalisation is a binary relationship between a pair of classes (a, b), represented using the *subclassOf* relation. Class a is termed a *subclass of* b, and class b a *superclass of* a. An instance of a subclass is also an instance of all the superclasses of the subclass. Classes on a higher level exercise control over their instances whether and how they may participate in generalisations and define attributes.

A generalisation may belong to a *generalisation set*. All generalisations in a generalisation set must share the same superclass. *Disjointness* and *covering constraints* may apply to generalisation sets. In the former (latter) case, an instance of the superclass must be an instance of at most (least) one of the subclasses of the generalisation set.

A class may have *attributes* and named *values*. An attribute describes what values the instances, including higher-order ones, of a class may and must have. The ability to describe higher-order instances is achieved using the *potency* of an attribute: an instance of a class has the same attributes as the class but the potency decremented by one. When a class with an attribute of potency 1 is instantiated, the instance has a corresponding value. Hence, an attribute of potency p describes the instances of order p of the class. In Figure 2, class *Ivanhoe* has the attributes *name* and *nrOfPages* bound to values *Ivanhoe* and 415, respectively.

An *association* is a relationship among a set of classes. An association defines a set of roles each of which is played by one or more classes, which is in contrast with associations (relations) in most previous conceptual modelling languages, such as UML and ER modelling, where a role must be played by exactly one class. Also in contrast with such languages, an association in NIVEL is also a class and may hence have higher-order instances. Consequently, an association in NIVEL may specify a relationship of interest between higher-order instances of the participating classes.

Cardinality constraints may apply to associations. Cardinality constraints in NIVEL resemble those in ER modelling, see [Thalheim, 1992], and to a lesser extent the multiplicities of association ends in UML. In addition to a lower and upper bound, a cardinality constraint in NIVEL includes a *potency* giving the order of instances of the association to which the constraint applies; customary cardinality constraints apply to first-order instances. The UML notation for multiplicities is used for cardinality constraints, with the added concept of potency denoted using superscript.

In Figure 2, *trade* is an assocation with three roles, *seller*, *buyer* and *book*. The role *seller* is played by *Person* and *Company*, the role *buyer* by *Person* and the role *book* by *Book*; the name of the role *book* can be omitted as the name equals the name of the class playing it.

2.3 Formal semantics

A formal semantics is given for NIVEL by translation to WCRL [Simons *et al.*, 2002], a general purpose knowledge representation language syntactically similar to logic programs; a full discussion of the formal semantics is given by [Asikainen and Männistö, 2009]. The symbol t will be used to denote the translation.

In symbols, we write: $t : \mathcal{N} \mapsto \mathcal{W}$, where $\mathcal{N}$ denotes the set of syntactically well-formed NIVEL models and $\mathcal{W}$ the set of WCRL programs. A model M entering the mapping t is termed an *input model*.

The formal semantics capture the notion of a *valid model*. Intuitively, the notion of validity pertains to the interrelations that may and must exist between model elements: for example, a class must not be an instance of itself. The mapping t is constructed in such a way that each stable model of $t(M)$, where $M \in \mathcal{N}$, corresponds to a valid NIVEL model.

An input model M need not be valid. In general, only a subset of the elements of M is found in each valid spun by M: an input model contains elements that may or may not appear in valid models. Intuitively, an input model spans a search space for valid models.

2.4 Extending NIVEL with role cardinalities

As discussed above, a role in an association may be played by more than one class. This is a generalisation over typical conceptual modelling languages, such as ER modelling [Chen,

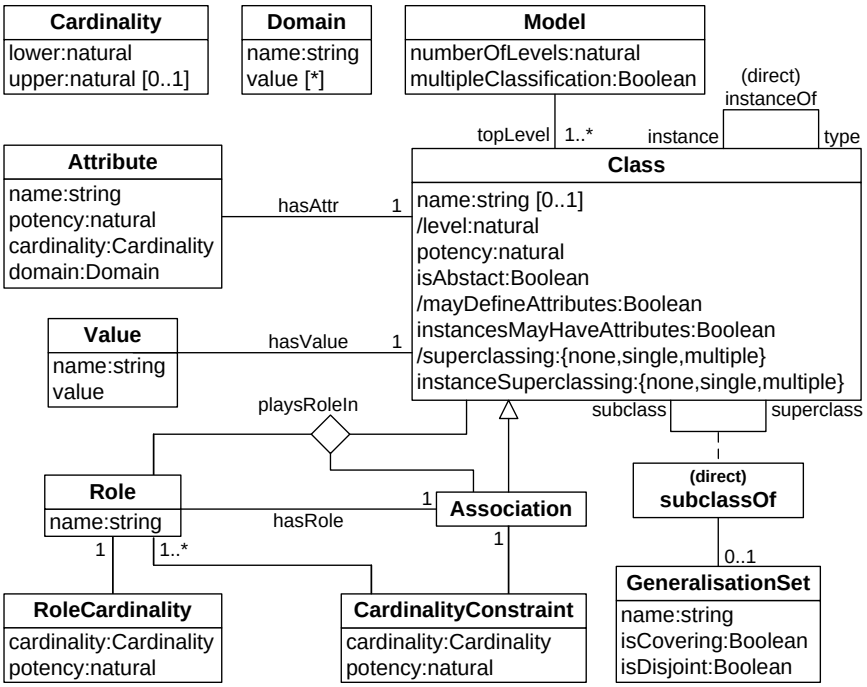

Figure 1: The abstract syntax of NIVEL given as a UML class model extended from [Asikainen and Männistö, 2009]

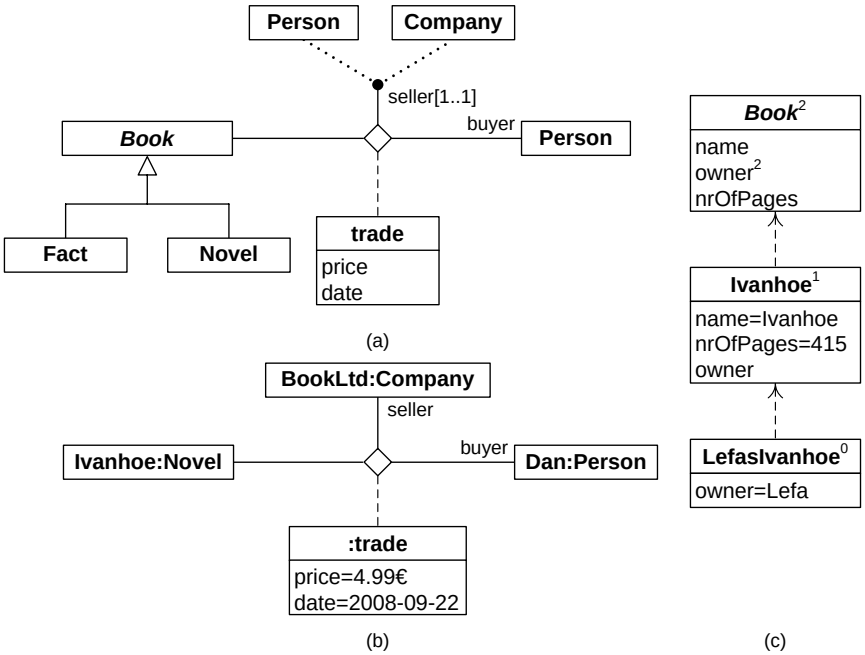

Figure 2: A NIVEL example: trades related to books

1976] and UML, where a role in a relation or an association may only be played by a single class or entity, respectively. However, in some cases it may be necessary to restrict the number of classes playing a role in the instances of an association, either first or higher order. Towards this purpose, we extend NIVEL with the notion of *role cardinality*; see Figure 1 for the abstract syntax.

A role cardinality relates to a role r in an association a and consists of a *cardinality* $[L..U]$ and a *potency* p. The intuitive semantics is that for an instance of order p of a, the number of classes playing role r must be at least L and at most U. The upper bound may be omitted, in which case there is no

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

constraint on the maximum number of classes playing role r. Note that the existing semantics of NIVEL implies that each role is played by at least one class, i.e., $L = 1$.

The notation for role cardinalities is illustrated in Figure 2 (a): a role cardinality of $[1..1]$ is defined for role *seller* of *trade*. Similar conventions as for UML cardinalities are adopted: If $L = U$, only one number is given as is done in the figure. A missing upper bound is denoted using the symbol "*".

Role cardinalities can be given a formal semantics in WCRL as follows using the symbols introduced in the previous paragraph and the predicates defined by [Asikainen and Männistö, 2009]. For a lower bound $L \geq 2$, we write:

$$\leftarrow \left\{playsRoleIn(C, r, A_i) : playsRoleIn_p(C, r, A_i)\right\} L - 1, \\ instanceOf_{tpo}(A_i, a, p), class(A_i)$$

and for an upper bound U (if one is defined):

$$\leftarrow U + 1 \left\{playsRoleIn(C, r, A_i) : playsRoleIn_p(C, r, A_i)\right\}, \\ instanceOf_{tpo}(A_i, a, p), class(A_i)$$

Above, the predicate $playsRoleIn(c, r, a)$ gives that class c plays role r in association a and $instanceOf_{tpo}(i, t, o)$ that i is an instance of t of order o.

3 Representing configuration knowledge using NIVEL

In this section, we show how configuration knowledge can be represented using NIVEL.

3.1 Levels of abstraction

Figure 3 illustrates the three levels of abstraction that underlie all product configuration frameworks, at least implicitly. The highest level is the *metalevel* containing the metamodel for the languages. In other words, the metalevel contains the concepts and their interrelations that are used to define configuration models on the level next below, termed the *model level*. The third and lowest level termed *instance level* contains *instances* and *configurations*. Intuitively, a configuration represents an individual product. A configuration model defines a set of configurations that are *valid* with respect to the model. The semantics of a configuration model usually concern the relation between a configuration model and configurations: some configurations are *valid* with respect to the configuration model.

The metalevel and the model level, and the model level and the instance level are related to each other in a similar manner: in both cases, the former can be characterised as a *model of* the latter. Further, in both cases, entities on the former, e.g., *Mainboard* and *processor* in Figure 3, can be termed *types* of the entities on the latter and entities on the latter, e.g., *:Mainboard* and *:processor* can be termed *instances* of the entities on the former.

Nivel We give the NIVEL representation of configuration modelling concepts under headings like this. The three levels of abstraction can be represented in NIVEL in a straightforward manner using NIVEL models with three levels. The

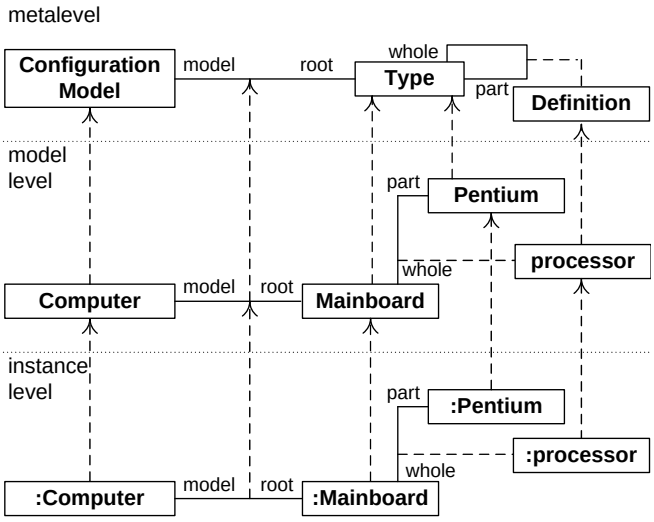

Figure 3: Levels of abstraction in product configuration frameworks

metalevel is mapped to the top level, i.e., level 0, of a NIVEL model. Entities on the model level are represented by NIVEL model elements on level 1; these model elements are instances of model elements on the top level. Similarly, entities on the instance level are represented by level 2 model elements.

3.2 Taxonomy

Types in configuration models can typically be organised into taxonomies using a binary relation known under different names in different approaches, e.g., *isa* [Soininen *et al.*, 1998] or *generalisation* [Felfernig *et al.*, 2001], with the terms *subtype* and *supertype* used to refer to the types playing the roles in the relation. The intuition underlying the relation is that an instance of a subtype is also an instance of its supertypes. This implies that the properties of a supertype become properties of its subtypes.

There are a number of issues that differentiate configuration modelling languages with respect to taxonomies. First, an important question related to generalisation is whether multiple superclasses are allowed for a type or not: some conceptualisations explicitly allow multiple superclasses [Soininen *et al.*, 1998] while others seem to equivocate on the issue [Felfernig *et al.*, 2001]. Second, in a configuration modelling language with modelling constructs such as structure, connection points and functions, one might wish to enable taxonomies for some constructs, e.g., components and functions, but disable them for others.

Nivel Taxonomies in configuration modelling languages can be represented in NIVEL using generalisations. It is possible to decide independently for each modelling concept whether its instances may be organised in taxonomies or not, and if yes, whether multiple superclasses (supertypes) are allowed or not. The double-headed arrow in Figure 4 (a) gives that a component type, i.e., and instance of *ComponentType*, may have any number of supertypes. Similarly, the single-

headed arrow in Figure 4 (e) has the semantics that a component type may have at most one supertype. By default, taxonomies are not allowed.

3.3 Compositional structure

Modelling the *structure* of configurable products is one of the key constructs in virtually all configuration modelling languages. The structure may be either physical or logical [Soininen *et al.*, 1998].

Intuitively, the structure pertains to part-whole relationships between entities, most importantly components. In other words, a component may be composed of other components, thus forming a compositional hierarchy. It is also said that a component is a *part* of the other component, termed *whole*, that is said to *contain* the part.

The compositional structure is specified using *part definitions* in component types. A part definition relates a *whole type* and *part type(s)*: depending on the method, there may be either many possible part types [Soininen *et al.*, 1998] or a single part type [Felfernig *et al.*, 2001]. In the latter case, the effect of multiple possible types can be achieved by defining a common supertype, if one does not already exist, for the desired set of types.

In addition to the above-disussed types, a part definition may have other properties. Most importantly, a *cardinality* is an integer range $[L..U]$ with the semantics that a whole in a valid configuration must have at least L and at most U parts by the part name. Further, a part definition contains a *part name* [Soininen *et al.*, 1998] that identifies the role in which instances of the possible part types are parts of instances of the whole type.

Nivel The NIVEL representations for the approaches by [Soininen *et al.*, 1998] and [Felfernig *et al.*, 2001] are illustrated in Figures 4 (a) and (e), respectively. In the latter approach, the effect of multiple possible types is achieved by defining a supertype for the desired set of types, cf. Figure 4 (f). Note that the cardinality of the *whole* role is restricted to 1 in both cases.

Figures 4 (b), (c) and (f) illustrate parts on the model level. Figures 4 (b) and (c) are both based on Figure 4 (a), with (b) using the basic NIVEL notation and (c) a more concise notation tailored. Figure 4 (f) shows a part definition based on Figure 4 (e).

At the instance level, Figures 4 (d) and (g) illustrate part relationships based on Figures 4 (c) and (f), respectively.

3.4 Root types and instances

The root, or possibly multiple roots, of the compositional hierarchy are commonly defined by associating one (or more) types as *root types* of a configuration model. The intuitive semantics is that only an instance of a root type may appear in a valid configuration without being a part of another component. It is said that each instance in a valid configuration must have a *justication* in one of these two ways.

Alternatively, instead of defining a root type for a configuration model, it is possible to define a Boolean attribute for component types with the semantics whether an instance of the component type may appear in a valid configuration without being a part of another component [Soininen *et al.*, 1998].

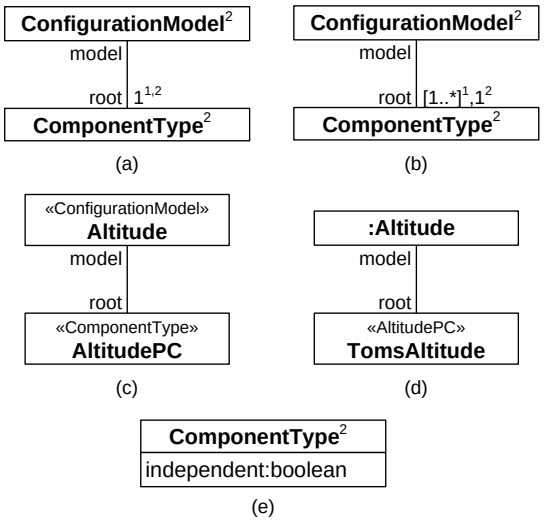

Figure 5: Representing configuration models and root component types, and configurations and root component instances in NIVEL. (a) A definition at the metalevel requiring that each configuration model has a single root component type and each configuration has a a single root component instance. (b) Similarly as above, but a configuration model may now have several root component types. (c) Configuration model *Altitude* with component type *AltitudePC* as its root component type. (d) An unnamed configuration of *Altitude* with *TomsAltitude* as its root component instance. (e) An alternative approach used by [Soininen *et al.*, 1998].

The number of root types in a configuration model and the number of root instances in a valid configuration may vary, the typical specification being exactly one root type and instance [Soininen *et al.*, 2000], while other authors allow multiple root instances [Soininen *et al.*, 1998; Kang *et al.*, 1998].

Nivel The NIVEL representations of different approaches to roots are represented in Figure 5. At the metalevel, Figure 5 (a) illustrates the case of a single root type (cardinality constraint 1 at the root end) and instance (1^2), whereas Figure 5 (b) the case of multiple root types ($[1..*]^1$) but a single root instance (1^2). The approach followed by [Soininen *et al.*, 1998] can likewise be represented using NIVEL, see Figure 5 (e).

3.5 Ports and connections

In addition to the compositional structure of components, it is in many cases of interest to model the possibilities for *interactions* between model elements. In product configuration, *points of interaction*, typically termed *ports*, and *connections* between these are standard means for capturing interactions within configurable products.

Just as compositional structure, the points of interaction and their interconnections may be either physical or logical [Soininen *et al.*, 1998]. Connection points may also have a *direction*, as is the case in Koala [van Ommering *et al.*, 2000]

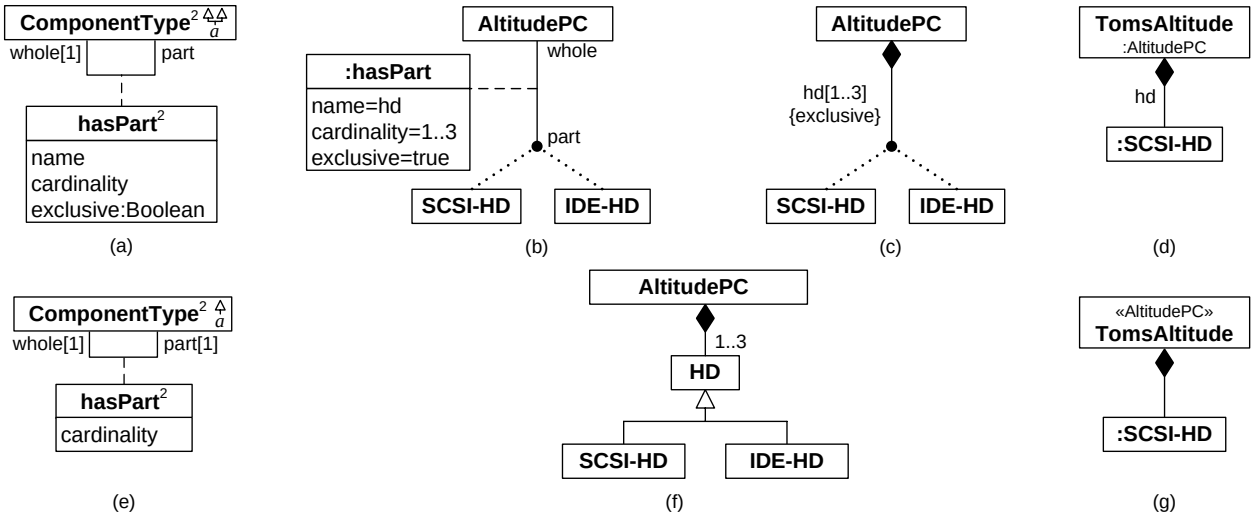

Figure 4: Representing compositional structure in NIVEL. (a) Definition of a part relation at the metalevel similar to found in [Soininen *et al.*, 1998]. Multiple possible part types may be defined. (b) A sample part definition in a configuration model. (c) The same definition as above represented using custom notation. (d) In a configuration, *TomsAltitude* has an instance of *SCSI-HD* as its part. (e)–(g) An alternative conception of compositional structure similar to [Felfernig *et al.*, 2001]. (e) A part definition has a single type for the part. (f) alternative type may be defined as subtypes of a common supertype. (g) Parts have no role names in configurations.

and Koalish [Asikainen *et al.*, 2003].

Just as for compositional hierarchy, an instance of a connection point type (e.g., a port instance) may not typically appear in a valid configuration without being justified by being a connection point of some entity.

As suggested by the term *connection point*, there may be *connections* between connection points. In different configuration modelling languages, different constraints apply to the connections that must and may exist between connection points. As an example, [Soininen *et al.*, 1998] require that the types of connected ports must be *compatible*. Also, there are differences in the number of connections a connection point may participate in. Finally, connections may either have a direction [van Ommering *et al.*, 2000; Asikainen *et al.*, 2003] or no direction, which is the typical case for physical and electronics products [Soininen *et al.*, 1998; Felfernig *et al.*, 2001].

Nivel The NIVEL representation of connection points similar to those in [Soininen *et al.*, 1998] is demonstrated in Figure 6 (a). Hence, a connection point at the model level is characterised by a *name* and a *cardinality*. Further, the definition of connections at the metalevel is illustrated in Figure 6 (b). Connections are symmetric in that the ports participating in instances of the association both play the same (unnamed) role. The role cardinality 2 of potency 2 implies that a connection is between exactly two ports on the instance level.

Port definitions on the model level are illustrated in Figures 6 (c) and (d), using basic NIVEL notation and a notation tailored for the purpose, respectively. Figures 6 (f) and (g) illustrate component-port relationships at the instance level, using a notation similar to Figure 6 (d). Figure 6 (h) shows a connection between the ports of Figures 6 (f) and (g).

Note that, according to NIVEL semantics, in order for two ports to be connected at the instance level, their respective types must be associated through the *connection* association at the model level. As an example, the association in Figure 6 (e) is needed to enable the connection in Figure 6 (f). Such first-order instances of *connection* correspond to compatibility definitions of [Soininen *et al.*, 1998].

An alternative definition of a connection relation is given as Figure 6 (i). The definition resembles the notion of connections in Koala [van Ommering *et al.*, 2000] and its derivative, Koalish [Asikainen *et al.*, 2003]. The defined connections are antisymmetric, with the roles *from* and *to*.

3.6 Attributes and values

A type in a configuration model may be characterised by *attributes*, also termed *attribute definitions* [Soininen *et al.*, 1998]. An attribute is typically characterised by a *name* and an *attribute value type*, or *domain*, the term used in NIVEL. In addition, an attribute may be characterised by a cardinality with semantics intuitively similar to the cardinality of a part definition, see Section 3.3, or a binary *necessity definition* that allows distinguishing between mandatory and optional attributes [Soininen *et al.*, 1998]. An attribute that may take more than one value is termed *set-valued* as opposed to a *single-valued* attribute.

A valid instance of a type with an attribute definition has a number of *values* corresponding to the attribute definition.

Nivel In NIVEL, the decision whether attributes may be defined is made for each modelling concept separately at the top level. As an example, Figures 4 (a) and (e) show how attributes are enabled for component types using the symbol "a" at the top-right corner. The default is that no attributes

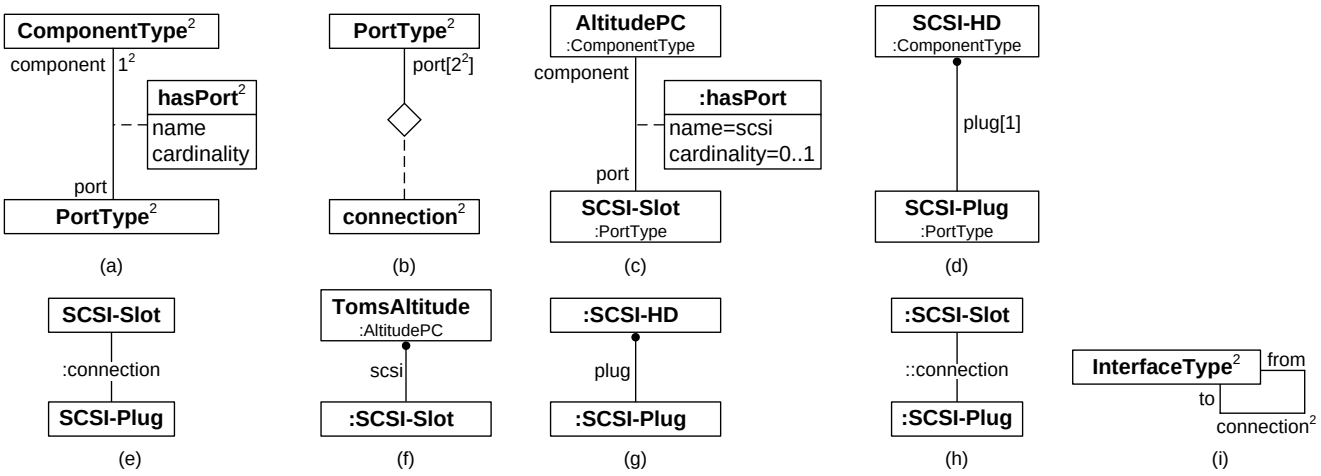

Figure 6: Representing ports and connections in NIVEL. (a) Definition of the *hasPort* relation between components and ports at the metalevel. (b) The definition of the *connection* relation. (c) An example port definition. (d) Another port definition represented using custom notation. (e) The port types *SCSI-Slot* and *SCSI-Plug* are compatible with each other. (f)–(g) Components have ports in configurations. (h) A connection between components in a configuration. (i) An alternative, non-symmetric definition of connections at the metalevel similar to the connections of Koala [van Ommering *et al.*, 2000].

may be defined for instances of a type. Attributes are by default set-valued. However, attributes can be easily constrained to be single-valued [Asikainen, 2008].

4 Discussion and comparison to previous work

In this section, we reflect the work presented in this paper with previous work on configuration modelling languages.

4.1 Evaluation

In this subsection, we discuss which aspects of configuration modelling languages can be fully captured with NIVEL using the kind of model fragments presented in Figures 4 through 6 and which aspects require additional constraints.

To begin with, the NIVEL semantics for the notions of generalisation, instantiation, attribute and value are as such applicable in configuration modelling languages. The semantics cover both well-formedness rules for each of the abovementioned modelling concepts and their interrelations. Further, NIVEL enables the concise representation of the abstract syntax for compositional structure as well as connection points and connections.

On the other hand, some of the semantic aspects of compositional structure, connection points and connections are not covered by NIVEL. In more detail, the notions of justification discussed in Section 3.3 and the semantics of cardinality attributes (cf. Figures 4 (a) and (e)) must be specified using additional constraints; [Asikainen and Männistö, 2009] provide an example set of such constraints. The same applies to the similar notions for connection points.

Further, the NIVEL semantics (for valid model) is more general than the customary definition of configuration task, i.e., finding a valid configuration of a configuration model m matching a given set of customer requirements R. Consequently, additional constraints are needed to restrict valid NIVEL models to valid configurations. Also, given a configuration model m, a set of instances large enough to cover any valid configuration of m must be generated; under suitable restrictiong, this can be achieved using a simple algorithm as has been shown by [Asikainen and Männistö, 2009].

Finally, certain kinds of configuration knowledge cannot be represented in NIVEL at all, or at least not in a straightforward manner. These kinds include *procedural knowledge* on the configuration task, or the *derivation process*, as the authors [Hotz *et al.*, 2006] call it; and resource-based modelling originally suggested by [Heinrich and Jüngst, 1991] and later incorporated in other approaches [Soininen *et al.*, 1998; Felfernig *et al.*, 2001; Hotz *et al.*, 2006].

4.2 Explicit knowledge representation

One of the contributions of this paper is to give more explicit yet concise and easily understandable definitions of configuration modelling languages. The contribution of such definitions can be seen by comparing our approach with previous conceptualisations of configuration knowledge.

In the configuration ontology of [Soininen *et al.*, 1998], only taxonomic relationships between concepts on the metalevel are represented using diagrammatic notation: although it is said that *Component type* is a *direct-subclass-of Configuration type*, the ways in which a component type or component instance may relate to other configuration types is only explained in natural language.

The conceptualisation by [Felfernig *et al.*, 2001] is even more parsimonious in describing the modelling concepts: the reader is forced to infer much of the abstract syntax of the conceptualisation from an example configuration model. Instead, emphasis is given on configuration knowledge based construction and diagnosis.

5 Conclusions and further work

We have shown how configuration modelling languages can be defined using a novel metamodelling language, NIVEL. In contrast to most, if not all, previous work, NIVEL enables the uniform representation at three levels of abstraction relevant to configuration modelling—metalevel containing the definition of configuration modelling concepts, model level containing configuration models, and instance level containing configurations describing individual products. Consequently, knowledge at all three levels can be explicitly represented. We believe that this substantially facilitates both understanding individual conceptualisations of configuration knowledge and comparing such conceptualisations with each other.

Further work is required to bring the ideas presented in this paper closer to practice. A suitable tool supporting NIVEL can be used as a product configurator. Of course, matching the level of usability and efficiency of the state-of-the-art configurators may require crafting a generic NIVEL tool to better support configuration purposes. On the other hand, such a tool based on NIVEL could accommodate a wide range of configuration modelling concepts. Hence, such a tool would in itself be configurable.

Acknowledgements

We gratefully acknowledge the financial support from the Technology Industries of Finland Centennial Foundation.

References

[Asikainen and Männistö, 2009] Timo Asikainen and Tomi Männistö. Nivel—a metamodelling language with a formal semantics. *Software and Systems Modeling*, in press, 2009.

[Asikainen *et al.*, 2003] Timo Asikainen, Timo Soininen, and Tomi Männistö. A koala-based approach for modelling and deploying configurable software product families. In Frank van der Linden, editor, *5th International Workshop on Product Family Engineering (PFE-5)*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer, 2003.

[Asikainen, 2008] Timo Asikainen. *A Conceptual Modelling Approach to Software Variability*. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2008.

[Atkinson and Kühne, 2002a] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22, 2002.

[Atkinson and Kühne, 2002b] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 22(4):290–321, 2002.

[Atkinson and Kühne, 2003] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[Atkinson and Kühne, 2007] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3), 2007. DOI: 10.1007/s10270-007-0061-0.

[Chen, 1976] Peter P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Felfernig *et al.*, 2001] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2):165–176, 2001.

[Gruber, 1992] Thomas R. Gruber. Ontolingua: A mechanism to support portable ontologies. Technical report, 1992.

[Heinrich and Jüngst, 1991] M. Heinrich and E. W. Jüngst. A resource-based paradigm for the configuring of technical systems from modular components. In *Seventh IEEE Conference on AI Applications (CAIA)*, pages 257–264, 1991.

[Hotz *et al.*, 2006] Lothar Hotz, Katarina Wolter, Thorsten Krebs, Sybren Deelstra, Jos Nijhuis, and John MacGregor. *Configuration in Industrial Product Families—The ConIPF Methodology*. IOS Press, 2006.

[Kang *et al.*, 1998] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[Kühne, 2006] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[Object Management Group, 2007] Unified Modeling Language: Superstructure, version 2.1.1. Technical Report formal/2007-02-05, Object Management Group (OMG), 2007.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[Soininen *et al.*, 1998] Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):357–372, 1998.

[Soininen *et al.*, 2000] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Unified configuration knowledge representation using weight constraint rules. In *ECAI'00 Workshop on Configuration*, 2000.

[Thalheim, 1992] Bernhard Thalheim. Fundamentals of cardinality constraints. In Günther Pernul and A Min Tjoa, editors, *11th International Conference on the Entity-Relationship Approach (ER '92)*, volume 645 of *Lecture Notes in Computer Science*, pages 7–23, 1992.

[van Ommering *et al.*, 2000] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

A Simple Evaluation Process for Configurability

Andreas Falkner and **Alois Haselböck**
Siemens AG Österreich, 1030 Vienna, Austria
e-mail: {andreas.a.falkner,alois.haselboeck}@siemens.com

Abstract

During the design of products and product families it is important as early as possible to balance the costs for better configurability with the profits which can be gained by it. For that purpose, we propose a simple yet comprehensive process based on an evaluation table containing configurable features and configuration criteria.
In a quick first step, the necessity and the profitability of each planned feature is estimated. In a second step, the features with a low profitability are analyzed in detail, resulting in a decision to skip them or to redesign them. The goal is to reduce overall configuration costs without compromising necessary requirements.

Motivation

Academic literature on the topic of product configuration like the series of workshops on configuration systems ([Tiihonen et al., 2008] and previous ones) focuses mainly on techniques (like constraint solving), algorithms, tools, etc. - generally speaking, on "doing things right". In praxis it is equally important to use those techniques in a useful and economic way - i.e. "doing the right things". Conferences on mass customization (like [Blecker et al., 2007]) cope with such topics, e.g. identification of configurable components, designing product families, etc.

In [Riitahuhta et al., 2000], the subject of “design for configuration” is comprehensively discussed. Designing modular systems starts with a study of the market, the potential customers and the available technology. This analysis leads to a product structure built from modular parts, where modularization is established and enhanced using relational matrices considering strategic and functional aspects of the product.

Hence, configurability and design for configuration is a broad topic with many important aspects. In this work, we specifically focus on an aspect with a high economic potential: the interface between product design and configurator design, reviewing the chosen flexibility/variability of the product. To which extent shall the product and its components be subject to configuration, i.e. customization to individual customers' needs?

The analysis of configurability and profitability shall be light-weight. Otherwise it would consume too many ressources or would not be accepted by the involved persons.

Configuration costs

From an abstract point of view, configuration is the process of discriminating a special solution from a set of possible solutions. Typically, this discrimination process is done stepwise in turn by the user and the configuration tool (i.e. configurator application).

Costs arise for both implementing and using the tool. Investing more into the tool can help significantly reduce the efforts of the users for their configuration work.

If the user is an engineer who configures a complex technical system (like a railroad switching system or a factory) as prerequisite of the delivery process, then the configuration costs can be a significant part of the overall system costs.

If the user is an end customer who - for example - configures and orders a consumer product directly via an internet application, then the number of inputs, the duration of the order session, and the presentation of questions and result contribute much to the user acceptance and to his decision to buy the product in the first place.

In both cases, the configuration tool must guide the user to or must generate solutions of acceptable quality. Bad solutions will cause additional costs and – even worse - bad reputation. So we demand that our configuration tool is doing the “right” thing (ignoring here that there usually is a range of possibilities how to guide the user and how to present the options and results).

The configuration tool must take over those parts in the solving process, which can be automatically derived. Essentially, this is ruling out illegal solutions and making decisions which are already determined in the current state of the configuration. Costs for implementation and maintenance of the configuration tool must not be underestimated and should be part of the budget right from the beginning of product development.

We can describe configuration as a decision tree. The nodes are the parameters to be configured, and the arcs are

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

the possible values for the parameters. Decisions are made alternately by the user and by the tool. Figure 1 shows such a decision tree, which is, of course, a very simple example with binary decisions only.

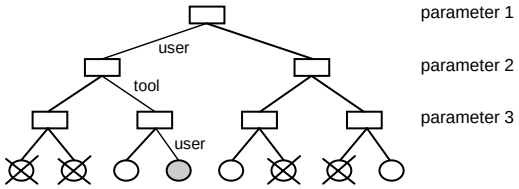

Figure 1: Configuration decision tree

The lowest level (the leafs) represents all possible combinations of the configuration parameters. They are partitioned into two groups: solutions and illegal combinations. The user makes those decisions which discriminate the path to his/her solution, while the tool makes those decisions which rule out illegal branches. Usually, a configurator should minimize the decisions to be made by the user and should guide the user to an optimal solution reaching various quality criteria, like fulfilling all the user requirements, being economical, using state-of-the-art components, etc.

There are many techniques for how a configuration tool can support the user and minimize her/his costs. E.g. it could establish some degree of consistency (partial or full, see [Bessiere, 2006]), and it may choose the order of parameters in an optimal way (see recommender systems [Felfernig and Burke, 2008]).

Another cost criterion for complex technical systems is optimization. If the decisions made by the user do not discriminate a single solution but a set of solutions which all meet the user's requirements, then the configuration tool must find the best solution according to some defined optimization criteria (e.g. minimum hardware costs, extendibility). This produces additional challenges and therefore costs for design, implementation and maintenance of the configurator.

A real-world example

One of our configurators in railroad safety domain allows to plan ETCS systems (ETCS = European Train Control Systems). The basic units in ETCS are the so-called datapoints, which lie between the track metals and transmit information to the passing train about the properties of the forthcoming section, like gradient and speed profile, signaling and stop points.

A datapoint consists of at least one concrete sending unit, a so-called balise. The size of a telegram a balise may send is quite restricted due to the short passing time of fast trains. Therefore, balises can be cascaded to increase the potential telegram length.

One of our customers demanded to use only one single balise per datapoint to save hardware and installation costs, assuming to reduce the entire costs. But it turned out that this optimization attempt didn't pay off.

At about a third of all datapoints, the telegrams where too big and did not fit into a single balise. For these cases, a second balise would have solved the problem in a simple way. But in order to fulfill the single-balise requirement, both configurator implementers and users had a lot of extra work. The configurator rules and constraints had to be tuned so that the telegrams were as small as possible. Additionally, the users had to manually optimize several telegrams to squeeze them into the balises.

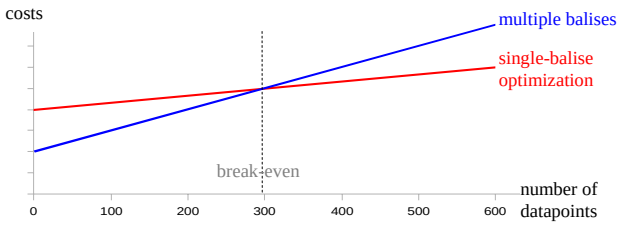

Figure 2: Example of a requirement evaluation

An analysis of the costs for the single-balise optimization is shown in figure 2. Costs are estimated for hardware, installation, maintenance and – this is the important point here – configuration. In the left part of the requirement evaluation graph, the version without the single-balise optimization (the blue line) clearly outperforms the single-balise optimization variant (the red line) because hardware costs don't play a dominant role in that region. From a certain value (which is around 300 datapoints), the single-balise optimization pays off because the hardware savings compensate the higher configuration costs. Such an analysis in an early phase of system design would have made this point clear.

Evaluation of configuration complexity

The preceding example shows that it is not enough to take the number of configuration parameters and decisions (i.e. the size of the decision tree) into account to estimate the costs for configuration user and tool. Rather must costs for each feature (e.g. group of parameters, their dependencies, and their optimization requirements) be evaluated separately.

To avoid unnecessary costs for the configuration of a product, or to prevent that expensive configuration costs eat up the additional benefit of a certain product feature, configuration architects should be involved in the product design in an early phase. Product and configurator designers should together identify those parameters and dependencies, which may be simplified or removed to reduce the overall product costs.

The rough steps in the product life-cycle are the product design and development, the tool design for logistics and sales (e.g. the configurator), and the sales processes. Product design, together with sales and marketing department, is responsible for the planning of the product features, of the hardware and software, and of the production, delivery, installation and sales processes. See also [Hvam et al., 2007].

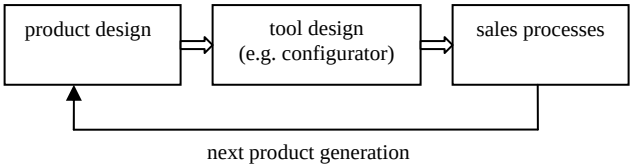

Figure 3: Product life-cycle

Specifically, product design defines the requirements for the product configurator, which represents the interface to the customer. On the one hand, these requirements define the building structures of the system, along with its variability to provide the flexibility necessary for the customers' needs. Configurator design now maps these structures to an appropriate data model and to dependencies and restrictions within this model (constraints, rules).

On the other hand, product design must provide – at least to a certain degree – requirements on the configurator usability and the optimization criteria. For example, for many consumer products it is state-of-the-art that the customer sees a picture of the current state of the product any time in the configuration session. Such demands aim at improved usability and acceptance of the configurator, and – eventually – of the product. But they may have high impacts on the realization costs of the configurator.

An early involvement of configurator designers in the product design may identify and avoid cost-ineffective requirements like in the real-world example above, thus increasing the overall profitability. We see the profitability here as the ratio of benefits or profits gained by the feature in question versus its implementation costs in the configuration tool - similar as seen in development processes like Scrum for prioritizing the product backlog (see e.g. [Watts and Haines, 2009]).

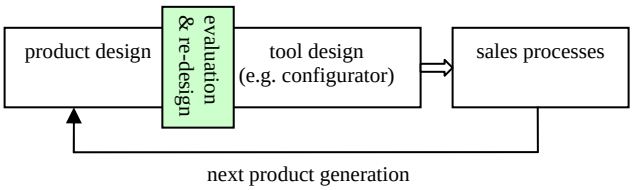

Figure 4: Product life-cycle with configurator requirements evaluation

We propose the following process for the start phase of configurator development:

- definition of the requirements of the product features, usability and optimization criteria
- quick evaluation of those requirements to find candidates which potentially may be simplified or removed in order to increase profitability
- detailed evaluation of those candidates and estimation of their cost-effectiveness
- re-design and final approval of the product features

Product design and configurator requirements

Product design must specify all requirements on the configurator, so that data model, dependencies, constraints, user-interface and input/output interfaces can be designed and realized.

It is convenient to do this from the *features* point of view. Features define the product how it can be designed and personalized by the customer. We take into account only features which depend on configuration (configurable features). They shall not be too small (so that we are not overwhelmed by details) and the estimation of their profits and costs must be reasonably possible. Examples:

- model type of a bicycle (mountain bike, city bike, or racing bike)
- size and material of a T-shirt
- type and power of the CPU in an electronic railroad interlocking system

A feature is implemented in the configurator data model as one or more configuration parameters (whose values are to be chosen by the customer during configuration), along with some internal data structures which carry all necessary information for a complete and consistent solution. Those are descriptions of the provided hardware, assembly possibilities, properties, costs, etc.

Dependencies and restrictions define the valid combinations of the configuration parameters and internal variables (e.g. choosing a certain module requires a special power supply).

Sales and marketing provide requirements for the user-interface. It must be specified, how the different features and their choices are presented to the customer. Answers for the following questions must be given:

- Is the configurator a web application or stand-alone?
- What are the skills of the customer (end customer, technical engineer)?
- Which widgets are to be used for the user-input?
- Are graphical representations and pictures necessary to present a certain feature?
- Is a preview picture of the current product state to be displayed?
- Are the current costs to be displayed?
- etc.

In addition, product design must specify all input/output interfaces of the configurator. Mainly in complex technical domains, a direct connection of the configurator to a PLM system (e.g. SAP) is required. Typical export data are hardware lists, assembly plans, and cost calculations.

All these requirements should be described in a schematic and structured form, so that the subsequent review steps can be managed efficiently.

Quick evaluation of configurator complexity

When a first version of the configurator requirement specification is available, product and configurator designers should evaluate the required features together. This is necessary to estimate benefits as well as cost needed to calculate the profitability. One goal of this evaluation is to find those

requirements with a realistic chance that a re-design (a simplification) cuts down the configuration costs without impairing the overall product functionality.

We recommend the evaluation of the product features along the following aspects:

- Is this feature necessary for the functioning of the product? Is it necessary for sales or marketing reasons? If the answer is a clear "yes", a detailed (and expensive) evaluation of this feature is not needed.
- What is the expected profit which this feature adds to the product? Since it is not always easy to estimate this value, we suggest to additionally specify the confidence in this estimation.
  Different to the costs criteria (see below), we have here only one accumulated value "expected profit". This value will usually be forcasted by the design team or by product management in a separate process step, taking several factors into account. We omit details here as we concentrate on the work of the configurator architect.
- What are the configurator costs for this feature?
  - for the data model part
  - for reasoning logics (constraints, rules)
  - for optimization criteria
  - for the user interface
  - for communication and data interfaces
  - for maintenance

An additional aspect to be considered is the costs for the user of the configurator. This may not be relevant for simple standard questions like "choose your preferred value from a list of possible values", but may be important for complex technical domains, where data which are difficult to collect must be provided by the user. Example from the railroad domain: The input of the reference positions of signals is required by certain accuracy. If this accuracy is in the range of a few meters, the positions can be read off from existing layout plans in a quite simple way. But if the required accuracy is decimeter, normally the track topology must be measured anew which is very time-consuming and expensive.

Suggested evaluation sheet

As an implementation and documentation for the process we propose an evaluation sheet as shown in figure 5. It contains all aspects mentioned in the previous section. Other aspects which affect the costs significantly can be added easily by extending the table when necessary - e.g. security (authorization, authentication) when it is relevant for the application.

The evaluation starts by writing the features which are subject to configuration one after the other in the first column of the table. They need not be too detailed - but the description must be clear enough for the participants to know their meaning. The level of refinement resembles that of a product backlog in Scrum style (see [Wikipedia:Scrum, 2009]). If necessary for accurate evaluation, the features may be split to more detailed ones later.

In the example in figure 5, the product is an individualized T-shirt which has an image or text printed on. Some data for the shirt and for the image are configurable. In addition, the configurator shall comprise some constraints and other features.

The next step is a quick assessment how necessary each feature is, e.g. from a technical view (like required by the product design), or by legal regulations, or by state-of-the-art standards.

Like all the other values, this is done in a simple scale coded as integer values: very large/high (4), large/high (3), medium (2), small (1), very small or null (0). We could use enumeration values like XL, L, M, S, and XS instead. Integers, however, have the advantage that they are easily ordered and can be summed and averaged (although we do not do so at present). In order to reach a sufficiently wide span of value sizes, we have chosen to use 5 values instead of only 4 values as typically used in the House of Quality approach in Quality Function Deployment (QFD), see [Wikipedia:HoQ, 2009].

Low values are marked by yellow or red, respectively - thus indicating features which are not as necessary as others.

Evaluation of configurability costs and gains

Product: *Individualized T-Shirt* values: 0 (very small) - 4 (very large)

		Profit		Costs							
Feature	necessary	expected	confidence	model	reasoning	optimization	user interface	data interfaces	maintenance	profitability	Comments
Shirt data (size, material, color, etc.)	4										*not evaluated*
Image (text, jpg, etc.)	4										*not evaluated*
Image data (size, color)	3	3	2	1	1		2			15	
Different colors of image and shirt	3	2	4	0	1		1			**20**	
Image size must fit to shirt	2	2	4	1	4				1	2	***complicated image processing***
Shirt on stock	4										*not evaluated*
Preview of configured shirt	3	**4**	3	1			1	1	1	**40**	
Discount (amount, special offers)	1	**3**	3	2	2		1	2	2	5	

Figure 5: Evaluation matrix

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Those marked in red are the favorites to be evaluated in depth. Features with the maximum value are absolutely necessary and need not evaluated. By that, the scope of evaluation is reduced significantly.

For the "red" and "yellow" features, the expected profit is entered next, together with the confidence in its rating - again in the scale 0-4 (XS, S, M, L, XL). High profits are printed in bold - thus giving the user a hint that higher costs could be accepted. In our example this is true for the feature "Discount" which has a high profit with high confidence and medium costs for various aspects.

Low and medium profits are marked with red and yellow, respectively - thus giving a hint where to look for cost reduction first (i.e. for cancelling configurability for this feature). In our example those are the constraints concerning color and size of shirts and images.

Now the configurator developers do a rough estimation of the implementation costs separately for the main aspects of the configurator - again in the scale 0-4 (XS, S, M, L, XL). Here it is not yet important to get "real" costs (not even story-points like in Scrum), but only a rough magnitude. The scale is to be seen relatively to the size of the product (meaning that large in the context of a small product would be small in the context of a complex product). Again, high values are marked with red, medium with yellow. The last column is meant for comments on the important values - in our example for the constraint on the size.

As an additional measure, the profitability is calculated by 10 * square of profit / sum of squares of each cost. Squaring is done in order to weight high values significantly higher - in analogy to the Quality of House values (0, 1, 3, 9). Nevertheless the value of the profitability is only a symbolic one, with values less than 5 to be considered very weak (marked red), values less than 10 weak (marked yellow) and those higher than 20 to be considered good (bold font).

The color code and the simple scale make it easy to identify the features with a bad smell, i.e. those which are not absolutely necessary, do not have a high profit, but have high costs or low profitability. They have to be analyzed in detail.

The following situations may arise:

- High profit, little costs, i.e. high profitability (like feature "Preview" in our example): no need to go into detail, just implement.
- High profit, high costs (like feature "Discount" in our example): verify profit expectations, estimate efforts in detail.
- Low profit, high costs, i.e. low profitability (like constraint on size in our example): no need to go into detail, discard feature (configurability). Only if it was rated with a very high necessity, consider simplifying (i.e. loosening the requirement for a "perfect" configurator) or search for alternatives.
- Low profit, low costs (like constraint on color in our example): consider implementation, if necessity is high enough.

Deep evaluation of hotspots

The hotspot features identified in the quick review step above (normally not more than a handful) must now be evaluated in deep detail. Costs and benefits must be estimated and compared. Often, this analysis depends on the number of systems which are expected to be sold. If the expected number of systems is below a certain break-even point, the feature is cost-ineffective (see the real-world example at the beginning).

If it turns out that an analyzed feature is cost-ineffective, product design and configurator design should together find an alternative. This could be either to completely abandon the feature, or to simplify it, e.g. by simplifying the user-interface design or by trimming down the possible options (which, of course, leads to reduced product flexibility).

Summary and outlook

We proposed a process for finding a design for a configurable product with a high profitability or for reducing configurator costs, respectively.

The process supports early involvement of configurator designers in product design thus identifying expensive features. It relies on two phases: quick evaluation of all configurable features, deep evaluation of hotspots.

At present, we do not have figures concerning the benefits of implementing this process for all our projects, but we have used it successfully in several situations. Furthermore, we are still adapting it to requirements and experiences arising in our project environments.

Areas of future work are:

- using a more detailed way for estimating the costs (e.g. story-points as used in Scrum)
- balance the costs for implementing the configurator and for users' configuration work (i.e. either invest more into the tool or invest more into engineers who use the tool)
- theoretical analysis of knowledge-base structures and dependencies to find “bad designs”
- check-list for deep analysis
- configuration anti-patterns

References

[Bessiere, 2006] Christian Bessiere. Constraint Propagation. In *Handbook of Constraint Programming*, pages 29-83. Elsevier, 2006.

[Blecker et al., 2007] Th. Blecker, K. Edwards, G. Friedrich, L. Hvam, F. Salvador. Innovative Processes and Products for Mass Customization. GITO 2007.

[Felfernig and Burke, 2008] Alexander Felfernig and Robin Douglas Burke. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*, Innsbruck, Austria, August 2008.

[Hvam et al., 2007] Lars Hvam, Niels Henrik Mortensen, Jesper Riis. Product Customization. Chapter *Specification Processes and Product Configuration*, pages 17-41. Springer 2007.

[Riitahuhta et al., 2000] A. Riitahuhta, A. Pulkkinen. Design for Configuration. A Debate based on the 5th WDK Workshop on Product Structuring. Springer 2000.

[Tiihonen et al., 2008] Juha Tiihonen, Alexander Felfernig, Markus Zanker, Tomi Männistö. *Proceedings of the Workshop on Configuration Systems*. 18th European Conference on Artificial Intelligence, July 2008.

[Watts and Haines, 2009] Geoff Watts and Jason Haines. Priority markets - A Free Market Approach to Managing the Product Backlog. *http://www.scrumalliance.org/articles/117-priority-markets*, February 2009.

[Wikipedia:HoQ, 2009] Wikipedia. House of Quality. *http://en.wikipedia.org/wiki/House_of_Quality*, March 2009.

[Wikipedia:Scrum, 2009] Wikipedia. Scrum (development). *http://en.wikipedia.org/wiki/Scrum_(development)*, March 2009.

Construction of Configuration Models

Lothar Hotz
HITeC e.V.
Department of Computer Science
University of Hamburg
hotz@informatik.uni-hamburg.de

Abstract

In this paper, a novel approach for creating configuration models is supplied by introducing a meta-knowledge base that enables the construction of configuration models. The meta-knowledge base represents all knowledge bases that can be expressed with a given configuration language, in the case of this paper, with the Component Description Language CDL. The meta-knowledge base itself is again represented with CDL and thus, at the metalevel it can use configuration tools that relay on CDL. With this approach inference techniques that are normally used for configuration of technical systems can be applied for the construction of configuration model, i.e. during knowledge acquisition and evolution.

1 Introduction

Knowledge-based configuration has its origin in the task of configuring physical components like drive systems [Ranze *et al.*, 2002] or elevators [Marcus *et al.*, 1988]. For example in [Günter, 1995] configuration is defined as "the composition of technical systems from parameterisable objects to a configuration, that fulfills a certain task" or Stefik defines in [Stefik, 1995] configuration tasks as tasks that "select and arrange instance of parts from a set". The focus is set on the composition of parts to aggregates and thus, on the compositional relation `has-parts`.

Naturally, in all approaches *descriptions* of objects are composed, not the physical objects themselves. By doing so, configuration can be understood as *model construction* [Buchheit *et al.*, 1995; Hotz and Neumann, 2005; Hotz, 2009]. From the configuration point of view model construction deals with the composition of arbitrary artifacts on the basis of a logical theory. Where a strict separation of the logical theory, i.e. the knowledge base or *configuration model*, and the logical model, i.e. the *configuration* or better *construction* is issued. Starting from a knowledge base a configuration system composes a construction that is consistent with the knowledge base, i.e. a logical model of the knowledge base is created.

Following this understanding of the configuration task the above mentioned `has-parts` relation is more a `has` relation, which is applied to various domains e.g. services [Tiihonen *et al.*, 2006], where e.g. a client *has* a certain insurance demand, or to software [Hotz *et al.*, 2004], where a software component *has* a certain feature, to scene interpretation [Hotz, 2006], where a certain scene description *has* observed or hypothized objects or actions. The `has` relation determines what part descriptions are to be integrated in a resulting construction (i.e. a system description). By taking up such a perspective sometimes considered conceptual mismatches [Tiihonen *et al.*, 2006], which may come up, when using configuration systems in non-physical domains, are avoided.

Taking a further step, one may look at configuration models as a type of software that is constructed during a knowledge-acquisition process. Thus, the questions arise: "Can the construction of configuration models be supported by configuration tools?" or "What are the parts that are composed in such an approach?" or "How does a configuration model that enables the configuration of configuration models (i.e. a *meta-configuration model*) look like?".

An application of such a meta-configuration model is naturally to support the knowledge-acquisition process needed for knowledge-based configuration systems. In a first phase of a knowledge-acquisition process the typically tacit knowledge about a domain is extracted by applying knowledge-elicitation methods and high interaction between a knowledge engineer and the domain expert (*knowledge-elicitation phase*). A model sketch is the result, which in turn is formalized during the *domain-representation phase*. During this phase a configuration model is created. The configuration model has to be expressed with the facilities of a configuration language. The meta-configuration model can be used to check such configuration models for being consistent with the configuration language. Thus, by using the meta-configuration model as a knowledge base of a configuration system, the domain-representation phase can be supported similarly to a configuration process.

In this paper, we will elaborate answers to the mentioned questions by first presenting a construction language, i.e. the *Component Description Language* CDL, which enables the description of domain objects (see Section 2). We than investigate in a concept for a configurator that enables the configuration of arbitrary configuration models, i.e. a meta configurator and its *meta-configuration model* (Section 3). We

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

partly implement such a meta configurator by using the configuration system KONWERK [Günter and Hotz, 1999]. A discussion and a summary is provided in Section 4 and Section 5 respectively.

2 The Component Description Language

2.1 A Sketch of CDL

The Component Description Language CDL introduced here is similar to existing other configuration languages as they are described in [Soininen *et al.*, 1998; Stumptner, 1997; Felfernig *et al.*, 2002; Cunis *et al.*, 1991; Günter, 1995]. The language mainly consists of two modeling facilities:

Concept Hierarchy Domain objects are described using *concepts*, a specialization hierarchy (based on the `is-a` relation), and structural relations. Concepts gather all properties, a certain set of domain objects has, under a unique name. A specialization relation relates a *super-concept* to a *sub-concept*, where the later inherits the properties of the first. 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 structural relationships like `has-feature` or `has-concept`. Parameters specify domain-object attributes with value intervals, sets of values (enumerations), or primitive values. Parameters and structural relations of a concept are also referred to as *properties* of the concept. *Instances* are instantiations of the concepts and represent concrete domain objects. When instantiated, the properties of an instance are initialized by the values or value ranges specified in the concepts.

Constraints *Constraints* summarize conceptual constraints, constraint relations, and constraint instance. *Conceptual constraints* consists of a condition and an action part. The condition part specifies a *structural situation* of instantiated concepts. If this structural situation is fulfilled by some instances (the instances *match* the structural situation), *constraint relations* that are formulated in the action part are instantiated to *constraint instances*.[1]

Constraint relations can represent restrictions between properties like `all-isp` or `create-instance`. Figure 1 shows the definition of the predefined constraint relations used in the following. The constraint relations `create-instance` and `integrate-instance` are later used for constructing structural relations and thus, provide main facilities for creating resulting constructions.

Knowledge processing is done by the *inference techniques* taxonomical reasoning, value-related computations like interval arithmetic, establishing structural relations, and constraint propagation. The structural relation as the main machinery causes the constructive notion of the language: if such a relation is given between a concept c and several relative concepts r, depending on what exists first as instances in the construction (c or one or more of the relative concepts r), instances for the other part of the relation are created and the construction increases.

A configuration process (or better *model-construction process*) applies these inference techniques in a certain way and constructs step-by-step a construction. At each step a *current partial construction* is issued. The knowledge needed for this processing is modeled by further modeling facilities, i.e. a *task description* and *procedural knowledge*. The task description is given in terms of an aggregate, which must be configured (the goal), and possibly additional restrictions such as choices of parts, prescribed properties, etc. Furthermore, the configuration process provides a stepwise composition of a construction. Each step is one of the following kinds of

[1] Thus, conceptual constraints are similar to rules, except the action part yields to instantiations of constraint relations not to changes in objects like rules do.

`integrate-instance <set1 instance1 instance2 set2>`
Integrate `instance1` into `set2` and `instance2` into `set1`. `instance1` and `instance2` than have established structural relations among them.
`all-isp <set type>`
Ensures that all objects in `set` are subtype of `type`.

Figure 1: Some predefined relations of CDL.

```
(define-relation :name has-elements
  :inverse element-of
  :mapping m-n)

(define-concept :name Door
  :specialization-of Opening
  :element-of
   ((:type Scene-Aggregate :min 0 :max 2)
     :=
    (:type Entrance :min 0 :max 1)
    (:type Balcony :min 0 :max 1)))

(define-concept :name Entrance
  :specialization-of Opening
  :has-elements
   ((:type Scene-Object :min 1 :max 3)
     :=
    (:type Door :min 1 :max 1)
    (:type Wall :min 0 :max 1)
    (:type Roof :min 0 :max 1)
    (:type Stairs :min 0 :max 1)))

(define-concept :name Balcony
  :specialization-of Scene-Aggregate
  :has-elements
   ((:type Scene-Object :min 1 :max 3)
     :=
    (:type Railing :min 1 :max 1)
    (:type Window :min 0 :max 1)
    (:type Door :min 0 :max 1)))
```

Figure 2: Example of a concept definition in CDL. The structural relation `has-elements` is defined, which relates one aggregate with several parts and one part with several aggregates. Furthermore, several concepts are defined with number restricted structural relations. The right side of the operator `:=` consists of the super-concept of all relative concepts and the total minimal and maximal number of those concepts. The left side restricts the number of each type.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

construction steps: *top-down structuring* (e.g. *aggregate instantiation*), *bottom-up structuring* (e.g. *part integration*), *instance specialization*, and *parameterization*. A step reduces a property value of an instance to a subset or finally to a constant. Procedural knowledge declaratively describes the selection of those steps and the inference techniques to be used.

Thus, adding facilities for task descriptions and procedural knowledge to CDL one gets a complete configuration language like the Configuration Knowledge Modeling Language CKML described in [Hotz *et al.*, 2006]. However, in this paper we concentrate on the first mentioned modeling facilities of concepts and constraints and try to express them with CDL again. CDL is fully described in [Hotz, 2009].

2.2 Parts of the Metamodel of CDL

For expressing the goals of this paper, we give more details for the definition of structural relations in CDL.

Concepts and constraints of CDL are given by an abstract syntax (see Figure 3), a concrete syntax (see Figure 2 for an example[2]), and several consistency rules.

For describing CDL with an abstract syntax, we introduce three facilities: a *knowledge element*, a *taxonomical relation* between knowledge elements, and a *compositional relation* between knowledge elements. However, these facilities are not to be mixed up with the above mentioned CDL facilities: *concepts*, a *specialization relation*, and *structural relations*. See Figure 3, a CDL concept is represented with a knowledge element of name $concept$, a CDL structural relation is represented with the knowledge element $relation\text{-}descriptor$. The fact that CDL concepts can have several structural relations is represented with a compositional relation with name $has\text{-}relations$. Similarly parameters are represented. Thus, the above mentioned modeling facilities of CDL are represented with these metalevel facilities.

In Figure 4 further parts of the metamodel are given for representing structural relations. The fact that a concept is related by a structural relation of other concepts (the relative concepts) is represented with three knowledge elements and three compositional relations in a cyclic manner.

Several consistency rules define the meaning of the syntactic constructs. For the structural relation, one rule defines that the types of the relative concepts of a structural relation have to be sub-concepts of the concept on the left side of the operator := ($rule$-5). Additionally consistency rules are given that check CDL instances, e.g. one rule defines when instances match a conceptual constraint ($rule$-6).

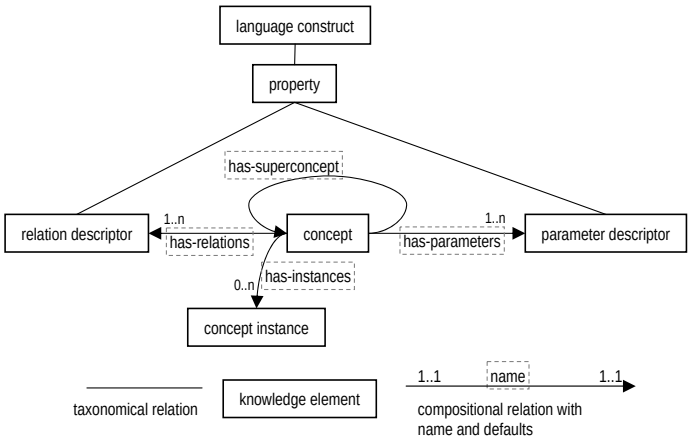

Figure 3: Metamodel for a concept of CDL

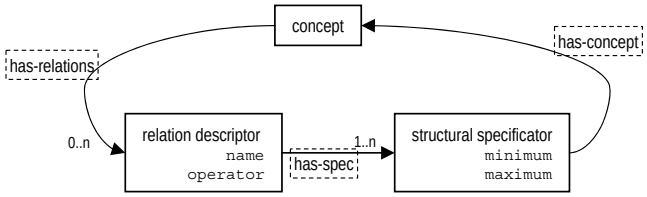

Figure 4: Metamodel for a structural relation of CDL

3 A Concept for a Meta Configurator

3.1 What CDL provides

The main feature of CDL is given by the use of its inference techniques like constraint propagation (see Section 2). By representing the knowledge of a domain with modeling facilities of CDL (like concepts with specialization, structural, and constraint relations) those inference techniques can be applied for model construction. This representation is basically a generic description of domain objects of a domain at hand. For the representation of concrete domain objects this description is instantiated. Such instances are related to each other through the relations. Furthermore, instances can be checked for concept membership.

What does this mean for the representation of CDL in CDL? In this case, the domain consists of CDL knowledge bases. A Meta-CDL knowledge base (Meta-CDL-KB) generically represents all knowledge bases that can be expressed with CDL (see Section 3.2). Doing so, the above mentioned inference techniques can be used for CDL knowledge bases. For example, a knowledge base G for a certain domain D (like the façde domain) can be created through instances of concepts of the Meta-CDL-KB. Examples are $concept\text{-}mm$ for representing concepts and $parameter\text{-}mm$ for representing parameters (see Figure 5). These concepts are related to each other e.g. $concept\text{-}mm\ has\text{-}parameters\ parameter\text{-}mm$. Through a configuration process, which applies the inference techniques of CDL in a certain way, a knowledge base G of a domain D can be created. Furthermore, a given knowledge base can be checked, if it can be constructed in principal with the Meta-CDL-KB, i.e. if it is a CDL knowledge base. An architecture that supports these tasks is given in Section 3.3.

[2]For the examples, the façade domain is used where the domain objects are parts of houses like balcony, door, stories. The purpose is to construct scene interpretations from façade images (see [Hotz, 2008]).

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

3.2 CDL in CDL

For the presentation of CDL in Section 2 three facilities are used, i.e. *knowledge elements*, *taxonomical relation*, and *compositional relation*. Those are mapped to the CDL constructs *concept*, *specialization relation*, and *structural relation* respectively. For example, the knowledge elements for describing the CDL facilities in Figure 3 concept, relation, and parameters are represented with the *meta*concepts $concept\text{-}mm$, $relations\text{-}descriptor\text{-}mm$, and $parameter\text{-}mm$ (see Figure 5).

Furthermore, the consistency rules of CDL have to be represented. This is achieved by defining appropriate constraints, which in turn use value-related computations (Section 2) for computing appropriate values. In Figure 6 a conceptual constraint is represented, which checks the types of a structural relation.[3]

Also instances can be represented on the metalevel by including a metaconcept $instance\text{-}mm$ for them. Through these instances also conceptual constraints and their matching instances can be represented (see Figure 7). Furthermore, the fact that instances fulfill a certain conceptual constraint is represented through establishing appropriate relations using the constraint relation `integrate-instance`. Note, that also self references can be described, e.g. a $concept\text{-}mm$ is related to itself via the $has\text{-}superconcept\text{-}mm$ relation (see also the loop in Figure 3).

3.3 A Meta-Knowledge Server

In this section, we describe the use of the Meta-CDL-KB for the construction of a CDL knowledge base for arbitrary domains. This use is realized by introducing a Meta-Knowledge Server (MKS) for supervising the construction of the CDL knowledge base. The MKS handles the current status of the evolving CDL knowledge base during the knowledge-acquisition process as well as the current status of CDL instances during a configuration process.

In Figure 8, we sketch the first case. the MKS uses the Meta-CDL-KB as configuration model M. Furthermore, MKS uses the model-construction process for supervising the construction of a configuration model G of a given domain. If e.g. a concept c of the domain is defined with `define-concept` the MKS is informed. The MKS observes the activities during the construction of the CDL knowledge base, i.e. during the domain-representation phase. The MKS

- supplies services like $check\text{-}knowledge\text{-}base$, $add\text{-}conceptual\text{-}constraint$,
- creates appropriate instances of Meta-KB-CDL metaconcepts (e.g. $concept\text{-}mm$ or $conceptual\text{-}constraint\text{-}mm$),
- applies the typical model-construction process by using procedural knowledge,
- uses constraint propagation for checking the consistency rules,
- completes the CDL knowledge base by including mandatory parts, and

[3] For a complete mapping of the CDL consistency rules to conceptual constraints see [Hotz, 2009].

```
(define-concept :name concept-mm
  :specialization-of named-domain-object-mm
  :concept-of-dom-mm (:type domain-mm)
  :superconcept-of-mm
   (:type concept-mm :min 0 :max inf)
  :in-some-mm (:type some-mm :min 0 :max inf)
  :has-superconcept-mm
   (:type concept-mm :min 0 :max 1)
  :has-relations-mm
   (:type relation-descriptor-mm :min 0 :max inf)
  :has-parameters-mm
   (:type parameter-mm :min 0 :max inf)
  :has-instances-mm
   (:type instance-mm :min 0 :max inf))

(define-concept :name relation-descriptor-mm
  :specialization-of named-domain-object-mm
  :relation-of-mm (:type concept-mm)
  :has-left-side-mm (:type some-mm :min 1:max 1)
  :has-right-side-mm (:type some-mm :min 0:max inf)
  :has-relation-definition-mm
   (:type relation-definition-mm :min 1:max 1))

(define-concept :name some-mm
  :specialization-of domain-object-descriptor-mm
  :parameters ( (lower-bound [0 inf])
                (upper-bound [0 inf]))
  :in-relation-left-mm
   (:type relation-descriptor-mm)
  :in-relation-right-mm
   (:type relation-descriptor-mm)
  :some-of (:type concept-mm))

(define-concept :name instance-mm
  :specialization-of named-domain-object-mm
  :instance-of-dom-mm (:type domain-mm)
  :instance-of-mm (:type concept-mm)
  :matching-instance-of-mm
   (:type conceptual-constraint-mm)
  :has-relations-mm
   (:type relation-descriptor-mm :min 0 :max inf)
  :has-parameters-mm
   (:type parameter-mm :min 0 :max inf))
```

Figure 5: Formalizing the knowledge elements shown in Figure 3 with CDL concepts.

- checks consistency of created parts of G.

The MKS integrates concepts of G as instances of metaconcepts of M in the current partial construction, which represents G at the metalevel. Changes in already defined concepts are represented by backtracking on the metalevel model-construction process. By using backtracking approaches, especially dependency-based backtracking (see [Hotz *et al.*, 2004; Ferber *et al.*, 2002]), dependencies of modeling decisions can be managed. Thus, changes of a CDL configuration model (i.e. during the knowledge-acquisition process or during evolution [Männistö and Sulonen, 1999]) are basically changes of G, i.e. changes of the currently constructed model on the metalevel. *Evolution is backtracking on the metalevel.*

Besides this construction of CDL knowledge bases the MKS can scrutinize CDL instances, which are created during a model-construction process. For this task, MKS is supplied with such instances (see Figure 9) and creates instances of the metaconcept $instance\text{-}mm$. By doing so consistency rules for instances represented as conceptual constraints on the metalevel can be checked.

Looking from the MKS perspective the construction of a CDL knowledge base can be seen as the interpretation of

```
(define-conceptual-constraint :name consistency-rule-5
  :structural-situation
   ((?c    :name concept-mm)
    (?rd   :name relation-descriptor-mm
           :relation-of-mm ?c)
    (?svt  :name some-mm
           :in-relation-left-mm ?rd)
    (?stdi :all :name some-mm
           :in-relation-right-mm ?rd))
  :constraint-calls
   ((all-isp ?stdi ?svt)))
```

Figure 6: A conceptual constraint representing consistency rule 5. The concepts of the right side of a relation descriptor has to be sub-concepts of the left side.

```
(define-concept :name conceptual-constraint-mm
  :specialization-of named-domain-object-mm
  :structural-situation
   (:type concept-expression-mm :min 1 :max inf)
  :constraint-calls
   (:type constraint-call-mm :min 1 :max inf)
  :matching-instances
   (:type instance-mm :min 0 :max inf))

(define-conceptual-constraint
  :name instance-consistency-rule-6
  :structural-situation
   ((?cc :name conceptual-constraint-mm)
    (?i :name instance-mm
        :self (:condition
                (instance-matches-cc-p *it* ?cc))))
  :constraint-calls
   ((integrate-instance-relation ?i
      (matching-instance-of ?i)  ?cc
      (matching-instances ?cc))))
```

Figure 7: Describing conceptual constraints with there matching instances on the metalevel.

an external system similar to the interpretation of an outside scene. MKS observes the construction of the CDL knowledge base and tries to integrate the observations by using the Meta-CDL-KB. This task is similar to scene interpretation where evidence in a scene is interpreted by constructing an interpretation on the basis of a model for anticipated scenes. Thus, similar implementations can be applied for the MKS like top-down and bottom-up structuring, spontaneous instantiation, and merging (see also [Hotz and Neumann, 2005; Hotz, 2006]).

We implemented parts of the meta configurator with the configuration system KONWERK [Günter and Hotz, 1999]. The Meta-CDL-KB could be used for constructing knowledge bases for a PC-domain. However, first experiments demand the need of highly interactive facilities for visualizing the complex relational structures of meta-level instances, e.g. visualizing which `some-mm` instance belongs to the which `concept-mm` during the configuration process.

4 Discussion

The model-construction view as it is emphasized in this work is a systematic generalization of structure-oriented configuration like it is provided by [Günter, 1995; Soininen *et al.*, 1998] and others. This is mainly achieved by focusing on the structural relation, which ensures existence of instances in the resulting construction. These instances build the constructed

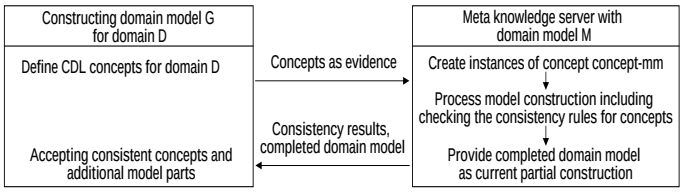

Figure 8: Meta-knowledge server applied to constructing a configuration model.

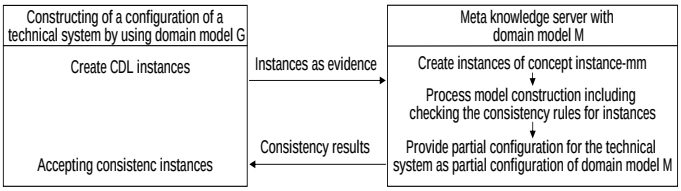

Figure 9: Meta-knowledge server applied to constructing a configuration.

model. This model is a description of the desired technical system, which is used e.g. for the production process. In this sense, also other configuration approaches like connection-based [Mittal and Frayman, 1989], resource-based [Heinrich and Jüngst, 1991], or function-based [Najman and Stein, 1992] can be seen as model-construction approaches. This view to configuration enables the concise application of configuration tools in environments like services, software, or like in this paper on metalevels. However, for model construction seldom supported facilities are needed as there are:

- The representation and processing of cyclic relational structures. Those techniques are sometimes avoided like in [Magro *et al.*, 2002; Arlt *et al.*, 1999].
- Sophisticated control mechanisms like bottom-up and top-down construction. Typically only top-down is emphasized in configuration systems.
- Connecting model construction with other external systems or the real world *during* the model-construction process demands spontaneous instantiation of concepts. In configuration systems only for creating the knowledge base external data like databases are used and the resulting configuration is exported for producing the configured system.

The creation of a metamodel for CDL with the aid of CDL has its tradition in self-referencing approaches like Lisp-in-Lisp [Brooks *et al.*, 1983] or the metaobject protocol, which implements CLOS (the Common Lisp Object System) with CLOS [Kiczales *et al.*, 1991]. Such approaches demonstrated the use of the respective language. In case of CDL the meta-knowledge server is enabled. It makes strong use of the implemented inference techniques of CDL like constraint propagation.

The meta-knowledge server is basically an implementation of a configuration tool on the basis of the Meta-CDL-KB, i.e. of a configuration model. A typical configuration tool is implemented with a programming language and an object model

implemented with it. During this implementation one has to ensure correct behavior of model construction and the inference techniques. By using CDL this behavior (e.g. the consistency rules) is declaratively modeled, not implemented. The bases for this realization are of course the implementation of value-computation methods and constraint mechanisms.

The here introduced meta-knowledge base has some relations to metamodeling approaches like described in [OMG, 2006; 2007; Kühne, 2006; Hesse, 2006]. Thus, in the following, we take a first glance to some aspects of metamodeling (see also [Asikainen and Männistö, 2009] for a deeper analysis of metamodeling). The main task of metamodeling is to specify modeling facilities that can be used for defining models, see for example [OMG, 2007]: "A metamodel is a model that defines the language for expressing a model". Or compiled to terms used here: "The Meta-CDL-KB is a configuration model that defines CDL, which in turn is used for expressing a configuration model" (see Figure 10). However, the notion of modeling is still not finally fixed (see [Kühne, 2006; Hesse, 2006]), or as [Hesse, 2006] says: "A complete and unanimously accepted theory of modeling is still emerging.". Besides these theoretical issues, in our approach a more pragmatical and operational view is taken, i.e. how to apply a metamodel for supporting the use of the language the metamodel defines. From this perspective, let us examine the Requirements Specification Language RSL [Kaindl *et al.*, 2007; Śmiałek *et al.*, 2007; Hotz *et al.*, 2009]. A metamodel defines elements typically used for specifying requirements as their are use-cases, scenarios etc. A tool (*RSL-Tool*) enables a requirements engineer to express her use-cases etc. through a user interface and the tool constructs a requirements specification expressed in RSL. Thus, the metamodel of RSL is used by the implementor of the RSL-tool, which in turn ensures a requirements specification that is compliant to the metamodel of RSL (see Figure 10). However, the implementation is done manually and specific for the RSL-metamodel. The creation of metamodel compliant models can be supported by a configuration tool.

A configuration tool supplies mainly three tasks:

1. It enables the expression of a configuration model that is consistent with the configuration language, which the tool implements. For this task, it performs consistency checks of given configuration models (or parts of it) with the language specification.
2. On the basis of the configuration model, the configuration tool supports the creation of constructions that are consistent with the configuration model. For this task, the tool interprets the logical expressions of the configuration model and creates constructions according to these definitions.
3. The configuration tool supplies user interfaces for expressing the configuration model and for guiding the construction process. The configuration model can be typically given in textual forms or with graphical user interfaces that enable the creation of concepts and constraints.

Thus, a configuration tool contains means for supporting the step from a domain model to a system specific model (see Figure 10). By introducing configuration models in the model chain as presented in Figure 10, an additional level is introduced, i.e. the domain-model level. This level represents all systems of a domain. The model for a system is an instantiation of the domain model. This instantiation is computed by a configuration tool. In our metamodeling approach based on the Meta-CDL-KB this instantiation facility is used for supporting the step from the configuration language to the domain model, i.e. the domain-representation phase. By applying the configuration tool to a domain model that contains every model of a language, i.e. by applying it to the Meta-CDL-KB, the construction of a domain model of an arbitrary domain is supported. This is achieved because of the general applicability of the language constructs of CDL, which are based on logic (see Section 2). Furthermore, other advantages of configuration tools, like a declarative representation of the configuration model, or the use of the inference techniques can thus applied to the Meta-CDL-KB.

A similar approach as supplied by the meta-knowledge server is provided by [Kienzler, 2000] who uses meta planning. A primary construction process is supported by a secondary analysation process on the metalevel. The configuration process is controlled by a meta planner. The meta planner is strongly coupled with the configuration process. However, it is realized by a further external implementation not in the configuration language itself.

Other approaches like [Dietrich *et al.*, 2004] also use a meta-model approach for supporting the configuration process. However, by using a configuration language for expressing the metamodel, in our approach a configuration tool can directly applied for making use of the metalevel.

5 Summary

The paper shows how a configuration language can be expressed with its own representation facilities. Thus, the parts that are composed in such a case are the modeling facilities the configuration language supplies, i.e. concepts, parameters, constraints etc. The configuration model contains concepts, parameters, constraints that again represent concepts, parameters etc. By doing so, inference techniques that are provided by the language can be used for constructing configuration models and thus, support the knowledge-acquisition process. In this case, the configuration tool is mainly used for checking the consistency of the constructed configuration models. Thus, the use of the inference techniques support the formal basis of such processes. Further work will emphasize user-interface tools that support the visualization and manipulation of highly structured relationships including cyclic structures that occur on the metalevel.

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, 2009] T. Asikainen and T. Männistö. A metamodelling approach to configuration knowledge representation. In *Proc. of the*

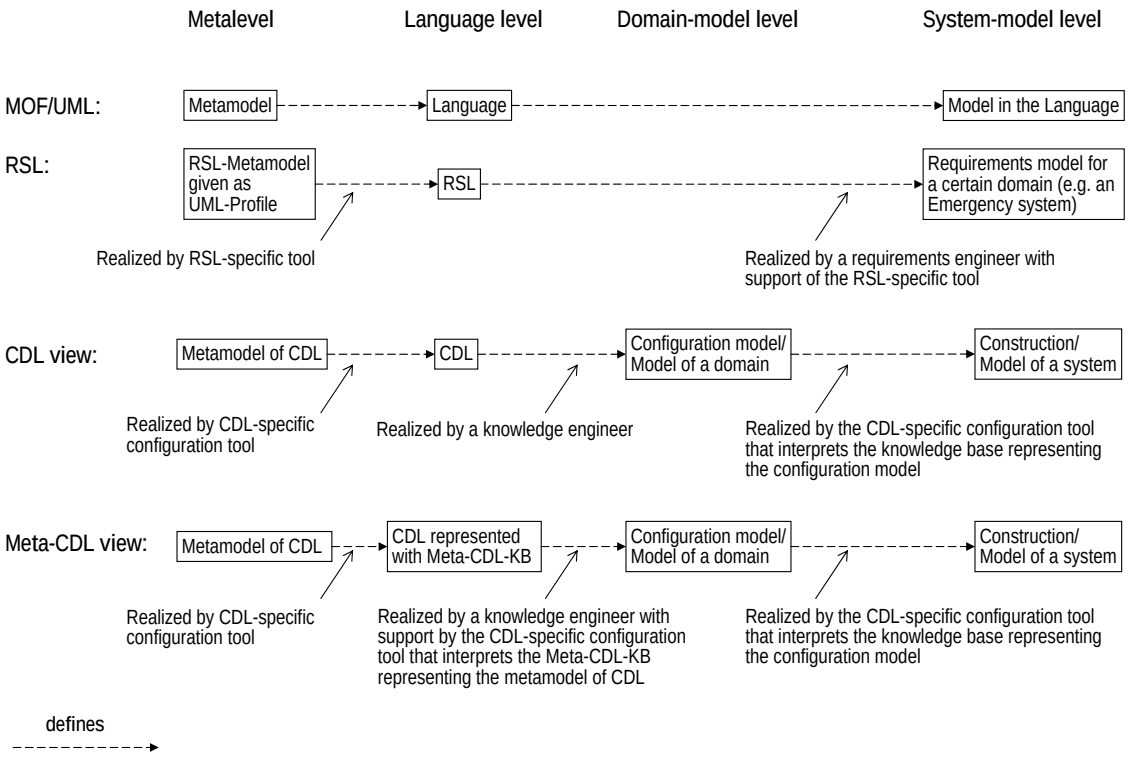

Figure 10: Relations between metamodels and domain models and their tool support.

Configuration Workshop on 22th European Conference on Artificial Intelligence (IJCAI-2009), Pasadena, California, 2009.

[Brooks *et al.*, 1983] R.A. Brooks, R.P. Gabriel, and L. Steele Jr. Lisp-in-Lisp: High Performance and Portability. In *Proc. of Fifth Int. Joint Conf. on AI IJCAI-83*, 1983.

[Buchheit *et al.*, 1995] M. Buchheit, R. Klein, and W. Nutt. Constructive Problem Solving: A Model Construction Approach towards Configuration. Technical Report TM-95-01, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, January 1995.

[Cunis *et al.*, 1991] R. Cunis, A. Günter, and H. Strecker (Hrsg.). *Das PLAKON-Buch*. Springer Verlag Berlin Heidelberg, 1991.

[Dietrich *et al.*, 2004] A.J. Dietrich, W. Hümmer, and Ch. Meiler. Meta model based Configuration Approach for mass-customizable Products and Services. In *Proceedings ot the 4th Workshop on Information Systems for Mass Customization (ISMC 2004)*, Madeira Island, Portugal, 2004.

[Felfernig *et al.*, 2002] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. A Joint Foundation for Configuration in the Semantic Web. In *Proc. of the Configuration Workshop on 15th European Conference on Artificial Intelligence (ECAI-2002)*, pages 89–94, Lyon, France, July 21-26 2002.

[Ferber *et al.*, 2002] A. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Re-engineering a Legascy Product Line. In *Proc. of 2nd Software Product Line Conference (SPLC-2)*, Lecture Notes in Computer Science, pages 235–256, San Diego, CA, USA, August 19-23 2002. Springer Verlag.

[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.

[Günter, 1995] A. Günter. *Wissensbasiertes Konfigurieren*. Infix, St. Augustin, 1995.

[Heinrich and Jüngst, 1991] M. Heinrich and E. Jüngst. A Resource-based Paradigm for the Configuring of Technical Systems from Modular Components. In *Proc. of 7th IEEE Conf. on Artificial Intelligence for Applications (CAIA'91)*, pages 257–264, Miami Beach, Florida, USA, February 24-28 1991.

[Hesse, 2006] W. Hesse. More matters on (meta-)modelling: remarks on thomas kühne's "matters". *Journal on Software and Systems Modeling*, 5(4):369–385, 2006.

[Hotz and Neumann, 2005] L. Hotz and B. Neumann. Scene Interpretation as a Configuration Task. *Künstliche Intelligenz*, 3:59–65, 2005.

[Hotz *et al.*, 2004] L. Hotz, T. Krebs, and K. Wolter. Dependency Analysis and its Use for Evolution Task. In *18th*

Workshop, New Results in Planning, Scheduling and Design (PuK2004). University of Oldenburg, 2004.

[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 *et al.*, 2009] L. Hotz, K. Wolter, S. Knab, and A. Solth. Ontology-based Similarity of Software Cases. In *submitted to International Conference on Knowledge Engineering and Ontology Development*, 2009.

[Hotz, 2006] L. Hotz. Configuring from Observed Parts. In C. Sinz and A. Haag, editors, *Configuration Workshop, 2006*, Workshop Proceedings ECAI, Riva del Garda, 2006.

[Hotz, 2008] L. Hotz. Modeling, Representing, and Configuring Restricted Part-Whole Relations. In J. Tiihonen, editor, *Configuration Workshop, 2008*, Workshop Proceedings ECAI, Patras, 2008.

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

[Kaindl *et al.*, 2007] Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John P Brogan, Kizito Ssamula Mukasa, Katharina Wolter, and Thorsten Krebs. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, CA, 1991.

[Kienzler, 2000] F. Kienzler. *Synthesis versus Analysis in model-based AI-Planning Systems? DIAKON - a auto-adaptive diagnostic Solution Approach for Action Planing and Configuration Problems (in German)*. PhD thesis, University of Ulm, 2000.

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

[Magro *et al.*, 2002] D. Magro, P. Torasso, and L. Anselma. Problem Decomposition in Configuration. In *Configuration Workshop, 2002*, Workshop Proceedings ECAI, Lyon, France, 2002.

[Männistö and Sulonen, 1999] T. Männistö and R. Sulonen. Evolution of Schema and Individuals of Configurable Products. In *Proc. of ECDM'99 - Workshop on Evolution and Change in Data Management*, Versailles, France, November 15-18 1999. Springer Verlag.

[Marcus *et al.*, 1988] S. Marcus, J. Stout, and J. McDermott. VT: An Expert Elevator Designer that uses Knowledge-based Backtracking. *AI Magazine*, pages 95–112, 1988.

[Mittal and Frayman, 1989] S. Mittal and F. Frayman. Towards a Generic Model of Configuration Tasks. In *Proc. of Eleventh Int. Joint Conf. on AI IJCAI-89*, pages 1395–1401, Detroit, Michigan, USA, 1989.

[Najman and Stein, 1992] O. Najman and B. Stein. A Theoretical Framework for Configurations. In *Proc. of Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: 5th International Conference, IEA/AIE-92*, pages 441–450, 1992.

[OMG, 2006] OMG. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*. Object Management Group, 2006.

[OMG, 2007] OMG. *Unified Modeling Language: Infrastructure, version 2.1.1, formal/07-02-06*. Object Management Group, 2007.

[Ranze *et al.*, 2002] 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. In *Eighteenth national conference on Artificial intelligence*, pages 845–852, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[Śmiałek *et al.*, 2007] Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, Albert Ambroziewicz, and Tomasz Straszak. Complementary use case scenario representations based on domain vocabularies. *Lecture Notes in Computer Science*, 4735:544–558, 2007.

[Soininen *et al.*, 1998] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a General Ontology of Configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), 12*, pages 357–372, 1998.

[Stefik, 1995] M. Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, CA, 1995.

[Stumptner, 1997] M. Stumptner. An Overview of Knowledge-based Configuration. *AI Communications*, 10(2):111–126, 1997.

[Tiihonen *et al.*, 2006] J. Tiihonen, M. Heiskala, K.-S. Paloheimo, and A. Anderson. Configuration of Contract Based Services. In C. Sinz and A. Haag, editors, *Configuration Workshop, 2006*, Workshop Proceedings ECAI, Riva del Garda, 2006.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration

Andreas Hau Nørgaard, Morten Riiskjær Boysen, Rune Møller Jensen
IT University of Copenhagen, Denmark
ahn@itu.dk, boysen@itu.dk, rmj@itu.dk

Peter Tiedemann
Configit A/S
pt@configit.com

Abstract

This paper demonstrates how to lower the average response time of search-based interactive configurators using over and under approximations of the configuration problem represented by binary decision diagrams (BDDs) on problems where fast configurators using monolithic BDDs are intractable. The paper introduces several ways to build the approximations and our experimental evaluation on industrial data shows that a search driven extension of the approximations substantially outperforms both purely search-based and purely BDD-based interactive configurators.

1 Introduction

Configuration Problems (CPs) occur whenever a product that can be configured needs to be tailored to specific requirements. Examples of this ranges from buying t-shirts or computers online to configuring large wind turbines all the way up to large data centers. These problems are a prime target for AI techniques either because their complexity is so high that even a trained user cannot oversee all requirements or because a user is untrained and must be guided e.g. during a purchase in an online store. Thus, in the case of the online store, solving configuration problems is important because it allows companies to use less resources for support, thus reducing their cost, and because it can give them a competitive advantage by making the purchase as simple and straightforward as possible. In the case where a trained person needs to configure large machineries, configuration becomes a question of increasing the productivity of the operator. Perhaps even more importantly, configuration technology aids in preventing costly invalid configurations by catching errors during the configuration instead of after the manufacturing process has begun when the cost of fixing the mistakes can be very high.

1.1 Interactive Configuration

A special form of the Configuration Problem is *Interactive Product Configuration* (IPC). In dealing with these problems, a user is interfacing directly with the configurator and needs to see the consequences of the choices he makes. This is in contrast with an automated system where, e.g., a partial assignment is given and the configurator has to complete the product. The requirements of interactive configuration are:

Complete Meaning that all valid configurations can be reached by the user. If the configurator is not complete, certain valid product configurations cannot be configured. This is very unfortunate, e.g. if the configurator is used to configure products in an online store as it would mean that some valid product configurations cannot be sold.

Backtrack Free Whenever the user selects a value, all values that cannot extend the current partial assignment to a solution will be removed. This means that a partial assignment is *always* extendable to a solution, hence the user never needs to backtrack. This is not a strict requirement but it is a very desirable property of an interactive configurator since backtracking can be very tedious to the user.

Fast Response Times It is important to display the consequences of an assignment to the user as fast as possible so the user does not grow impatient with the configurator. How fast this must happen depends on the type of user and the environment the configurator is used in.

Arbitrary Order of Assignments The user must be allowed to make assignments to the variables in any order the user likes.

1.2 Search

Using backtracking search to solve various kinds of CSPs (not just configuration) is a commonly used technique. When performing the search, the *solver* chooses a variable and branches on it, thereby obtaining a reduced problem, after which the solver chooses a variable again to branch on, continuing in that fashion until either the CSP is proven unsatisfiable or a solution has been found. Much work has been put into improving the basic search by using consistency techniques and heuristics to reduce the search tree. These techniques are invaluable in a modern solver. Unfortunately, since the search tree is potentially exponential in size, these techniques give very little guarantees on the performance with

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

respect to computation time.

1.3 Compiled Representations

Another way to solve a CP is through *compilation*: The entire set of solutions to the problem is stored in some compact form. This is the preferred technique for interactive configuration since the representation of solutions only has to be built once and can be shared by users afterwards. There are many ways to store the solution space, including *Binary Decision Diagrams* (BDDs) [Bryant, 1986], *Multi-valued Decision Diagrams* (MDDs) [Kam *et al.*, 1998], and *Cartesian Product Tables* (CPTs) [Møller, 1995].

There exists polynomial time algorithms to do valid domain calculations on these data structures, thus giving good guarantees on the performance. However, there is no such thing as a free lunch; since CPs are NP-complete, the compilation phase might take exponential time, and, even worse, the output might take up exponential space. It can be shown that BDDs and MDDs require exponential space for the `all-different` constraint [van Hoeve, 2001], which is vital in modeling configuration problems involving placement.

The main contribution of this paper is to show a way to implement a complete backtrack free interactive configurator capable of handling configuration problems with better run-time performance than a purely search-based configurator or a BDD-based configurator on a realistic problem inspired by a real-world data center configuration problem. In this paper, we show that it is possible by using a combination of both techniques.

The results were obtained by combining the Gecode solver [Schulte *et al.*, 2009] with CLab [Jensen, 2004], which is a BDD-based configuration library. We used BDDs for storing approximations of the configuration problem that makes it possible to *eliminate* some of the searches performed by the solver in the valid domain computation. Our computational results are on industrial data from Configit A/S [Andersen and Hulgaard, 2007] and show that a substantial lower average response time of the configurator can be achieved in this way compared to a purely BDD-based or a purely search-based approach.

The idea to use approximations to speed up interactive configurations was first presented in [Tiedemann, 2008], but the author did not provide any tests or implementation. A previous study in [Subbarayan *et al.*, 2004] compared a purely search-based configurator with a BDD-based configurator, showing that the latter had better run-time performance in most cases. However, the study did not involve global constraints and all problem instances could be represented in a monolithic BDD. Furthermore, some ideas were presented in the study for creating an efficient search-based configurator; these ideas are extended in this paper. Previous research has been conducted in extracting no-goods from constraints represented as BDDs [Subbarayan, 2008] but the paper does not mention the use of search results for building the BDDs. It focuses entirely on extracting small no-goods from a static BDD. In [Subbarayan *et al.*, 2006] the authors used BDDs to build a hybrid SAT solver. However, the work does not include configuration problems, nor does it use BDDs to store the results of time consuming searches. Thus, the main contributions of this paper are the implementation and test of a hybrid configurator using BDDs for good- and no-good recording (good-recording is described in [Cheng and Yap, 2006]) and a solver for problems that are intractable to be represented entirely as BDDs.

The remainder of this paper is organized as follows: In Section 2, we present the concept of interactive configuration and show two different ways of implementing it. One uses a search-based solver the other uses BDDs. In Section 3 we show how to combine a solver with BDDs to obtain better run-time performance than what is possible if using each technique alone. Section 4 shows how to use and build the approximations and Section 5 show an alternative way to build the approximations by using the search results. In Section 6 we show the empirical results obtained and, finally, Section 7 concludes on the results. These results are obtained on a data center configuration example, provided by Configit, that models the configuration of a large-scale data center.

2 Backtrack Free Interactive Configuration

A configuration problem C is a triple (X, D, F), where

- X is a set of variables $x_1, x_2, \ldots, x_n$
- D is the Cartesian product of their finite domains $D = D_1 \times D_2 \times \ldots \times D_n$
- $F = \{f_1, f_2, \ldots, f_m\}$ is a set of propositional formulas over atomic propositions $x_i = v$, where $v \in D_i$, specifying the conditions that the variable assignments must satisfy. Each formula is inductively defined by $f \equiv x_i = v \mid f \wedge g \mid f \vee g \mid \neg f$

Furthermore, the solution space S of C is defined as the set of all complete assignments that satisfy all requirements. In other words, S is a set containing all the valid configurations to the configuration problem.

An interactive product configurator (IPC) enables the configuration process as described in Section 1.1. The main task of an IPC is to compute the set of *valid domains* VD for a configuration problem where $VD = \{VD_1, VD_2, \ldots, VD_n\}$. VD_i denotes the valid domain for variable x_i where $VD_i \subseteq D_i$. Thus, any assignment $\{(x_i = v) \mid v \in VD_i\}$ will never require the user to backtrack. However, a user might choose to remove an assignment if the consequences of the assignment are not desirable, thus allowing manual backtracking.

Let VD_i denote the valid domain for variable x_i, then $VD_i \subseteq D_i$. Thus, any assignment $\{(x_i = v) \mid v \in VD_i\}$ will never lead to the user backtracking.

These are the assignments that are guaranteed to be extendable to a solution. Once the valid domains have been computed the user can make a valid assignment. These two steps are repeated until the product has been configured (all variables have been assigned), see Algorithm 1.

2.1 Search-based Configuration

The simplest way, albeit very naive, to calculate the valid domains using a search based solver is shown in Algorithm 2. This algorithm enumerates all possible assignments $(x_i =$

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Algorithm 1 An informal definition of the IPC algorithm

```
1: procedure IPC
2:     read and process configuration problem
3:     while not all variables assigned do
4:         VD ← COMPUTEVALIDDOMAINS
5:         user makes a valid assignment
```

v_{ij}) where $\{(x_i, v_{ij}) \mid x_i \in X, v_{ij} \in VD'_i\}$. Before the assignments are enumerated, the solver is instructed to prune as many values as possible using the PROPAGATE procedure. This runs all propagators to fix-point, thereby possibly removing values that are never part of the solution. Since the propagators do not generally yield generalized arc consistency with respect to the conjunction of all constraints there can still be values left in VD' that cannot be part of a solution. An assignment is added to the existing configuration problem where after this augmented problem is tested for satisfiability. If the augmented problem is satisfiable it is known that $v_{ij} \in VD_i$. This step is repeated for all possible assignments. This method performs a search for all (x_i, v_{ij}) that *might* be valid. Hence it performs $\sum_{i=1}^{n} |D_i|$ searches. In Section 3 we describe several ways to improve this initial algorithm.

Algorithm 2 A naive way to determine the valid domains

```
1: procedure CVD-NAIVE(C)
2:     VD' ← PROPAGATE(C)
3:     for all x_i ∈ X do
4:         VD_i ← ∅
5:         for all v_ij ∈ VD'_i do
6:             if C|x_i = v_ij is satisfiable then
7:                 VD_i = VD_i ∪ v_ij
```

2.2 BDD-based Configuration

A binary decision diagram (BDD) is a rooted directed acyclic graph. A BDD has one or two terminal nodes[1], labeled 1 or 0, and a set of variable nodes. The terminal node labeled 0 is denoted by T_0 and the terminal node labeled 1 is denoted by T_1. Each variable node is an internal node in the BDD and has exactly two outgoing edges marked *low* and *high*. A BDD represents a boolean function f on a set of n boolean variables $f : \mathbb{B}^n \to \mathbb{B}$. The value of the boolean function, given an assignment of the variables, can be found by recursively traversing the BDD. The traversal begins at the root and continues to a terminal node. Whenever a variable is assigned to true the high branch of the corresponding node along the path is taken. If a variable is assigned to false the low branch of the corresponding node is taken. If the path ends at a terminal labeled 1 the assignments means the value of the function is true. If the path ends at a terminal labeled 0 the value of the function is false. An introduction to BDDs and some of the basic algorithms used on them can be found in [Andersen,].

A reduced ordered binary decision diagram (ROBDD) [Bryant, 1986] is a BDD with the two additional properties of being ordered and reduced. A BDD is said to be ordered when all paths from the root node to a terminal node respect a given variable ordering, meaning that the variables associated with the nodes will be met in the order defined. A BDD is said to be reduced when all nodes where the low and high branches leading to the same node are removed and when all nodes are unique. A node is unique if no other node exists that has the same associated variable and branches to the same destinations on the high and low branches respectively. If such a duplicate node exist it can be removed by collapsing the two nodes into a single node. In the rest of this paper we only use ROBDDs and since it is a De facto standard to use the abbreviation BDD to mean a reduced ordered binary decision diagram we will follow the convention and consequently write BDD from now on when we refer to a reduced ordered binary decision diagram.

BDDs have been widely used in verification, but it was later discovered that they are also well suited for configuration problems [Hadzic *et al.*, 2004]. However, a configuration problem can have variables with finite integer domains whereas a BDD only has boolean variables. Fortunately, an integer variable x_i can be encoded efficiently in a BDD using $k_i = \lceil log_2 |D_i| \rceil$ boolean variables $x_i^0, \ldots, x_i^{k_i - 1}$. Furthermore, these variables are places in *layers* so all boolean variables encoding the same finite domain variable are placed in the same layer and all finite domain variables define a unique layer. The BDD nodes comprising the layer i are denoted by V_i.

Example: A simple example of a CP is shown in Figure 1. The constraints corresponds to the relations $x_1 < x_2$, $x_1 < x_3$ and $x_2 \neq x_3$.

$$
\begin{aligned}
X &= \{x_1, x_2, x_3\} \\
D &= \{\{1,2,3\}, \{1,2,3\}, \{1,2,3\}\} \\
C &= \{((x_1, x_2), \{(1,2), (1,3), (2,3)\}), \\
&\quad\ ((x_1, x_3), \{(1,2), (1,3), (2,3)\}), \\
&\quad\ ((x_2, x_3), \{(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)\})\}
\end{aligned}
$$

Figure 1: A simple CP

This simple example can be represented by a BDD as seen in Figure 2.

2.3 BDD-based Valid Domain Computation

In order to use BDDs for IPC we need to be able to perform assignments and compute the valid domains. Assignments can be made by using the standard APPLY BDD operation by conjoining the BDD representing $x_i = v$ onto the BDD G_1 representing the current solution space restricted by the assignments made so far in the configuration process. The complexity of the restriction operation for variable x_i in BDD G_1 is thus $O(|G_1| \cdot \lceil log_2 |D_i| \rceil)$.

The *Compute Valid Domains* (CVD) operation determines from a BDD representing a configuration problem what values that are guaranteed to be in the solution space and extensible to a full assignment. The valid domain computation works

[1] A terminal node has out-degree zero

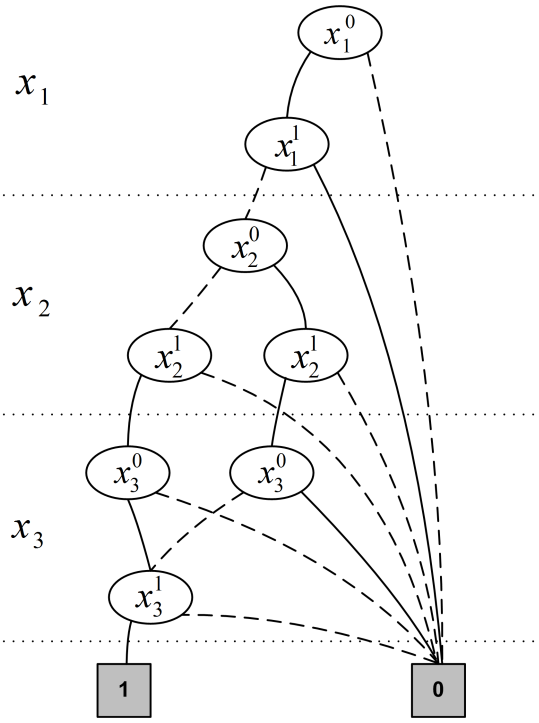

Figure 2: The CP from Figure 1 encoded as a BDD with high edges shown as solid lines and low edges shown as dashed lines. The integer variables are encoded in little-endian format and the variable ordering is $x_1^0 < x_1^1 < x_2^0 < x_2^1 < x_3^0 < x_3^1$. The layers are shown as the horizontal dashed lines.

by *probing* the layers. Each value $v \in D_i$ is tested by traversing the i'th layer from all nodes in V_i with incoming edges from the preceding layers until support has been found or all nodes in a layer have been probed. If all traversals for v end in T_0 there is no support for v, so $v \notin VD_i$. To avoid probing the same nodes while checking for support for v, all nodes are checked if they have already been probed with v. If a node has already been checked, the traversal is stopped since the traversal will end in T_0 (Since it has been probed earlier, that probe failed. Otherwise, support would have been found and the probing for v would stop). The checking ensures that a node in v_i is only checked once for each $v \in D_i$. Thus, the worst case complexity of the compute valid domains operation over a BDD is $O(\sum_{i=1}^{n} |V_i| \cdot |D_i|)$. For a detailed explanation of the algorithms used for computing valid domains over a BDD see [Hadzic *et al.*, 2007].

Example: Assume that a valid domain computation is performed on the BDD from Figure 2 and that the algorithm is about to test the valid domain of x_3. First, the value 1 is tested. There are two nodes with incoming edges from preceding layers. The probing starts from the left-most node with the binary encoding of 1. In this probing, the traversal ends up in T_0 after having gone though the node labeled x_3^1. The right-most node in the layer is then probed, but with the binary encoding of 1, this leads directly to T_0. Thus, $1 \notin VD_3$. When probing for support for 2, the left most node is again chosen as the start, but this leads directly to T_0. The right-most node is then used to start a traversal, and after passing through the node labeled x_3^1 the traversal ends in T_1, so $2 \in VD_3$. When checking for the value 3 the traversal beginning from the left-most node ends in T_1, so there is also support for 3. The result of the probing is that $VD_3 = \{2, 3\}$.

3 BDD and Search-based Hybrid Configurator

To be able to make a fair comparison between the performance of a search-based configurator and our hybrid configurator, the algorithm behind the search-based configurator needs to be improved. In the following we will present a series of improvements to the naive algorithm shown in Algorithm 2.

Only a very small part of the information provided by the solver in Algorithm 2 is actually used, namely whether a single value is part of the valid domain of a variable or not. The search result has a lot more information than that; all the assignments in the search result are part of the valid domains of their respective variables. The naive algorithm can therefore be improved in two ways. Firstly, the result is traversed and all assignments are stored as part of the valid domains. Secondly, before a search is started it is checked whether the value v_{ij} has already been verified to be part of VD_i. If it is the search is simply skipped.

Additionally, we can use former valid domain results to speed up the valid domain computation since when an assignment is made the solution space can never grow, meaning that $S_{|x_i=v_{ij}} \subset S$. This implies that a search is redundant if an assignment $(x_i = v_{ij})$ has been discovered as invalid by a previous search but has not been pruned by propagation yet

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

because of the non-increasing property of the solution space. This information can be fed back to the solver and propagation mechanism by posting the unary inequality constraint $x_i \neq v_{ij}$ whenever a search fails. The value v_{ij} is thus removed from the current domain of x_i and hence augments the solver with information that propagation alone could not detect. This can improve propagation and increase the search performance for other variables.

Finally, it is an invariant in the configurator that once the domain has size 1 it cannot shrink any more. If it could, the configurator would not be backtrack free. It is therefore possible to skip the search if all but the last value v_i^{last} in VD'_i has been found to be invalid. Therefore, $x_i = v_i^{\text{last}}$ must be a valid assignment.

3.1 Hybrid Configurator

The search preventing hybrid configurator utilizes a combination of BDDs and search-based techniques. The basic idea is to avoid as many searches as possible by using BDD-based approximations.

Over- and Under-Approximations

An over-approximation of a CP with solution space S is a CP with solution space $S_o \supseteq S$. An under-approximation of a CP with solution space S, is a CP with solution space $S_u \subseteq S$. Given an over-approximation CP_o of a CP and a partial assignment (PA), CP_o can be used to determine if PA is not extendable to a solution in CP. However, it cannot be used to determine whether it is extendable to a solution. Conversely given an under-approximation CP_u of a CP, CP_u can be used to determine if a partial assignment PA is extendable to a solution in CP but CP_u cannot be used to determine if PA is not extendable to a solution. Thus, if the two approximations are used together a search is only needed when neither approximation is able to determine whether PA is definitely extendable to a solution or definitely not.

This relation is shown in Figure 3. The picture shows the Cartesian product of the domains of the variables in a CP. The grey area to the left of the curved line represents the solution space and the white area to right of the curved line represents the non-solutions (the set of full assignments that violate one or more constraints). The box with the bold dashed line represents the under-approximation and the box with the bold solid line represents the over-approximation. As the drawing shows we need to perform a search for elements in the set $S_o \backslash S_u$.

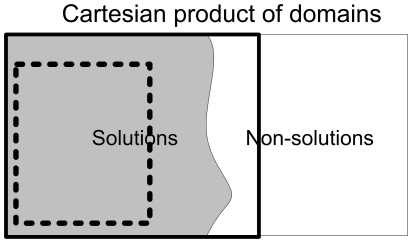

Figure 3: The relation between the solution space and the over- and under-approximation for a CP.

4 Using the Approximations

As described in Section 2.1 a search-based configurator uses a two-step approach, by first propagating and then searching whenever an assignment has been made and you want to find the new valid domains. Using an over-approximation changes the two-step approach into what we could call a two-and-a-half-step approach because we need to utilize the over-approximation after the propagation step in order to avoid the search step as often as possible. As mentioned, we can avoid a search for all values not in the valid domains of the over-approximation restricted to the current partial assignment since these will clearly not be in the domain of the CP. Furthermore, all values in the valid domain of the under-approximation restricted to the current partial assignment can be added to the valid domains of the CP before the search phase because $S_u \subseteq S$.

The valid domain computation including all the optimization from Section 3 and the approximations can be seen in Algorithm 3 where BDD^o is the BDD representing the over-approximation and BDD^u is the BDD representing the under-approximations.

Algorithm 3 Solver-based valid domain computations algorithm using an over- and under-approximation

```
1:  procedure CVD-SP(C)
2:      VD' ← PROPAGATE(C)
3:      VD^o ← COMPUTEVALIDDOMAINS(BDD^o)
4:      VD^u ← COMPUTEVALIDDOMAINS(BDD^u)
5:      VD ← VD^u
6:      for all x_i ∈ X do
7:          if |VD'_i| = 1 then
8:              VD_i = VD'_i
9:              continue
10:         for all v_ij ∈ VD'_i do
11:             if v_ij ∈ VD_i then
12:                 continue
13:             else if VD_i = ∅ ∧ v_ij = v_i^last then
14:                 VD_i ← {v_ij}
15:             else if v_ij ∉ VD_i^o then
16:                 continue
17:             else if C_|x_i=v_ij is satisfiable then
18:                 S ← solution to search
19:                 for all (x_k, v_k) ∈ S do
20:                     VD_k = VD_k ∪ v_k
21:             else
22:                 C ← C_|x_i≠v_ij
23:         VD' ← PROPAGATE(C)
```

4.1 Constructing an Over-Approximation

Given a CP with solution space S, an over-approximation of this is also a CP (which we call CP′). Since we represent our over-approximation by a BDD, a simple way to construct CP′ is by removing all constraints from the original CP that are intractable to represent in a BDD. This would make CP′ less restricted than the original CP and therefore $S \subseteq S_o$, which was the requirement.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

4.2 Constructing an Under-Approximation

We construct the under-approximation CP$''$ by putting additional constraints on the CP we are approximating. As mentioned, `alldifferent` constraints put an exponential lower bound on the number of nodes in a BDD. Since the under-approximation needs to be at least as strict as the original CP we cannot remove the `alldifferent` constraint from the under-approximation. We have therefore investigated what additional restrictions to add to a CP that contains an `alldifferent` constraint in order to limit the amount of nodes generated in the BDD. The way we have attained this is by limiting the combinations of values the variables involved in an `alldifferent` constraint can have. This is done by limiting the domain of each variable in such a way that the union of the limited domain of all the variables is still the complete domain.

Example: If, for example, we have 10 variables with $D_i = \{1, 2, \ldots, 10\}$, we can *slice off* one value from each variable so the domains become $D_1 = \{1, 2, \ldots, 9\}$, $D_2 = \{1, 2, \ldots, 8, 10\}$, $D_3 = \{1, 2, \ldots, 7, 9, 10\}$, etc. If we continued with the example and we wanted to do a *domain slice* of half the values, the domains would become $D_1 = \{1, 2, \ldots, 5\}$, $D_2 = \{2, 3, \ldots, 6\}$, $D_3 = \{3, 4, \ldots, 7\}$, etc. To avoid making the under-approximation too narrow, we always construct the complement set of values when we slice off values of the domains. In the last example given, the complement domain values would be $D'_1 = \{6, 7, \ldots, 10\}$, $D'_2 = \{1, 7, \ldots, 10\}$,
$D'_3 = \{1, 2, 8, 9, 10\}$, etc. After slicing the domains, the `alldifferent` constraint becomes

$$Alldiff(x_1, x_2, \ldots, x_n) \wedge$$
$$(x_1 \in D_1 \wedge x_2 \in D_2 \wedge \ldots \wedge x_n \in D_n \vee$$
$$x_1 \in D'_1 \wedge x_2 \in D'_2 \wedge \ldots \wedge x_n \in D'_n)$$

5 Search Driven Approximations

An alternative way of constructing the approximations is by building it over time. We can achieve this by noting each time we perform a search to find a solution in the CP given a partial assignment (PA) that takes an excessive amount of time and does not find a solution. Each time this happens, we can conjoin an additional constraint on to the over-approximation of the form ¬PA. By doing this we are using the over-approximation as a way of performing no-good recording [Hawkins and Stuckey, 200].

In the case of the under-approximations, we are interested in the case where we perform a search that takes an excessive amount of time and actually finds a solution. In this case we can extend the under-approximation by setting it equal to the disjunction of the solution found and the existing under-approximation.

6 Results

In this section we compare the search-preventing configurator(s) with the purely search-based configurator. When building the over- and under-approximations for the various problems using the search results as described above, we added all results that took more than $10\,\mathrm{ms}$.

The different configurators use these abbreviations:

CVD-S The purely search-based configurator.

CVD-R The search-preventing hybrid configurator described using the statically built over- and under approximations.

CVD-CB The search-preventing hybrid configurator using over- and under-approximations built purely from search results.

CVD-WB The search-preventing hybrid configurator using the statically built over- and under approximations augmented with results gathered while performing search. This configurator is thus a combination of the two described above.

We have not made experiments with a purely BDD-based configurator since it is intractable to represent all but the smallest problems in a monolithic BDD.

One of the problems we have tested the hybrid configurator on is the data center configuration problem. The data center configuration problem is a modular problem in the sense that we are configuring a data center that consists of a series of racks, that each consists of a series of servers, that each consists of a series of boards. The problem contains `alldifferent` constraints to ensure that specific pieces of hardware are placed only once. In particular, each of the servers in a rack can only be placed once, and each of the boards of a particular server can only be placed once. Furthermore, the data center configuration problem contains local constraints to ensure the right configuration of specific hardware pieces. For our purpose we have focused on the configuration of a single rack.

We have performed tests with 5 differently sized data center configuration problems. The sizes are 4, 6, 8, 10 and 11 servers. For each of the problem instances we have created the under-approximation by slicing half of the domains of the variables in the `alldifferent` constraint representing the constraint that each server can only be used once. It is worth noting that the maximum number of servers we can have in a monolithic BDD representing the data center configuration problem is 10. For this reason, we have tried to see how little we could slice of the domains in the under-approximation representing the data center configuration problem with 11 servers, and still be able to contain it in the under-approximation BDD. The limit we found is 4 values sliced of each domain of 11 values. The problem instances in the experimental results are listed as `dcNN-SS` where `NN` denotes the number of servers in the problem instance and `SS` denotes the number of values sliced from the domains. All tests were performed on an Intel Core 2 Duo 6600 $2.4\,\mathrm{GHz}$ Dual Core Processor workstation with $2\,\mathrm{GB}$ RAM running Windows XP Professional SP3.

The result of these tests are shown in 3 tables where Table 1 shows the maximum valid domain computation times. As can be seen, CVD-CB performs the best overall. We attribute this to the fact that CVD-CB cuts of all those searches that takes too long and has a relatively small size BDDs compared to the BDDs used by CVD-WB and CVD-R.

The average valid domain computation times are shown in

Table 2. It is apparent that for the smallest problems the overhead of using BDDs is not made up by the searches skipped. We see, however, that when the problem size grows it more than makes up for it. CVD-CB and CVD-WB perform best. Furthermore, we see that purely search-based (CVD-S) and CVD-Reg are about the same.

In Table 3 we see as expected that the maximum number of searches is performed in CVD-S and the least is performed in CVD-WB. If we assume that the searches skipped are evenly distributed among those that are fast and those that are slow then this is an important fact since it decreases the likelihood that CVD-WB run into a search that takes an extremely long time.

	Max CVD time [ms]			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	**16**	**16**	**16**	**16**
dc6-3	**31**	32	32	32
dc8-4	47	63	47	**47**
dc10-2	94	63	**47**	157
dc10-5	79	79	78	**63**
dc11-4	110	172	**78**	329
dc11-5	125	157	**79**	188

Table 1: Max time of the valid domain computations of the search preventing CVD algorithms.

	Average CVD time [ms]			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	**2**	4	6	5
dc6-3	**7**	11	10	11
dc8-4	17	22	16	**15**
dc10-2	28	25	**21**	24
dc10-5	28	33	24	**22**
dc11-4	44	55	**31**	42
dc11-5	44	54	**30**	35

Table 2: Average time of the valid domain computations of the search preventing CVD algorithms.

	Searches performed			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	1082	785	927	**578**
dc6-3	2116	2110	**1138**	1183
dc8-4	3796	3800	1406	**1281**
dc10-2	5222	3016	1601	**1084**
dc10-5	5358	5268	1656	**1477**
dc11-4	7560	7362	1690	**1622**
dc11-5	7560	7384	1543	**1444**

Table 3: Searches performed of each of the search preventing CVD algorithms.

7 Conclusion

This paper has introduced three new algorithms that combine BDDs and backtracking search for backtrack-free interactive configuration. Our results show that the performance of these algorithms dominate purely search- or BDD-based approaches.

7.1 Directions and Future Work

Another approach to constructing hybrid configurators is to augment a propagator-centric solver by BDD-based propagators. The idea is that several constraints can be represented by a single BDD and thereby improve propagation strength since there is strong n-consistency between the constraints in the BDD.

We tested this idea of implementing a BDD-propagator in Gecode but no improvement of runtime was achieved even though we did get stronger propagation.

Future work could go into exploring new ways of constructing the approximations, which enable them to be as close to the original problem as possible and at the same time limit the amount of space needed to represent them. To achieve this, new data structures could be tested for representing the approximations. Interesting data structures could be MDDs [Kam *et al.*, 1998], Tree-of-BDDs [Subbarayan, 2005], and cartesian product tables [Møller, 1995]. It could also be investigated whether it would be beneficial to use different data structures for two approximations. Furthermore, experiments should be done on other problems to further validate the techniques used in the paper.

References

[Andersen,] Henrik Reif Andersen. An introduction to binary decision diagrams. `http://www.configit.com/fileadmin/Configit/Documents/bdd-eap.pdf`.

[Andersen and Hulgaard, 2007] Henrik Reif Andersen and Henrik Hulgaard. Configit software, 2007.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, aug 1986.

[Cheng and Yap, 2006] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, pages 78–82. IOS Press, 2006.

[Hadzic *et al.*, 2004] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune Møller Jensen, Henrik Reif Andersen, Henrik Hulgaard, and Jesper Møller. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, pages 131–138. DTU-tryk, 2004.

[Hadzic *et al.*, 2007] Tarik Hadzic, Rune Møller Jensen, and Henrik Reif Andersen. Calculating valid domains for BDD-based interactive configuration. *CoRR*,

abs/0704.1394, 2007. informal publication; informal publication.

[Hawkins and Stuckey, 200] Peter Hawkins and Peter J. Stuckey. A hybrid BDD and SAT finite domain constraint solver. In P. Van Hentenryck, editor, *Proceedings of the Practical Applications of Declarative Programming, 8th International Symposium*, volume 3819 of *LNCS*, pages 103–117. Springer, 200.

[Jensen, 2004] Rune M. Jensen. CLab: A C++ library for fast backtrack-free interactive product configuration. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, page 816. Springer, 2004.

[Kam *et al.*, 1998] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4:9–62, 1998.

[Møller, 1995] Gert Møller. *On the Technology of Array Based Logic*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1995.

[Schulte *et al.*, 2009] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. Software download and online material, 2009. `http://www.gecode.org`.

[Subbarayan *et al.*, 2004] Sathiamoorthy Subbarayan, Rune M. Jensen, Tarik Hadzic, Henrik R. Andersen, and Henrik Hulgaard. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application*, pages 97 – 111, aug 2004.

[Subbarayan *et al.*, 2006] Sathiamoorthy Subbarayan, Lucas Bordeaux, and Youssef Hamadi. On hybrid SAT solving using tree decompositions and BDDs. Technical Report MSR-TR-2006-28, Microsoft Research (MSR), March 2006.

[Subbarayan, 2005] Sathiamoorthy Subbarayan. Integrating csp decomposition techniques and bdds for compiling configuration problems. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[Subbarayan, 2008] Sathiamoorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with BDDs. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2008.

[Tiedemann, 2008] Peter Tiedemann. *Compiled Data Structures and Global Constraints in Constraint Processing*. PhD thesis, ITU, 2008.

[van Hoeve, 2001] Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. informal publication.

Industrial requirements for interactive product configurators

Matthieu Quéva[*] and **Christian Probst**[*] and **Per Vikkelsøe**[†]
[*]DTU Informatics, Technical University of Denmark
{mq,probst}@imm.dtu.dk
[†]Microsoft Development Center Copenhagen
pvikkels@microsoft.com

Abstract

The demand for highly customized products at low cost is driving the industry towards Mass Customization. Interactive product configurators play an essential role in this new trend, and must be able to support more and more complex features. The purpose of this paper is, firstly, to identify requirements for modern interactive configurators. Existing modeling and solving technologies for configuration are then reviewed and their limitations discussed. Finally, a proposition for a future product configuration system is described.

1 Introduction

Increasing market demand concerning customization and market pressure from competitors force enterprises to adapt their production and selling processes. Indeed, today's customers demand products with lower prices, higher quality, and faster delivery, but they also want products customized to match their unique needs. In many industrial areas, Mass Production is nowadays replaced by Mass Customization [Pine, 1993], which provides customers with highly customized products and low unit costs. One of the essential tools enabling Mass Customization is a product configuration system. In a product configuration system, a configurable product is defined by a set of components, options, or more generally attributes that can be chosen by the user. Some of these attributes are bound together by constraints limiting the number of possible combinations. The configuration task thus takes as input a model representing the structure and the constraints of the product (*product knowledge*), and aims at finding a configuration satisfying all the constraints defined in the model, as well as the requirements given by the end-user. It can also output a price, or a specification of the product to be manufactured, usually as a *bill-of-materials* and *operations routes*. Interactive configurators display the possible combinations of the product's components and options to an end-user. When the user chooses among the possibilities, the configurator computes the consequences of these choices on the possible values available for the other attributes for example. In this paper, we will focus on this type of configurators. Two main challenges arise when dealing with interactive product configuration. At *modeling time*, there is a need to find efficient and easy-to-use ways for the design engineers to express the product knowledge. The more complex the product is, the more important this phase is. The second issue concerns how to solve, at *configuration time*, the constraints expressed at modeling time. When the different attributes of the product are instantiated, there is a need for an efficient solving engine, capable of solving all types of constraints defined previously. In such an interactive process, the end-user should be assisted through meaningful explanations and indications on how to satisfy his requirements in the case the solution is not directly available.
Solving the configuration problem has received a lot of attention from the research area [Amilhastre *et al.*, 2002; Mailharro, 1998; Mittal and Falkenhainer, 1990], while the modeling problem has been less covered [Aldanondo *et al.*, 2003; Felfernig *et al.*, 2002]. In this paper, we aim at identifying requirements for industrial use of product configurators. We then present a review of different techniques and technologies currently available for both modeling and solving the configuration problem. Finally, we propose new directions to explore for building state-of-the-art configurators.

2 Requirements analysis

In this chapter, we use a Home Multimedia Station (HMS) as a case study to discuss the requirements analysis for product configurators. The requirements are derived from various literature as well as discussion with industrial partners. We first present general requirements in the first section, followed by more specific features. For most of the requirements, an example is given through the case study.

2.1 General modeling requirements

A modeling environment for product configuration should:

- be *easy-to-use*: The persons that will interact with the modeling environment are usually design engineers, often possessing only basic programming skills. The modeling environment should therefore be accessible without advanced training in programming, and support easy development through tools for a fast implementation. Also, the terms used should be based on a widely accepted terminology, e.g. following Soininen et al.'s ontology of configuration [1998].

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

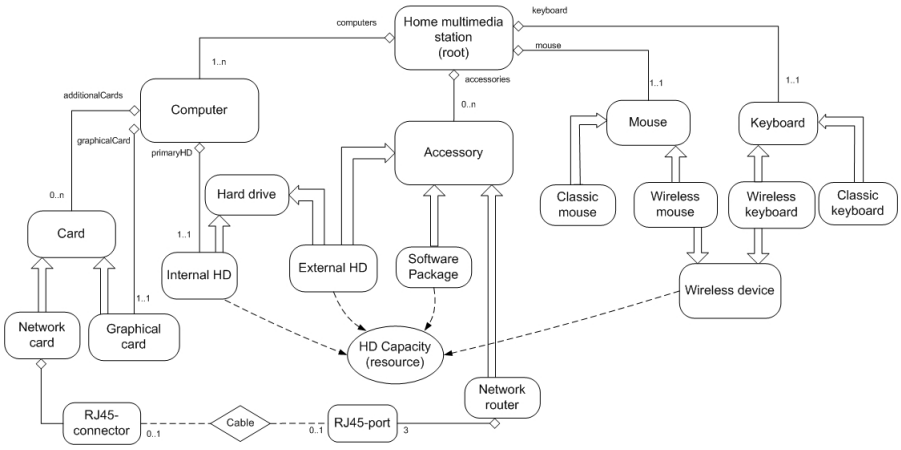

Figure 1: Structure of the Home Multimedia Station (HMS) case study[1]

- support *object-oriented modeling*: This approach has been favored by many researchers ([Hvam *et al.*, 2008; Rossi *et al.*, 2006]): it is indeed very suitable for product modeling, as product components can naturally be seen as objects. The object-oriented structure of the Home Multimedia Station can be seen in Figure 1.
- provide a *graphical representation*: This is important to the user for an easy understanding and a lower maintenance effort [Janitza *et al.*, 2003]. Graphical modeling languages will be presented in Section 3.
- be *extensible*: Companies use many applications around configurators. The system should provide an easy integration of CAD tools, databases, ERP or other systems in the configuration process.

2.2 Structure modeling

One part of modeling in configuration deals with representing the structure of the product. Several key features can be highlighted:

- *(dynamic) partonomy* (or part-of) relations define a subcomponent hierarchy in the product model. The multiplicity of these relations corresponds to the number of subcomponents to consider, which allows the reuse of component models. The possibility of specifying an unbound multiplicity permits to have a dynamic structure. The indefinite maximum in the "accessories" partonomy in the Home Multimedia Station models the possibility of adding a potentially unlimited number of accessories depending on the user's requirements.
- *taxonomy* (or kind-of/specialization) relations permit the use of generic base components to group features that are common to several subcomponents, which makes modeling and maintenance of the model much easier. Both the wireless and the classical mouses are specializations of the mouse component in the case study.
- *component groups* are a simple yet important feature in product modeling when it comes to product maintenance. Indeed, this makes it much easier to organize product knowledge data, as it allows to structure the model and its components according to specific criteria. The HMS model could be split into three groups: Computer Parts, Accessories, and Input Devices.
- definition of *units*: A product can be complex and can contain more than one data type with a specific unit. It should then be possible to declare different units, in order to make the model more realistic and the maintenance easier. The "price" of the Computer would be in *dollars*, while the "size" of the Internal HD would be in *inches*, although they are both real numbers.
- *connection ports* represent non-hierarchical relations between components that can be located in different subtrees of the model. Specific data can also be added to these relations, like the Cable component for the connection between the "RJ-45 Connector" and the "RJ-45 Port" in the HMS model.
- *default values* permit to provide the end-user with a capable default configuration very quickly, while still allowing the user to change some attributes.
- *hidden and locked attributes*: Using locked attributes to provide read-only information to the end-user or hidden attributes for internal computations offer increased flexibility to the model designer while reducing the complexity of the model for the customer.
- *production attributes*: Industrial product configurators are usually meant to be integrated with production management software, like ERPs. This includes mapping the configuration output to Bill-Of-Materials (BOMs) and operations routes that can be used in sales and manufacturing. Allowing the definition of production attributes that model how the BOMs and routes will be constructed from the product model's components is a great step towards an automatic generation of these production data.

[1] Part-of relations are represented with UML-like aggregations symbols and multiplicity, while taxonomy relations are represented using plain arrows. Dashed arrows represent use of resources.

2.3 Constraint modeling

Another important aspect of product modeling is the definition of constraints on the model. Requirements for constraint modeling include:

- a panel of *built-in functions and constraints* available to the modeler: aside from simple arithmetic and logical constraints, advanced functions (e.g. sum), constraints (e.g. allEqual) or quantification (e.g. forAll) provide a much greater support to the product modeler.
- *table constraints*: More and more real product data is coming from tables representing allowed combinations of attributes/components. The ability to declare table constraints (or product catalogs) directly (instead of more complex formulas) simplifies by far the creation and usability of the model.
- *Continuous domains* for attributes give a much more precise representation of specifically tailored products for example, or just attributes involved in advanced arithmetic formulas.
- Products are often configured according to the *resources* they produce/consume. The modeling of these resources and how they increase/decrease must be defined to handle such cases. The HMS model involves a resource called "HD Capacity", which is produced by the hard drives and consumed by both wireless devices' drivers and software packages.
- *Soft (or prioritized) constraints* are constraints that may be violated if they are overridden by a user selection or indirectly as a consequence of a constraint with higher priority. Modeling with soft constraints permits to introduce a notion of uncertainty that can be used by the modeler to guide the configuration process with recommendation or simulate preferences for example. Such a constraint could be used in the HMS case study to recommend the user to choose a bigger internal hard drive if a specific software package is chosen.
- *layout constraints*: The same combination of components can result in different configurations when their layout is involved. One-dimensional positioning can be needed in the HMS model to order the list of cards in the computer, while more advanced positioning (2-D or 3-D) can be required to organize the disposition of the parts inside the computer box. Even more complex layout problems can be solved by an interaction with CAD tools [Aldanondo *et al.*, 2001].
- Defining *optimization (or cost) functions* helps the modeler specify how to calculate a value that then can be minimize or maximize at some point of the configuration, once the other user requirements are met. One cost function (to be minimized) declared in the HMS represents the overall price of the Home Multimedia Station as a function of the price of the components chosen.

2.4 Development and runtime support

Aside from product modeling, configurators also need to provide convenient tools to help understanding and solving the constraints defined in the model:

- The task of creating a product model is not only about defining the structure and constraints. Most of the modelers' time is usually spent in *debugging* the model, so that it behaves as it is intended to. Providing a convenient way to debug product configuration models is thus a priority for a product configurator.
- A must-have feature for a good configurator is the ability to provide *explanations* at configuration time. These explanations are given to the end-user when the configuration is over-constrained, or to provide guidance if he wants to force the selection of a value that is not allowed by the solving engine.

3 Product Knowledge Modeling

Product Knowledge Modeling represents a significant part in the configuration process. It consists in defining the model of a product family that will then be configured by the end-user. Development and maintenance of product knowledge bases are of primary importance, and the representation formalism must be thoroughly considered when choosing a product configuration system. Major vendors of configuration systems already use declarative knowledge modeling [Moller *et al.*, 2001].

Modeling languages are used to represent knowledge in a structured way. They can be categorized into two types: *graphical* and *textual* languages. Graphical languages use diagrams with symbols to express the different concepts, while textual languages use standardized keywords to structure the knowledge representation, that is then interpreted in an abstract syntax.

In the next sections, we will discuss two graphical languages, UML and SysML, both accompanied by a constraint textual language called OCL, as well as the textual modeling language EXPRESS.

3.1 UML and OCL

The *Unified Modeling Language (UML)* is an international standard defined in 1997 by the Object Management Group (OMG). This general-purpose object-oriented language is very well-known as it is widely used in industrial software development, and so is of prime choice for configuration.

The UML class diagram is worth our interest, as product modeling in configuration mainly deals with the structure and the constraints of the product. Several UML relations are of interests for configuration: the *association*, that establishes a semantical relationship between two components, and can be used to model *connection ports*; the *composition* (or composite aggregation), a parent-child relationship that can represent *partonomy* relations; and the *generalization*, used to model inheritance in UML for data and code reuse, and that can represent *taxonomy* relations in configuration.

Secondly, UML 2.0 contains an extension mechanism called *stereotypes*. A stereotype allows designers to extend UML by creating new model elements from existing ones. The new nodes are then stereotyped, which is reflected graphically by adding a name enclosed by guillemets above the name of another element. A stereotype can contains attributes, called tagged values.

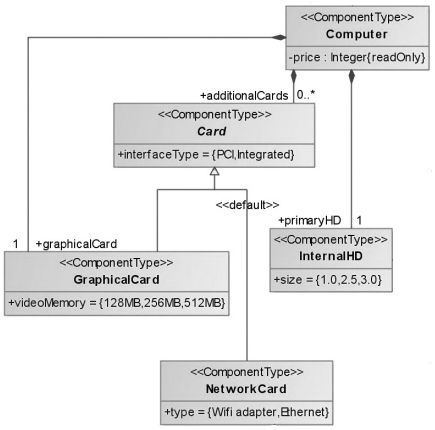

Figure 2: UML representation in the HMS case study

The *Object Constraint Language (OCL)* is an extension to UML that allows to write standardized constraints. It is actually a textual language that provides constraints and object query expressions that cannot be expressed using notations-like diagrams. OCL is a pure specification language, which means that an OCL expression will not have any side effects. Indeed, when an OCL expression is evaluated, it simply returns a value, and does not change anything in the model.

Product modeling using UML and OCL. Several researchers have showed interest in UML associated with product configuration. For example, UML is used by Hvam et al. [2008] in their "Procedure for building product models", along with another representation called *Product Variant Master*. They mainly focus on defining the object-oriented structure of the product model using UML, while more in-depth data (such as attributes and constraints) are stored in tables called *CRC cards*.
Felfernig et al. [2002] go further by defining a UML meta model architecture, i.e. a formalism to represent product configuration concepts and constraints using both UML and OCL. They automatically translate their model into an executable logical architecture, using the *XML Metadata Interexchange (XMI)* format - an XML-based OMG standard for the exchange of UML models.
The UML representation of a part of the Home Multimedia Station defined in Section 2 can be seen in Figure 2. This figure shows three partonomy and two taxonomy relations in the sub-model of the Computer. The OCL language can be used to describe constraints, e.g. that a computer that has a (primary) graphical card with 512 Mo of memory must cost at least 500 dollars:

```
context Computer inv:
self.graphicalCard.videoMemory = ''512Mo''
implies self.price >= 500
```

UML exhibits other interesting features for configuration, such as the use of packages. This allows the user to decompose its model into different groups of elements, thus permitting a better structure in the model.

UML/OCL limitations. The association UML/OCL provides an interesting object-oriented modeling experience, and the notoriety of UML among industry makes it an ideal candidate for product knowledge representation. As a graphical language, it also gives the design engineer a clear overview of the product model, making it more easy to see relationships between different components.
However, UML and OCL are not designed specifically for product modeling (and configuration), and thus miss interesting features. Although it is possible to adapt it to product configuration through the use of stereotypes, UML concepts are aimed at software engineering, and the transition can be difficult for the modeler. For example, it is not possible to declare units, and OCL falls a bit short when it comes to define table or soft constraints.
Finally, as a graphical language, its interpretation remains an issue. Indeed, the model must be interpreted in order to be integrated into a knowledge base and a configuration system. The work of [Felfernig *et al.*, 2002] goes in this direction, but more remains to be done in order to provide strong model checking and debugging facilities to the modeler.

3.2 SysML

The *Systems Modeling Language (SysML)* [SysML, 2001] is a recent modeling language developed as a joint initiative of OMG and the International Council on Systems Engineering (INCOSE). It is actually a UML profile, and thus inherits the characteristics of UML. The aim of SysML is to represent systems and product architectures, as well as their behavior and functionalities, where UML was used for software engineering. The development team of SysML aimed on the first hand at limiting the concepts too close from software engineering, and on the other hand at simplifying UML original notations by limiting the number of diagrams available, in order to make it easier to use.

Product Modeling using SysML. Modeling using SysML is very similar than doing so with UML, except that almost no user-defined stereotype is needed. Along with everything imported from UML, SysML defines *units* and *dimensions*. It is also possible to define objective (or optimization) functions and parametrized constraints using *Parametric diagrams* in SysML. This allows to represent constraints in diagrams where the parameters can be linked to the different attributes of the model's components, although the constraints' text still has to be expressed using OCL.
Finally, due to its full list of product-oriented diagrams, SysML gives the modeler the possibility to integrate the configuration model into a much wider product model, as UML does with software.

SysML limitations. Although SysML brings new capabilities relative to product modeling compared to UML, it still suffers from similar issues. For example, the constraints are still defined using OCL, and the matter of the interpretation of the model for model checking and debugging remains the same. The SysML extension provides more diagrams prestereotyped for product modeling (e.g. block, units, optimization functions,...), but still lacks some essential product con-

figuration concepts like resources for example.

3.3 STEP and EXPRESS

This section introduces the International Standard ISO 10303, which is referenced as *STEP (STandard for the Exchange of Product data)*. STEP was first released in 1994, and is published as a series of Parts. The goal of STEP is to allow the exchange of data describing a product between Computer Aided system (CAD, CAM, ...etc). It uses the EXPRESS language to formalize the semantics of the data, and the 20 series of Parts specify the standard data exchange mechanisms (e.g., data file or API access).
EXPRESS [EXPRESS, 2004] is thus an object-oriented data modeling language standardized as the Part 11 of STEP. It consists of two different representations: textual, or graphical (called *EXPRESS-G*). However, EXPRESS-G is not able to represent all details that can be formulated in the textual form, on which we will concentrate in this part.

Product Modeling using EXPRESS. Models in EXPRESS are organized according to *schemas*. These schemas permit to group the different elements of the model in relevant scopes, in the same way as UML packages. Components in EXPRESS are defined as *entities*, and are composed by *attributes*, that can be of basic types or entities themselves (partonomy). Taxonomy relations can also be represented through *abstract* classes and *subtypes*:

```
SCHEMA HomeMultimediaStationFactory;
  USE FROM ComputerParts;...
  ENTITY HomeMultimediaStation;
    price: DOLLAR;
    computers: SET[1:?] OF Computer; ...
  END_ENTITY;
  ENTITY Card ABSTRACT SUPERTYPE; ... END_ENTITY;
  ENTITY GraphicalCard SUBTYPE OF (Card); ... END_ENTITY;
  ...
END_SCHEMA;
```

Finally, named types and *units* can also be declared, clarifying the meaning and context of the variables of these types. Constraints can also be associated to each entities or types/units, through a *WHERE* clause.

```
TYPE DOLLAR = INTEGER;
WHERE SELF >= 0;
END_TYPE;
```

Although few built-in functions are available, EXPRESS allows *user-defined functions* using a full procedural programming language.

EXPRESS limitations. EXPRESS is a powerful language for product modeling, suitable for many product configuration problems. It contains nice features, such as units and constants declarations, as well as dynamic multiplicity or the possibility to define functions. However, the EXPRESS language is too general and is not suitable for knowledge engineers that are not expert in the language itself, mainly because of its lack of configuration-specific keywords. The definition of functions requires advanced programming skills, and writing a complex model without these functions can be very difficult or impossible, as a lot of functions are not built-in (min/max, sum, ...).

3.4 Discussion

Choosing a modeling language for product configuration is not a trivial task. Although graphical languages such as UML (and SysML) provide a clear and well-known representation of the product structure, interpreting the model is an issue, and advanced verification mechanisms may not be easy to built upon them. On the other hand, textual languages like EXPRESS provide a flexible formalism for modeling, but may be difficult to apprehend for a product modeler with few programming skills. Finally, specific features for product configuration are often missing, such as product catalogs and production attributes integration, or complex constraints (layout, soft, optimization functions, ...).

4 Solving the configuration

Proposing a language expressive enough to ideally model product families is not sufficient: the configuration system must be able to support this language and propose sufficient solving mechanisms. The combinatorial nature of configuration problems has led towards a wide use of *Constraint Satisfaction Problems (CSP)*.
Others topics of interest are model debugging and explanations. Indeed, both the modeler and the end-user must be assisted when using an interactive configurator. Modeler should be able to have a clear view of the running model and its constraints during design phase, while help should be provided to the end-user when he wants to force a value selection or when the configurator has reached an over-constrained choice with no solution.
In this section, we first recall the original definition of CSP and compare several dynamic extensions. We then review the trends in explanation generations and debugging.

4.1 Constraint Satisfaction Problems

The original CSP is a triple $\mathcal{P} = \langle X, D, C \rangle$ where:

- X is an n-tuple of variables $X = \langle x_1, x_2, ..., x_n \rangle$,
- D is a corresponding n-tuple of *domains* $D = \langle D_1, D_2, ..., D_n \rangle$, representing, for each variable x_i, the set of possible values it can take,
- C is a t-tuple of constraints $C = \langle C_1, C_2, ..., C_t \rangle$ restricting the values that the variables can simultaneously take.

Such problems are usually solved using search and consistency techniques. Search techniques are used to explore the solution space of the problem, the most famous one being *backtracking*, where the algorithm assigns each variable, tests for all the constraints, and then backtracks if no solution is to be found. Consistency techniques are used to reduce the domains of the variables during the solving, while keeping the problem consistent with the constraints. The most used consistency algorithm is arc-consitency (AC) 3, that research has been going on trying to improve (e.g. AC-3.3 in [Lecoutre *et al.*, 2003]). Other types of consistency techniques have also been investigated: path- [Bessière *et al.*, 2005] or k-consistency - although the complexity of the algorithms goes increasing. These techniques are aimed at problems on finite domains: working with continuous domains is

often more perilous [Benhamou *et al.*, 1999]. Finally, hybrid techniques can be used to improve the efficiency of consistency algorithms. Such techniques involve sharing of common subexpressions within the constraints [Hentenryck *et al.*, 1997], reshaping of the constraint network into a DAG (Direct Acyclic Graph), or reformulation of constraints [Benhamou and Granvilliers, 1997].

4.2 Extensions to CSP

Extensions to the classical CSP have been developed to permit the resolution of dynamic problems like product configuration. Mittal and Falkenhainer [1990] define a *Dynamic Constraint Satisfaction Problem (DCSP)*, where some variables are initially *active* while other not, and constraints are classified in two categories: *activity constraints*, that activate or deactivate variables, or *compatibility constraints* (similar to the constraints in the classic CSP).
This formulation includes algorithms only based on backtracking techniques, and has served as a basis for several others frameworks:

- a revision of the DCSP by [Soininen and Gelle, 1999], equivalent in complexity to the original CSP, and where activity constraints are generalized.
- Composite CSP (CCSP) [Sabin and Freuder, 1996] have been introduced to model a hierarchical structure between variables or constraints, using metavariables as placeholders for subproblems.
- In CSPe [Véron and Aldanondo, 2000], a state attribute is associated to each variable, giving the possibility to represent the activity of the variable, but can be also used for other purposes. The advantage of this formulation is that it can be solve using classic algorithms.
- More advanced algorithms for DCSP (renamed Cond-CSP) are developed in [Gelle and Faltings, 2003; Sabin *et al.*, 2003], and even further by Geller and Veksler [2005] with the ACSP.

Although well-studied, these dynamic CSP can only represent optional variables, and thus problems with an unbound number of variables cannot be solved with those methods. Two different approaches have thus been studied to solve that problem. Stumptner et al. [1998] describe the Generative CSP (GCSP), where constraints with metavariables can be used to express generic relations. Mailharro [1998] defines another framework, capable of satisfying on-demand generation of component in configuration. His approach is based on *constrained set variables*, which can contain a special value (*wildcard*) that represents the set of all components that have not been instantiated yet. Although a solving methodoly is presented, optimal algorithms in the number of value queries could be investigated.
One noticeable difference in these propositions is the representation of the product model using constraint satisfaction. Mailharro's approach focus on exploiting the component structure of the problem during solving. This kind of hybrid structure-based and constraint-based approach produces a constraint model much closer to the product model itself. This permits to design solving mechanisms specific to configuration, and that can reason on the structure of the product. It could also be of great help for giving debugging feedback to the modeler, thanks to its expressiveness.
The algorithms for solving these different problem representations are not always optimal, while time is a very important factor when dealing with interactive configuration. Precompilation techniques have thus been studied to circumvent those issues. The idea is to preprocess the constraint model at compile time into an efficient representation, using Binary Decision Diagrams [Hadzic *et al.*, 2004] or automata [Amilhastre *et al.*, 2002; Fargier and Vilarem, 2004]. However, generative problems or continuous domains are still an issue.

5 Explanations and Debugging

An efficient solving is not the only feature a constraint solver for interactive configuration should provide. Explanations and debugging support are necessary to help the user react intelligently when confronted to the modeling and configuration of a product.
Explanations are used to assist the end-user when answering the following questions: if there is a conflict, what are the reasons for inconsistency? Why is this feature selected by the solver engine? Why is this feature unavailable? Research work on explanations has recently increased, with the development of the QuickXplain algorithm by Junker [2004], that computes the minimal conflict set of a problem. In [2004], Friedrich proposes an improvement in the definition and the computation of the explanations in order to avoid the problem of *spurious explanations*. Recently, other approaches such as *corrective* [O'Callaghan *et al.*, 2005] and *representative explanations* [O'Sullivan *et al.*, 2007] have been developed to provide more intuitive explanations to the users.
Debugging support must also be provided to the user when it comes to test the product model. Indeed, an important part in the development of a model is actually verifying whether the model does what it is expected to do. Constraint debugging has been well studied in the research world, including visualization tools [Der, 2000] or the generic trace format from the *OADymPPaC project* [2007]. But few of these techniques have been specially targeted at product configuration, and are thus difficultly accessible to a classic design engineer. The concept of *model-based diagnosis* has been adapted to configuration by Felfernig et al. [2004] in order to tests the knowledge base with positive and negative test cases. Recently, Krebs [2008] proposed algorithms to identify relevant and irrelevant components for a specific product type (using segmentation of the product model tree), as well as detecting reachable component, using preprocessing to reduce the algorithm's complexity in some cases.
We strongly believe that research should focus on adapting debugging tools for product configuration, in order to alleviate the work of product modelers and testers, and limit the amount of time they spent on debugging their models.

6 Towards a future configuration system

In this section, we explore propositions for a new product configuration framework, based on the state-of-the-art technologies and the implementation of a new modeling language (Fig. 3).

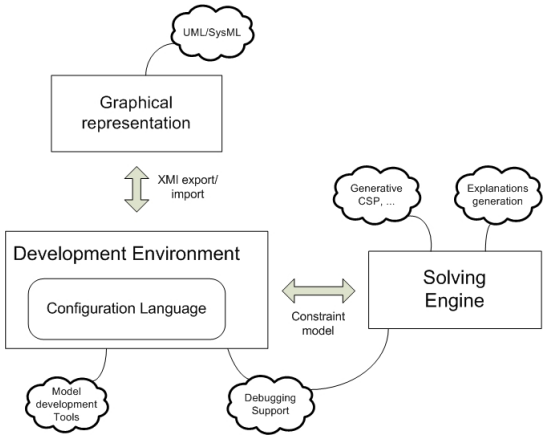

Figure 3: Product configuration framework

6.1 A new modeling language

As mentioned earlier, the language supporting the modeling of the product is of great importance for the configuration process. We propose the creation of a new declarative language that would be:

- *textual*: that type of language facilitates the interpretation and techniques like static analysis and optimization of the implemented model. Those techniques could for example be applied to perform model-checking, or propose correction of the model (e.g. by detecting unreachable values in attributes' domain). Integration with other systems such as a CAD software or a database would also be easier with such a language.
- *accessible* for model designers: the language should integrate keywords derived from configuration terminology, such as "product", "component", "constraint", "attribute" or even "BOM".
- providing a *graphical* representation: an export to a graphical language such as UML (or SysML) using the XMI format would permit to give a clear view of the model representation to the modeler, along with favoring the exchange of modeling data. Data about the model's constraints could be converted in OCL (when possible), or exported as text.
- designed according to the *modeling requirements* from Section 2, so that the language provides as many necessary features as possible for the model designer.
- aimed at enhancing designers' *productivity*: firstly, working with a textual language is often faster than through user interfaces, especially in an industrial environment where models are complex and contain a relatively big amount of data; secondly, such a language can be integrated in a development environment such as Microsoft Visual Studio, providing language services such as syntax-highlighting, code auto-completion, structured projects and many others. Last but not least, using a product modeling specific language would ease the integration with existing systems like ERPs or other tools (Word/Excel, ...).

6.2 Solving engine

The solving engine supporting the modeling language should at least implement the *Generative CSP* framework. Indeed, the dynamic possibilities of this framework are necessary to ensure the broadest panel of options available for the model. On the other hand, such a framework may not be as fast as simpler ones, and that is why other solutions may be envisaged. An interesting idea would be to perform an analysis of the model and determine what framework would be the most appropriate to use, improving execution times when possible. An explanation generation module will have to be integrated into the solving system. Such a module should integrate the recent state-of-the-art techniques, such as the *QuickXplain* algorithm [Junker, 2004] but could also experiment with corrective and representative explanations for example.

6.3 Advanced debugging support

Model-based diagnosis [Felfernig *et al.*, 2004] could provide interesting debugging options when designing a product model. The test cases could be gathered from previous configuration runs to make sure new model's additions do not create inconsistencies with old configurations. It is also worth investigating the automatic generation of the test cases: just after the creation of the model, by looking for potential weaknesses (e.g. targeting special values such as 0, infinity, or constructs like dynamic aggregations), or from a model proven correct, in order to test future modifications.
We also propose the exploration of debugging through *breakpoints*. The development of the model in an advanced environment such as Visual Studio gives the possibility to easily assign breakpoints to some parts of the model. Those breakpoints could target attributes and/or constraints, and be triggered when the attributes are modified, or the constraints provoke a change in the other assignments. A graphical overview of the constraint system could then be presented to the user.

7 Conclusion

Product configuration is a recent field of interest for both research and industry. As a consequence, the features and technologies needed for configuration systems are always evolving. We presented in this article a list of requirements for state-of-the-art product configuration systems, illustrated by a case study. We also described major existing modeling languages in the context of configuration, and discuss their limitations when it comes to configuration-specific features. Graphical languages fall short when it comes to automatic validation of the model. On the other hand, textual languages like EXPRESS are powerful but not suited for configuration model designers with few programming skills.
We then reviewed existing techniques for solving configuration problems, highlighting the need for advanced debugging support integrated with product modeling. As a solution, we made some propositions for a future product configuration framework, based on a new textual product modeling language integrated into a development environment. The development of this framework is the main objective of future work, associating static analysis, constraint solving and constraint debugging.

References

[Aldanondo *et al.*, 2001] M. Aldanondo, J. Lamothe, and K. Hadj-Hamou. Configurator and cad modeler:gathering the best of 2 worlds. *IJCAI Configuration Workshop*, 2001.

[Aldanondo *et al.*, 2003] M. Aldanondo, K. Hadj-Hamou, G. Moynard, and J. Lamothe. Mass customization and configuration: Requirement analysis and constraint based modeling propositions. *Integr. Comput.-Aided Eng.*, 10(2):177–189, 2003.

[Amilhastre *et al.*, 2002] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic csps—-application to configuration. *AI*, 135(1-2):199–234, 2002.

[Benhamou and Granvilliers, 1997] F. Benhamou and L. Granvilliers. Automatic generation of numerical redundancies for non-linear constraint solving. *Reliable Computing*, 3(3):335–344, 1997.

[Benhamou *et al.*, 1999] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising hull and box consistency. *Proc. ICLP*, pages 230–244, 1999.

[Bessière *et al.*, 2005] C. Bessière, J.-C. Régin, R. HC Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *AI*, 165 (2):165–185, 2005.

[Der, 2000] *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl)*, 2000.

[EXPRESS, 2004] EXPRESS. *10303-11 ISO - Part 11: The EXPRESS language reference manual*, 2004.

[Fargier and Vilarem, 2004] H. Fargier and M.-C. Vilarem. Compiling csps into tree-driven automata for interactive solving. *Constraints*, Vol. 9 Issue 4:263–287, 2004.

[Felfernig *et al.*, 2002] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Configuration knowledge representation using uml/ocl. *LNCS*, Jan 2002.

[Felfernig *et al.*, 2004] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *AI*, 152:213–234, 2004.

[Friedrich, 2004] G. Friedrich. Elimination of spurious explanations. *Proc. ECAI'04*, 2004.

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

[Geller and Veksler, 2005] F. Geller and M. Veksler. Assumption-based pruning in conditional csp. *Proc. CP'05*, 3709:241–255, Oct 2005.

[Hadzic *et al.*, 2004] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *Proc. PETO'04*, 2004.

[Hentenryck *et al.*, 1997] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. 1997.

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

[Janitza *et al.*, 2003] D. Janitza, M. Lacher, M. Maurer, U. Pulm, and H. Rudolf. A product model for mass-customisation products. *LNCS*, 2774:1023–1029, 2003.

[Junker, 2004] U. Junker. Quickxplain: preferred explanations and relaxations for over-constrained problems. *Proc. AAAI'04*, pages 167–172, 2004.

[Krebs, 2008] Thorsten Krebs. Debugging structure-based configuration models. *Proc. ECAI'08*, 2008.

[Lecoutre *et al.*, 2003] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithm. *Proc. CP'03*, pages 480–494, 2003.

[Mailharro, 1998] D. Mailharro. A classification and constraint-based framework for configuration. *AI EDAM*, 12:383–395, Sep 1998.

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

[Moller *et al.*, 2001] J. Moller, H. R. Andersen, and H. Hulgaard. Product configuration over the internet, 2001.

[OADymPPaC, 2007] OADymPPaC. *Generic Trace Format for Constraint Programming - Version 2.1*, Jan 2007.

[O'Callaghan *et al.*, 2005] B. O'Callaghan, B. O'Sullivan, and E. C. Freuder. Generating corrective explanations for interactive constraint satisfaction. *Proc. CP'05*, 2005.

[O'Sullivan *et al.*, 2007] B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. *Proc. CP'07*, 2007.

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

[Rossi *et al.*, 2006] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. 2006.

[Sabin and Freuder, 1996] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. pages 153–161, 1996.

[Sabin *et al.*, 2003] D. Sabin, E. C. Freuder, and R. J. Wallace. Greater efficiency for conditional constraint satisfaction. *LNCS - CP 2003*, pages 649–663, 2003.

[Soininen and Gelle, 1999] T. Soininen and E. Gelle. Dynamic constraint satisfaction in configuration. *Proc. of AAAI Workshop on Configuration*, Jan 1999.

[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.

[Stumptner *et al.*, 1998] M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(4):307–320, 1998.

[SysML, 2001] SysML. *OMG Systems Modeling Language (OMG SysML) v1.0 Specification*, 2001.

[Véron and Aldanondo, 2000] M. Véron and M. Aldanondo. Yet another approach to ccsp for configuration problem. *Proc. ECAI'00*, pages 59–62, 2000.

Argumentation based constraint acquisition

Kostyantyn Shchekotykhin and **Gerhard Friedrich**
University Klagenfurt, Austria
Department of Intelligent Systems and Business Informatics
firstname.lastname@ifit.uni-klu.ac.at

Abstract

Efficient acquisition of constraint networks is a key factor for the applicability of constraint problem solving methods. Current techniques ease knowledge acquisition by generating examples which are classified by a domain expert. However, in addition to this classification, an expert can usually provide arguments *why* examples should be rejected or accepted. Generally speaking domain specialists have partial knowledge about the theory to be acquired which can be exploited for knowledge acquisition. Based on this observation we discuss the various types of arguments a knowledge engineer can formulate. For the processing of these types of arguments we developed a knowledge acquisition algorithm which gives the knowledge engineer the possibility to input arguments in addition to the classification of examples. The result of this approach is a significant reduction of the number of examples which must be classified.

1 Introduction

Constraint networks are among the most successful technologies for the implementation of knowledge based systems in various domains, such as configuration and recommender systems. The positive aspect of this technology is an explicit knowledge representation, which can be exploited for validation and explanation generation. However, the downside is the significant engineering effort required for knowledge base formulation and maintenance. In order to ease this problem, [Bessière *et al.*, 2007] proposed a knowledge acquisition process that generates examples to be classified by a knowledge engineer. These examples are produced s.t. (roughly speaking) the number of possible knowledge bases contained in the learning space is reduced by half. While this is a major step towards efficient knowledge acquisition, the acquisition process can be further improved by exploiting additional knowledge. Usually the domain expert is not only able to classify an example but can also provide pieces of knowledge (which we call arguments) that justify the classification.

In our solution we follow the approach of [Bessière *et al.*, 2007] by characterizing the version space (i.e. the set of all possible constraint networks for a given vocabulary) using a propositional theory but introduce arguments for constraint acquisition. Note, arguments for learning classification rules were pioneered in [Mozina *et al.*, 2007]. We borrow the idea that additional knowledge about examples improves the learning process. However, since both the underlying learning method and the target language are different we follow an other route for constraint acquisition. In particular, our novel contributions are: First, we show the broad range of possible types of arguments a domain expert can formulate. Complete positive and negative examples in the sense of [Bessière *et al.*, 2007] are treated as special cases in this generalized view. Next, we show the consequences on the version space depending on the type of argument. Finally, we provide a constraint network acquisition algorithm based on a generalized view of arguments. This algorithm generates examples by employing the method proposed in [Bessière *et al.*, 2007]. In addition to [Bessière *et al.*, 2007] the knowledge engineer has the possibility to provide a set of arguments which are exploited to reduce the version space thus reducing the number of examples that must be evaluated by the knowledge engineer. E.g. our experimental study shows a reduction by roughly 50% to 75% for two arguments per example.

The paper is structured as follows. In Sec. 2 we provide an introduction to constraint learning. The types of possible arguments, their relation to examples, and their consequences on the version space is described in Sec. 3. The implementation of our method is presented in Sec. 4 followed by the results of our experiments in Sec. 5.

2 Constraint learning overview

In order to be able to learn a target problem description an expert should define the underlying formal language of the learner. In the case of constraint network learning an expert has to specify a *vocabulary* that includes a set of variables $X : \{x_1, \ldots, x_n\}$ along with a (finite) set of domain values D and a set of allowed constraints $\mathcal{L} : \{b^1, \ldots, b^m\}$. Each constraint $b \in \mathcal{L}$ with arity n can be defined on a set of variables $X_b \subseteq X$ where $|X_b| = n$ specifies a restriction on the allowed combination of domain values for the set of variables X_b.

For simplicity hereafter we assume that every constraint considered by the learning algorithm is binary, that is $|X_b| = 2$. For example, given a constraint library $\mathcal{L} : \{\geq\}$ and a set of variables $X : \{x_1, x_2\}$ one can define a constraint $\geq_{1,2}$ that

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

specifies a "greater than" relation between variables x_1 and x_2. Note, this assumption does not affect the generality of the proposed approach since every high-order finite-domain constraint can be reduced to a set of binary constraints.

Constraints from the library $\mathcal{L}$ together with the set of variables X define the *inductive bias* of the learning algorithm B. A bias B is a subset of $\{b^i_{j,k}\}$ for all $1 \leq j,k \leq n$, $1 \leq i \leq m$. For instance, given the library $\mathcal{L} : \{\leq, \geq, \neq\}$ together with the set $X : \{x_1, x_2\}$ an expert can construct the bias $B : \{\leq_{1,2}, \leq_{2,1}, \geq_{1,2}, \geq_{2,1}, \neq_{1,2}\}$.

A *training set* E^f is the second required input required for the constraint acquisition algorithm. Such a set should be formed by an expert and consists of a set of instances E (training examples) and a *classification function* $f : E \rightarrow \{0, 1\}$. The classification function partitions the given instances into positive e^+ and negative e^- examples.

Each instance $e \in E$ is a mapping from the set of *all* variables X to the set of domain values $e(x_j) \in D$. If a pair $(e(x_j), e(x_k))$ is an element of a binary constraint $b_{j,k} \in B$ then we say that the instance e *satisfies* the constraint $b_{j,k}$. In the opposite case the instance is *rejected* by the constraint. An instance e is called a *solution* for a set of constraints $C \subseteq B$ if it satisfies all constraints in C and a *non-solution* otherwise.

[Bessière *et al.*, 2005] define the *constraint network acquisition problem* as generating a constraint set $C \subseteq B$ that correctly classifies all training examples in E^f, i.e. every positive example $e^+ \in E^f$ is a solution and every negative example $e^- \in E^f$ is a non-solution of C. In this case C is termed to be *consistent* with the training set E^f.

CONACQ [Bessière *et al.*, 2005] is a SAT-based algorithm that uses version spaces [Mitchell, 1982] to acquire constraint networks. The hypothesis space corresponds to the bias. Given a bias B and a set of training examples E^f the *version space* $V_B(E^f)$ is defined as the set of constraint networks $C \subseteq B$ which are consistent with E^f.

CONACQ encodes the version space $V_B(E^f)$ as a propositional theory K. The propositional variables of K are symbols $b \in B$ representing constraints of the bias B. In an interpretation of K a propositional variable b with truth value 1 indicates that the constraint b is included in a constraint network. Truth value 0 represents the exclusion of constraint b. $models(K)$ denotes the set of all models of K. A function $\phi(m)$ transforms a model m of K to a constraint network by mapping propositional symbols with truth values of 1 to corresponding constraints. All constraints which correspond to symbols with truth value 0 are omitted. Therefore negative facts $\neg b$ in K should be understood *not* as the negation of constraint b itself, but as a description that b is absent in all constraint networks characterized by K. Thus, $models(K)$ is used to represent all possible constraint networks consistent with E^f.

The constraint acquisition algorithm takes the bias B and a set of examples as an input. For each example e CONACQ finds a set $\kappa(e)$ of constraints $b \in B$ where b is unsatisfiable with example e. If e is a positive example then the unit clauses $\{\neg b\}$ are added to K for all $b \in \kappa(e)$. In the case e is a negative example then the clause $\{\bigvee_{b \in \kappa(e)} b\}$ is added. Each learning step finishes by applying unit propagation to simplify the propositional theory K and to check consistency.

Table 1: Clauses learned after processing the training set E^f_T

E^f_T	Clauses added to K
e^+_1 $(1,2,3)$	$\neg \geq_{1,2} \wedge \neg \geq_{1,3} \wedge \neg \geq_{2,3}$
e^-_2 $(1,2,1)$	$\neq_{1,3} \vee \leq_{2,3}$

Table 2: Clauses learned from generated examples E^f_G

E^f_G	Propositional clauses added to K	Constraints fixed by K
e^-_3 $(1,3,2)$	$\leq_{2,3} \vee \geq_{1,2} \vee \geq_{1,3}$	$\leq_{2,3}$
e^+_4 $(1,2,2)$	$\neg \neq_{2,3} \wedge \neg \geq_{1,2} \wedge \neg \geq_{1,3}$	$\neg \neq_{2,3}$
e^+_5 $(2,1,3)$	$\neg \leq_{1,2} \wedge \neg \geq_{1,3} \wedge \neg \geq_{2,3}$	$\neg \leq_{1,2}$
e^+_6 $(2,1,2)$	$\neg \neq_{1,3} \wedge \neg \leq_{1,2} \wedge \neg \geq_{2,3}$	$\neg \neq_{1,3}$
e^+_7 $(3,1,2)$	$\neg \leq_{1,3} \wedge \neg \leq_{1,2} \wedge \neg \geq_{2,3}$	$\neg \leq_{1,3}$
e^-_8 $(1,1,2)$	$\neq_{1,2} \vee \geq_{1,3} \vee \geq_{2,3}$	$\neq_{1,2}$

Example 1 *Assume that an expert provided a constraint library $\mathcal{L} = \{\geq, \leq, \neq\}$ along with a set of variables $X = \{x_1, x_2, x_3\}$ and a domain $D = \{1, 2, 3\}$. The constraint bias B is defined as $\{\geq_{i,j}, \leq_{i,j}, \neq_{i,j}\}$ for all $1 \leq i < j \leq 3$. Furthermore, the expert defined a training set $E^f_T = \{e^+_1, e^-_2\}$ (see Table 1) aiming to acquire a target constraint network C_T that contains two constraints $\{\neq_{1,2} \wedge \leq_{2,3}\}$. After processing these examples* CONACQ *will output the propositional theory K presented in Table 1.*

Example 1 shows that due to incompleteness of the provided training set E^f_T various different constraint networks are consistent with the examples. [Bessière *et al.*, 2007] subsequently suggest an algorithm to generate missing examples until all $b \in B$ are *fixed*, that is either b or $\neg b$ is entailed by K. From the given propositional theory K and a set of constraints L the algorithm iteratively generates extensions of K s.t.: (a) K is satisfiable and (b) there exists $m \in models(K)$ such that constraint network $\phi(m)$ has at least one solution. Such a solution is then presented as an example to an expert, who classifies it as positive or negative. Under the best performing optimistic-in-expectation strategy the algorithm iteratively generates such examples, which in the best case fix exactly one constraint $b \in L$, thus reducing the version space by half, regardless of user classification.

Example 2 *For the case given in Example 1 the algorithm generates examples that fix at least one constraint $b_{i,j}$. The generated examples and the learned clauses are presented in Table 2. Note, unit propagation will reduce clauses s.t. after removing the clause generated for e^-_2 from K after e^-_8 all elements of the bias are fixed.*

However, experts may give arguments to supplement examples. E.g. for e^-_2 the expert may give the constraint $\leq_{2,3}$ as a reason why this example is negative. These arguments represent additional information and can be easily formulated by an expert in the validation step. In the following section we will provide a general definition of arguments and their impact on the version space.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

3 Argumentations and examples

The task of argumentation based learning is to generate a logical theory C based on a set of arguments ARG provided by a knowledge engineer. In our case this logical theory is a set of constraints. Consequently, we define an argument arg as a set of constraints, i.e. an argument is a subset of the bias and the set of all possible value assignments, i.e. $arg \subseteq B \cup \{x_i = v_i | x_i \in X, v_i \in D\}$. Since an example can be represented as a set of constraints $x_i = v_i$ where x_i is a variable and v_i is an element of the domain D, we can treat both examples and arguments uniformly. In the following we require that arguments are satisfiable. Arguments and constraints are interpreted w.r.t. a vocabulary. An *interpretation for a vocabulary* is an assignment of a value of the domain to each variable of the vocabulary. The concepts of models, satisfiability, and entailment are defined as usual.

Definition 1 *Given a set of arguments ARG and a logical theory C. C is valid w.r.t. ARG iff C is consistent and the following conditions hold for all arguments of ARG:*

1. *$C \cup arg^+$ satisfiable where arg^+ is called a* **positive** *argument for C. The set of positive arguments is denoted by ARG^+.*
2. *$C \cup arg^-$ unsatisfiable where arg^- is called a* **negative** *argument for C. The set of negative arguments is denoted by ARG^-.*
3. *$C \models arg^N$ where arg^N is called a* **necessary** *argument for C. The set of necessary arguments is denoted by ARG^N.*
4. *$C \not\models arg^{-N}$ where arg^{-N} is called* **too-strong** *for C. The set of too-strong arguments is denoted by ARG^{-N}.*
5. *$arg^S \models C$ where arg^S is called a* **sufficient** *argument for C. The set of sufficient arguments is denoted by ARG^S.*
6. *$arg^{-S} \not\models C$: arg^{-S} is called* **too-weak** *for C. The set of too-weak arguments is denoted by ARG^{-S}.*

The set of arguments ARG is $ARG^+ \cup ARG^- \cup ARG^N \cup ARG^{-N} \cup ARG^S \cup ARG^{-S}$.

By exploiting negation, the third case is equivalent to the second by verifying if $C \cup \neg arg^N$ is unsatisfiable. Likewise, the forth case can be reduced to the first case by verifying if $C \cup \neg arg^{-N}$ is satisfiable. Therefore, if the negation of constraints is supported by a reasoning system only Cases 1 and 2 need to be considered. The set of necessary and too-strong arguments can be included in ARG^- and ARG^+.

In the following we specify for each argument type the clauses added to the propositional theory K characterizing the version space. For this characterization we employ the concept of conflicts.

Definition 2 *The set of constraints B' for a bias B is a conflict for argument arg (set of constraints) where $B' \subseteq B$ iff $B' \cup arg$ is unsatisfiable. A minimal conflict is a conflict which does not contain a proper subset which is a conflict.*

Property 1 *If arg^+ is a positive argument and $B' \subseteq B$ is a conflict for arg^+ then K contains the clause $\bigvee_{b \in B'} \neg b$. It is sufficient to consider only minimal conflicts B'.*

This property ensures that every set of conflicting constraints are excluded by adding corresponding clauses to K since arg^+ can be empty.

Given the vocabulary defined in Example 1, if a user specifies a positive argument arg^+ to be $\{x_1 = 1, x_3 = 2\}$ then the minimal conflicts are $\{\geq_{1,3}\}, \{\geq_{1,2}, \geq_{2,3}\}, \{\geq_{1,2}, \neq_{1,2}\}, \{\geq_{2,3}, \leq_{2,3}, \neq_{2,3}\}, \{\neq_{1,2}, \leq_{2,3}, \neq_{2,3}\}$. Consequently, clauses like $\{\neg \neq_{1,2} \vee \neg \leq_{2,3} \vee \neg \neq_{2,3}\}$ are added to K.

Note, we can construct cases where the number of all minimal conflicts can increase exponentially in the size of the bias. However, if some minimal conflicts are not added to K then the domain expert should validate more examples, thus computational costs can be traded for knowledge acquisition costs.

Property 2 *If arg^- is a negative argument and $CONF$ is the set of minimal conflicts for arg^- then $\bigvee_{B' \in CONF} \bigwedge_{b \in B'} b$ is contained in K.*

If a user specifies the previous example argument as negative then the shown conflicts are interpreted as a conjunction and connected by a disjunction. However, Property 2 can be reformulated in order to avoid the disjunction of all minimal conflicts.

Property 3 *If arg^- is a negative argument and $B' \subseteq B$ s.t. $arg^- \cup (B - B')$ is satisfiable then the clause $\bigvee_{b \in B'} b$ is contained in K. It is sufficient to consider only sets B' where for each $B'' \subset B'$, $arg^- \cup (B - B'')$ is unsatisfiable.*

If for the example given above a user specifies the negative argument $\{x_1 = 1, x_3 = 2\}$ then the clauses $\{\geq_{1,3} \vee \geq_{1,2} \vee \leq_{2,3}\}, \{\geq_{1,3} \vee \geq_{1,2} \vee \neq_{2,3}\}$, and $\{\geq_{1,3} \vee \geq_{2,3} \vee \neq_{1,2}\}$ can be added to K. Note, such clauses correspond to the minimal hitting sets of the conflicts. As previously stated not all clauses need to be added to K since example generation proceeds until all b in B are fixed.

As in [Bessière *et al.*, 2005] examples may be specified completely, i.e. every variable of X is assigned a value. In the case that arg^- is a complete example then the previous property can be simplified since minimal conflicts are singletons. Consequently, the previous property corresponds to the case in [Bessière *et al.*, 2005] where the complete example is a negative one.

Property 4 *If arg^S is a sufficient argument and $b \in B$ and $arg^S \not\models b$ then the clause $\neg b$ is contained in K.*

Let us assume that the user knows that $x_1 = 2$ and $x_2 = 1$ together with the domain constraint on x_3 is a sufficient argument, i.e. in case $x_1 = 2$ and $x_2 = 1$ all values of x_3 are allowed. In this situation $\{\neg \leq_{1,2}\}, \{\neg \geq_{2,3}\}, \{\neg \neq_{2,3}\}, \{\neg \leq_{1,3}\}, \{\neg \geq_{1,3}\}, \{\neg \neq_{1,3}\}$ are added to K.

Note, complete specified examples correspond to sufficient arguments. If e^S is a completely specified example then $e^S \models C$. Consequently, the previous property corresponds to the case in [Bessière *et al.*, 2005] where the complete example is a positive one.

Property 5 *If arg^{-S} is a too-weak argument and B' is the set of all $b \in B$ where $arg^{-S} \not\models b$ then add $\bigvee_{b \in B'} b$ to K.*

If a user specifies a too-weak argument arg^{-S} to be $\{x_1 = 1, x_3 = 1\}$ then the following clause $\{\neq_{1,2} \vee \geq_{1,2} \vee \neq_{2,3} \vee$

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Table 3: Clauses learned from argued examples ARG_T provided by an expert

ARG_T	Clauses added to K
$arg_1^S\ (1,2,3)$	$\neg \geq_{1,2} \wedge \neg \geq_{1,3} \wedge \neg \geq_{2,3}$
$arg_1^N\ (\neq_{1,2})$	$\neq_{1,2} \vee \geq_{1,3} \vee \geq_{2,3}$
$arg_1^{-N}(\leq_{1,3})$	$\neg \leq_{1,3} \wedge (\neg \leq_{1,2} \vee \neg \leq_{2,3})$
$arg_2^{-S}(1,2,1)$	$\geq_{1,2} \vee \neq_{1,3} \vee \leq_{2,3}$
$arg_2^N\ (\leq_{2,3})$	$\leq_{2,3} \vee \geq_{1,2} \vee \geq_{1,3}$
$arg_2^{-N}(\neq_{1,3})$	$\neg \neq_{1,3}$

$\leq_{2,3} \vee \neq_{1,3}\}$ is added to K. Every negative argument is a too-weak argument. In case the too-weak argument is a completely specified example then this argument is also a negative argument. Because of this one-to-one relation in case of complete examples, complete examples can be treated either as sufficient or too-weak arguments.

Example 3 *Assume that the same constraint bias B was provided as in Example 1. Instances in training set E^f are complete examples and are processed as sufficient and too-weak arguments. Moreover, for each example the expert provided two arguments arg^N and arg^{-N} defined in terms of bias constraints. Note, the negation of each arg^N in ARG^N is added to ARG^- and the negation of each arg^{-N} in ARG^{-N} is added to ARG^+.*

The learning algorithm exploits the arguments according to the properties given above. Clauses inserted by the learner into the propositional theory K for each argument are presented in Table 3. For necessary arguments arg^N and too-strong arguments arg^{-N} we depict only those clauses needed for fixing the constraints in the bias. Note, in practical settings, if K entails $\neg b$ then the constraint b is removed from the bias and thus clauses become smaller. E.g. in the second line of Tab. 3 clause $\{\neq_{1,2} \vee \geq_{1,3} \vee \geq_{2,3}\}$ is simplified to $\{\neq_{1,2}\}$.

The resulting theory K obtained after unit propagation is $\neg \geq_{1,2} \wedge \neg \geq_{1,3} \wedge \neg \geq_{2,3} \wedge \neq_{1,2} \wedge \neg \leq_{1,3} \wedge \leq_{2,3} \wedge \neg \leq_{1,2} \wedge \neg \neq_{1,3}$. The example generation algorithm will generate only one complete example for this theory, e.g. $(1,2,2)$ which we assume to be classified as a positive argument. By processing this example the argumentation-based learning algorithm can fix the last missing constraint $\neg \neq_{2,3}$ and the method outputs the constraint network $C_T = \{\neq_{1,2}, \leq_{2,3}\}$.

4 Implementation

The argumentation-based constraint acquisition algorithm described in this section relies on two general algorithms: QUICKXPLAIN [Junker, 2004] and HS-TREE [Reiter, 1987]. Given a set B of possibly unsatisfiable constraints and a set BK of constraints considered to be correct, QUICKXPLAIN returns: (a) *consistent* in the case when the set $B \cup BK$ is consistent, (b) $\emptyset$ if BK is inconsistent, or (c) a minimal conflict $B_C \subseteq B$. On each call QUICKXPLAIN computes only one minimal conflict. Since a set of constraints B may contain more than one minimal conflict we can use the HS-TREE algorithm to find: (a) the set of all minimal conflicts $CONF$, and (b) the set HS of all minimal hitting sets of conflicts. The computation of minimal conflicts within HS-TREE is done by QUICKXPLAIN and therefore both algorithms take the same inputs [Friedrich and Shchekotykhin, 2005]. Each minimal hitting set $B_{HS} \in HS$ is a set of constraints s.t. $(B \setminus B_{HS}) \cup BK$ is satisfiable. During computation the generation of conflicts and hitting sets can be limited for avoiding combinatorial explosion. This strategy is sufficient since the constraint acquisition algorithm continues to generate solutions until all the constraints in the given bias are fixed.

Algorithm 1: Iterative learning algorithm

Input: set of constraints B
Output: set of constraints C
1 $CONF \leftarrow$ CONFLICTS(HS-TREE$(B, \emptyset)$)
2 **if** $CONF \neq \emptyset$ **then**
3 $\quad K \leftarrow \bigwedge_{B_C \in CONF} \left(\bigvee_{b \in B_C} \neg b\right)$
4 **else** $K \leftarrow \emptyset$
5 $K \leftarrow$ SIMPLIFY(K, B)
6 $LS \leftarrow \{B\}$
7 **while** HASUNFIXEDCONSTRAINTS(B, K) **do**
8 $\quad S \leftarrow$ GENERATEEXAMPLE(K, LS, B)
9 $\quad$ **if** $S = \emptyset$ **then return** "inconsistency"
10 $\quad ARG \leftarrow$ VALIDATE(S)
11 $\quad K \leftarrow$ LEARNCONSTRAINTS(ARG, K, B)
12 $\quad$ **if** $K = \emptyset$ **then return** "inconsistency"
$\quad LS \leftarrow$ GETDISJUNCTIONS(K)
13 $\quad$ **if** $LS = \emptyset$ **then** $LS \leftarrow \{B\}$
14 **return** CREATECONSTRAINTS(K)

The constraint acquisition algorithm (see Algo. 1) takes the set of bias constraints B generated from the given vocabulary as input. In the first step the consistency of the bias is verified by applying HS-TREE. For an inconsistent bias HS-TREE outputs a set $CONF$ of minimal conflicts. According to Prop. 1 for each conflict $B_C \in CONF$ the algorithm adds corresponding clauses to the propositional theory K. Additional constraints are fixed in K by applying SIMPLIFY. This procedure exploits unit propagation, redundancy rules, and backbone detection as proposed by [Bessière *et al.*, 2005] in order to fix bias constraints and to reduce the size and number of clauses in K. Next, the algorithm initializes an ordered set of sets LS by inserting the bias into it. During the learning process LS is used to store sets of unfixed constraints, namely all constraint sets corresponding to non-unary disjunctions in K. Hence, the elements of LS are constraint sets.

The main loop (Line 7) of Algo. 1 includes three stages: generation of an example (Line 8), validation of the solution by an expert (Line 10), and learning of the version space (Line 11). The algorithm stops when all bias constraints are fixed.

Complete examples are generated by GENERATEEXAMPLE which implements the example generation algorithm presented in [Bessière *et al.*, 2007] (see Sec. 2 for the basic idea). Note, GENERATEEXAMPLE outputs complete examples or $\emptyset$. If $\emptyset$ is returned by GENERATEEXAMPLE then the provided arguments exclude all possible constraint networks hence *inconsistency* is returned by Algo. 1. The complete example is

Algorithm 2: LEARNCONSTRAINTS algorithm

Input: set of arguments ARG, set of clauses K, set of constraints B
Output: set of clauses K

foreach *argument* $a \in ARG$ **do**
  if $a \in ARG^- \vee a \in ARG^N$ **then**
    if $a \in ARG^-$ **then**
      $B_{HS} \leftarrow$ HITTINGSET(HS-TREE(B, a))
    else $B_{HS} \leftarrow$ HITTINGSET(HS-TREE$(B, \neg a)$)
    $K \leftarrow K \wedge \left(\bigvee_{b \in B_{HS}} b\right)$
  if $a \in ARG^+ \vee a \in ARG^{-N}$ **then**
    if $a \in ARG^+$ **then**
      $B_C \leftarrow$ QUICKXPLAIN(B, a)
    else $B_C \leftarrow$ QUICKXPLAIN$(B, \neg a)$
    $K \leftarrow K \wedge \left(\bigvee_{b \in B_C} \neg b\right)$
  if $a \in ARG^S \vee a \in ARG^{-S}$ **then**
    $B' \leftarrow \{b \in B : a \cup \{b\}$ unsatisfiable$\}$
    if $a \in ARG^S$ **then**
      $K \leftarrow K \wedge \left(\bigwedge_{b \in B'} \neg b\right)$
    else $K \leftarrow K \wedge \left(\bigvee_{b \in B'} b\right)$
$K \leftarrow$ SIMPLIFY(K, B)
return K

classified by an expert in the validation step (VALIDATE) and added either to ARG^S or ARG^{-S}. Furthermore, the set of arguments ARG can be extended by the expert by specifying additional arguments justifying the classification decision.

Next, Algo. 1 calls LEARNCONSTRAINTS(ARG, K, B) which processes all given arguments one by one. For each argument, Algo. 2 introduces changes to the propositional theory K as defined in Sec. 3. Depending on processing requirements we invoke either HS-TREE or QUICKXPLAIN to identify a minimal conflict or a minimal hitting set in the bias B given an argument as a background theory. If an argument provided by an expert is inconsistent then both QUICKXPLAIN and HS-TREE return $\emptyset$. Hence, inconsistent arguments are ignored by the constraint acquisition algorithm. During the final step the algorithm calls SIMPLIFY(K, B) to validate and to simplify the generated propositional theory.

The last steps of the main loop update the set LS so that the new set includes all constraint sets that correspond to non-unary positive disjunctions of the propositional theory K. The bias is added in case LS is empty. Note, the actual implementation deletes those b from B where K entails $\neg b$. When all constraints $b \in B$ are fixed, the algorithm returns the target constraint network C_T to the expert.

5 Experimental results

The main goal of the evaluation was to show the impact of argumentation-based constraint acquisition on the number of questions posed. Therefore, in our experiments we compared the number of examples generated to learn a desired constraint network using the original approach of [Bessière *et al.*, 2007] with the approach presented in this paper. Furthermore, we evaluated how the number of arguments influences the number of generated examples.

Similar to [Bessière *et al.*, 2007] the argumentation-based learner was applied to three problems: random binary problems, Sudoku puzzle and Schur's lemma. In order to generate the random binary problem the system created a constraint bias given the library $\{\leq, <, =, \neq, >, \geq\}$ and a set of 14 integer variables with domain size $|D| = 20$. Next the generator randomly selected a constraint and added it to the network. If the network had solutions the generator randomly added another constraint. The iterations continued until a given size of a soluble network was achieved. The resulting network was used as the target network for the learner. The second problem implements a standard 9×9 grid Sudoku puzzle and a reduced 4×4 (employed in [Bessière *et al.*, 2007]). Both Sudoku problems were evaluated using a constraint library $\{\geq, \leq, \neq\}$. The constraint library for Schur's lemma[1] was defined as a set of ternary constraints just as in [Bessière *et al.*, 2007] $\{$ALLDIFF, ALLEQUAL, NOTALLDIFF, NOTALLEQUAL$\}$.

For each problem we performed multiple tests in order to get average results describing the approximate performance. The generation of arguments as well as validation of examples was implemented using task-specific checkers which were aware of the target constraint network and thus were able to validate generated examples. Moreover, the checkers provided a given or maximum possible number of arguments for each example. The later is done to simulate expert's actions during the argumentation process. We randomized the argumentation as follows: first the validation system generates a set of possible arguments for a given example using elements of the bias as necessary or too-strong arguments. Then the system randomly selects the required number of arguments and returns them to the learner. In order to compute a set of possible arguments for a positive example, the checker gets all unfixed constraints UC that are accepting the example. If $c \in UC$ is included in the target constraint network then c is possible for ARG^N else c is possible for ARG^{-N}. For a negative example the algorithm generates the possible arguments as previously described with the exception that UC is the set of all unfixed constraints that are rejecting the example.

Example 4 *Consider the Sudoku learning problem, where a generated complete example includes the following constraints* $e_i : \{x_{1,1} = 1, x_{1,2} = 2, x_{1,3} = 3, \ldots\}$. *The validation system evaluates this example as positive and randomly generates the arguments* $a_1^N : \{x_{1,1} \neq x_{1,2}\}$ *and* $a_1^{-N} : \{x_{1,1} \leq x_{1,3}\}$.

The implementation of the presented approach uses the SAT4J and CHOCO CSP solver. The example generation parameters were set to values that correspond to the *optimal-in-expectation* strategy.

Figure 1 shows the performance of the argumentation-based approach in terms of the number of examples to be validated and arguments provided for the described problems. The case with zero arguments is equivalent to the ap-

[1] Problem 15 of CSPLib available from: http://www.csplib.org

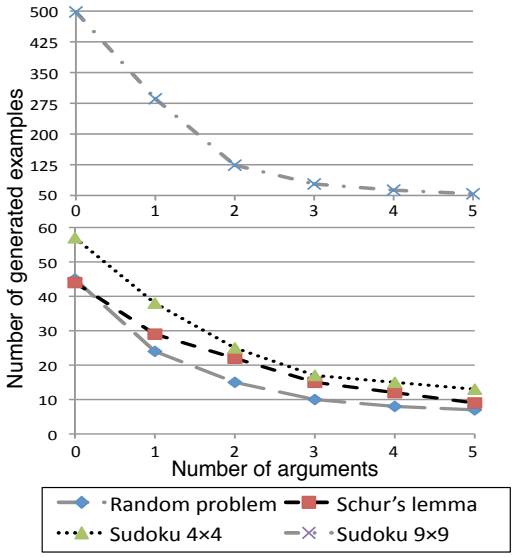

Figure 1: Performance of the learner in terms of the numbers of required examples and given arguments. Random Problem ($|X| = 14, |C_T| = 40$), Sudoku 4×4 ($|X| = 16, |C_T| = 72$), Sudoku 9×9 ($|X| = 81, |C_T| = 864$), and Schur's lemma ($|X| = 8, |C_T| = 12$).

proach presented in [Bessière *et al.*, 2007]. As it can be seen from Figure 1, the number of examples to be classified by an expert decreases considerably depending on the number of provided arguments. The experiment shows, however, that increasing the number of arguments (cases with 4 or more arguments per example) does not guarantee a constant improvement in performance. Regarding the runtime performance we can report that the generation of an example (GENERATEEXAMPLE) together with the processing of arguments (LEARNCONSTRAINTS) for our largest test problem (i.e. the 9×9 Sudoku) requires in average 16 sec. for the 2 argument case. Other average runtimes are: 7 sec. for 0 arg., 10.4 sec. for 1 arg., 18.7 sec. for 3 arg., 22.9 sec. for 4 arg., and 27.6 sec. for 5 arg. These results indicate the applicability of the proposed approach in interactive settings.

6 Related work

As discussed above our approach extends the work of [Bessière *et al.*, 2007] in order to allow the domain expert to provide arguments during the knowledge acquisition process. Arguments are also exploited in the area of learning classification rules introduced in [Mozina *et al.*, 2007]. In this approach the expert can provide "because of" and "despite of" arguments in addition to the usual attribute/classification vector. This arguments are exploited by an enhanced version of the CN2 rule generation algorithm. The main differences to our approach is the underlying learning approach and the language exploited to express a theory.

The work of [Rossi and Sperduti, 2004] aims at the acquisition of preferences. We regard this as important step for effective knowledge acquisition. However, learning preferences is beyond the scope of this paper. In particular, we leave it as an open problem to investigate argumentation techniques for enhancing the acquisition of preferences.

7 Conclusions

Our work was motivated by the fact that efficient knowledge acquisition is still an important problem to be solved in order to make knowledge based systems competitive. Since constraint based systems are among the most successful problem solving approaches in many application domains, we focused on the acquisition of constraints. In particular, we built on the version space approach pioneered by [Bessière *et al.*, 2007]. We enhanced this method by giving the domain expert the ability to provide arguments, since specialists are not only able to classify examples but to formulate partially the desired knowledge base. We investigated various types of arguments and discussed the consequences on the version space in case such arguments are given. Based on this analysis we have presented the implementation of a learning method where the expert can provide arguments in addition to classification information. We validated our method based on the test problems formulated in [Bessière *et al.*, 2007]. Our tests have shown a significant reduction of the number of examples a knowledge engineer has to classify depending on the number of arguments provided.

References

[Bessière *et al.*, 2005] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *ECML*, pages 23–34, 2005.

[Bessière *et al.*, 2007] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 50–55, 2007.

[Friedrich and Shchekotykhin, 2005] Gerhard Friedrich and Kostyantyn Shchekotykhin. A General Diagnosis Method for Ontologies. *Proceedings of the 4th International Semantic Web Conference (ISWC-05)*, pages 232–246, 2005.

[Junker, 2004] Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Association for the Advancement of Artificial Intelligence*, pages 167–172, San Jose, CA, USA, 2004.

[Mitchell, 1982] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[Mozina *et al.*, 2007] Martin Mozina, Jure Zabkar, and Ivan Bratko. Argument based machine learning. *Artificial Intelligence*, 171(10-15):922–937, 2007.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 23(1):57–95, 1987.

[Rossi and Sperduti, 2004] Francesca Rossi and Allesandro Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.

On Solving Complex Rack Configuration Problems using CSP Methods

Wolfgang Mayer and **Marc Bettex** and **Markus Stumptner**
University of South Australia
Advanced Computing Research Centre
Adelaide, Australia
ciswma|bettex|mst@cs.unisa.edu.au

Andreas Falkner
Siemens AG Österreich
Vienna, Austria
Andreas.A.Falkner@siemens.com

Abstract

Constraint Satisfaction Techniques have been widely applied to the configuration of complex systems in various domain. While much progress has been achieved on algorithms and theoretical characterisations of special cases that can be solved efficiently, many of these results make strong assumptions. In this paper we investigate different modelling and search techniques to assess their suitability for on-line configuration of a complex prototypical configuration problem. We found that CSP solving techniques perform poorly on our problem, while local search and repair techniques may achieve good performance while permitting simpler constraint formalisms.

1 Introduction

Assembly and configuration of larger systems from individual modular parts has been a central topic of interest, leading to a number of different technical solutions to address this problem in a variety of domains (Stumptner and Soininen; 2003). Constraint satisfaction based search techniques have been particularly successful in solving this task in certain domains, such as the telecommunications sector (Fleischanderl et al.; 1998), where entire switching systems are assembled from racks and modules.

Similar techniques have recently been employed to address other domains. For example, the design and planning of railway systems and the underlying infrastructure also requires that the results adhere to legal requirements and technical constraints. Similarly, the composition and configuration of distributed service processes can be phrased as a configuration problem (Thiagarajan et al.; 2009).

Typically, modular ("object-oriented") knowledge representation techniques are applied to capture the relevant properties of components and their related constraints, which may then be used to check configurations for compliance and extend partial configurations with additional components (Fleischanderl et al.; 1998). Modular knowledge representation is convenient for knowledge acquisition and is complemented with a custom-built constraint solving implementation that may be used to reason about configurations (Fleischanderl et al.; 1998).

While custom constraint frameworks are easily implemented to check a (partial) configuration for consistency with given constraints, completing a partial configuration ("*solving*") or altering a configuration to repair constraint violations is a challenging problem. Since some of the newer application areas have significantly more complex constraints than earlier applications, traditional solving approaches that are based only on backtracking or back jumping algorithms are no longer effective and may lead to inefficient exploration of the space of possible configurations. Instead, advanced Constraint Satisfaction techniques would be desired that can handle complex arithmetic expressions, relations, graphs, and other approximations. In the configuration context, a combination of advanced domain reduction and search procedures allows a solver to detect infeasible configurations early and focus search. For example, using integer linear programming techniques to better estimate the number of required components may allow to prune the search space more efficiently. Similar combinations of multiple CSP techniques has already led to significant results in other domains (Van Hentenryck and Michel; 2005).

While advanced CSP technology has great potential to improve configuration efficiency, it remains challenging to implement the necessary algorithms within the custom configuration system. Rather, ways to exploit existing implementations, for example the ECLiPSe suite (Apt and Wallace; 2007), would be desired. Hence, methods to express selected aspects of the configuration task as standard CSP problem have been developed (Kızıltan and Hnich; 2001; Narodytska and Walsh; 2007). In particular, the elimination of equivalent solutions due to symmetries —in our context typically caused by interchangeable components or permutations of components— has been thoroughly researched in the CSP community.

However, most representations of configuration problems address only a small subset of the entire configuration do-

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

main and must be extended to cover complex configuration problems. Moreover, existing symmetry reduction techniques typically require strong assumptions that may not hold for configuration problems in practice.

In this paper, we investigate different ways to encode a prototypical configuration problem as a standard CSP to assess which techniques are suitable for semi-interactive configuration. Our sample problem revolves around a toy example that is derived from a real-world configuration problem and includes complex constraints that are challenging to handle for most constraint solvers. Complex global constraints together and global cost function that should be optimised render this problem interesting from both a theoretical and a practical perspective.

Our results indicate that constraint technology is no silver bullet and must be applied carefully. Unsurprisingly, standard CSP heuristics alone are often insufficient to derive useful solutions and must be complemented with domain-specific heuristics. More surprisingly, symmetry reduction techniques that have shown significant improvements on other benchmarks exhibit poor performance when applied to our configuration domain. In contrast to our expectations, advanced symmetry reduction strategies designed to focus search lead to performance that is significantly worse than solvers not exploiting this information. We also report that traditional backtracking solvers scale poorly on any encoding, while incremental refinement-based solvers can quickly arrive at solutions that are close (within 25%) to the (estimated) optimum. Hence, we advocate the use of heuristics combined with iterative repair-based search procedures to attack semi-interactive configuration where configurations need not be optimal but must be delivered timely. We also show that replacing complete search procedures with incomplete heuristics does not necessarily sacrifice the quality of the resulting configurations.

The remaining paper is organised as follows: in Section 2 we introduce the problem used throughout our case study. In Section 3 we evaluate matrix-based translations of our problem, followed by a tailored CSP representation that more directly encodes the modular KB constraints in Section 4. We then discuss local search procedures in Section 5 before concluding the paper by summarising our findings and avenues for further work in Section 6.

2 Showcase Problem

Our investigations are based on the "Showcase House" configuration problem where a given list of items ("Things") must be allocated to Cabinets in different Rooms while respecting certain constraints and preference criteria (Falkner and Schreiner; 2008). While the problem seems trivial at first, it is interesting because it incorporates constraints and properties that can also be found in more technical domains like design and assembly of complex systems and software processes.

A conceptual model of the problem is shown in Figure 1. In this domain, a house (of given size) contains a number of Rooms where each can accommodate multiple Persons and Cabinets. Cabinets are used to store "Things" and have a fixed capacity. Cabinets may be either low or high, where low Cabinets may be stacked on other (low) Cabinets. Each

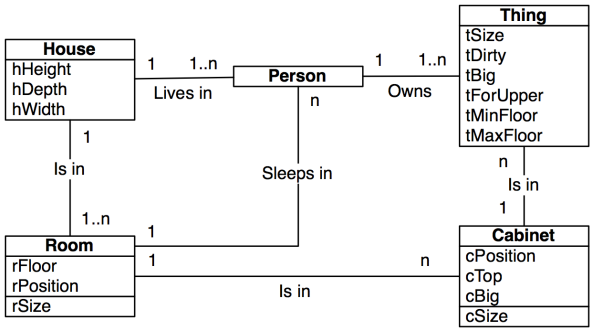

Figure 1: House Configuration Model

Thing is characterised by its size, whether it is clean or dirty, and whether it can be stored in an upper Cabinet. Cabinets may contain only clean or only dirty Things to ensure dirty Things cannot soil clean ones, and lower Cabinets cannot accommodate big Things. Furthermore, restrictions on the floor level where a thing can be stored apply. Each thing belongs to a Person. Since floor space and Cabinets are expensive, Things should be allocated to as few Cabinets as possible, while respecting the allocation constraints.

In addition to the mandatory constraints, a good solution not only minimises the number of Cabinets used, but also adheres to certain preference criteria. In particular, a thing should be stored in the Room of its owner, and, should this be impossible, in a vacant Room. The option where a Thing is stored in another Person's Room is least preferred, since this is likely to cause many disruptions.

The requirements and preferences can be expressed formally in a CSP framework. For example, assume a model where each entity is uniquely identified by some identifier and where properties of entities as well as relations between entities are represented as functions. Then, the constraint that a Cabinet may contain either only clean or only dirty Things can be expressed as a logical sentence as follows (Bettex; 2009):

$$\forall t_1, t_2 \in T : t_1 \neq t_2 \wedge dirty(t_1) \neq dirty(t_2) \Rightarrow cabinet(t_1) \neq cabinet(t_2)$$

The remaining requirements can be formalised similarly. While this logical representation can be derived directly from the problem statement, it may may not be best-suited to the configuration task. Since the logical model can only express "hard" constraints, it is difficult (or at best inefficient) to handle the preference criteria. Here, (soft) CSP satisfaction methods may be more suitable.

In the following sections we investigate different constraint-based modelling approaches that have been proposed for similar configuration tasks to assess which approaches are suitable to capture complex domain constraints like the above.

3 Configuration as Matrix CSP

A variety of constraint satisfaction techniques have been proposed to address the configuration problem of modular electronic systems, where modules of different types must be consistently assembled into a (the smallest) set of racks of different kinds. In particular, Kızıltan and Hnich (2001) compare

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

$$\begin{array}{r} \\ t_1: \\ t_2: \\ t_3: \\ t_4: \\ \\ high: \\ cDirty: \end{array} \begin{array}{c} \begin{array}{cc} \textit{big} & \textit{dirty} \end{array} \\ \begin{array}{cc} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{array} \\ W_A \\ \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 1 & 0 \end{array} \end{array} \quad \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

(a) Model A

$$\begin{array}{r} \\ T_1: \\ T_3: \\ T_4: \\ \\ high: \\ cDirty: \end{array} \begin{array}{c} \begin{array}{ccc} n & \textit{big} & \textit{dirty} \end{array} \\ \begin{array}{ccc} 2 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \\ W_B \\ \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 1 & 0 \end{array} \end{array} \quad \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

(b) Model B

Figure 2: Two Matrix Models

and combine different complementary models in order to focus the search for solutions. Racks and modules correspond to Cabinets and Things in our problem. In this section, we adapt two of their models to our problem domain and summarise our empirical results.

We consider a simplified variant of the configuration problem, where the number of Cabinets are minimised but the preference constraints related to Persons and associated Rooms are left aside. This simplification may lead to solutions that are not optimal in the full problem. We refrain from extending the matrix CSPs to the more detailed representation since our experiments showed that even the simplified version is computationally intractable. The poor performance of the matrix CSP for the simplified representation discouraged us from attempting to formalise the problem in full; this would require a multi-dimensional matrix, which, we conjecture, would prohibit the solving efficiency required for (semi-)interactive configuration of large problem instances. The full problem is addressed in Section 4.

3.1 Model

Matrix-like CSP models are a special class of CSP models that have been proposed to model relationships between entities (Kızıltan and Hnich; 2001). In matrix models, the properties of entities are typically expressed as variables indexed by entity type (or even entity instance), while a matrix $W = (w_{ij})$ represents the associations between entities. In this framework, a given configuration problem is translated into a matrix CSP problem and is subsequently solved using a standard CSP solver; the solution of the CSP problem determines the configuration.

Kızıltan and Hnich (2001) discuss two different modelling approaches:

(A) Rows and columns of W represent single entity *instances*, where the elements of W range over $\{0, 1\}$ to indicate a relationships between entity instances i and j.

We adapt this models for our house configuration problem as follows; a similar representation was also described in Falkner (2009):

Each row $w_{.j}$ of W represents a single Thing, whose properties are represented as constants. Figure 2a shows an example, where four things ($t_{1\ldots4}$) are placed in three cabinets ($c_{1\ldots3}$). The properties of concrete things are represented as constant vectors. For our case study, the properties big and $dirty$ are modelled.

Each column $w_{i.}$ of W represents a unique Cabinet, where $w_{ij} = 1$ iff t_i is placed in c_j. A Cabinet's properties are modelled as constraint variables, whose values are constrained by the $w_{i.}$. We use variable $high_j$ to indicate that a Cabinet contains big Things, and $cDirty_j$ to reflect that at least one dirty Thing is placed in c_j. Constraints ensure that each Thing is assigned to one Cabinet

$$\sum_j w_{ij} = 1$$

and that Things with incompatible properties (clean/dirty) are assigned to different Cabinets (columns):

$$cDirty_j = \max_i(w_{ij} dirty_i),$$
$$cDirty_j \neq dirty_i \Rightarrow w_{ij} = 0$$

Other constraints are modelled similarly. Preferences are expressed as minimisation criterion, where the number of Cabinets (columns) where at least one thing is placed is minimal:

$$\arg\min_W \sum_j sign(\sum_i w_{ij}).$$

(B) Rows of W represent entity *types*, where each element $w_{ij} \in \mathbb{N}$ denotes the *number* of entities (modules, Things) i that are related to an instance of the entity (racks, Cabinets) represented by column j.

For example, the first row of matrix W_B shown in Figure 2b represents two indistinguishable things of type T_1. Each row $w_{i.}$ of W represents a set of n_i Things that share the same properties. In contrast to model A, only one row is required to represent multiple instances. The representation of columns (Cabinets) remains the same as in the previous model. The constraint that each Thing must be allocated to some Cabinet, while not exceeding the Cabinet's capacity, is captured as follows:

$$\sum_j w_{ij} = n_i, \quad \sum_i w_{ij} \leq size_j$$

Since each row represents potentially many Things, the constraint model is likely to be smaller than the previous one. The compact representation may also allow to solve more efficiently, since equivalent solutions where rows corresponding to equivalent Things are interchanged need not be explored separately.

3.2 Symmetry Reduction

While the models can be solved using standard finite-domain constraint solvers, both suffer from inefficient search caused by different representations of semantically equivalent variable assignments. This "symmetry" problem is well-known in the CSP literature and a number of techniques have been developed to reduce this effect (Frisch et al.; 2003). Typically, symmetries are eliminated from the search by imposing additional constraints that enforce an ordering on the variable assignments to eliminate equivalent solutions.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Both models introduced in the previous section suffer from many redundancies, stemming from indistinguishable Things and Cabinets. Following Kızıltan and Hnich (2001), we impose restrictions on the Cabinet's assignments such that (in this order) (i) high Cabinets must occupy columns with smaller indexes than low Cabinets, (ii) "dirty" Cabinets must come before "clean" ones, and (iii) adjacent Cabinets of the same type are ordered with increasing free space. For example, the latter constraint is expressed as follows:

$$high_j = high_{j+1} \wedge cDirty_j = cDirty_{j+1} \Rightarrow \sum_i size_i w_{ij} \geq \sum_i size_i w_{i,j+1}$$

In addition, one could enforce a lexicographic ordering on the columns of W (Frisch et al.; 2003). Since this additional constraint slowed search considerably (see next section), we decided to not use it in our experiments. Kızıltan and Hnich (2001) argue that neither model is strictly better than the other and show that a combined approach can significantly improve symmetry elimination (although not all symmetries could be eliminated). Their best model backtracks only $\approx 5K$ times, down from $164M$ for the original Model A. However, their improved model does not directly apply to our problem domain, since different requirements on entity types and their constraints prohibit the same symmetry elimination in our problem domain.

3.3 Results

Similar to the experiments described by Kızıltan and Hnich (2001), Falkner (2009) reported that Model A outlined above performed poorly on all but the most trivial house configuration instances. While a solution for 10 Things could be found in a few seconds, problems for 20 Things could not be solved in 30 seconds.

We reimplemented both models in the ECLiPSe CLP framework (Apt and Wallace; 2007) to investigate whether symmetry reduction techniques could help to increase efficiency, and whether different labelling heuristics had a significant impact on the solving process. We chose the ECLiPSE CLP implementation since it offers an integrated framework where a variety of different CSP solving libraries can simultaneously be combined with logic programming and other search procedures. This was advantageous in particular for the model described in Section 4, where constraints of different type must be considered in unison. Based on earlier experiences with CSP libraries such as Minion and Choco, we are confident that the implementation used in our work is of comparable efficiency as other state-of-the-art generic CSP solvers. Encodings using SAT solvers are also expected to perform poorly due to a large number of global and arithmetic constraints increases clause sizes.

Our evaluation was based on a test suite of over 200 generated problem instances of varying size (from 10–450 Things). We focused our investigations on Model B, which showed better performance on our example problem.

Our results are summarised in Figure 3:[1] It can be seen that without symmetry reduction, *a* solution to problems of

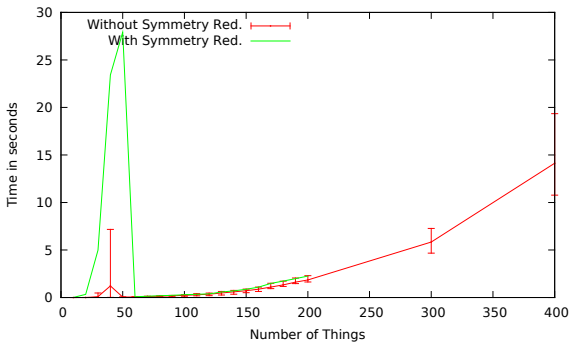

Figure 3: Results for the Matrix CSP model, Approach B

size up to 450 Things can be solved in a few seconds. Larger problems could not be investigated due to memory exhaustion (> 256Mb).

However, we were able to prove for only three of all problems that the solution found by our solver was indeed optimal. In all other cases, optimality could not be established within the set time limit of two minutes per problem. Instead, we estimated the quality of our solutions by comparing each with a lower bound for each solution.[2] We found that the solutions are almost all within factor 1.25 of the lower bound. Hence, for configuration problems where an absolute optimum is not required, matrix representations without symmetry reduction may be an appropriate CSP representation.

From the discussion in Kızıltan and Hnich (2001), we anticipated that adding symmetry reduction constraints would significantly boost the efficiency of our solver. However, our experiments contradicted our expectations: while the symmetry breaking constraints removed much of the redundant assignments, the overhead of evaluating the necessary constraints by far outweighed the benefit for all but the most trivial examples. Most symmetry-breaking constraints are "global" for each column or row and must therefore be evaluated frequently. This overhead by far outweighs the benefit for larger examples. Using symmetry reduction, we could prove optimality in 14 cases (up from 3), but could not find solutions for most problems exceeding 50 Things. Note that Figure 3 only shows times for successfully solved problem instances; cases where no solution could be found within the time limit have been omitted. Hence the graphs for solving with and without symmetry reduction appear similar, while the number of successfully solved cases actually drops to around 15% compared to the "plain" CSP. Hence, in practice, symmetry reduction for this complex constraint optimisation problem does not lead to any improvement.

Furthermore, we confirmed that the labelling strategies used by the CSP solver have considerable impact. In our experiments, we used a variable and value selection strategy that explores variables of W column-wise, with smaller domain values first. This strategy has shown the best results for this problem.[3] This can be explained by the fact that this strat-

[1] The vertical bars represent minimum and maximum values.

[2] This can easily be computed from the total size of all clean (dirty) Things and the fixed Cabinet capacity.

[3] Row-wise labelling produces exactly the same results. Using

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

egy partly incorporates the symmetry reduction constraints (empty/full Cabinets before full/empty ones) without incurring the same overhead as explicit constraints. In fact, we found this strategy to be backtrack-free for most problems. Surprisingly, the well-known "most-constrained" variable selection heuristic performed poorly; we believe that is due to the large number of global constraints and the additional overhead necessary to search for a suitable variable. Similarly, the "anti-first-fail" variable selection strategy that selects the variable that is least constrained required excessive time and failed to solve many instances.

4 Custom Symmetry Reduction

While rephrasing the configuration problem as a standard CSP has the advantage that available off-the-shelf solvers can be applied, our experiments indicate that this approach may not be the best solution for complex problems.

In this section, we pursue a different approach that is more closely aligned with the conceptual model introduced in Section 2, but applies partial symmetry reduction by transforming the problem. In contrast to the previous section, we include preference constraints into our model.

4.1 Transformed Problem

The case study given in Section 2 exhibits the following symmetries (other than those introduced by the matrix-like representation):

(i) Rooms on the same floor not occupied by a Person are indistinguishable;

(ii) The identity of Cabinets is insignificant. Swapping any two Cabinets (together with their properties, such as low/high, clean/dirty, etc.) results in an equivalent solution;

(iii) Cabinets of the same type (low/high, clean/dirty, upper/lower) within a Room are indistinguishable; their contents can be swapped or mixed without changing the quality of the solution.

We eliminate Symmetries i and ii by restructuring the model into an equivalent representation that does not exhibit such equivalent solutions. We did not attempt to address Symmetry iii, but it is believed that imposing constraints on the order and number of some Things allocated to Cabinets in a Room may eliminate redundancy further.

The structure of our revised conceptual model is shown in Figure 4. The Cabinets and their properties that are explicitly represented in the conceptual model in Figure 1 have been transformed into attributes of Rooms and Things. For each Room, the number of each type of Cabinet is represented as an attribute, with constraints enforcing that a lower Cabinet exists for each upper one, and that the Cabinets can fit within a Room. While the number of Cabinets is represented, their specific positions within a Room are not, hence reducing Symmetry ii.

To address Symmetry i, Rooms that are unoccupied are merged into a single empty Room per floor, with the Room size being the sum of all merged Rooms.

domain splitting or maximal domain values instead also results in the same solutions but requires up to three times as much time.

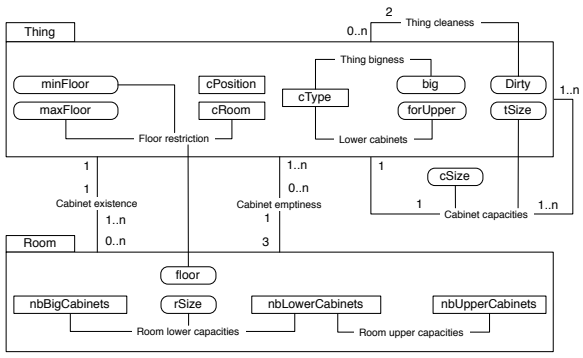

Figure 4: Symmetry-reduced CSP (Bettex; 2009)

Associations between Things and Cabinets are represented as attributes of Thing, capturing the Room, Cabinet type and index within a Room. The constraints are adjusted accordingly. For example, the constraint that clean and dirty Things must be stored in different Cabinets is enforced by a constraint operating on the attributes of Thing:

$$dirty(t_1) \neq dirty(t_2) \Rightarrow cRoom(t_1) \neq cRoom(t_2) \vee cType(t_1) \neq cType(t_2) \vee cPos(t_1) \neq cPos(t_2)$$

Additional constraints are necessary to ensure the Cabinets occurring in attributes of Thing actually exist in the Rooms, and that empty Cabinets should be prohibited (to ensure minimality of the solution). The former constraint for lower Cabinets can be formalised as follows:

$$type(t) = \text{lower} \Rightarrow nbLowerCab(cRoom(t)) \geq cPos(t)$$

Note that this constraint can only be evaluated if attribute *cRoom* of a Thing has been assigned. Our implementation uses techniques that are similar to Constraint Handling Rules (Frühwirth; 1998) to suspend such indirect constraint until they can be evaluated. Our implementation does not apply domain reduction to the index part of such constraints ($cRoom(t)$ in the example), and only checks consistency once variables have been instantiated sufficiently.

We represent the preference criteria for optimal solutions as a lexicographic ordering over the number of Cabinets, the number of Things stored in a Room occupied by someone else than the Thing's owner, and the number of Things stored in vacant Rooms.

Since many of our constraints rely on certain variables being instantiated, variable and value ordering heuristics are critical for solving. The structure of the constraints (Things rely on their Room assignment, and Cabinets in a Room are determined by the Things assigned to them) and initial empirical results lead to a strategy that instantiates the variables belonging to a Thing together without intervening other assignments. Within a Thing, its Room and type of Cabinet are assigned first, where the owner's Room and lower Cabinets are tried first. At this point most constraints relating Things to Rooms can be evaluated to check consistency and prune the possible values of attributes in Room. Once the Things' variables have been assigned, we proceed with the Rooms, with lower values first.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

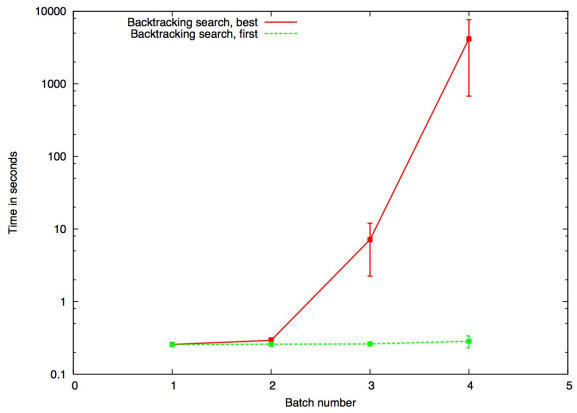

Figure 5: Results for the Custom CSP model

4.2 Results

We attempted to solve our CSP model using a standard backtracking solver using the interval domain and constraint suspension (Rossi et al.; 2006) to obtain a baseline for further evaluation with different search techniques (presented in Section 5). The results are shown in Figure 5. Note that the scale of the ordinate is *logarithmic*.

It can be seen that this approach does not scale; even for trivial problems with four Rooms and twelve Things, problems can take up to several hours to solve optimally. The figure also shows that *some* (possibly sub-optimal) solution is found quickly; however, the problem instances are very small and this observation may not uphold for larger instances. Regarding the quality of the solutions, we find that the (non-exhaustive version of our) backtracking solver produces solutions that are approximately 1.6 times worse than the estimated optimum.

Overall, as informed readers may have expected, complete search techniques based on backtracking are insufficient to devise solutions from our model. This is most likely due to weak consistency enforcement and delayed pruning due to constraint suspension. We investigate alternative search procedures using the same model in the next section.

5 Local Search

The constraint representation outlined in the previous section is effective only for *checking* a candidate solution assignment, where the values of the variables in Thing and Rooms are known. While building such an assignment using finite-domain CSP techniques has proven ineffective (see previous section), it may be possible to use the same constraint representation to *improve* a given solution. In this section, we investigate whether *local search techniques* (Van Hentenryck and Michel; 2005) that iteratively improve an invalid assignment may be suitable to address this problem.

Constraint-based local search and optimisation techniques arrive at an (optimal) solution by altering fully instantiated but possibly invalid assignments to reduce the constraint violations or to improve the overall cost. Many variations of the general scheme exist, with each employing different strategies to navigate from one assignment to the other and to stop the overall process (Van Hentenryck and Michel; 2005).

The variant used in our work is similar to Simulated Annealing (Russel and Norvig; 2003), where probabilistic thresholds are applied to guide exploration. Starting with an initial solution, constraint satisfaction techniques are employed to find minimal sets of constraints that imply infeasibility of a solution ("conflicts"). Once conflicts have been identified, one or more of the involved variables are chosen and assigned different values. Suitable values are determined by domain-specific heuristics. As a result, the previously violated constraint is satisfied, but other constraints may be violated in the new assignment. Hence, the overall quality of a solution may deteriorate rather than improve in a single step.

If the quality of the resulting candidate solution is better than that of the previous assignment it is used as the basis for further exploration. Otherwise, it may be kept (with some probability), or another alternative assignment may be explored. This process continues until all constraints are satisfied and no improvement could be achieved over a number of iterations, or until a threshold on the number of iterations (or time) has been reached.

Since this approach does not require to consider partial assignments, the drawback of delayed and suspended constraint evaluations observed with the backtracking solver are not an issue. The restriction to *complete* assignments also is not a limitation on the problems that can be handled. Partial configurations and dynamically expanding problems can be addressed by using techniques where the relevant scope of variables and constraints is expanded incrementally, as for example in the generative CSP framework (Fleischanderl et al.; 1998).

5.1 Solving Heuristics

To generate the initial solution, we leverage the *Best Fit Decreasing* heuristics from the related class of *Bin Packing Problems* (Lodi et al.; 2002). We consider the Things in order of high before low and in decreasing size. Starting with all Rooms empty, we assign each Thing to a Cabinet in a Room such that all constraints remain satisfied, possibly creating a new Cabinet if no suitable Cabinet can be found. Ties between cost-equivalent Cabinet assignments are resolved by minimising the remaining space in a Cabinet. Otherwise, the Thing is placed in a random location, disregarding any violated constraint. As a result, an initial solution is created that may contain violated constraints and that may not be minimal.

To improve upon the initial solution, heuristics that repair constraint violations are applied iteratively. Conflicts implied by the given candidate assignment are identified using ECLiPSe's conflict monitoring mechanisms (Cheadle et al.; 2003–2006). For each constraint we identify possible repair actions. For example, the constraint that Room capacities must not be exceeded is repaired by selecting and moving cabinets to other Rooms until the capacity constraint is satisfied. Similarly, to ensure only clean or only dirty Things are in a Cabinet, the clean or dirty Things must be moved to another location.

Since local repair choices may critically influence the further progress of the overall optimisation process, we allow our solver to spend considerable time choosing between possible repair actions. In particular, we apply local constraint prop-

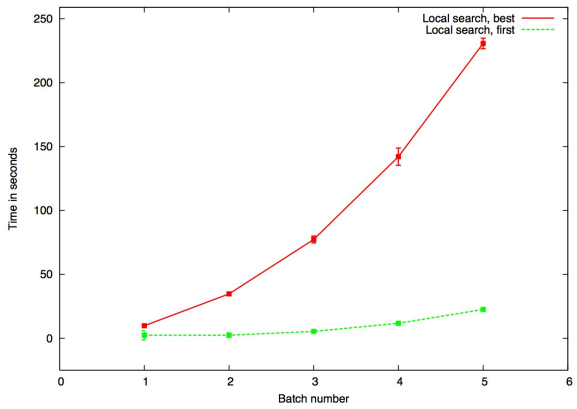

Figure 6: Results for the Custom CSP model

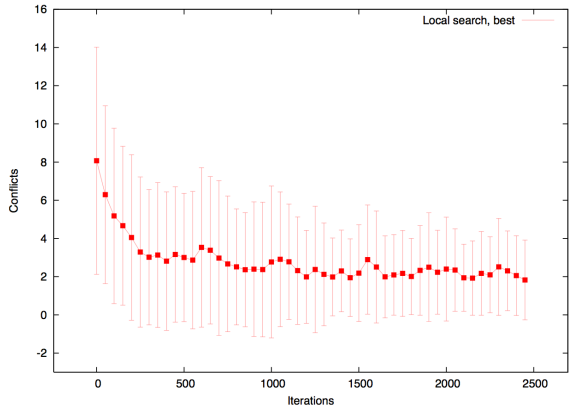

Figure 7: Conflict evolution

agation for different alternative repair assignments to determine the number of conflicts in each assignment. The locally conflict-minimal assignment is then chosen as basis for the subsequent iteration.

While this process is not guaranteed to result in an optimal solution, our results indicate that, typically, good-quality solutions within factor 1.2 of the estimated optimum can be found.

While the repair heuristics incorporate domain-specific knowledge, such as preferred ordering of Rooms, the local minimisation approach allows us to reduce the domain-specific aspects compared to imperative domain-specific solving algorithms. By utilising constraints that are close to the conceptual domain model rather than encodings tailored to low-level finite-domain CSP representations, we are able to largely structure repair actions in terms of domain concepts, which may be an advantage when considering the maintenance of knowledge bases that evolve over time. Using the local repair paradigm, constraints can be used to assess validity of a configuration and repair actions can often be designed and considered without considering the entire minimisation problem. It is not necessary to explicitly factor the search and minimisation strategies into the solving algorithms; this aspect is instead covered by the local exploration strategy that searches for combinations of repair actions that are suitable for achieving a goal or for finding a close assignment.

5.2 Results

Figure 6 summarises the results for our example problem. For this experiment, we selected five batches of example problems from our library, where problems in batch n contained n floors with $6n$ Rooms, $2n$ Persons, and $20n$ Things. We report the averages over five repeated runs.

For each problem an initial candidate assignment was devised using the Bin Packing heuristics introduced in Section 5.1. We then identified conflicts and repeated the repair process until either a conflict-free assignment was found, or up to a maximum of 2500 iterations.

While we always accept assignments with better cost valuation, the probability that an assignment is accepted that contains more conflicts or that has a worse cost estimate is 0.75. Compared to traditional probabilistic optimisation techniques, this value may seem unusually high; however, since we consider only a single repair action at a time, resolving one conflict often requires to create a number of intermediate invalid configuration before another, better configuration can be reached. Figure 7 depicts the evolution (arithmetic mean and variance) of conflicts as the search progresses. It can be seen that while the initial solution may suffer from a considerable number of conflicts, this number drops quickly but keeps oscillating until a solution is found.

Figure 6 summarises the time required for our algorithm. The case where only a valid solution is sought is labelled "*first*", whereas the algorithm that continues for 2500 iterations is labelled "*best*". It is observed that the time required for each solver increases polynomially with the problem size; hence the algorithm scales well to non-trivial problems. Furthermore, since the time required to find the first solution is little and scales well with problem size, the local search strategy can also be used in an "anytime" framework where the solving process can still provide a good solution even if stopped early.

Comparing the absolute time values with those reported in Section 3 is difficult, since the matrix representation of the problem captured only a subset of all constraints. Furthermore, our current implementation is not optimal. We are aware that our local search implementation currently suffers from excessive constraint checking that could be eliminated by using advanced implementation techniques and different constraint systems that handle global constraints more efficiently.

Comparing the quality of the solutions derived by the "best" and the "first" algorithms, we observe that both are virtually indistinguishable, with both being within 10% of the estimated optimum. Both are much better than the non-complete backtracking-based solver (which produced solutions exceeding by as much as 60%). Since differences between the "first" and "best" solution are negligible, the effort required to pursue further optimisation may not be justified in practice.

6 Discussion and Conclusion

We investigated three approaches to solving a prototypical configuration problem by using finite-domain constraint satis-

faction and constraint optimisation techniques. While there is considerable work on efficiently solving particular classes of problems, our configuration domain includes constraints and irregularities that render many of those techniques inapplicable. Our experiments aimed at assessing the suitability of CSP technology for the configuration of complex problems that do not fit squarely in the categories researched previously. Our findings can be summarised as follows:

- The globally optimal solution cannot be found in reasonable time for most non-trivial problem instances. (More precisely, it cannot be proved in reasonable time that the derived solution is indeed optimal.)
- Adding constraints for symmetry reduction seems ineffective for this type of problem. While the backtracking behaviour is indeed much reduced, the overhead caused by non-local constraints that span many variables make this approach slow and unsuitable in practice.
- Heuristic search techniques together with heuristics adapted from Bin Packing problems are able to arrive at good solutions quickly for medium to large-sized problems. However, considerable transition periods between regions of feasible configurations and the evaluation of global constraints remain the critical factors in this process.
- The first valid solution that is found with local search is often close to the optimum, and further search may be limited if time is critical.
- Specifying local repair actions for constraints in combination with generic conflict-driven optimisation procedures can yield an effective configuration framework where complex imperative solving strategies need not be asserted explicitly. Furthermore, this formalism may be more amenable to devising constraint representations that closely follow the domain entities instead of requiring complex translations to structurally different finite-domain CSPs.
- The time required by local search procedures can be predicted more easily than that of conventional CSP search algorithms. Hence, this technique may be preferred for interactive configuration scenarios.

From our experiments we identify the following areas where progress may be most beneficial for future improvements:

- The synthesis of local repair actions from declarative knowledge bases, in particular from domain constraints and optimisation criteria.
- On-line tuning and learning approaches to help choosing suitable algorithm parameters (probabilities, thresholds) and to focus more effectively on promising repairs, depending on non-local contexts and properties of the configuration instance under consideration.
- The identification of criteria which can be used to select a "good" CSP encoding for a problem described by a structured, "object-oriented" knowledge base that is close to the conceptual domain models.

References

Apt, K. R. and Wallace, M. (2007). *Constraint Logic Programming using Eclipse*, Cambridge University Press, New York, NY, USA.

Bettex, M. (2009). *House configuration problem using constraint optimization*, Master's thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia. Joint work with the Aritificial Intelligence Laboratory, Swiss Federal Institute of Technology.

Cheadle, A. M., Harvey, W., Sadler, A. J., Schimpf, J., Shen, K. and Wallace, M. G. (2003–2006). *ECLiPSe – A Tutorial Introduction*, Cisco Systems, Inc.

Falkner, A. (2009). Choco implementation (partly) of s'upreme showcase house, *Technical report*, Siemens AG, PSE, Vienna, Austria.

Falkner, A. and Schreiner, H. (2008). Two decades' experience in developing product configurators – mastered challenges and remaining issues, *in* J. Tiihonen, A. Felfernig, M. Zanker and T. Männistö (eds), *Proceedings of the Workshop on Configuration at the 18th European Conference on Artificial Intelligence*, Patras, Greece.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H. and Stumptner, M. (1998). Configuring large-scale systems with generative constraint satisfaction, *IEEE Intelligent Systems* **13**(4).

Frisch, A. M., Jefferson, C. and Miguel, I. (2003). Constraints for breaking more row and column symmetries, *in* F. Rossi (ed.), *CP*, Vol. 2833 of *Lecture Notes in Computer Science*, Springer, pp. 318–332.

Frühwirth, T. W. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1-3): 95–138.

Kızıltan, Z. and Hnich, B. (2001). Symmetry breaking in a rack configuration problem, *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, Seattle, WA.

Lodi, A., Martello, S. and Monaci, M. (2002). Two-dimensional packing problems: A survey, *European Journal of Operational Research* **141**(2): 241–252.

Narodytska, N. and Walsh, T. (2007). Constraint and variable ordering heuristics for compiling configuration problems, *in* M. M. Veloso (ed.), *Proceedings International Joint Conf. on AI*, pp. 149–154.

Rossi, F., Beek, P. v. and Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science Inc., New York, NY, USA.

Russel, S. and Norvig, P. (2003). *Artificial Intelligence A Modern Approach*, 2 edn, Prentice Hall.

Stumptner, M. and Soininen, T. (2003). Special issue on configuration, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **17**(1).

Thiagarajan, R., Mayer, W. and Stumptner, M. (2009). Generative composition of web services, *CAiSE Forum*, CEUR Workshop Proceedings, Amsterdam, The Netherlands. Forthcoming.

Van Hentenryck, P. and Michel, L. (2005). *Constraint-Based Local Search*, The MIT Press.

Configuring Models for (Controlled) Languages

Mathias Kleiner (INRIA) and **Patrick Albert (ILOG)*** and **Jean Bézivin (INRIA)**

Abstract

Conceptual schemas (CS) are core elements of information systems knowledge. A challenging issue in the management processes is to allow decision makers, such as business people, to directly define and refine their schemas using a controlled language. The recently published Semantics for Business Vocabulary and Rules (SBVR) is a good candidate for an intermediate layer: it offers an abstract syntax able to express a CS, as well as a concrete syntax based on structured English. In this article, we propose a method for extracting a SBVR terminal model out of an English text and then transform it into a UML class diagram. We describe a model-driven engineering approach in which constraint-programming based search is combined with model transformation. The use of an advanced resolution technique (configuration) as an operation on models allows for non-deterministic parsing and language flexibility. In addition to the theoretical results, preliminary experiments on a running example are provided.

1 Introduction

Conceptual schemas (CS) are widely used in industry as formal representations of information system's knowledge. A CS is often the central element on which relies a set of operations: validation, simulation, usecases generation, etc. The UML/OCL combination is the de-facto standard for specifying a CS. However, designing, maintaining and refining a CS currently requires important technical skills. Stakeholders rely on IT-specialists to model their requirements. A recent trend in software engineering (requirements engineering) is to propose ways to facilitate this communication, by allowing decision makers to express their needs in natural language, which can then be transformed into a formal representation.

In the business context, the Object Management Group (OMG) has recently published the SBVR (Semantics for Business Vocabulary and Rules) recommandation. SBVR provides a metamodel for business concepts and statements which can be used to define a CS. The specification also proposes a structured English form. However parsing natural language into a SBVR model is a difficult problem. In particular, if one wishes to build domain specific languages using non context-free grammars, the different interpretations that can be made for one sentence disqualifies tools based on deterministic algorithms, such as existing model driven engineering (MDE) tools (ATL[Jouault and Kurtev, 2005], QVT[QVT, 2008]) or most controlled languages parsers (ACE[Schwitter and Fuchs, 1996]). Instead, the problem requires to *search for* a solution using advanced AI techniques.

This article describes an automatic translation of controlled English into a SBVR concrete model and a UML model of the described CS, allowing to build flexible domain specific languages. The originality of our approach is that it combines constraint programming techniques with model transformation tools in a MDE framework. It is composed of three main operations. The first task is a syntactical and grammatical analysis of the text, which is directly related to the well-known and challenging field of language parsers: we describe a parser based on the configuration of a constrained model. The second task is the transformation of the resulting model into a SBVR model. The third task is the transformation of the SBVR model into a UML model of the CS. The integration of configuration in MDE as an advanced transformation tool is an important and innovating contribution of this work.

Plan of article

Section 2 briefly introduces the technologies involved in our approach. We also present an overview of the whole process and a running example. Section 3 introduces the language parser. Section 4 shows how the resulting model is transformed into a SBVR model. Section 5 proposes a transformation from SBVR to UML. Validation, implementation and experiments are presented in Section 6. Finally, we discuss related work in Section 7 and conclude.

2 Context of the work

2.1 Brief introduction to SBVR

SBVR is an OMG standard [SBVR, 2008] intended to be the basis for describing business activities in natural languages. It is an attempt to build the bridge between Business Users and software artifacts, enabling non-IT specialists parameterize and evolve the business logic embedded into applications.

[1]Partially funded by the French ANR IdM++ project

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

SBVR standardizes a set of concepts for the definition of business specific Controlled Languages (declarative languages whose grammar and lexicon have been limited in order to eliminate part of the ambiguity. See [Schwitter and Fuchs, 1996] as a historical paper, or more recently [Kaljurand, 2008]). Business Rules Systems (ILOG JRules, Drools) are popular controlled languages used to explicitly model the business logic in a growing number of applications.

We will not describe SBVR exhaustively in this article. However a look at the Figures 1 and 2 might provide a feeling of the sophistication of the meta-model and of the approach that nicely separates logical formulations from Meanings.

2.2 Brief introduction to MDE and model transformation

Model Driven Engineering is an emerging research area that considers the main software artifacts as typed graphs. This comes from an industrial need to have a regular and homogeneous organization where different facets of a software system may be easily separated or combined.

In MDE, models are considered as the unifying concept. The MDE community has been using the concepts of terminal model, metamodel, and metametamodel for quite some time. A terminal model is a representation of a system. It captures some characteristics of the system and provides knowledge about it. MDE tools act on terminal models expressed in precise modeling languages. The abstract syntax of a modeling language, when expressed as a model, is called a metamodel. A language definition is given by an abstract syntax (a metamodel), one or more concrete syntaxes, and a definition of its semantics. The relation between a model expressed in a language and the metamodel of this language is called conformsTo. This should not be confused with the representationOf relation holding between a terminal model and the system it represents. Metamodels are in turn expressed in a modeling language called metamodeling language. Its conceptual foundation is itself captured in a model called metametamodel. Terminal models, metamodels, and metametamodel form a three-level architecture with levels respectively named M1, M2, and M3. A formal definition of these concepts may be found in [Jouault and Bézivin, 2006]. The principles of MDE may be implemented in several standards. For example, OMG proposes a standard metametamodel called Meta Object Facility (MOF).

The main way to automate MDE is by providing transformation facilities. The production of model Mb from model Ma by a transformation Mt is called a model transformation. When the source and target metamodels are identical (MMa = MMb), we say that the transformation is endogenous. When this is not the case (MMa $\neq$ MMb) we say the transformation is exogenous. In this work we use ATL (AtlanMod Transformation Language), a QVT-like model transformation language [Jouault and Kurtev, 2005] allowing a declarative expression of a transformation by a set of rules.

2.3 Brief introduction to configuration

Configuring is the task of composing a complex system out of generic components [Junker, 2006]. Components, also called objects or model elements in the sequel, are defined by their types, attributes, and known mutual relations. The acceptable systems are further constrained by the request: a set of problem-specific and/or user-specific requirements, represented by a fragment of the desired system (i.e interconnected objects). From a knowledge representation perspective, configuration can be viewed as the problem of finding a graph (i.e a set of connected objects) obeying the restrictions of an object model under constraints. From a model driven approach, it can be viewed as the problem of finding a finite model that conforms to a metamodel. More precisely, we consider the process as a model transformation where the source model is the request and the target model is the solution. The configuration model acts as the target metamodel. A *relaxed* version of this metamodel acts as the source metamodel. This relaxed metamodel is obtained by the removal of all constraints: minimum cardinalities are set to zero, attributes are optionals and OCL constraints are removed. The request, which is thus a set of target model elements with incomplete knowledge (for instance linked elements and attributes values are undefined), is therefore conformant to the source metamodel. In this context, the configurator *searches* for a target model, completing the source and creating all necessary model elements so that the result (if any) is conformant to the target metamodel.

Various technical approaches have been proposed to handle configuration problems: extensions of the CSP (Constraint Satisfaction Problem) paradigm [Stumptner and Haselböck, 1993; Sabin and Freuder, 1996], knowledge-based approaches [Stumptner, 1997], logic programming [Soininen *et al.*, 2001], object-oriented approaches [Mailharro, 1998; Junker and Mailharro, 2003]. Configuration has traditionally been used with success in a number of industry applications such as manufacturing or software engineering. More recently, the expressive power of configuration formalisms has proven their usefulness for artificial intelligence tasks such as language parsing [Estratat and Henocque, 2004]. A deeper introduction to configuration can be found in [Junker, 2006].

In the sequel we propose to use Ilog JConfigurator [Junker and Mailharro, 2003] as a language parser for SBVR. In this model-oriented approach, a configuration model (in our context, the target metamodel) is well-defined as a set of classes, relations and constraints. The UML/OCL language combination may be used to this purpose [Felfernig *et al.*, 2002].

2.4 Process overview

Figure 3 sketches the overall process in a model-driven engineering framework. The input is an English text, close to the form of structured English that has been proposed in the SBVR specification [SBVR, 2008]. The text is injected into a model thanks to a simple metamodel for sentences and words annotating the position of words in the text. A simple transformation uses a lexicon to label each word with a set of possible syntactical categories. We have then defined a metamodel, called *Syntax*, where we adapted configuration grammars[Estratat and Henocque, 2004] to SBVR and the model-driven engineering context. The text (as labeled words) is fed into a constraint-based configurator using the relaxed version of Syntax. The result of the configuration process, acting as a syntax and grammar analysis, is a finite

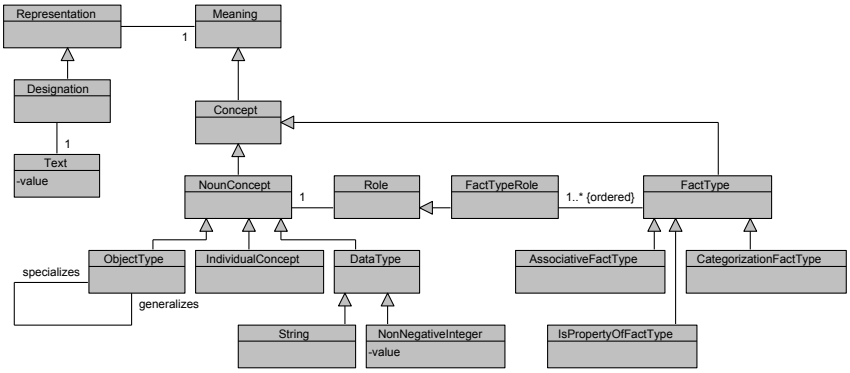

Figure 1: Extract of the SBVR metamodel: meanings

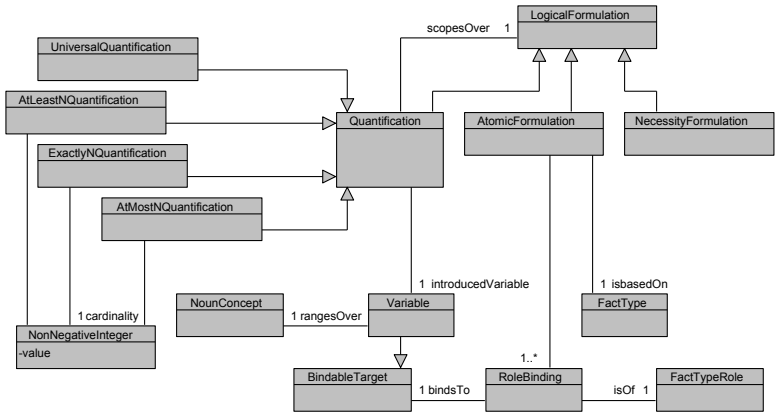

Figure 2: Extract of the SBVR metamodel: logical formulations

model that conforms to the Syntax metamodel. This model is then transformed with a usual model transformation tool (here, ATL) to a model conforming to the SBVR metamodel through a set of rules using the grammatical dependencies found during configuration. This SBVR terminal model may then be processed again with ATL to obtain a corresponding UML model.

2.5 Running example

An example will be used throughout the paper to illustrate the approach. The considered text is composed of three sentences defining a simple CS:

(1) Each company sells at least one product.
(2) Each product is sold by exactly one company.
(3) A software is a product.

This relatively simple example still embeds many important concepts. From the language parsing viewpoint it uses nouns, active and passive verbs, as well as different quantifiers. From the modeling viewpoint it shows the notions of classes, inheritance and relations with cardinalities. In the following Sections, we will show how each main task is applied on those sentences.

3 Parsing English language for SBVR

Parsing natural languages is one of the major challenges of AI. Considering the difficulty of the task, many efforts have been focused on the more accessible field of controlled languages [Kittredge, 2003], where ambiguities are discarded from the language. Among the existing approaches, [Estratat and Henocque, 2004] shows how property grammars [Blache and Balfourier, 2001] can be captured into a configuration model in order to parse a subset of French with a constraint-based configurator. The resulting parser does not inherit the deterministic behaviour of most parsers and is designed to be adapted to different grammars. We have modified and extended this method for English and SBVR. In our MDE approach, the proposed configuration model is defined as a metamodel called Syntax.

3.1 Syntax metamodel

A fragment of this metamodel (most classes and relations) is presented in Figure 4. Syntax captures three main informations from the input text: syntactical categories, grammatical dependencies and SBVR semantics.

Syntactical categorization
In order to obtain a syntactic tree from a sentence, we have adapted the property grammar model from [Estratat and

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

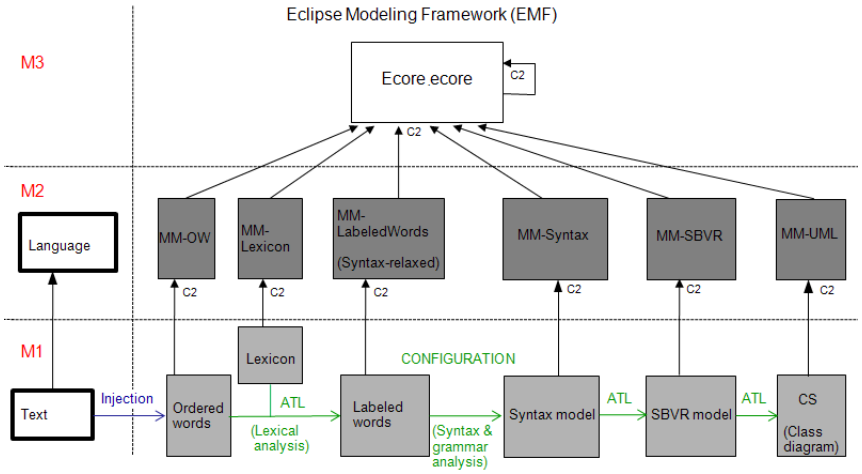

Figure 3: Process overview

Henocque, 2004] to English. The main class of the model is *Cat*, which denotes a syntactical category. A category is *terminal* when it is directly associated to a single word. Such categories include *NCat* (noun), *VCat* (verb) or *DCat* (determiner). Those categories may be further specialized: a verb is either transitive (*TVCat*) or intransitive (*ITVCat*). The possible categories of a given word are obtained with the lexicon in the previous transformation. A category is *non-terminal* when it is composed of other categories. *SentenceCat* (sentence), *NPCat* (noun phrase), *VPCat* (verb phrase) are the main non-terminal categories. A set of constraints further defines the acceptable categorizations. Such constraints involve for instance the categories constituents or relative positions. Here are some example constraints specified in OCL:

- *Each verb phrase for which the verb is transitive is composed of at least one noun phrase*. This constraint applies to the relation *isComposedOf* of a category:

```
context VPCat
inv: head.oclIsTypeOf(TVCat)
        implies isComposedOf->
          exists( elt.oclIsTypeOf(NPCat) )
```

- *A verb phrase is always preceded by a noun phrase*. The constraint applies to the attributes *begin* and *end* of categories, obtained from their associated word(s):

```
context SentenceCat
inv: isComposedOf->exists(
        elt.oclIsTypeOf(NPCat)
        and elt.end < vp.begin )
```

Grammatical dependencies

We have extended the syntactic model so that grammatical dependencies appear as explicit relations between categories. Similarly to the syntactical part, a set of constraints defines the acceptable constructions. Here are some example constraints specified in OCL:

- *The subject of an active verb occurs before the verb phrase*:

```
context VPCat
inv: (head.voice = 'active')
        implies head.subject.end < begin
```

- *The head of a verb's subject shares the same plural*:

```
context VCat
inv: plural = subject.head.plural
```

SBVR semantics

We have extended the metamodel with the main concepts of the SBVR metamodel. SBVR semantics are assigned to syntactical categories through the *expresses* relation. Again, a set of constraints governs the possible assignments. Here are some examples:

- *A transitive verb expresses a fact type.*:

```
context TVCat
inv: not expresses.oclIsUndefined()
        and expresses.oclIsKindOf(FactType)
```

- *The head of a subject of a verb expresses either an object type or an individual concept*:

```
context VCat
inv: subject.head.expresses.
        oclIsKindOf(ObjectType)
        or subject.head.expresses.
        oclIsKindOf(IndividualConcept)
```

About SBVR concepts singularity A critical issue in assigning SBVR semantics to categories is the one of concepts singularity. More precisely, the same SBVR concept may be expressed in different sentences (or even in the same sentence). Consider for instance the first two sentences of our running example: the concepts "Company", "Product" and "To sell" are expressed multiple times. We obviously wish to avoid creating duplicate SBVR elements in the resulting model. A set of constraints forces the uniqueness of SBVR elements based on equivalency statements. In the case of elements of class *ObjectType*, the disambiguation is done on the words base designation:

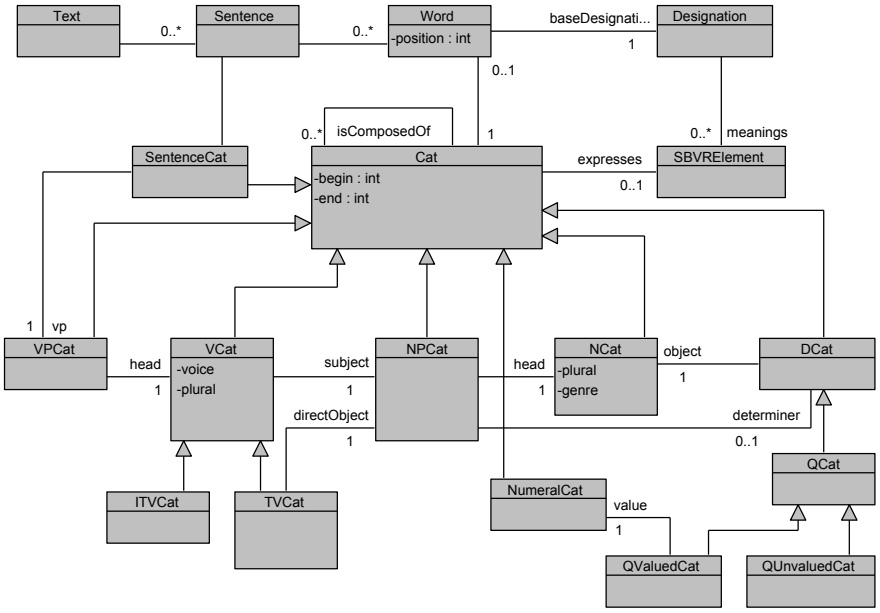

Figure 4: Extract of the Syntax metamodel: syntax and grammar

```
inv: NCat.allInstances()->
       forAll(n1,n2 : NCat |
       (n1.word.baseDesignation =
        n2.word.baseDesignation)
       = (n1.expresses = n2.expresses))
```

Note that since the base designation is used, different forms of the same word are still recognized (i.e "products" and "product" are matched). The same principle is applied to other SBVR elements such as fact types.

3.2 Parsing process and result

As explained previously, the input of the configuration process is a model of a relaxed version of the target metamodel. In our context, the input is a set of interconnected objects of type Text, Sentence, Word, Designation and Cat. Indeed, for each word, the preceding transformation, using the lexicon, has provided its properties (plural, voice, etc.), base designation (the base designation of the word "has" is "to have"), and candidate syntactical categories (the word "one" may be a noun, a numeral or an adjective).

The result of the configuration process is one (or more) terminal model(s) satisfying the configuration model constraints, when such a model exists. Figure 5 shows (a fragment of) the generated model for the first sentence.
It should be noted that this parsing process is not deterministic: due to language ambiguities, multiple solutions may be valid for the same request. For instance, consider the sentence "MyCode is a software". Without other sentences or lexicon information, it may not be possible to decide whether the *NounConcept* "MyCode" is an *ObjectType* (a specialization of Software) or an *IndividualConcept* (an instance of Software). Rather than arbitrary deciding on one model, search allows to generate all valid solutions which can be later compared, or even to optimize the target model based on some preferences. In this regard, our approach offers a higher flexibility than most existing parsers.

4 Transforming the syntactical model into a SBVR model

The model produced by the parsing process exhibits the SBVR semantics expressed by (groups of) words. Using this information together with grammatical dependencies between elements, we are able to construct a complete SBVR model of the input text. This is achieved with model transformation using the ATL language [Jouault and Kurtev, 2005]. Presenting each rule of this transformation in details is outside the scope of this article. However we propose an overview of its main principles.[1]

4.1 Mapping overview

A first straightforward mapping is obvious: each SBVRElement of the Syntax metamodel has its counterpart in the SBVR metamodel and therefore implies the creation of the target model element. However the relations between SBVR model elements are not exhibited in the source model and may require additional (intermediate) SBVR elements. A set of rules therefore allows to derive them from the grammatical relations of the source model.

As an example, consider the following rule that generates a (binary) AssociativeFactType and its roles from a transitive verb, its subject and direct object. It may be informally expressed as follows: "For an AssociativeFactType B expressed by a verb V in the source model, create an AssociativeFactType B' in the target model, with two roles R1 and R2, where

[1] Source code and documentation of all presented transformations have been submitted as a contribution to the Eclipse ATL project and are available on http://www.eclipse.org/m2m/atl/

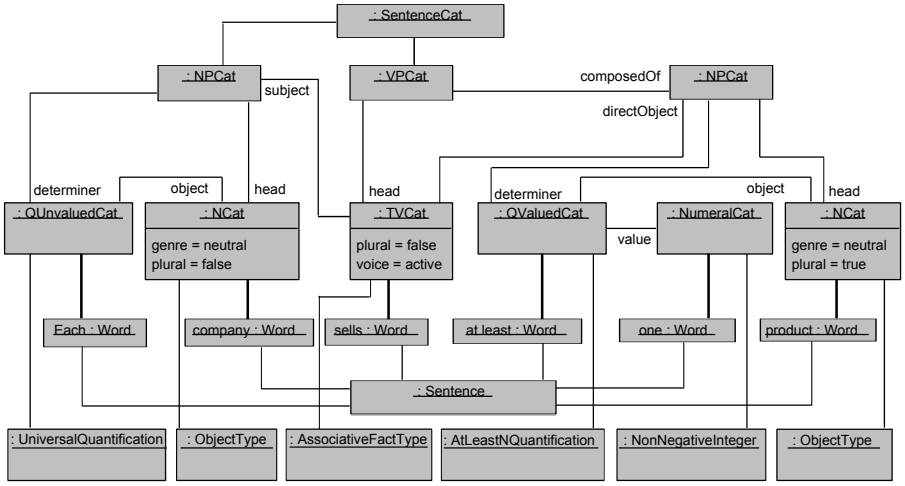

Figure 5: Running example: fragment of a Syntax terminal model

R1's nounConcept is the target NounConcept of V's subject, and R2's nounConcept is the target NounConcept of V's direct object". The rule creates intermediate elements (the roles) and uses them to relate SBVR elements. Note that some of these elements (target NounConcepts of the subject and direct object) are created by a different rule.

Moreover, the transformation allows to create attribute values from a source information having a different datatype. Indeed, consider the word "one" in the first sentence of our running example. In the source model, the word is associated to the category *NumeralCat*, expressing a *non-negative integer* in SBVR semantics. The rule that creates the target model element is able to assign a value to the attribute *value* of type *Integer*.

4.2 Transformation process and result

Once the transformation is complete, we obtain a model that conforms to SBVR, leaving aside the syntactical and grammatical information of the text. Figure 6 shows a fragment of the generated SBVR model for the first sentence of our running example.

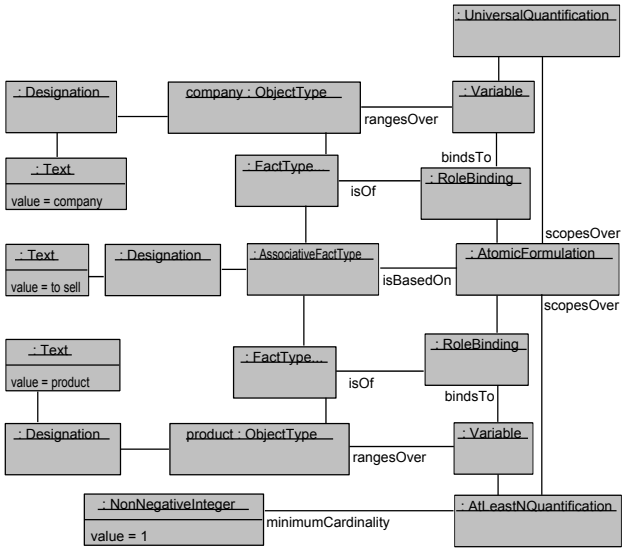

Figure 6: Running example: fragment of the SBVR terminal model

5 Transforming the SBVR model into a UML model

Once a valid SBVR model has been generated, it is possible to transform it into a corresponding UML model of the CS using ATL. The target metamodel is of UML class diagrams.

5.1 Mapping overview

Some examples of the mapped concepts are presented in Table 1 where the dotted notation is used to navigate classes attributes and relations. The correspondence between concepts is quite natural: an ObjectType becomes a Class, an AssociativeFactType becomes an Association, a CategorizationFactType denotes inheritance (Generalization in UML), an IsPropertyOfFactType refers to an attribute, an IndividualConcept becomes an InstanceSpecification, etc. Linked concepts and values are also quite explicit. However, most of the rules do not realize a straight one-to-one mapping but imply additional conditions, target elements from other rules, etc. For instance, consider the mapping for the SBVR concept *AtLeastNQuantification*. The *Property* for which *lowerValue* is assigned is the one obtained by transforming the *AssociativeFactType* that is target of the relation *AtLeastNQuantification.scopesOver.isBasedOn*. Ordered relations also play a role: the first role of a categorization denotes the general class, whereas the second one refers to the specific class.

5.2 Transformation process and result

Once the transformation is complete, we obtain a UML specification of the CS. Figure 7 shows a UML terminal model

SBVR concept	UML Concept
ObjectType	Class
ObjectType.Designation.Text.value	Class.name
DataType	DataType
IndividualConcept	InstanceSpecification
AssociativeFactType	Association
AssociativeFactType.Designation.Text.value	Association.name
AssociativeFactType.FactTypeRole#1	Property (Association.memberEnd)
AssociativeFactType.FactTypeRole#1.nounConcept	Property.classifier
CategorizationFactType	Generalization
CategorizationFactType.FactTypeRole#1	Generalization.general
AtLeastNQuantification.minimumCardinality.value	Property.lowerValue
AtMostNQuantification.maximumCardinality.value	Property.upperValue

Table 1: Excerpt of the mapping from SBVR metamodel concepts to UML metamodel concepts

obtained for our running example.

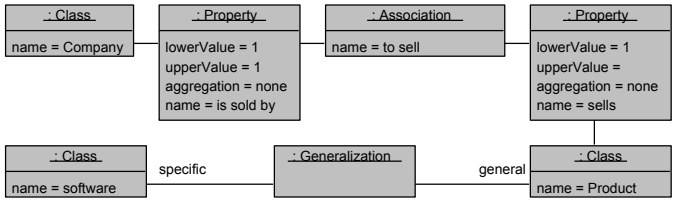

Figure 7: Running example: fragment of the UML model

6 Implementation and experiments

6.1 Implementation

The proposed approach has been integrated into an Eclipse-based model-driven engineering framework. To those aims, each presented metamodel has been defined using the KM3 metamodelling language [Jouault and Bézivin, 2006], which offers an automatic conversion to the EMF's ECore format [ECORE, 2009]. These ECore metamodels are source and target metamodels of the proposed ATL transformations. The configuration tool JConfigurator has its own modelling language and currently offers only XML inputs and outputs. Therefore the Syntax metamodel is also defined directly within the tool as the configuration model. At runtime, the configuration request model is projected to XML in order to be parsed, and the XML representation of the solution is then injected into a model. This model (in XMI format) is passed over to the remaining ATL transformations.

6.2 Experiments

We provide early experiments on the running example, conducted on an Intel Core2Duo 3Ghz with 3GB of RAM. Table 6.2 shows the results. We first parsed each sentence separately and then multiple sentences at once. Only the configuration step is displayed since ATL transformations run in negligible times (less than 0.2 seconds).

Parsing is efficient for separated sentences but the time required for the search task quickly increases with the whole text. This is due to the size of the source model which directly impacts the search space of the configurator. On the other hand, the ATL transformations are able to handle larger models. Splitting the tasks is thus positive for performance and could be investigated further so as to reduce the complexity of

`sentence(s)`	Text to Syntax		
	Time	*Vars*	*Constraints*
(1)	0.26	527	1025
(2)	0.20	526	1022
(3)	0.19	475	885
(1)+(2)	0.38	973	2819
(1)+(2)+(3)	1.41	1328	5312

Table 2: Experiments on the example (times in seconds)

the configuration model to the minimum required for syntactical analysis. We currently focused on a straightforward integration of the configurator with a configuration model covering the whole Syntax metamodel. Future steps are, on the one hand, to extract the combinatorial core of the metamodel, and on the other hand, to allow a further specified request through different relaxation levels of the source metamodel. It should however be emphasized that these are early experimental results on a known hard problem. No optimizations have been applied to the configuration engine such as heuristics or symmetry breaking techniques, which are known to drastically reduce the computation times. Another alternative envisioned is to perform an incremental parsing of the text, sentence by sentence, and then use the ATL multiple source capabilities to unify the resulting SBVR models. The successful parsing of our example however proves the feasibility of our approach.

7 Related work

In [Cabot *et al.*, 2009], a procedure for performing the reverse transformation is described: from a UML/OCL description of a CS, the authors show how it can be transformed into a SBVR terminal model using ATL, and then paraphrased with structured English text. Combining the two approaches is thus promising. Indeed, designing a CS often requires several discussions between stakeholders for refinements, and maintaining a CS leads to frequent evolutions. The combination would allow to switch from one representation to another automatically.

There have been previous research on using constraint programming techniques in MDE, mostly about animation of relational specifications or model verification. [Cabot *et al.*, 2007] transforms UML/OCL specifications into a constraint satisfaction problem in order to check satisfiability. [Dinh-Trong *et al.*, 2006] proposes a similar method, although the underlying solver (Alloy [Jackson, 2000]) is based on SAT. Both approaches inherit the limitations of the target formalism in which specifications must be translated, whereas the configuration paradigm is expressive enough to directly capture models representation. Moreover, to the best of our knowledge, this is the first time that a constraint-based search is embedded in MDE as a model transformation tool.

With respect to the domain of controlled language parsers, our approach differs from most existing methods (such as ACE[Schwitter and Fuchs, 1996]). Indeed these parsers do not accept ambiguous grammars (i.e not context-free), whereas we are able to parameterize the level of accepted

ambiguities, thus allowing to define a trade-off between language coverage and computation efficiency.

8 Conclusion and future research

We have described a method which allows parsing a CS specification expressed in English text into a UML class diagram. To those aims, we proposed SBVR as an intermediate layer. The originality of our approach is the use of an advanced object-oriented constraint-programming search technique (configuration) as a model transformation tool integrated in a MDE environment. Early experiments are provided as a proof-of-concept. Moreover, the proposed parser is flexible with respect to language coverage and disambiguation, allowing to build domain specific languages that are not restricted to context-free grammars. There are many perspectives to this work. First, the metamodels can be extended to capture an increased portion of English and SBVR. The expressed meanings will then probably require to generate OCL constraints in addition to UML. Other target formalisms can also be considered such as OWL or Rule Systems. The experiments clearly show that there is a need for performance improvement in the search-based transformation. The leading direction is to reduce the search space by isolating the metamodel's combinatorial core, thus further decomposing the problem. Finally, the described use of configuration as a model transformation could benefit to other complex transformations that require searching for a target model instead of applying deterministic rules to the source model.

References

[Blache and Balfourier, 2001] Philippe Blache and Jean-Marie Balfourier. Property grammars: a flexible constraint-based approach to parsing. In *IWPT*. Tsinghua University Press, 2001.

[Cabot *et al.*, 2007] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *ASE*, pages 547–548. ACM, 2007.

[Cabot *et al.*, 2009] Jordi Cabot, Raquel Pau, and Ruth Ravents. From uml/ocl to sbvr specifications: a challenging transformation. *Information Systems Elsevier*, 2009.

[Dinh-Trong *et al.*, 2006] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test uml design models. In *ISSRE*, pages 95–104. IEEE Computer Society, 2006.

[ECORE, 2009] EMF : http://www.eclipse.org/modeling/emf/, 2009.

[Estratat and Henocque, 2004] Mathieu Estratat and Laurent Henocque. Parsing languages with a configurator. In *Proceedings of the European Conference for Artificial Intelligence ECAI'2004*, pages 591–595, August 2004.

[Felfernig *et al.*, 2002] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Configuration knowledge representation using uml/ocl. In *UML*, volume 2460 of *LNCS*, pages 49–62. Springer, 2002.

[Jackson, 2000] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT FSE*, pages 130–139, 2000.

[Jouault and Bézivin, 2006] Frédéric Jouault and Jean Bézivin. Km3: A dsl for metamodel specification. In *FMOODS*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.

[Jouault and Kurtev, 2005] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[Junker and Mailharro, 2003] Ulrich Junker and Daniel Mailharro. The logic of (j)configurator : Combining constraint programming with a description logic. In *proceedings of IJCAI'03*. Springer, 2003.

[Junker, 2006] Ulrich Junker. *Configuration*, volume Handbook of Constraint Programming, chapter 26. Elsevier, 2006.

[Kaljurand, 2008] Kaarel Kaljurand. Ace view - an ontology and rule editor based on controlled english. In *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[Kittredge, 2003] R. I. Kittredge. *Sublanguages and controlled languages*. Oxford University Press, 2003.

[Mailharro, 1998] Daniel Mailharro. A classification and constraint-based framework for configuration. *AI in Engineering, Design and Manufacturing, (12)*, pages 383–397, 1998.

[QVT, 2008] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0*, 2008.

[Sabin and Freuder, 1996] Daniel Sabin and Eugene C. Freuder. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[SBVR, 2008] *Semantics of Business Vocabulary and Business Rules (SBVR) 1.0 specification: http://www.omg.org/spec/SBVR/1.0/*, 2008.

[Schwitter and Fuchs, 1996] Rolf Schwitter and Norbert E. Fuchs. Attempto controlled english (ace) a seemingly informal bridgehead in formal territory (poster abstract). In *JICSLP*, page 536, 1996.

[Soininen *et al.*, 2001] Timo Soininen, Ilkka Niemela, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming*, pages 195–201, 2001.

[Stumptner and Haselböck, 1993] Markus Stumptner and Alois Haselböck. A generative constraint formalism for configuration problems. In *Advances in Artificial Intelligence: Proceedings of AI*IA'93*, pages 302–313. Springer, 1993.

[Stumptner, 1997] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2):111–125, June 1997.

Characterization of 26 configuration models

Juha Tiihonen
Department of Computer Science and Engineering
Helsinki University of Technology
Juha.Tiihonen@tkk.fi

Abstract

Characterization of configuration problems has remained limited. This work characterizes 26 configuration models with numerous indicators of size and degree of application of modeling mechanisms including inheritance and application of advanced compositional structure. The original goal of modeling was to evaluate and demonstrate the applicability of WeCoTin configurator and PCML modeling language to industrial configuration problems. The main contribution of the paper is in providing probably the first multi-faceted detailed characterization of a relatively large number of configuration problems. Additionally, aspects for characterizing configuration models were identified.

1 Introduction

Due to active research in the configuration domain, several formalisms for representing configuration knowledge have been proposed, and configurators are used to support day-to-day business in many companies. However, characterization of practical configuration problems has remained limited in the literature. Of course, a number of individual cases have been documented thoroughly, most importantly the R1/XCON [Barker, et al. 1989] and the VT/Sisyphus elevator configuration problem [Schreiber, et al. 1996], not forgetting characterizing the domain and configuration models when describing configurator implementations, e.g. [Fleischanderl, et al. 1998]. Further, classification of configuration tasks has been proposed [Wielinga, et al. 1997, Haag. 2008], which requires characterization of the tasks as a basis for classifications.

There are several reasons why configuration tasks should be characterized. These include enabling evaluation of effectiveness of specific representation formalisms and modeling constructs, gaining understanding on the nature of different configuration tasks, which in turn could enable supporting tools that better match practical problems and facilitates development of benchmarks, and enabling classification of configuration tasks. Configuration models, related configuration tasks and their IT support could be characterized from several perspectives. These include the size of the models, computational performance, and complexity, effectiveness of specific modeling techniques related to specific configuration tasks as defined by the company offering. Less technical views cover aspects of practical interest such as the proportion of cases covered by the configuration models, completeness of the models in terms of business requirements, usability and different aspects of utility provided and sacrifices required.

This work reports a part of evaluation of the WeCoTin configurator [Tiihonen, et al. 2003] and especially its modeling capabilities. 26 configuration models were created to evaluate and demonstrate the applicability of WeCoTin and PCML to real industrial configuration problems, which demonstrates the high-level efficacy of the constructs. The models will be characterized with numerous indicators of size and degree of application of different modeling constructs. Five tables provide characterizations models.
This paper is structured as follows. In Section 2 the applied modeling language will be described. Section 3 gives an overview of the configuration models and their background. Section 4 describes component type hierarchy, overall configuration model size, and price modeling. Section 5 details the compositional structure of the models, Section 6 attributes, and Section 7 constraints. Finally, discussion, future work and conclusions are presented in Section 8.

2 Product Configuration Modeling Language

The configuration models have been modeled with *PCML* (Product Configuration Modeling Language) [Tiihonen, et al. 2003, Peltonen, et al. 2001]. PCML is object-oriented, declarative and it has formal implementation-independent semantics. The semantics of PCML is provided by mapping it to weight constraint rules [Soininen, et al. 2001]. The basic idea is to treat the sentences of the modeling language as short hand notations for a set of sentences in the weight constraint rule language (*WCRL*) (Soininen et al. 1998). PCML covers a practically important subset of a synthesized ontology of configuration knowledge [Soininen, et al. 1998].

The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define the parts and

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Model	Company & Description Status and validation
1 Compressor FM	Gardner Denver. Compressor family FM series. 18-40kW compressors. Internal view for sales persons. Demonstrated integration to automatic manufacturing completion (EDMS2), and e-commerce site Intershop 4. Delivery time calculation. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Company workers had a few dozen configuration sessions over the web. Model used in empirical performance testing. Manually analyzed the number of possible configurations, which matches the number of answer sets (configurations) generated by the inference engine.
2Compr FM sc	Gardner Denver. Model 1 Compressor FM above augmented with 13 pre-selection packages to represent agreed-on customer standards with their defaults, usually enforced with soft constraints. Behaves as intended.
3 Compr FS	Gardner Denver compressor family, 11-18 kW FS series compressors. Internal view for sales persons. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. In addition, company workers had configuration sessions over the web. Model used in empirical performance testing. The manually analyzed number of possible configurations matches the number generated by the inference engine.
4 Compr FX	Gardner Denver compressor family, 4-10kW FX series compressors. Internal view for sales persons. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. In addition, company workers had configuration sessions over the web. Model used in empirical performance testing. The manually analyzed number of possible configurations matches the number generated by the inference engine.
5Compr FL	Gardner Denver compressor family, 45-80 kW FL series compressors. Internal view for sales persons. Complete model, seems to work, no validation in company.
6Compr M	Gardner Denver compressor family. 75-160kW M series compressors. Internal view for sales persons. Complete model, seems to work, no validation in company.
7 KONE old	KONE maintenance service contracts, older company offering. Some additional options that can be specified after contract made (billing options, e-notification). Some one-time service extras like long-term maintenance plan, condition check. Extensive on-line help texts for salespersons developed together with company, shown with the description mechanism. Complete model with extras. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Installed to product manager computer for test use.
8 KONE new	KONE maintenance service contracts. Options of newer maintenance service contracts. Additional options that can be specified after contract made (billing options, extensive e-services). Some one-time service extras like long-term maintenance plan, condition check. Extensive on-line help texts for sales persons. Complete model with extras. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Installed to product manager computer for test use.
9 Bed	Not public 1. Real hospital bed product line based on order forms and additional information. Complete model. Demonstration to company and focus group on matching company view of product configurability.
10Fireplace	Not public 2. Modular fireplace. Technology demonstration with CAD vendor. Simple product. Integrated with 3D CAD to visualize configuration changes. Complete model. Validation with CAD vendor.
11 Patria Pasi	Patria Vehicles. Military vehicle. For company internal systematic documentation of available productized options. Complete model with respect to standardized offering. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Configuration model validated in internal test use in company.
12 Dental	Not public 3. Real integrated dental unit and patient chair. Most difficult to configure product of the company. Complete model. Demonstration to company, brief focus group, expert validation.
13 X-ray	Not public 3. Real X-ray unit for dentists. Product designed with ease of configuration in mind. Complete model. Demonstration to company, brief focus group, expert validation.
14 Vehicle	Not public 4. Self-moving machine industry product. Real product based on order forms and interviews. Partial model for demonstration purposes representing about half of the sales view of the product. Numerous optional parts and some simple constraints were excluded. Despite omissions the model reflects quite well the nature of sales configuration of this vehicle. Test-used by company stakeholders over the web. Functionality found satisfying "better than our commercial product". The manually analyzed number of possible configurations matches the number generated by the inference engine.
15 Insur 1	Tapiola group. Insurance coverage for families (persons, travel, car, home, cottage) as a combination of insurance products. Demonstration model with a subset of the whole offering, Discretized large domain specification attributes. Demonstrations to company.
16 Insur 2	Tapiola group. Comprehensible insurance coverage for family's person related risks. Non-traditional risk-oriented (not insurance product oriented) way of asking which coverage is desired. These selections are satisfiable with a combination of real products. Mapping to products not performed in model. Demonstration model. Demonstration to company and focus group.
17 Insur 3	Tapiola group. Experimented 4-world model objects-world questions. Solution-world with detailed model of real offered car-related insurance coverage. HUT internal, no company validation.
18 Insur 4	Tapiola group. Comprehensive insurance coverage for families and property. Tested some 4-worlds model ideas. Objects: persons, home, leisure-time apartment, vehicles (cars, motorcycle, boat), forest, domestic animals (dogs, horses, cats). Solutions corresponding real insurance products and their availability, modeled in detail. However, risks coverage for home and cottage is selectable with excessively small granularity. HUT internal, no company validation.

19 Mob Subscr 1	Elisa (Radio-linja). Demonstration on configuring mobile subscription and its value-added services while taking into account phone capabilities. Model covers only aspects considered interesting for the demonstration. Information acquired from public company web-site. Demonstration model, demonstrated to company stakeholders.
20 Mob Subscr 2	Elisa. Real mobile subscription + phone bundle as basis. Reverse-engineered from company web site + added needs analysis questions to identify suitable product options with soft constraints HUT internal validation against offering., presentation in an open seminar for the Finnish industry.
21 Mob Subscr 3	Telia-Sonera. Demonstration on selecting mobile subscription and some value-added services based on usage characteristics. Implemented with hard and soft constraints. Based on public information from company website. No validation. Behaves as intended by the modeler.
22 Broadband	Telia-Sonera. Real broadband subscription product line. 4 worlds model demonstration. Re-engineered from company website, added customer and needs analysis questions, and aspects of delivery process that are configured based on selected options. Soft constraints warn when selections do not correspond to needs. Complete model with extras. Demonstration to the company.
23 Linux	Debian Linux Familiar distribution configuration model over 6 points of time and several software versions. Information gathered from Debian Linux version compatibility lists, configuration model generated via script by mapping package descriptions into PCML [Kojo, et al. 2003]. A newer version than that in Kojo et al. was characterized. No validation, "seems and behaves right", although with slow performance.
24 Iced	None. Demonstration: minimal fictive car model for ICED conference article describing WeCoTin configurator. No validation. Behaves as intended by the modeler.
25Wecotin car	BMW. Demonstration model based on a subset of a real car, identified configuration rules from company website No validation. Behaves as intended by the modeler.
26 CarDiss	BMW. Demonstration model based on 25 WeCoTin Car, with some extra fictive features to demonstrate higher cardinality. No validation. Behaves as intended by the modeler.

Table 1. Identification, description, status and validation of the configuration models.

properties of their *individuals* that can appear in a configuration. A component type defines its compositional structure through a set of *part definitions*. A part definition specifies a *part name*, a non-empty set of *possible part types* (*allowed types* for brevity) and a *cardinality* indicating the possible number of parts. A component type may define properties that parametrize or otherwise characterize the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition* indicating if the property must be given a value in a complete configuration. Component types are organized in a *class hierarchy* where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner. When a type inherits data from a supertype, the type can use the inherited data "as such" or it can modify the inherited data by means of *refinement*. Refinement is semantically based on the notion that the set of potential valid individuals directly of the subtype is smaller than the set of valid individuals directly of the supertype. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type can be used in a configuration. Constraints associated with component types define conditions that a correct configuration must satisfy. A constraint expression is constructed from references to parts and properties of components and constants such as integers. These can be combined into complex expressions using relational operators and Boolean connectives.

3 Model background, identification, characterization, status and validation

PCML and WeCoTin have been used to model and configure the complete sales view of 14 real products or services, and partial sales view of 8 products or services. In the complete sales views all known configurable options of the products or services have been modeled. Configuration models of some products or services contained extra features that are not normally taken into account during sales configuration. Three additional demonstration models are included in the characterizations. Some configuration models were created in early phases of WeCoTin construction, some after completion of the development project.

In most cases, order forms, brochures, and other documentation were used as a basis for modeling, and company representatives were contacted for additional information before showing the results as demonstrations. In some cases it was possible to re-engineer configuration model information from company websites.

The WeCoTin modeling tool was instrumented to provide the characterizing metrics based on static configuration model analysis presented in Tables 2-5. The configuration models come from the following domains:

- Eight models are from machine industry and come from three companies (5 compressors (1 twice), an undisclosed vehicle, and one military vehicle).
- Three models from two companies are from healthcare domain (2 dentist equipment product families, a hospital bed family.)
- Four models from two companies are from telecommunications domain (3 mobile and 1 broadband subscriptions).
- Three models from one company are from insurance domain.
- Two models from one company represent two generations of maintenance contracts of elevators.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

- One model is software configuration (Debian Linux Familiar with package versions).
- One model demonstrates configuration of a modular fireplace.
- Three models are pure demonstration models – two are based on a subset of a real car, and one is fictional.

Table 1 identifies the models and briefly characterizes the domain of each configuration model. Each configuration model is identified with a unique numeric identifier that remains the same in each table. Model names have been abbreviated in later tables due to space constraints.

Table 2 details the degree of configuration model completeness and model validation status. Five models from three companies have been test-used by company representatives. In addition, configuration demonstrations and immediately following focus groups have been used for validation of additional seven configuration models.

4 Taxonomy, model size, and pricing

Table 2 provides characterization of component type hierarchy used in the models, overview of model size, and .information on price modeling.

One model (23, Linux) was significantly larger than others and semi-automatically generated. Discussions on model characterizations exclude this model, but averages and totals are calculated with and without it.

Numbers of abstract, concrete, and total component types contribute to the size of a configuration model, and are shown in corresponding columns of Table 2. The total number of component types varied from one 1 to 626, the median was 9 and average was 24. The number of direct subtypes of abstract types (other than root of component type hierarchy `Component`) ("Subtypes") characterizes the number of component types organized in a type hierarchy. Interpreted as a percentage "% as subtypes", the figure varied from 0% to 100%, with average without Linux being 59% and median 46%.

Each selectable attribute or part of a component individual being configured generates a *question* during a configuration process. The number of questions in a configuration model ("Questions") roughly characterizes the size of each configuration model and the related configuration task. In a typical configuration model without redundant concrete component types, each question might have to be answered while configuring a product. All possible questions may not be asked in a configuration session, because an individual of a specific type is not necessarily selected into a configuration, or if some attributes or parts are defined to be invisible to the user or to have a fixed value. On the other hand, if several individuals of a component type are in a configuration, the number of questions may be multiplied. The average was 61 questions per configuration model, and roughly 5.4 questions per concrete type (excluding Linux).

Model	Total types	Abstract types	Concrete types	Subtypes	% as subtypes	Questions	% questions in root	Constraints	Price
1 C FM	9	2	7	4	44	31	58	17	adv
2 CFm sc	9	2	7	4	44	31	58	17	adv
3 C FS	3	0	3	0	0	24	88	14	adv
4 C FX	1	0	1	0	0	20	100	23	adv
5 C FL	9	2	7	4	44	28	64	13	no
6 C M	3	0	3	0	0	23	91	14	no
7 KO old	5	0	5	0	0	28	79	13	no
8 Ko new	15	3	12	7	47	77	4	1	no
9 Bed	31	8	23	27	87	34	76	10	basic
10 Firepl	7	1	6	4	57	4	75	0	no
11 Pasi	5	1	4	2	40	79	95	13	no
12 Dental	64	11	53	43	67	109	3	36	no
13 X-ray	11	2	9	4	36	37	41	3	no
14 Vehicl	28	4	24	9	32	24	75	7	basic
15 Ins 1	8	2	6	5	63	30	20	4	no
16 Ins 2	62	13	49	56	90	49	20	0	no
17 Ins 3	11	3	8	5	45	41	29	14	no
18 Ins 4	37	11	26	34	92	242	5	84	no
19 Mob 1	4	0	4	0	0	18	56	6	basic
20 Mob 2	39	9	30	38	97	65	25	28	basic
21 Mob 3	5	1	4	3	60	21	38	6	no
22 Broad	66	15	51	64	97	485	1	43	no
23 Linux	626	1	625	624	100	4369	14	2380	no
24 Iced	8	2	6	5	63	4	75	3	basic
25 Wcar	6	1	5	2	33	10	60	3	basic
26CarDis	10	2	8	5	50	12	58	3	basic
Total	1082	96	986	949		5985		2755	
Total no Linux	456	95	361	325		1526		375	
Average	18	4	14	13	48	227	50	106	
Avg no Linux	24	5	19	18	59	61	52	15	
Median	9	2	7	5	46	31	58	13	
Min	1	0	1	0	0	4	1	0	
Max	626	15	625	624	100	4369	100	2380	

Table 2. Use of pricing mechanisms, company look, and component type hierarchy in the configuration models.

Especially simpler models were often centered on the configuration type that is the root of the compositional hierarchy. The degree of such concentration is characterized by the proportion of questions defined in the configuration type. Column "% questions in root" specifies this proportion. On the average about half (50%), and median 58% of questions were in the root component type, with a large scale of variation.

The total number of constraints ("Constraints") specified in the component types of each configuration model varied

largely, but usually remained in a few dozen at maximum. More details will be given in Section 2.5.

Column "Price" indicates which of two price calculation mechanisms, if any, was used. The "basic" mechanism considers additive prices: each component individual can have a base price determined by its type, and each attribute value can specify additional price. The sum of component individual and their attribute value prices is the price of the configuration. Four real products and three demonstration models applied this pricing mechanism.

A more advanced calculation mechanism ("adv") [Nurmilaakso. 2004] performs definable calculations as function of the current configuration, configuration model, and external data. Values are provided to calculations when condition expressions examining a configuration evaluate to true. Three products were priced with this mechanism.

Prices were often omitted either due to indicated sensitivity or to constrain resource usage. The simple mechanism would have been sufficient for other products except compressors and insurance products.

5 Compositional structure

Table 3 exhibits details of applying compositional structure in the modeled cases. An indication on the number of parts in a configuration model is given by the number effective parts in concrete types of the model. The number of effective parts ("Effective parts") is the sum of inherited and locally defined parts in concrete types. The average number of effective parts per concrete component type ("Eff. parts / concrete") characterizes the breath of the configuration tree and average application of the compositional structure as a modeling mechanism. On the average, 10 parts were defined in each configuration model. Median was 4. The average of 0.6 effective parts in each concrete type indicates moderate usage of the compositional structure.

Inheritance in the compositional structure was used on the average less frequently than direct part definitions in concrete types: Eight models applied inheritance of parts whereas parts were introduced in 25 models. On the average, seven part definitions were defined in concrete types ("Part def in concrete"), and one part per configuration model was defined in abstract types ("Part def in abstract"). Four of the effective 10 part definitions were inherited. Three inherited part definitions were applied as such ("Non-refined inherited"), and one was refined, e.g. to restrict the set of allowed types ("Refined inherited"). Some models applied part inheritance significantly more. For example, in model 12 Dental, 26 (79%) of 33 effective part definitions were inherited ("% inherited parts"). Out of these 6 were refined. The eight models applying part inheritance had 31% (80) of their 257 effective part definitions inherited.

Many configuration models concentrated part definitions on the configuration type. An average configuration type contained 4 part definitions ("Part def in conftype"). In 12 models all parts were defined in the configuration type. The average percentage of part definitions in the configuration type was 70% ("% part def in conf type").

The cardinality of a part definition defines how many component individuals must realize the part in a consistent and complete configuration. On the average, 6 part definitions were optional, that is, with cardinality with 0 to 1 ("0 to 1 cardinality"), and 2 part definitions were obligatory with cardinality 1 to 1 ("1 to 1 cardinality"). Only one demonstration model contained one part definition with a larger maximum cardinality than one ("max cardinality 2+").

Model	Effective parts	Eff. parts / concrete	Part def in concrete	Part def in abstract	Part def in conf type	% part def in conftype	Non-refined inherited	Refined inherited	% inherited parts	0 to 1 cardinality	1 to 1 cardinality'	max cardinality 2+	Enumerated allowed	Effective allowed	% allowed saved	Max allowed
1 C FM	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
2 C Fm sc	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
3 Com FS	1	0.3	1	0	1	100	0	0	0	0	1	0	2	2	0	2
4 com FX	0	0.0	0	0	0	-	0	0	-	0	0	0	0	0	-	0
5 com FL	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
6 com M	1	0.3	1	0	1	100	0	0	0	0	1	0	2	2	0	2
7 KO old	2	0.4	2	0	2	100	0	0	0	1	1	0	4	4	0	2
8 Ko new	19	1.6	10	3	2	11	5	4	47	11	2	0	14	17	18	3
9 Bed	3	0.1	3	0	3	100	0	0	0	0	3	0	5	32	84	12
10 Firepla	2	0.3	2	0	2	100	0	0	0	1	1	0	2	5	60	4
11 Pasi	2	0.5	2	0	2	100	0	0	0	1	1	0	3	3	0	2
12 Dental	33	0.6	7	13	1	3	20	6	79	18	2	0	33	80	59	8
13 X-ray	5	0.6	3	1	2	40	0	2	40	3	1	0	8	8	0	3
14 Vehicl	16	0.7	16	0	12	75	0	0	0	12	4	0	20	24	17	3
15 Insur 1	10	1.7	10	0	6	60	0	0	0	9	1	0	10	10	0	1
16 Insur 2	30	0.6	22	4	10	33	8	0	27	24	2	0	26	29	10	4
17 Insur 3	12	1.5	12	0	11	92	0	0	0	10	2	0	13	13	0	2
18 Insur 4	53	2.0	30	5	12	23	27	0	51	35	0	0	35	46	24	4
19 Mob 1	3	0.8	3	0	3	100	0	0	0	3	0	0	3	3	0	1
20 Mob 2	13	0.4	13	0	12	92	0	0	0	8	5	0	15	27	44	5
21 Mob 3	1	0.3	1	0	1	100	0	0	0	1	0	0	3	3	0	3
22 Broad	32	0.6	30	1	4	13	2	0	6	10	21	0	32	53	40	15
23 Linux	624	1.0	624	0	624	100	0	0	0	624	0	0	624	624	0	1
24 Iced	2	0.3	2	0	2	100	0	0	0	0	2	0	2	5	60	3
25 Car	2	0.4	2	0	2	100	0	0	0	1	1	0	3	4	25	2
26 CarDis	3	0.4	3	0	3	100	0	0	0	1	1	1	4	7	43	3
Total	881		805	30	724		65	15		773	61	1	881	1019		
Total no Linux	257		181	30	100		65	15		149	61	1	257	395		
Average	34	0.7	31	1	28	72	3	1	16	30	2	0	34	39	19	4
Avg no li	10	0.6	7	1	4	70	3	1	17	6	2	0	10	16	20	4
Median	4	0.6	3	0	2	92	0	0	0	1	1	0	6	7	0	3
Minimum	0	0.0	0	0	0	3	0	0	0	0	0	0	0	0	0	0
Maximum	624	2.0	624	13	624	100	27	6	79	624	21	1	624	624	84	15

Table 3. Compositional structure of the configuration models.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

A part is realized with individuals(s) of allowed types. A part definition explicitly enumerates the allowed types. The number of directly enumerated allowed types ("Enumerated allowed") is often smaller than the number of effectively allowed component types ("Effective allowed"), because specifying a supertype as an allowed type effectively specifies the concrete subtypes as allowed types. An average configuration model directly specified 10 and effectively 16 component types in the average 8 part definitions. The average percentage of "savings" was 20% ("% allowed saved"). The relatively low number of allowed types is partially explained by relatively often occurring optional parts with only one effective allowed type. The maximum number of effective allowed types in a part definition ("Max allowed") was on the average and median 4 types, and maximally 15.

6 Attributes

Table 4 exhibits details of applying attributes in the modeled cases. A rough indication on the number of attributes in a configuration model is given by the number effective attributes of the configuration model. The number of effective attributes ("Effective attributes") is the sum of inherited and locally defined attributes in concrete types. Concrete types had a total of 1269 effective attributes. Average without the Linux model was 51 effective attributes and the median was 25. The average of average number of effective attributes per concrete component type ("Effective / concrete") was 4.8, and median of averages was 3.7.

Inheritance of attributes was applied more frequently than inheritance of parts, but less frequently than direct attribute definitions in concrete types: 16 models applied inheritance of attributes whereas all 26 models defined attributes.

On the average a model contained 26 attribute definitions in concrete types ("Defs in concrete"). The 5 attribute definitions in abstract types ("Defs in abstract") expanded to an average of 25 effective attributes in concrete types. The average percentage of inherited attributes in concrete types ("% Inherited") was 23%. Some models applied attribute inheritance more significantly, e.g. 44 - 89% of effective attributes were inherited in some larger models.

Of the 1269 effective attributes, 51% (642) were defined locally ("Defs in Concrete"), and the remaining 49% (627) were inherited. 122 attribute definitions in abstract types ("Defs in abstract") were inherited as such into 537 attributes in subtypes ("Non-refined inherited"), and into 168 attributes in refined form ("Refined inherited"), a total of 705 inherited attributes.[1] Often attribute definitions were concentrated on the configuration type, 54% (14) of the 26 models specified at least 50% of effective attribute definitions there. An average configuration type defined 11 attributes ("Def in config type"). The average percentage of attribute definitions in the configuration type was 46%. ("% part def in conf type").

Model	Effective attributes	Effective / concrete	% Inherited	Attr. definitions	Defs in concrete	Defs in abstract	Def in config type	Boolean	Enumerated string	Integer	Refined inherited	Non-refined inherited	Optional attr. defs	Maximum domain	Domain 1	Domain 2 to 3	Domain4 to 10	Domain 11+
1 FM	27	3.9	22	24	21	3	16	7	13	4	2	4	0	61	0	17	6	1
2 Fm sc	27	3.9	22	24	21	3	16	7	13	4	2	4	0	61	0	17	6	1
3 FS	23	7.7	0	23	23	0	20	5	13	5	0	0	0	51	0	16	6	1
4 FX	20	20.0	0	20	20	0	20	5	10	5	0	0	0	44	0	12	7	1
5 FL	24	3.4	17	22	20	2	16	7	12	3	2	2	0	20	0	13	7	1
6 M	22	7.3	0	22	22	0	20	5	14	3	0	0	0	15	0	11	8	1
7 K old	26	5.2	0	26	26	0	20	8	9	4	0	0	0	7	2	14	5	0
8 k new	58	4.8	81	29	11	18	1	12	7	2	23	24	0	4	0	18	3	0
9 Bed	31	1.3	0	31	31	0	23	17	13	1	0	0	2	7	0	26	5	0
10 Fire	2	0.3	0	2	2	0	1	0	2	0	0	0	0	2	0	2	0	0
11 Pasi	77	19.3	3	76	75	1	73	39	37	0	0	2	0	5	0	67	9	0
12 dent	76	1.4	70	48	23	25	2	28	19	1	3	50	8	10	0	41	7	0
13 xray	32	3.6	44	25	18	7	13	10	7	8	2	12	2	4	8	16	1	0
14Vehi	8	0.3	0	8	8	0	6	3	4	1	0	0	0	22	0	5	2	1
15 Ins1	20	3.3	10	19	18	1	0	4	6	9	0	2	0	11	1	9	8	1
16 Ins2	19	0.4	58	12	8	4	0	4	2	5	0	11	0	4	5	4	2	0
17 Ins3	29	3.6	0	29	29	0	1	17	6	4	0	0	3	5	4	20	3	0
18 Ins4	189	7.3	26	159	140	19	0	144	3	2	0	49	1	5	2	146	1	0
19 Mo1	15	3.8	0	15	15	0	7	10	2	3	0	0	2	13	0	10	2	3
20 Mo2	52	1.7	29	40	37	3	4	25	10	2	15	0	1	5	0	35	2	0
21 Mo3	20	5.0	60	12	8	4	7	12	0	0	0	12	0	2	0	12	0	0
22 Broa	453	8.9	89	81	51	30	0	51	12	3	119	361	1	436	5	58	1	2
23 Linu	3745	6.0	67	1253	1249	4	1	0	1248	3	1248	1248	0	6	551	691	9	0
24 Iced	2	0.3	0	2	2	0	1	0	1	1	0	0	0	2	0	2	0	0
25Wcar	8	1.6	25	7	6	1	4	4	3	0	0	2	0	5	0	5	2	0
26 Diss	9	1.1	22	8	7	1	4	5	3	0	2	2	0	5	0	6	2	0
Total	5014			2017	1891	126	276	429	1469	73	1416	1785	20		578	1273	104	13
Tot. no Linux	1269			764	642	122	275	429	221	70	168	537	20		27	582	95	13
Average	193	4.8	25	78	73	5	11	17	57	3	54	69	1	31	22	49	4	0
Avg no Linux	51	4.8	23	31	26	5	11	17	9	3	7	21	1	32	1	23	4	0
Median	25	3.7	19	24	21	1	5	7	8	3	0	2	0	7	0	15	3	0
Minimum	2	0.3	0	2	2	0	0	0	0	0	0	0	0	2	0	2	0	0
Maximum	3745	20.0	89	1253	1249	30	73	144	1248	9	1248	1248	8	436	551	691	9	3

Table 4. Attributes in the configuration models.

Attribute value types were distributed as follows: of average 31 attribute definitions per configuration model, 17 were Boolean ("Boolean"), 9 were enumerated strings (""Enumerated string"), 3 integers ("Integer"), and 2 unconstrained strings. In total there were 56% (429)

[1] The 705 inherited attributes includes 78 (705-627) attributes inherited to abstract types.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Boolean, 29% (221) enumerated string, 9% (70) integer, and 6% (44) unconstrained string attribute definitions. Unconstrained strings specified additional details such as customer names, addresses, etc. aspects that do not require inference.

Attribute domain sizes remained quite small. A domain of at least 11 possible values ("Domain 11+") was present in about 2% (13) of 717 attribute definitions with a fixed domain. The maximum domain size ("Maximum domain") varied significantly – the largest domain was 436 possible values, in this case enumerated string values. The most common domain size was 2 to 3 possible values ("Domain 2 to 3") in 81% (582) of attribute definitions. 13% (104) of attribute domains were of size 4-10 ("Domain 4 to 10"). Domain of size 1 ("Domain 1") was encountered in 4% (27) of cases, mostly created through attribute value refinements.

20 attributes were defined as *optional*, ("Optional attr. defs") meaning that it is possible to specify in a complete configuration that no value will be assigned to the attribute.

7 Constraints

The number of constraints varied significantly from 0 to 84 (2380 with Linux), an average of 15 per model. The median was 13. On the average, two of the constraints were soft, and the rest were hard. Totally 44 constraints were defined in abstract component types, of these 40 were hard and 4 soft. As with other modeling constructs, definition of a constraint in a supertype causes inheritance to subtypes.

It is not trivial to characterize complexity of constraints. A simple syntactic metric based on parse tree complexity of the resulting constraint expression was calculated. For example, consider the following a constraint:

```
Active_Cruise_Control_Requires_BiXenon
( Cruise_control = true ) implies
($config.Headlights individual of BiXenon)
```

The "complexity" of the example constraint is seven (7). Complexity of a literal, a constant, a variable, a component type reference, element access, an ID-expression, or an element reference in the expression is one. Each operator application counts one plus complexity of each argument.

Typical constraints were small, almost half (45%) of the constraints were of roughly the same complexity as the example constraint above, and 36% a bit more complex.

- 12% (45) of constraints had complexity 0-5
- 45% (170) of constraints had complexity 6-10
- 36% (135) of constraints had complexity 11-20
- 4% (14) of constraints had complexity 21-50
- 1% (4) of constraints had complexity 51-100
- 1% (4) of constraints had complexity 101-1000
- 1% (3) of constraints had complexity over 1000

Maximum constraint complexity varied significantly. The median was as low as 13, and average without Linux was 235. The maximum complexity was 1319. All the compressor models had a large table constraint specifying feasible combinations of values of 5 attributes, each with a relatively large number of rows, which explains the high average.

8 Discussion, future work, and conclusions

8.1 Limitations

Modeling and evaluation of modeling mechanisms contains author bias. All modeling was performed by researchers who were involved in development of the system.

The companies whose products were modelled were either existing or potential research partners. In other words, the sample of companies was not selected e.g. to cover most challenging cases such as telecommunications networks.

8.2 Modeling mechanisms

Some partial models were created due to resource constraints, or when the purpose of modeling was attainable with partial modeling. In other words, capabilities of PCML or WeCoTin did not limit the scope of modeling. However, Floats, fixed point numbers or integers with very large domain would have been useful in the insurance and compressor domains. In the insurance case some specification variables such as a desired amount of monetary coverage were specifiable with arbitrary monetary amounts, which can lead to very large domains. Apartment size and desired coverage were discretized in model "16 Insur 1". In compressor models, the company had calculated and validated combinations of specification variable values that produce a specific nominal capacity, represented in a table constraint. Further, a specific percentage of capacity loss is encountered in high altitude use environments. Calculating this would have been more convenient with floating or fixed point arithmetic.

Application of the compositional structure was important but less frequent than anticipated. A partial explanation is that it was often considered more practical to model alternative or optional components as enumeration or Boolean attributes rather than as a part, if there was no need to configure details of the selected component individuals.

Part definitions with cardinality were useful: the mechanism provides a convenient way to model selecting at most one or exactly one component individual to a role in product structure out of several alternatives. This capability prevents the need for a number of extra constraints. For example, some commercial systems require that each alternative is specified as optional, and a mutual exclusivity constraint is required [Damiani, et al. 2001]. However, sometimes the mechanism was a bit clumsy: in case of an optional part (cardinality 0 to 1), and exactly one allowed type, it was difficult to invent a name for the component type and for the part. A bit surprisingly, large cardinalities were not needed in these configuration models.

Applying inheritance saved modeling effort in larger models significantly. Almost half (49%) of effective attributes were inherited, and one definition in a supertype created in average 4.4 effective attributes. Refinement of inherited attributes and parts was a useful mechanism for

limiting the domain of allowed values or allowed types. created through statistics. Refinement facilitated the application of inheritance also in cases where some subtypes have a narrower range of allowed values or types. Inheritance related to compositional structure was also useful, but was applied only in about 31% of the models. This mechanism was generally used in larger models, where almost a third of part definitions were inherited.

There was no need for explicit resource balancing or satisfaction in the modeled cases. There was no need for topological modeling, e.g. ports in our modeled cases. However, when modeling services and their delivery processes [Tiihonen, et al. 2007], there was a need to assign different stakeholders as resources that participate in different service activities. This assignment can be somewhat clumsily modeled with attributes. However, allocation of responsibilities to different, dynamically defined stakeholders could be more naturally modeled as connections between the activities and stakeholders.

8.3 Future work

The amount of work required to create a configuration model depended to a large extent on the knowledge acquisition and validation work. Collecting reliable statistics on total effort of creating and maintaining configuration models remains future work.

Performance evaluations are important characterization of configuration models. In previous work, performance of some of the models has been evaluated, and was found satisfactory[Tiihonen, et al. 2002]. However, performance should be tested with a larger and more representative set of configuration models.

8.4 Conclusions

The main contribution of this paper is providing, to our knowledge, the first multi-case in-depth characterization of configuration models and analysis of utility of modeling mechanisms. A combination of taxonomic hierarchy with inheritance and strict refinement, compositional structure with the concept of part definitions, attributes, and constraints for expressing consistency requirements of a configuration seem to be able to effortlessly capture a significant subset of sales configuration problems. The utility of inheritance in configuration was shown through significant application of the mechanism, especially when related to attributes, and to a lesser but still significant extent to parts.

In addition, this work provides an initial proposal for a framework for characterizing configuration models.

Acknowledgements

We thank A. Anderson, A. Martio, J. Elfström, K. Sartinko, M. Heiskala, M. Pasanen, and T. Kojo for modeling and knowledge acquisition; A. Martio and R. Sulonen for acquisition of the cases; Gardner Denver Finland, KONE, Patria,Tapiola Group for sharing product information; and TEKES for funding WeCoTin, ConSerWe, and Cosmos

References

Barker, V. E., O'Connor, D. E., Bachant, J., & Soloway, E. (1989). Expert systems for configuration at digital: XCON and beyond. Communications of the ACM, 32(3), 298-318.

Damiani, S. R., Brand, T., Sawtelle, M., & Shanzer, H. (2001). Oracle configurator developer User's guide, release 11i Oracle Corporation.

Fleischanderl, G., Friedrich, G., Haselbock, A., Schreiner, H., & Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. Intelligent Systems and their Applications, IEEE [See also IEEE Intelligent Systems], 13(4), 59-68.

Haag, A. (2008). What makes product configuration viable in a business? Proceedings of ECAI 2008 Workshop on Configuration Systems, Patras, Greece. 53-54.

Kojo, T., Männisto, T. And Soininen, T. (2003). Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. In Software Configuration Management (ICSE Workshops SCM 2001 and SCM 2003 Selected Papers), 86-101.

Nurmilaakso, J. (2004). WeCotin.calc documentation. Unpublished manuscript.

Peltonen, H., Tiihonen, J., & Anderson, A. (2001). Configurator tool concepts and model definition language. Unpublished manuscript.

Schreiber, A. T., & Birmingham, W. P. (1996). Editorial: The Sisyphus-VT initiative. International Journal of Human-Computer Studies, 44(3-4), 275-280.

Soininen, T., Niemelä, I., Tiihonen, J., & Sulonen, R. (2001). Representing configuration knowledge with weight constraint rules. Proceedings of the AAAI Spring Symp.on Answer Set Programming: Towards Efficient and Scalable Knowledge, , 195–201.

Soininen, T., Tiihonen, J., Mannisto, T., & Sulonen, R. (1998). Towards a general ontology of configuration. AI EDAM, 12(04), 357-372.

Tiihonen, J., Heiskala, M., Paloheimo, K., & Anderson, A. (2007). Applying the configuration paradigm to mass-customize contract based services. Paper presented at the Extreme Customization: Proceedings of the MCPC 2007 World Conference on Mass Customization & Personalization, Massachusetts Institute of Technology, MA, USA. paper ID MCPC-134-2007, section 7.5.3.

Tiihonen, J., Soininen, T., Niemelä, I., & Sulonen, R. (2003). A practical tool for mass-customising configurable products. Paper presented at the Proceedings of the 14th International Conference on Engineering Design, Stockholm, Sweden. Paper 1290.

Tiihonen, J., Soininen, T., Niemelä, I., & Sulonen, R. (2002). Empirical testing of a weight constraint rule based configurator. Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence, Lyon, France, 2002. 17–22-17-22.

Wielinga, B., & Schreiber, G. (1997). Configuration-design problem solving. Expert, IEEE [See also IEEE Intelligent Systems and their Applications], 12(2), 49-56.

Interactive Configuration and Time Estimation of Civil Helicopter Maintenance

É. Vareilles[a], C. Beler[a], E. Villeneuve[(a,b)], M. Aldanondo[a] and L. Geneste[b]
[a]: Université de Toulouse - Mines d'Albi
[b]: Université de Toulouse - ENI de Tarbes
Corresponding author: elise.vareilles@mines-albi.fr

Abstract

This communication is a prospective study which looks at the possibility of configuring the maintenance of civil helicopters and of estimating the maintenance time during the pre-check phase. This study is one part of a project called *Helimaintenance*, funded by the French Government with the agreement of the *Aerospace Valley* Association. It involves academic laboratories and SMEs. In the first section we present the problem and one of the solution investigations. In the second section, we present a configuration model based on constraints, relevant to maintenance configuration and time estimation. In the third and final section we present the first possibility we have identified for coupling this model to a CBR. Through out the paper we use one particular example to illustrate our proposal.

1 Introduction

The aim of this communication is to present a prospective study on the development of a configuration system which will simultaneously enable the interactive configuration of, and time estimation for civil helicopter maintenance. This problem originates from a French project called *Helimaintenance* which aims, using experts' know-how and process optimization, to reduce the cost of civil helicopter maintenance by 30%. The first step of our work package will be to model the maintenance of civil helicopters following the constructors' documentation and to estimate the time needed by taking into account the life of the helicopter and the workload of the maintenance outfit.

1.1 The Problem of Helimaintenance

According to [Norme NF X60012, 2006], maintenance means any of the operations required to maintain or re-establish a product in a specified state or to guarantee a predetermined service. In the aeronautic field average maintenance costs during the life of a craft are higher than the initial purchase costs for the products. We must notice that helicopter maintenance cost represents 45% of the overall life cycle cost. In order to reduce the cost of maintenance, some constructors are trying to control these costs right from the initial development phase of the products [Poncelin and al., 2006]. The goal of the *Helimaintenance* project is to optimize the maintenance process and reduce the cost of maintenance by capitalizing on and reusing experts know-how.

Figure 1: Helicopter example

The essential point of civil helicopter maintenance is that, in order to maintain the airworthiness of a craft, the *Maintenance Report Board* or *MRB* given by the constructors must be follow absolutely to the letter. The *Maintenance Report Board* defines the general cycle of maintenance for a family of helicopters, see Fig. 1: it specifies the time interval between maintenance operations, the type of maintenance operation, and the exact record describing each maintenance operation for a given type of helicopter, with all its possible options.

The maintenance schedules are established by the constructors in terms of exposure to the conditions that can cause failures. The most widely used usage parameters are:

- calendar time (weeks, months, years) to take into account fatigue due to the helicopter age,
- and usage factors (number of landings and take-offs, flight cycles, flight hours) to take into account part failures particular to the use of helicopters.

There are normally three types of maintenance services:

- complete service *CS*: a helicopter is completely dismantled and all of the parts are tested and replaced if necessary,
- interim service *IS*: only the critical parts are tested and changed if necessary (blades, rotor, for example),
- mini service *MS*: the less critical parts are tested and replaced if necessary (air-conditioning, seats or oil changes for example).

Every service records give an indication of the time needed to carry out the associated maintenance. In theory, the con-

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

structors can estimated the time for all the helicopter configurations.

Our aim is firstly to remodel the *MRB* with a specific helicopter and its maintenance processes and options in mind, and secondly, to help those who maintain it to estimate the time needed by taking into account the life and use of helicopter and also some information relative to the workshop where it will be maintained.

The time estimation is critical in order to give a correct quote for a maintenance operation, to deal with any possible risks caused by the operations itself and more importantly, to avoid keeping the helicopter out of service any longer than necessary.

1.2 Solution Investigations

According to [D.C. Brown, 1985] and [Coyne and al., 1990], product design can be characterized with respect to a degree of recurrence in: creative, innovative and routine design. In the case of extreme routine design called configuration, all design possibilities can be investigated completely. Many authors such as [Tsang, 1993], [Sabin and Freuder, 1996] have shown that product configuration could be efficiently modeled and aided when considered as a CSP (Constraints Satisfaction Problem).

In the same way, we can consider the maintenance of helicopters as configuration problems by the fact that all maintenance operations have already been identified and characterized in the *MRB*. The *MRB* has to be personalized depending on the options, the life of the helicopter and its use: the type of service and the set of service records describing each operation of maintenance have to be determined. Thanks to the description of a helicopter (for instance, its age, its type, its set of options: air-conditionning, winch, additionnal seats) and the *MRB*, the maitenance operators determine the subset of operations to be done: they configure the *MRB*.

The time estimation of a maintenance operation can also be represented by a continuous CSP by the fact that the *MRB* gives a therotical time to carry each maintenance operation for the different helicopter types. This theoritical time is usually lower than the real time spent to carry out the maintenance because of several factors such as unforseen events, workshop capacity, etc. The experts in maintenance are able to give an idea of the real time and the parameters that have an impact on it.

A CSP is defined by a set of variables, a set of finite domains (one for each variable) and a set of constraints linking the variables [Montanari, 1974]. The variables can be either discrete or continuous. The constraints can either be of compatibility, when defining the possible or forbidden combinations of values for a set of variables (lists of compatible values, mathematical expressions, temporal relations), or of activity, when allowing for the activation of a subset of variables and constraints [Mittal and Falkenhainer, 1990].

As we are at the initial state of the project, the example used to illustrate our proposition is the time estimation of the maintenance of three types of helicopters (the *Puma*, the *Gazelle* and the *Dolphin*) after customize of the *MRB* and the theoretical time needed to carry out the maintenance.

1.3 Organization of the paper

The aim of this communication is therefore to propose a global approach based on CSP to estimate the time of maintenance of helicopters. The rest of the paper is organized as follows. In the second section we present the configuration model of the maintenance time estimation. We show that different kinds of constraints are necessary in order to make this model. In the third section we put forward some ideas on the coupling of a *Case Based Reasoning* and the CSP model in order to validate, extract and update maintenance knowledge.

2 Maintenance Time Estimation

The aim of this section is to present the maintenance time estimation of a helicopter. The model is described as a CSP and outlines the kind of variables and constraints necessary to make it.

2.1 Theoretical Maintenance Time Estimation

Up to the present time, very few parameters have been identified to evaluate the theoretical time of helicopter maintenance. There are many reasons why only critical parameters have been modeled from now:

- the *MRB* for a helicopter family is so large that it can take several filing cabinets,
- all that exists at the moment are paper versions of the *MRB*. An *Enterprise Resource Planning* will be going to be deployed in the SMEs to manage several enterprise processes and the *MRBs*,
- in our study we have concentrated on the time estimation of a maintenance service, we take into consideration only the parameters directly linked to the theoretical time: the type of helicopter and the type of service.

In order to determine the theoretical maintenance time given by the *MRB*, we need to know:

- the helicopter type: a symbolic variable *HT* for the helicopter type must be defined. Its domain is {Puma, Tiger, Dolphin },
- the kind of service: a symbolic variable *ST* for service type must be defined. Its domain is {MS, IS, CS } with MS meaning mini service, IS meaning interim service and CS, complete service,
- the theoretical time: a continuous variable *TT* for theoretical time must be defined. Its domain is [120, 2000] hours according to the *MRB* of the considered helicopters.

Compatibility constraints represent the permissible or forbidden combinations of parameter values. These kinds of constraints can be described as tables of permissible values or as mathematical expressions. In our case, a compatibility table exists expressing the relation between the type of helicopter, variable *HT*, the type of visit, variable *VT* and the theoretical time to carry the maintenance, variable *TT*, as described in table 1.

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.

Table 1: Example of theoretical maintenance time estimation

HT	ST	TT
Puma	MS	≤ 140
Dolphin	MS	≤ 140
Gazelle	MS	≤ 120
Puma	IS	≤ 140
Dolphin	IS	0
Gazelle	IS	[140, 550]
Puma	CS	[600, 1200]
Dolphin	CS	[800, 2000]
Gazelle	CS	[1200, 1800]

2.2 Real Maintenance Time Estimation

The theoretical time estimation can be modulated by a large number of factors, some of them relevant to the use of the helicopter and the risk of an unforeseen event during maintenance, and the others down to the workload of the workshops.

The final or real maintenance time estimation *FT* results from the multiplication of the theoretical time extracted from the *MRB* by two modulation parameters: the first one, c^1 is linked to the use of the helicopter and the second one c^2 to the workload of the workshop.

The final maintenance time is computed by:

$$FT = TT * c^1 * c^2$$

Parameters characterizing the Risk of an Unforeseen Event

The experts in helicopter maintenance have identified 2 external parameters which have an impact on the real maintenance time:

- the type of flight condition, which can be normal or severe,
- the conditions in use, which can be normal or unusual.

These two parameters are used to qualify the severity of the atmosphere, which can be high, medium or normal. This particular parameter is used in the computation of the final maintenance time. The experts have associated to each of its symbolic value a numerical value: hihg matches with number 3, medium with number 2 and normal with number 1.

So we have then introduced into the model:

- a symbolic variable *FC* for the flight condition. Its domain is {normal, severe},
- a symbolic variable *UC* for the conditions in use. Its domain is {normal, unusual},
- a numerical variable *AS* for the severity of the atmosphere. Its domain is {1, 2, 3}.

The relation between the flight, variable *FC*, the conditions in use, variable *UC* and the severity of the atmosphere, variable *AS*, is described in table 2.

The experts have also decided to give more influence to:

- the age of the helicopter,
- the number of flight hours.

As these parameters are used to compute the final maintenance time, we have modeled them with continuous variables:

Table 2: Characterization of the severity of the atmosphere

FC	UC	AS
normal	normal	1
normal	unusual	2
severe	normal	2
severe	unusual	3

- a variable *HA* for the helicopter age. Its domain is {1, 2, 3}, 1 meaning that the helicopter age is under 5 years, 2 meaning that it is between 5 and 25 years, and 3 meaning that it is more than 25 years.
- a variable *FH* for the number of flight hours. Its domain is {1, 2, 3 }, 1 meaning that the helicopter has flown less than 1000 hours, 2 meaning that it has flown between 1000 and 5000 hours and 3 meaning that it has flown longer than 5000 hours.

These scales of age and flight hours have been defined by the experts.

The risk of an unforeseen event is calculated by a mathematical formula linking the previous numerical parameters. The experts have given more importance to the atmosphere severity than to the two other aspects. The risk of an unforeseen event is the continuous variable *UR* and is then computed by :

$$UR = 2 * AS + HA + FH$$

Its domain is therefore $\{[4, 12]\}$.

The unforeseen event risk is linked to the modulation parameter c^1 by a continuous compatibility table, as shown in table 3.

Table 3: First modulation parameter values

UR	c^1
$[4, 6]$	$[1, 1.05]$
$[6, 10]$	$[1.05, 1.2]$
$[10, 12]$	$[1.2, 1.3]$

Parameters characterizing the Workshop Capacity

The workload of the workshop has an enormous impact on the real time necessary for a maintenance operation. If the mechanics are too busy or if the tools or machines are unavailable, the time it takes to maintain one helicopter or a fleet of helicopters will increase. In order to take these factors into account, we have added to our model some information relative to the workload in the workshop.

The experts in helicopter maintenance have identified 2 external parameters which have an impact on real maintenance time:

- the average mechanics' availability, which can be busy or free,
- the average machines availability, which can be busy or free.

We have modeled them with symbolic variables:

- a variable *HA* for the average human availability. Its domain is {busy, free },

- a variable *MA* for the average machines availability. Its domain is {busy, free }.

These two parameters are directly linked to the second modulation parameter c^2 by a compatibility table, as shown in table 4.

Table 4: Second modulation parameter values

HA	MA	c^2
free	free	$[1, 1.1]$
free	busy	$[1.1, 1.35]$
busy	free	$[1.1, 1.35]$
busy	busy	$[1.35, 1.5]$

Final Maintenance Time Estimation

As we have already said, the final maintenance time estimation *FT* results from the multiplication of the theoretical time given by the *MRB* by two modulation parameters c^1 and c^2.

Its domain is then computed thanks to interval analysis, proposed by [Moore, 1966]:

$$\begin{aligned} \text{FT} &= TT * c^1 * c^2 \\ &= ([1, 550][600, 2000]) \bigotimes [1, 1.3] \bigotimes [1, 1.5] \\ &= [1, 3900] \end{aligned}$$

We can see that taking into account the use of the helicopter characterizing the risk of an unforeseen event, and the workload of the workshop the final maintenance time can be effectively multiplied by two compared to the theoretical time given by the constructors.

The general constraints model is shown in figure 2. The squares represent the variables and the lines the constraints (the bold lines represent the numerical ones).

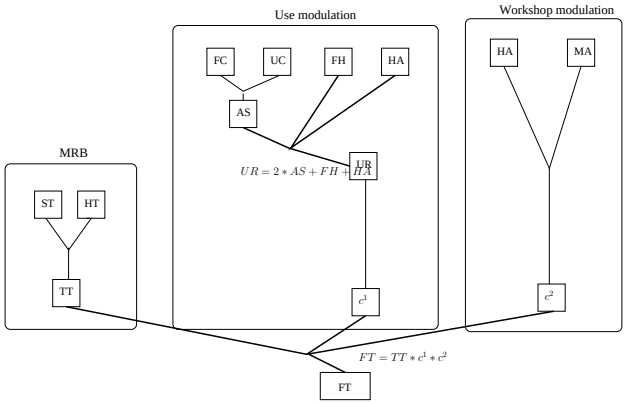

Figure 2: General architecture of model

Even if our first model is quite simple,it has shown that this kind of tool is of great interest for the industrialists involved in the project. Such a tool could help speed up decision time in quoting for any maintenance delay. Nevertheless, to be really efficient, the model must be upgraded by more knowledge provided by the *MRB* and workshop know-how. Once we have finally tuned our model, it must be validated under real conditions.

3 Coupling CSP and CBR

In Case-Based Reasoning or *CBR* [Maher *et al.*, 1995], expertise is embodied in a library of past cases, rather than being encoded in classical rules. Each case typically contains a description of the problem, plus a solution and/or the outcome. The knowledge and reasoning process used by experts to solve the problem is generally not recorded, but is implicit in the solution. In order to find a solution to a new problem, the user describes her/his problem through a list of parameters and after all the user's inputs, the problem described is matched against the cases in the base.

A similarity function [Kolodner, 1993] makes it possible to detect and classify the most similar or the most adaptable cases. We must point out that if the user's problem does not match with any past cases, the system will return the nearest possible ones. The cases retrieved provide ballpark solutions that must actually be adapted by the user to fit her/his current problem.

In the *Helimaintemance* project, each maintenance visit represents a particular case to stock, re-use and adapt. In that respect, we need to have a specific parameter corresponding to the time really spent to maintain a given helicopter in a given situation in order to have an idea about the time usually spent on a given maintenance.

As CSP and CBR are close enough, at least in their formulation, we have considered and are about to investigate different approaches associating CSP and CBR.

3.1 Sequential estimating process: CBR first then CSP

In this first situation, the CSP could help the user to adapt her/his case to the current problem, as has been proposed in [Geneste and Ruet, 2001], [Weigel R and Torrens, 1998] and [L., 1998]. This could be useful in re-estimating the time needed using the example of a previous situation which may be only very slightly to a current one. For this situation, the set of variables used for CBR and CSP is roughly the same. To this purpose, the constraint model can be used to make small changes in the variable values in order to get close to the investigated case, as shown in Fig. 3.

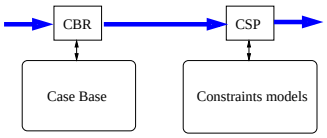

Figure 3: CBR-CSP sequencing

3.2 Sequential estimating process: CSP first then CBR

This situation is the opposite of the first one and considers that the known variable values (set KV) that should be inputted in the CBR process are not numerous enough to discriminate efficiently between cases, as shown in Fig. 4. The idea is therefore to use the domain knowledge captured by the

constraints of the CSP model in order to process some pruning of variables with some constraint filtering. The remaining variable values are still consistent with the inputs and the variables that have a single value domain can be added to set KV. The previous set (KV) increases and that provides a more efficient retrieval step of the CBR process.

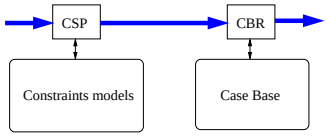

Figure 4: CSP-CBR sequencing

3.3 Building or tuning CSP cased based model

This situation considers the elaboration of the CSP model used in the previous section. Constraints used in CSP models correspond to domain knowledge that is obtained most of the time from human experts or scientific or technical publications. The domain knowledge for time estimation problem is very weak, it is therefore necessary to consider other knowledge sources. The idea is to consider the knowledge embedded in each case. The proposition is, using different computation techniques: pattern recognition, regression analysis or data mining techniques, to extrapolate from past cases, rules or mathematical relations which enable to estimate maintenance duration, as shown in Fig. 5.

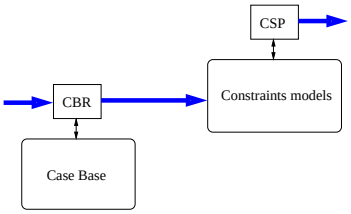

Figure 5: Building or tuning CSP

3.4 Checking case validity with a CSP

This situation considers a new occurrence of an effective maintenance operation with associated time estimation. In order to strength the quality of the case base, the idea is to check each case validity before storing, as shown in Fig. 6. To this purpose, the idea is to use the knowledge embedded in the constraint model to validate the new case. All the variable values describing the case are progressively inputted into the CSP in order to be sure that the new problem is conformed the model. At the end of the process, it is quite probable that the final variable values will not fit exactly due to the natural variance of the estimation.

3.5 Dealing with incomplete knowledge

Our final idea of associating CBR and CSP based reasoning is relevant to the possibility of processing with partial knowledge, as shown in Fig. 7. This means for CBR dealing with stored cases that are incomplete: some variable values can not

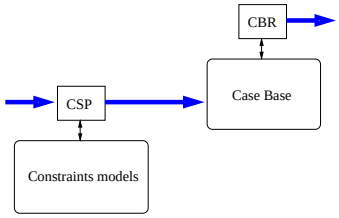

Figure 6: Checking cases validity

be provided because they have been manually lost, deleted, erased. For the CSP model, this means that some variables, important for describing the case, are not linked with the others by constraints (rules not defined, unstable relation, no clear correlation) [Inakoshi H. and N., 2001]. Two ideas are considered. The first considers the cases in order to replace a missing constraint. Let us assume an estimation problem described with 20 variables V_i, $I = [1, 16]$ and two constraints networks linking V_j, $j = [1, 8]$ and V_k, $k = [9, 16]$. It is clear that some knowledge between the two is missing. The idea is to use retrieved cases in order to link the two constraint problems. The aim of the second one is to be able to operate on incomplete stored cases. Let us assume an estimation problem described with 20 variables and a case where a value for five of them is not given while the 15 others fit rather well. The case retrieved and the constraint model can be used to provide the five missing values.

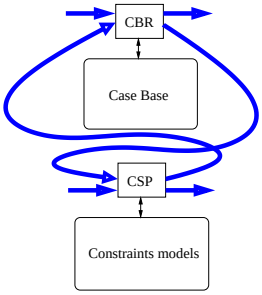

Figure 7: Adding knowledge

This kind of approach seems very interesting for the mechanics because they will be able to capitalize and re-use old experiments easily. But they must still keep in mind that having formalized knowledge as in the CSP model is useful in order to understand why a particular effect has such an impact on the maintenance process. That's why, we think that having both these tools, and coupling them together could help reduce the cost of the civil helicopter maintenance.

4 Conclusion

The aim of this communication has been to present a prospective study on the time estimation of civil helicopter maintenance.

Firstly, we have presented our problem and one of the solution investigations. Then we have suggested a way of remodeling maitenance documentation to give a more accurate

idea of the theoritical time needed to carry out a given maintenance operation and have shown a method of remodulating this time by considering several factors linked to the use of the helicopter and the workload of the workshop, seen as CSP.

In the final section we have put forward some ideas about the coupling of CSP approaches with CBR approaches in order to take advantage of both to improve the way of estimating the maintenance time.

All the propositions have been validated by the maintainers. The example is available at `http://cofiade.enstimac.fr/cgi-bin/cofiade.pl` under the name *Helimaintenance*.

As we are still in the very earliest stage of the project, some improvements and updating of the model will be necessary. The next phase of our work package, is to model the part of the *MRB* which corresponds to the determination of the theoretical time, to develop the CBR tool able to stock maintenance cases, to improve the CSP model of time-estimation and to investigate the coupling of the CSP and the CBR.

Acknowledgments

The authors wish to acknowledge the *Aerospace Valley* Association and all partners in the project, and in particular the experts of IXAIRCO for their involvement in the building of the CSP model.

References

[Coyne and al., 1990] R.D. Coyne and al. *Knowledge-Based Design Systems*. Addison-Wesley, 1990.

[D.C. Brown, 1985] B. Chandrasekaran D.C. Brown. Expert systems for a class of mechanical design activity. In *Computer-Aided Design*, pages 259–282, North-Holland, 1985.

[Geneste and Ruet, 2001] L. Geneste and M. Ruet. Experience based configuration. In *International Joint Conference on Artificial Intelligence Workshop on Configuration*, Seattle, USA, 2001.

[Inakoshi H. and N., 2001] Ohta Y. Inakoshi H., Okamoto S. and Yugami N. Effective decision support for product configuration by using cbr. In *Workshop ICCBR*, 2001.

[Kolodner, 1993] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publisher, 1993.

[L., 1998] Purvis L. A cbr integration from inception to productization. In *AAAI Technical Report*, 1998.

[Maher *et al.*, 1995] M. Maher, D. Zhang, and M. Balachandran. *Case-Based Reasoning in Design*. Lawrence Erlbaum Associates, 1995.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI*, pages 25–32, Boston, US, 1990.

[Montanari, 1974] U. Montanari. Networks of constraints: fundamental properties and application to picture processing. In *Information sciences*, volume 7, pages 95–132, 1974.

[Moore, 1966] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Norme NF X60012, 2006] *Norme NF X60012*, 2006. Maintenance. - Termes et dfinitions des lments constitutifs des biens et de leur approvisionnement.

[Poncelin and al., 2006] Guillaume Poncelin and al. *Product Life Cycle Aspects*, chapter Design to maintenance cost by the control of the products environment, pages 95–107. Springer, December 2006. DOI: 10.1007/978-2-287-48370-7.

[Sabin and Freuder, 1996] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[Tsang, 1993] E. Tsang. *Foundations of constraints satisfaction*. Academic Press, London, 1993.

[Weigel R and Torrens, 1998] Faltings B.V Weigel R and M. Torrens. Interchangeability for case adaptation in configuration problem. In *AAAI Technical Report*, 1998.

Author Index

Markus Stumptner and Patrick Albert, Editors.
Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09),
July 11–13, 2009, Pasadena, CA, USA.